View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Integrating verification, testing, and learning for cryptographic protocols

Martin Oostidijk, Vlad Rusu, Jan Tretmans, Rene de Vries, Tim Willemse

To cite this version:

Martin Oostidijk, Vlad Rusu, Jan Tretmans, Rene de Vries, Tim Willemse. Integrating verification, testing, and learning for cryptographic protocols. Integrated Formal Methods, 2007, Oxford, United Kingdom. inria-00564231

HAL Id: inria-00564231

https://hal.inria.fr/inria-00564231

Submitted on 8 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Integrating Verification, Testing, and Learning for Cryptographic Protocols*

M. Oostdijk[1,4], V. Rusu[2], J. Tretmans[1,3], R.G. de Vries[1], and T.A.C. Willemse[1,4]

[1] Radboud University, Nijmegen, NL
[2] Irisa/Inria Rennes, FR
[3] Embedded Systems Institute, Eindhoven, NL
[4] Eindhoven University of Technology, NL
[5] Riscure, Delft, NL

Abstract. The verification of cryptographic protocol *specifications* is an active research topic and has received much attention from the formal verification community. By contrast, the black-box testing of actual *implementations* of protocols, which is, arguably, as important as verification for ensuring the correct functioning of protocols in the "real" world, is little studied. We propose an approach for checking secrecy and authenticity properties not only on protocol specifications, but also on black-box implementations. The approach is compositional and integrates ideas from verification, testing, and learning. It is illustrated on the Basic Access Control protocol implemented in biometric passports.

1 Introduction

The verification of cryptographic protocols has been an active research topic for at least the last two decades. Early approaches consisted in developing dedicated logics for specification and inference rules [1,2,3], which a user applied "by hand". More recently, automatic, or, at least, computer-assisted techniques have emerged. These include model checking [4,5], theorem proving [6,7] and combinations of these two techniques [8]. Other approaches are based on term rewriting techniques [9] sometimes combined with abstract interpretation [10,11,12]. The above list, albeit incomplete, shows that most formal verification techniques have been applied, or adapted to, cryptographic protocol verification.

The situation is quite different in the area of testing of black-box implementations of protocols[1]. A thorough search of computer science research bibliographies revealed only a few related works. Closest to ours is [14], where an executable implementation is instrumented (hence, it is not really a black box) to detect violations of security properties. In other works [15,16], various source-code verification techniques have been applied to source-code implementations.

* This work was done while the second author was visiting the university of Nijmegen.

[1] Here, by "implementation" we mean black-box executable code, which is controllable/observable only through some interfaces, e.g., like in conformance testing [13].

J. Davies and J. Gibbons (Eds.): IFM 2007, LNCS 4591, pp. 538–557, 2007.
© Springer-Verlag Berlin Heidelberg 2007

All these works assume in one way or another that some kind of source code of the protocol is available, which may not always be the case.

Outside the academic world, practitioners have also developed empirical approaches for testing security protocols. The tester (or "cracker") will try to find whatever information might leak from a protocol implementation, by applying ad-hoc techniques such as sending arbitrary messages, trying to overflow buffers, or attempting to break cryptography by brute force. Some of these techniques were tried when testing the new Dutch biometric passport [17].

Clearly, there are differences between verification and testing techniques for cryptographic protocols. Some are the usual differences between the two: formal verification may prove either presence or absence of errors in specifications, while testing may only prove the presence of errors in implementations. Other differences are specific to the present field of interest:

- in verification, specifications are often very partial, in the sense that only some inputs and outputs are specified. Of course, only the specified behaviour, together with that of an implicit "intruder", e.g., following the so-called Dolev-Yao model [18] can be verified. This amounts to saying that the intruder does not "try" to feed the honest agents with messages that they do not " expect" (i.e., whose reception is not specified in the protocol).
- testing for security is not limited to the behaviour of the protocol as described by an (incomplete) specification; rather, the protocol's behaviour outside the specification is also targeted, in the hope that violations of security properties will be observed.

The two techniques are different, yet complementary: verification proves correctness of the specification against a given set of security properties, whereas testing checks correctness of the implementation with respect to the specification and, outside the specification, with respect to the security properties. In this paper we propose an approach that builds on this complementarity and, moreover, performs *learning* from implementations to "complete" the incomplete specifications. The approach can be roughly described as follows:

1. the protocol's specification is automatically verified against a given set of security properties (e.g., secrecy and authenticity properties);
2. if the properties hold on the protocol's specification, we proceed with the learning step, which consists in augmenting each agent's specification with a set of new behaviours, obtained by testing/interacting with their respective implementations; then, the process continues at Step 1;
3. otherwise, the verification finds a violation of a property on the protocol's specification, and produces a counterexample. Then, we attempt to execute the counterexample on the black-box implementation:
   (a) if the attempt succeeds, a violation of a property on the protocol's global implementation has been found, and the procedure terminates.
   (b) if the attempt does not succeed, the last learning step is responsible; hence, it is modified, and the process is continued at Step 2.

The global procedure can terminate by reporting a violation of a security property by the protocol's implementation, or when all "representative" traces up to a certain length have been learned. In the latter case, the conclusion is that the implementation satisfies the security properties, provided that the testing/learning was "exhaustive". This "exhaustiveness" condition is formally defined in the paper, and notions of soundness and (theoretical) completeness of the approach are formally defined and proved.

The rest of the paper is organised as follows. In Section 2 we introduce the model of IOSTS (Input-Output Symbolic Transition Systems), which we use for writing specifications of cryptographic protocols. In Section 3 we present the ingredients of our approach: verification of security properties (secrecy, authentication) expressed using observers (which are IOSTS augmented with certain acceptance conditions) against protocol specifications expressed as parallel compositions of IOSTS; and learning new behaviours of a specification by testing a black-box implementation of the protocol and observing/interpreting the results. In Section 4 our approach integrating verification, testing, and learning is defined. The approach is demonstrated on the Basic Access Control protocol, which is part of the security mechanisms implemented in biometric passports [19].

2 Models

The IOSTS model (*Input/Output Symbolic Transition Systems*) is inspired from I/O automata [20]. We specialise here IOSTS for modelling security protocols. The symbolic data that our protocol-oriented IOSTS manipulate are of three main *sorts*: the sort $\mathcal{M}$ of *messages*, the sort $\mathcal{K}$ of *keys*, and the sort $\mathcal{N}$ of *nonces*. Keys and nonces are subsorts of messages. We define a *composition* (i.e., concatenation) function $''.'' : \mathcal{M} \times \mathcal{M} \mapsto \mathcal{M}$ as well the decomposition of a composed message into its components, using the usual axiomatic way. We also define a (symmetrical) *encryption function* $''\{\}_'' : \mathcal{M} \times \mathcal{K} \mapsto \mathcal{M}$, with the property that $\{\{m\}_k\}_k = m$, for all $m : \mathcal{M}$ and $k : \mathcal{K}$[2]. We enrich this signature with the usual Boolean sort, and obtain a simple equational theory, with the usual notion of *terms* (closed as well as with free variables). We denote by *free*(trm) the set of free variables of a term trm. Each term has a *smallest sort* (with the convention that the sorts $\mathcal{M}$, $\mathcal{K}$, and $\mathcal{N}$ are ordered such that $\mathcal{K}$ and $\mathcal{N}$ are *smaller* than $\mathcal{M}$). We shall need the following notion: a term trm_1 is *sort-compatible* with a term trm_2 if the smallest sort of trm_1 is smaller than or equal to the smallest sort of trm_2. The semantics of closed terms is given by the usual *initial algebra* of our signature.

2.1 Syntax of IOSTS

Definition 1 (IOSTS). *An IOSTS is a tuple* $\langle V, P, C, \Theta, L, l^0, \Sigma^?, \Sigma^!, \Sigma^\tau, \mathcal{T}\rangle$ *where*

[2] In this paper we only use symmetrical encryption. If needed, asymmetrical encryption can also be defined in a similar way.

- *V is a finite set of* state variables[3]*, P is a finite set of* formal parameters*, and C is a finite set of* symbolic constants*, all of which can be of any of the above sorts $\mathcal{M}$, $\mathcal{K}$, or $\mathcal{N}$.*
- *Θ is the* initial condition*, a term of Boolean sort, with free$(\Theta) \subseteq V$,*
- *L is a nonempty, finite set of* locations *and $l^0 \in L$ is the* initial location*,*
- *$\Sigma^?$ is a set of* input actions*, $\Sigma^!$ is a set of* output actions*, and Σ^τ is a set of* internal actions*. For each action $a \in \Sigma^? \cup \Sigma^!$, its* formal parameter list *par$(a) = \langle p_1, \ldots, p_k \rangle \in P^k$ ($k \in \mathbb{N}$) is a tuple of pairwise distinct formal parameters. We say that the action a* carries *the formal parameters in par(a). By convention, internal actions $\tau \in \Sigma^\tau$ carry no parameters.*
- *$\mathcal{T}$ is a nonempty, finite set of* transitions*. Each transition is a tuple $\langle l, a, G, A, l' \rangle$ made of:*
  - *a location $l \in L$, called the* origin *of the transition;*
  - *an action $a \in \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ called the* action *of the transition;*
  - *a term G of Boolean sort, with free$(G) \subseteq V \cup P$, called the* guard*;*
  - *an* assignment *A, which is a set of expressions of the form $(x := A^x)_{x \in V}$ such that, for each $x \in V$, A^x is a term that is sort-compatible with x, and free$(A^x) \cap P \subseteq$ free(G);*
  - *a location $l' \in L$ called the* destination *of the transition.*

For an IOSTS $\mathcal{S}$ we shall denote by $V_\mathcal{S}$ its set of state variables, by $C_\mathcal{S}$ its set of symbolic constants, by $L_\mathcal{S}$ its set of locations, etc. In graphical representations of IOSTS, the identifiers of input actions are followed by the the "?" symbol, and the identifiers of output actions are followed by the "!" symbol. These symbols are not part of the action's name, but are only used to easily identify inputs and outputs. Input and output actions are also called *visible* or *observable* actions; this in contrast with *internal* actions, which are not observable from the environment. Guards that are identically *true* are not depicted, and a variable not present in the assignments of a transition is assumed to be left unmodified.

The difference between state variables, constants and formal parameters is that a state variable's value can be modified, whereas a symbolic constant's value cannot. However, both state variables and constants always *have* values, while a formal parameter, say, p, has a value only during the instantaneous firing of the transition labelled by the action carrying p[4].

Example 1. The two IOSTS depicted in Figure 1 describe, respectively the behaviour of a terminal (in the left-hand side) and of a biometric passport (in the right hand side), executing the Basic Access control protocol [19]. This protocol is designed to ensure that the passport and the terminal mutually authenticate, and generate a certain secret session key, by which all communication between the two - after successful completion of Basic Access Control - will be encrypted.

Initially, both passport and terminal know a certain key K (different from the session key), which is a symbolic constant. The terminal initiates the protocol by sending a certain command, Get_Chal!, to the passport. Upon reception, the

[3] Not to be confused with the free variables appearing in terms of our signature.
[4] Cf. Section 2.2 on the formal semantics of IOSTS.

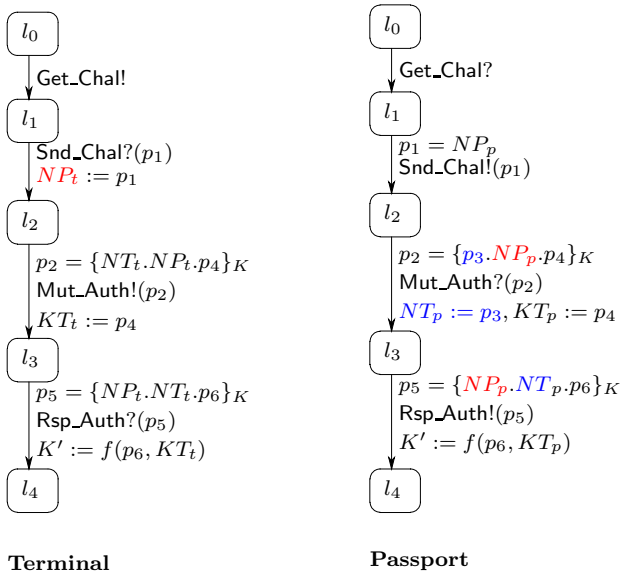

Fig. 1. Sample IOSTS: Basic Access Control in the Biometric Passport

passport replies by a Snd_Chal! response, carrying a formal parameter p_1 of sort nonce, whose value is equal to NP_p, the passport's nonce (a symbolic constant). The terminal receives this value and memorises it in its state variable NP_t, which is the terminal's copy of the passport's nonce. (State variables/symbolic constants of the passport are subscripted by p, those of the terminal, by t.)

Then, the terminal sends a Mut_Auth! output, carrying a formal parameter $p_2 = \{NT_t.NP_t.p_4\}_K$, that is, an encryption under K of a sequence consisting of: the terminal's nonce NT_t (a symbolic constant), the previously memorised passport's nonce NP_t, and a certain arbitrary value p_4 of sort $\mathcal{K}$. The value p_4 is stored in the terminal's variable KT_t containing so-called *key material* (to be used later). The passport accepts the output only if, on the passport's side, the formal parameter p_2 contains, under the encryption with the same key K, the passport's nonce NP_p, surrounded by two arbitrary values: p_3 and p_4. On the same transition, these values are stored, respectively, in the state variables NT_p, i.e., the passport's copy of the terminal's nonce, and KT_p (*key material*).

Next, the passport outputs a Rsp_Auth! response, together with a formal parameter p_5, of the form $p_5 = \{NP_p.NT_p.p_6\}_K$, i.e., an encryption with the same original key K of a concatenation of the nonces NP_p and NT_p, together with an arbitrary value p_6, to serve as *key material* of the passport. This response is accepted by the terminal only if it is able to decrypt and decompose its formal parameter p_5 and to find, at the beginning, its copies NP_t, NT_t of the nonces.

Finally, on their last transitions, both passport and terminal compute a new session K' as a function f, not specified here, of the key material exchanged [19].

Note that the behaviour of the passport and terminal in Basic Access Control is not completely specified by the above IOSTS (which closely follows the informal documents [19]). For example, nothing is said about what happens if a Get_Chal? input is received in a location different from l_0, or if the formal parameter carried by the Mut_Auth! output is not of the expected form. Later in the paper we shall make this specification more "complete" by means of *learning*.

2.2 Semantics of IOSTS

The semantics of IOSTS is described in terms of labelled transition systems.

Definition 2. *An input-output labelled transition system (IOLTS) is a tuple* $\langle S, S^0, \Lambda^?, \Lambda^!, \Lambda^\tau, \rightarrow\rangle$ *where* S *is a possibly infinite set of* states, $S^0 \subseteq S$ *is the possibly infinite set of* initial states, $\Lambda^?$, $\Lambda^!$, *and* Λ^τ *are possibly infinite sets of* input, output, *and* internal *actions, respectively, and* $\rightarrow \subseteq S \times (\Lambda^? \cup \Lambda^! \cup \Lambda^\tau) \times S$ *is the* transition relation.

Intuitively, the IOLTS semantics of an IOSTS $\langle V, C, P, \Theta, L, l^0, \Sigma^?, \Sigma^!, \Sigma^\tau, \mathcal{T}\rangle$ explores the reachable tuples of values (hereafter called *valuations*) of the variables of the IOSTS. Let $\mathcal{V}$ denote the set of valuations of the state variables, and Π denote the set of valuations of the formal parameters P. Then, for a term E with $\mathit{free}(E) \subseteq V \cup P$, and for $\nu \in \mathcal{V}$, $\pi \in \Pi$, we denote by $E(\nu, \pi)$ the value obtained by evaluating E after substituting each state variable by its value according to ν, and each formal parameter by its value according to π. In particular, when the term E does not include parameters, i.e., $\mathit{free}(E) \subseteq V$, the value obtained by evaluating E after substituting each state variable by its value according to ν is denoted by $E(\nu)$. For $P' \subseteq P$ and for $\pi \in \Pi$, we denote by $\pi_{P'}$ the restriction of the valuation π to a subset set $P' \subseteq P$ of parameters, and let $\Pi_{P'} \triangleq \{\pi_{P'} | \pi \in \Pi\}$.

Definition 3. *The semantics of an IOSTS* $\mathcal{S} = \langle V, C, P, \Theta, L, l^0, \Sigma^?, \Sigma^!, \Sigma^\tau, \mathcal{T}\rangle$ *is an IOLTS* $[\![\mathcal{S}]\!] = \langle S, S^0, \Lambda^?, \Lambda^!, \Lambda^\tau, \rightarrow\rangle$, *defined as follows:*

- *the set of states is* $S = L \times \mathcal{V}$,
- *the set of initial states is* $S^0 = \{\langle l_0, \nu\rangle | \Theta(\nu) = \mathit{true}\}$,
- *the set of input actions, also called the set of* valued inputs, *is the set* $\Lambda^? = \{\langle a, \pi'\rangle | a \in \Sigma^?, \pi' \in \Pi_{\mathit{par}(a)}\}$,
- *the set of output actions, also called the set of* valued outputs, *is the set* $\Lambda^! = \{\langle a, \pi'\rangle | a \in \Sigma^!, \pi' \in \Pi_{\mathit{par}(a)}\}$,
- *the set of internal actions is* $\Lambda^\tau = \Sigma^\tau$,
- $\rightarrow$ *is the smallest relation in* $S \times (\Lambda^? \cup \Lambda^! \cup \Lambda^\tau) \times S$ *defined by the following rule:*

$$\frac{t\colon \langle l,a,G,A,l'\rangle \in \mathcal{T} \quad \pi \in \Pi \quad \nu \in \mathcal{V} \quad G(\nu,\pi) = \mathit{true} \quad \pi' = \pi_{\mathit{par}(a)} \quad \nu' = A(\nu,\pi)}{\langle l,\nu\rangle \xrightarrow{\langle a,\pi'\rangle} \langle l',\nu'\rangle}$$

The rule says that the transition t is fireable when control is in its origin location l, and its guard G is satisfied by the valuation ν of the state variables

and π of the formal parameters. If this is the case, then the system moves to the destination location l, and the assignment A maps (ν, π) to ν', *via* the valued action $\langle a, \pi' \rangle$, where π' restricts the valuation π to the formal parameters $par(a)$ carried by the action a (if any; remember that internal actions do not carry parameters). Intuitively, this is because $par(a)$ are the only formal parameters "visible" from the environment. In the sequel, we let $\Lambda \triangleq \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$.

Definition 4 (Execution). *An execution fragment is a sequence of alternating states and valued actions* $s_0\alpha_0 s_1\alpha_1 ... \alpha_{n-1}s_n \in S \cdot (\Lambda \cdot S)^*$ *such that* $\forall i = 0, n-1, s_i \xrightarrow{\alpha_i} s_{i+1}$. *An* execution *is an execution fragment starting in an initial state. We denote by* $Exec(\mathcal{S})$ *the set of executions of the IOLTS* $[\![\mathcal{S}]\!]$.

Definition 5 (Trace). *The* trace $trace(\rho)$ *of an execution* ρ *is the projection of* ρ *on the set* $\Lambda^! \cup \Lambda^?$ *of valued actions. The set of traces of an IOSTS* $\mathcal{S}$ *is the set of all traces of all executions of* $\mathcal{S}$*, and is denoted by* $Traces(\mathcal{S})$.

We shall sometimes need to restrict the traces of an IOSTS in a given *environment*. This operation, together with the *parallel product* operation, defined below, allows for communication of values between IOSTS and enables us to formally define the interactions between agents in a protocol. These interactions ar similar to those encountered, e.g., in coordination models such as Linda [21].

Intuitively, an environment is an unordered channel connected to an IOSTS, and may contain, zero, one, or several instances or each valued action in $\Lambda^! \cup \Lambda^?$ (hence the multiset structure):

Definition 6 (environment). *For an IOSTS* $\mathcal{S}$*, an* environment *is a multiset* $\mathcal{E} : \Lambda^! \cup \Lambda^? \to \mathbb{N}$ *of valued inputs and outputs of the IOSTS.*

Then, the traces of an IOSTS that are "admissible" in an environment are those traces obtained by taking valued inputs from the environment and adding valued outputs to it. In the following definition, $\cup$ and $\setminus$ denote the usual union and difference operations on multisets.

Definition 7 (traces in environment). *A trace* $\sigma \in Traces(\mathcal{S})$ *is admissible in an environment* $\mathcal{E} \in \mathbb{N}^{\Lambda^! \cup \Lambda^?}$ *if the pair* $(\sigma, \mathcal{E})$ *belongs to the following recursively defined* admissibility *relation:*

- *Any pair* $(\epsilon, \mathcal{E})$ *where* ϵ *denotes the empty trace, is admissible,*
- *if* $(\sigma, \mathcal{E})$ *is admissible,* $\alpha \in \Lambda^!$ *is a valued output, and* $\sigma \cdot \alpha \in Traces(\mathcal{S})$*, then* $(\sigma \cdot \alpha, \mathcal{E} \cup \{\alpha\})$ *is admissible,*
- *if* $(\sigma, \mathcal{E})$ *is admissible,* $\alpha \in \Lambda^?$ *is a valued input,* $\alpha \in \mathcal{E}$*, and* $\sigma \cdot \alpha \in Traces(\mathcal{S})$*, then* $(\sigma \cdot \alpha, \mathcal{E} \setminus \{\alpha\})$ *is admissible.*

We denote by $Traces(\mathcal{S}, \mathcal{E})$ *the set of traces of* $\mathcal{S}$ *that are admissible in the environment* $\mathcal{E}$.

Lemma 1 (Monotonicity of admissible traces). *For all IOSTS* $\mathcal{S}_1$, $\mathcal{S}_2$ *and environment* $\mathcal{E}$*: if* $Traces(\mathcal{S}_1) \subseteq Traces(\mathcal{S}_2)$ *then* $Traces(\mathcal{S}_1, \mathcal{E}) \subseteq Traces(\mathcal{S}_2, \mathcal{E})$.

2.3 Parallel Product

The *parallel product* of two IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ will be used in specification and verification (for defining the protocol, and its interaction with the intruder and with "observers" for security properties). The parallel product operation is defined only for *compatible* IOSTS, defined as follows:

Definition 8. *Two IOSTS* $\mathcal{S}_1$,$\mathcal{S}_2$ *are* compatible *if* $V_{\mathcal{S}_1} \cap V_{\mathcal{S}_2} = \emptyset$, $P_1 = P_2$ *and* $C_1 = C_2$.

Definition 9 (Parallel Product). *The parallel product* $\mathcal{S} = \mathcal{S}_1 || \mathcal{S}_2$ *of two compatible IOSTS* $\mathcal{S}_i = \langle V_i, P_i, C_i, \Theta_i, L_i, l_i^0, \Sigma_i^?, \Sigma_i^!, \Sigma_i^\tau, \mathcal{T}_i \rangle$ $(i = 1, 2)$ *is the IOSTS defined by the following components:* $V = V_1 \cup V_2$, $P = P_1 = P_2$, $C = C_1 = C_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $L = L_1 \times L_2$, $l^0 = \langle l_1^0, l_2^0 \rangle$, $\Sigma^! = \Sigma_1^! \cup \Sigma_2^!$, $\Sigma^? = \Sigma_1^? \cup \Sigma_2^?$, $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$. *The set* $\mathcal{T}$ *of symbolic transitions of the parallel product is the smallest set satisfying the following rules:*

$$\frac{\langle l_1, a_1, G_1, A_1, l_1' \rangle \in \mathcal{T}_1, \quad l_2 \in L_2}{\langle \langle l_1, l_2 \rangle, a_1, G_1, A_1 \cup (x := x)_{x \in V_2}, \langle l_1', l_2 \rangle \rangle \in \mathcal{T}}$$

$$\frac{\langle l_2, a, G_2, A_2, l_2' \rangle \in \mathcal{T}_2, \quad l_1 \in L_1}{\langle \langle l_1, l_2 \rangle, a_2, G_2, A_2 \cup (x := x)_{x \in V_1}, \langle l_1, l_2' \rangle \rangle \in \mathcal{T}}$$

The parallel product allows each IOSTS to fire its transitions independently of the other one. We also note that the parallel product is associative and commutative (up to the names of locations).

Example 2. The Basic Access Control protocol depicted in Fig. 1 can be formally modelled as $\mathit{Traces}(\mathit{Terminal} || \mathit{Passport}, \emptyset)$, that is, as the traces of the parallel composition of *Terminal* and *Passport* that are admissible in the empty environment $\emptyset$ (cf. Definition 7). This initially empty environment is enriched by the outputs of the agents, which also consume inputs from it. We shall see in the next section that the full protocol, including the intruder, can be modelled in a similar manner.

Lemma 2 (Monotonicity of traces in the parallel product). *For three IOSTS* $\mathcal{S}_1, \mathcal{S}_1', \mathcal{S}_2$ *such that* $\mathcal{S}_1, \mathcal{S}_2$ *are compatible and* $\mathcal{S}_1', \mathcal{S}_2$ *are compatible, if* $\mathit{Traces}(\mathcal{S}_1') \subseteq \mathit{Traces}(\mathcal{S}_1)$ *then* $\mathit{Traces}(\mathcal{S}_1' || S_2) \subseteq \mathit{Traces}(\mathcal{S}_1 || S_2)$.

In the sequel, whenever two IOSTS are composed by the parallel product operation, we implicitly assume that they are compatible.

3 Verification, Testing, and Learning

In this section we present the ingredients of our approach, which are verification, testing, and learning. The approach itself is presented in the next section.

3.1 Expressing Security Properties Using Observers

We represent security properties, such as secrecy and authentication, using *observers*, which are IOSTS equipped with a set of *recognising* locations. Observers can be seen as an alternative to, e.g., temporal logics and, for some temporal logics such as LTL, formulas can be translated into "equivalent" observers; see, e.g., [22] for a transformation of *safety* LTL formulas into observers.

Definition 10 (recognised traces). *Let $F \subseteq L$ be a set of locations of an IOSTS $\mathcal{S}$. An execution ρ of $\mathcal{S}$ is* recognised by *F if the execution terminates in a state in $F \times \mathcal{V}$. A trace is recognised by F if it is the trace of an execution recognised by F. The set of traces of an IOSTS $\mathcal{S}$ recognised by a set F of locations is denoted by Traces$(\mathcal{S}, F)$.*

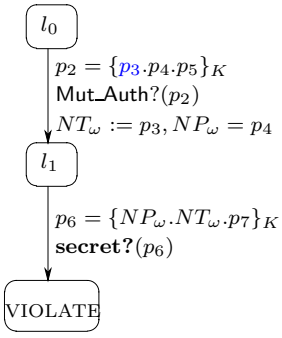

Fig. 2. Observer ω for the authentication to Terminal

Example 3. The passport authenticates itself to the terminal by demonstrating its ability to decrypt the terminal's nonce, and by sending that nonce (within is an encrypted tuple of messages) back to the terminal by the Rsp_Auth command. The observer depicted in Figure 2 expresses a scenario where the *intruder* gains enough information in order to authenticate itself to the terminal. The observer starts by observing the Mut_Auth? input from the terminal, and, by decrypting the parameter p_2 of the input (which is an encrypted sequence of three messages), it memorises the first and second messages in the sequence in its variables NT_ω and NP_ω, respectively. Then, the observer waits for a certain **secret?** input, emitted only by the *intruder* (defined below), carrying a parameter of the form $\{NP_\omega.NT_\omega.p_7\}_K$. Intuitively, when the intruder emits this parameter, it can also emit Rsp_Auth$(\{NP_\omega.NT_\omega.p_7\}_K)$, hence, it can also authenticate itself to the terminal. Hence, upon reception of the **secret?** input, the observer enters its *Violate* location, which expresses the violation of the authentication property.

In addition to authentication properties, secrecy properties (and in general all *safety* properties) can be defined using observers in a similar way.

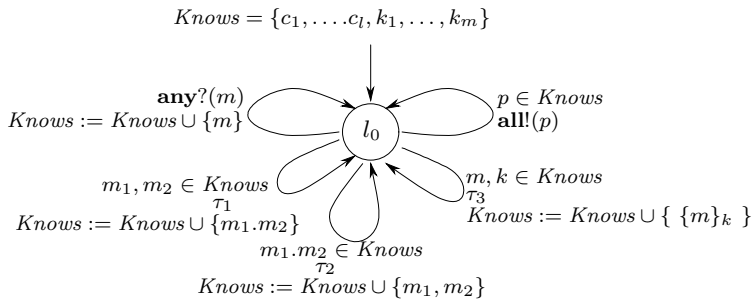

Fig. 3. Template IOSTS for generic intruder

Definition 11 (recognised traces of product). *For IOSTS $\mathcal{S}$ and ω and $F \subseteq L_\omega$ a subset of locations of ω, we denote by Traces$(\mathcal{S}||(\omega, F))$ the set of recognised traces Traces$(\mathcal{S}||\omega, L_\mathcal{S} \times F)$.*

Lemma 3 (monotonicity of recognised traces). *For IOSTS $\mathcal{S}_1, \mathcal{S}_2$, ω, and $F \subseteq L_\omega$, if Traces$(\mathcal{S}_1) \subseteq$ Traces$(\mathcal{S}_2)$ then Traces$(\mathcal{S}_1||(\omega, F)) \subseteq$ Traces$(\mathcal{S}_2||(\omega, F))$.*

3.2 The Intruder

The Basic Access Control protocol will be modelled as a parallel product between the terminal and passport IOSTS, depicted in Figure 1, together with observers for security properties (such as that depicted in Figure 2) and an intruder, whose general structure is given in Figure 3 as a "template" IOSTS. A "template" IOSTS is just like an IOSTS, except that it has "generic" actions, which are abbreviations for any (input, output) actions in a given set of actions.

The generic intruder depicted in Figure 3 reacts to "any" input **any?**(m), by adding the formal parameter m to the variable *Knows*, which is a state variable encoding the intruder's current knowledge. This state variable is initialised as a certain set $\{c_1, , \ldots, c_l, k_1, \ldots k_m\}$ $(l, m \geq 1)$ of symbolic constants. The constants c_i will be used as *nonces*, whereas the constants k_j will be used as *keys*[5]. The knowledge of the intruder is then closed under the concatenation, deconcatenation, and encryption operations, which is modelled by the transitions labelled by the internal action τ_1, τ_2, and τ_3, respectively. For example, the transition labelled τ_3 in Figure 3 can be fired whenever some message m and key k belong to the current knowledge *Knows*, and, by firing the transition, the intruder adds the encrypted message $\{m\}_k$ to its knowledge. The intruder sends messages by the **all!**(p) output, where p is any term in the current knowledge.

The above model of the intruder corresponds to the Dolev-Yao model [18], with a few limitations: finitely many nonces and keys, and symmetrical encryption only. The full Dolev-Yao model can also be encoded as a template IOSTS.

[5] Finitely many nonces is a usual approximation in cryptographic protocol verification. Infinitely many nonces can be generated by using a function symbol *nonce* : $\mathbb{N} \mapsto \mathcal{N}$.

3.3 Modelling the Protocol, Performing the Verification

We now have all the ingredients for defining the protocol and its verification. We specify the protocol as the *admissible traces* in the empty environment $\emptyset$ (cf. Definition 7) of the *parallel product* (cf. Definition 9) $Terminal||Intruder||Passport$, where *Terminal* and *Passport* are the IOSTS depicted in Fig. 1, and *Intruder* is obtained from the template depicted in Fig. 3 by letting **any?** denote any element in the set $\{\mathsf{Get_Chal?}, \mathsf{Snd_Chal?}, \mathsf{Mut_Auth?}, \mathsf{Rsp_Auth?}\}$, and **all!** denote any element in $\{\mathsf{Get_Chal!}, \mathsf{Snd_Chal!}, \mathsf{Mut_Auth!}, \mathsf{Rsp_Auth!}, \mathbf{secret!}\}$. We let initially $Knows = \{c_1, k_1\}$, where c_1 is a symbolic constant of sort $\mathcal{N}$ and k_1 is a symbolic constant of type $\mathcal{K}$. That is, the intruder uses one nonce to send to the terminal, and has one key k_1. Note that the traces of the product have been restricted to those admissible in the empty environment $\emptyset$, to which the agents (including the intruder) add outputs, and from which agents consume inputs.

Once the specifications of the two agents and of the intruder are known, and once the property is expressed by an observer ω with a set *Violate* of recognising locations, the verification problem becomes: decide whether

$$Traces([Terminal||Intruder||Passport||(\omega, Violate)], \emptyset) = \emptyset \tag{1}$$

where, for an IOSTS $\mathcal{S}$ and an observer $(\omega, Violate)$, we denote (cf. Definition 11) by $Traces(\mathcal{S}||(\omega, Violate))$ the set of recognised traces $Traces(\mathcal{S}||\omega, L_{\mathcal{S}} \times Violate)$.

To decide whether Equality (1) holds, we proceed as follows: we translate the IOSTS *Terminal*, *Intruder*, and *Passport* into the language of the Maude tool [23]. We choose this particular tool because it is well adapted to modelling IOSTS and their parallel compositions. Then, Maude's rewriting engine checks whether the *Violate* location, or set of locations, of the observer are reachable in the parallel composition of the modules. If yes, then the property is violated, otherwise, the property holds. Of course, in general (in the presence of loops), the reachability analysis may not terminate. Hence, for practical reasons, we restrict the analysis to traces of a certain length $n \in \mathbb{N}$, which is a parameter of our global approach (including verification, testing, and learning). The protocol depicted in Figure 1 does satisfy the property defined by the observer depicted in Figure 2. Below is an example of a negative result.

Example 4. Consider the IOSTS $Terminal_1$ in Fig. 4, which is very much like *Terminal* (Fig. 1) except for the fact that, in the guard of the transition from l_2 to l_3, the variable NP_t equals NT_t. That is, instead of sending back its own nonce as it should, $Terminal_1$ sends back the *same* nonce it received from the passport[6]. Then, the intruder may just copy the message p_2 and send it back *via* a $\mathsf{Rsp_Auth!}$ command, which the terminal accepts as "valid" authentication. In this way, the intruder manages to "fake" the identity of the passport. The intruder also generates a **secret**! output with the same parameter

[6] Note that, in the correct protocol in Fig. 1, on the transition from l_2 to l_3 of *Terminal*, we have $NP_t \neq NT_t$: by the protocol's semantics, we have $NP_t = NP_p$, and by the initial algebra semantics, the constants NP_p and NT_t are different as no equality between has been specified.

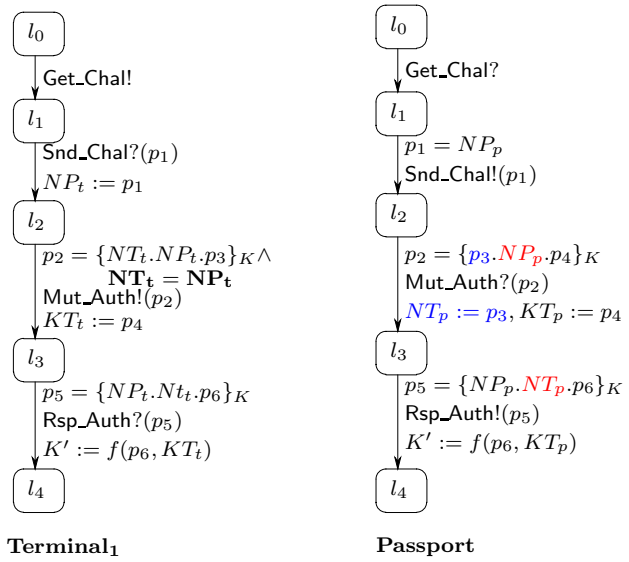

Fig. 4. Erroneous protocol, which violates authentication property

as Rsp_Auth!, which makes the observer enter its *Violate* location; formally, we have $Traces([Terminal_1||Intruder||Passport||(\omega, Violate)], \emptyset) \neq \emptyset$.

3.4 Learning by Testing

Another ingredient of our approach is *testing/learning*. Intuitively, each agent's specification, say, A, may be "augmented" using information obtained by interacting with the corresponding (unknown) implementation I_A. If a trace $\sigma_A \in Traces(I_A) \setminus Traces(A)$ is observed, then transitions are added to A, such as to include the trace σ_A, called thereafter an *example*; and this part of the learning process is called *adding examples*. The ultimate goal of the *adding examples* process is that, eventually, $Traces(I_A) \subseteq Traces(A)$ and $Traces(I_B) \subseteq Traces(B)$ hold, in which case no more examples can be added. In general, infinitely many examples, in the above sense, must be added before the process terminates[7].

[7] There are two sources of infinity: infinite *breadth* due to the values of the parameters carried by the actions, and infinite length (or depth) of the traces. In order to make this process finite we can resort to *uniformity* and *regularity* hypotheses [24]. In our context, *uniformity hypotheses* say that the valued inputs can be partitioned into finitely many classes, and that it is enough to stimulate the implementation with one input in each class in order to obtain all possible (equivalent) outputs; and *regularity hypotheses* state that it is enough to bound the length of the testing/learning step to a certain natural number n.

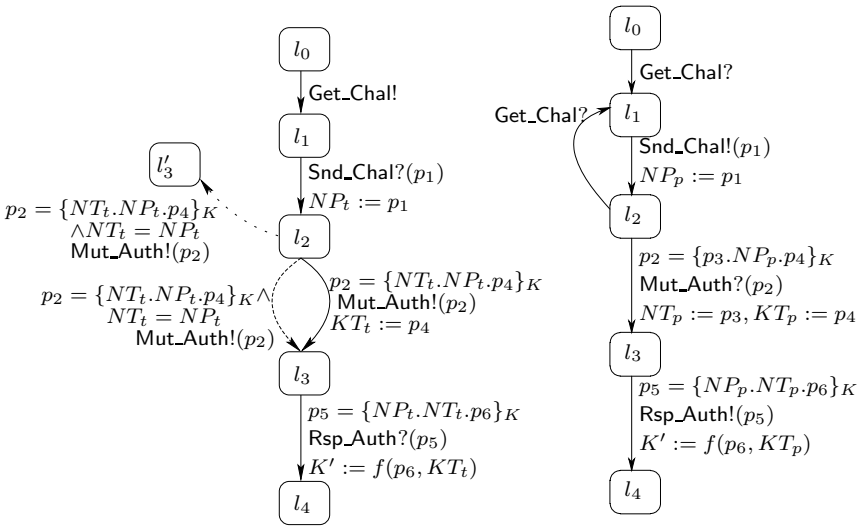

Fig. 5. Terminal (left) and Passport (right), after learning

Example 5. Assume that, by interacting with the passport's implementation[8] we obtain the *example* that, after the first time the nonce was sent via the Snd_Chal! output, a new Get_Chal? input results in sending the nonce once again. Then, we add to the specification the transition labelled Get_Chal? from l_2 to l_1. The resulting specification $Passport_2$ of the passport is depicted in the right-hand side of Figure 5. Next, assume that, by interacting with the terminal, we discover that the Mut_Auth! command emits a sequence in which the first element is equal to the terminal's copy NP_t of the passport's nonce. Then, we add a transition to the specification; the left-hand side of Figure 5 shows two possibilities for adding the new transition (drawn with dashed, and dotted lines, respectively): either to a new location l'_3, or to the existing location l_3. We denote by $Terminal_2$ the IOSTS obtained by adding the latter transition (depicted using a dashed line).

The second part of the learning process deals with *removing counterexamples.* Let again A and B denote the honest agents in a protocol and C denote the intruder. Then, as seen in Section 2, the protocol is modelled by the set of traces $Traces(A||C||B, \emptyset)$, and a *counterexample* is a trace $\sigma \in Traces(A||C||B, \emptyset)$, which violates the property φ, and which is not in $Traces(I_A||I_C||I_B, \emptyset)$. That is, a trace showing that φ is not satisfied by the protocol's global specification, but that cannot be "reproduced" on the protocol's implementation. The existence of such "spurious" counterexamples indicates that the learning was imperfect; they have to be removed from the specifications A, B in order to "fix" the learning.

Example 6. We have seen in Example 4 that if the terminal re-uses the nonce of the passport instead of its own in its Mut_Auth output (as does $Terminal_2$ in

[8] Due to confidentiality issues about the case study, these examples are fictitious.

the left-hand side of Figure 5), the protocol is incorrect as it is able to "authenticate" the intruder via the Rsp_Auth input. Then, a model checker generates a trace σ, illustrating the authentication property's violation, which, if the protocol's *implementation* does not violate our property, is not reproducible on the implementation. That is, σ is a *counterexample* for the property described by the observer in Fig. 2.

We now give more details on the *adding examples* and *removing counterexamples* procedures. These procedures are only one way, among many, to perform learning. We do not know of any existing learning techniques for infinite-states systems such as IOSTS, but for finite-state automata such techniques exist [25].

Adding examples. The intuition is that we want to preserve as much as possible the control structures of the honest agent's current specifications A, B. Let $\sigma_A \in \mathit{Traces}(I_A) \setminus \mathit{Traces}(A)$. Then, σ_A can be decomposed as $\sigma_A = \sigma'_A \cdot \alpha \cdot \sigma''_A$, where $\sigma'_A \in \mathit{Traces}(A)$, $\alpha \in \Lambda^?_A \cup \Lambda^!_A$, $\sigma'_A \cdot \alpha \notin \mathit{Traces}(A)$, and $\sigma''_A \in (\Lambda^?_A \cup \Lambda^!_A)^*$. Then, let $\alpha = \langle a, \pi \rangle$, let $\langle l, \nu \rangle$ be any state of $[\![A]\!]$ in which the IOLTS $[\![A]\!]$ may be after firing the sequence σ'_A. We add one new transition t to the IOSTS A in order to "include" the valued action $\alpha = \langle a, \pi \rangle$. The transition t has:

- origin l;
- action a;
- guard G, chosen by the user to be some predicate G on the variables V_A and parameters P_A such that $G(\nu, \pi) = \mathit{true}$; by default, G is chosen to be the *complement* of the union of the conditions under which a may be fired in l;
- assignments are also chosen by the user - by default, the identity assignments $(x := x)_{x \in V_A}$;
- the destination is defined as follows:
  - if, by choosing the destination of t to be a location $l' \in L_A$, the whole sequence σ_A becomes a trace of the resulting IOSTS, then we let the destination of t be l'; if several choices are possible for l', then one is chosen;
  - otherwise, we let the destination of t be a new location $l'' \notin L_A$.

In the right-hand side of Figure 5, the transition t labelled Get_Chal? from l_2 to l_1 has been added to the passport's specification, as a result of the observation that, after one Snd_Chal! output, a new Get_Chal? input produces another Snd_Chal!. Here, the existing location l_1 has been chosen as the destination of t. The left-hand side of Figure 5 shows two different ways of adding a transition to the terminal's specification: the transition drawn with a dashed line goes to an existing location, whereas the one depicted with a dotted line goes to a newly created location.

Removing counterexamples. This procedure is called whenever the last call to *adding examples* leads to a violation of the property under verification by the the protocol's augmented specification. (It is assumed that the protocol is initially correct, hence, any counterexample may only exist because of the last added

example.) Hence, removing counterexamples consists in undoing the effect of the last "adding examples" operation, and in proposing another way of adding the last example (in terms of the transition t to be added to one of the agent's specifications). A marking mechanism is used to ensure that a different choice is proposed if the removing counterexamples operation is called repeatedly for the same example. These procedures are illustrated in the next section.

4 Our Approach

The proposed approach deals with the problem of establishing whether or not $I_A||I_C||I_B \models \varphi$ holds, for *black-box* implementations I_A, I_B of the honest agents and I_C of the intruder, assumed to have some (unknown) representations in terms of finite or infinite IOLTS (Input-Output Labelled Transition Systems), and a security property φ represented using an observer $(\omega, Violate)$[9].

The corresponding specifications of the agents are denoted by A, B, C. We assume that $\mathit{Traces}(I_C) = \mathit{Traces}(C)$, that is, our model of the intruder is correct. The approach is presented in Figure 6 in pseudocode.

The outer `while` loop is executed as long as a certain Boolean flag `inconclusive` holds true. The flag is set to `false` and will be reset to `true` if the function's inner `while` loop, described below, fails to deliver a "conclusive" result.

The inner `while` loop can be executed as long as $A||C||B \models \varphi$. The latter formula denotes a verification problem, to be solved by means of model checking as described in Section 3.3. In each iteration of the inner `while` loop, one of the two honest agent's specifications, A and B (in alternation) is augmented using the `add-example` mechanism informally described in Section 3.4. The inner `while` loop may terminate in two situations: when, after a number of iterations, the `done`() function returns `true` (intuitively, this happens if all "representative" traces up to a bounded length have been explored *and* $A||C||B \models \varphi$ still holds); or when $A||C||B \models \varphi$ fails to hold.

- in the first situation, the result is that *if the learning process was* exhaustive *i.e., such that* $Traces(I_A) \subseteq \mathit{Traces}(A)$ and $\mathit{Traces}(I_B) \subseteq \mathit{Traces}(B)$ holds, the conclusion is that *the property* φ *also holds on the protocol's implementation.* That is, we have established the correctness of the protocol's implementation without actually executing it. This is a consequence of the *theoretical completeness* theorem given below.

  Note that, since I_A, I_B are black boxes, the hypothesis that $Traces(I_A) \subseteq \mathit{Traces}(A)$ and $\mathit{Traces}(I_B) \subseteq \mathit{Traces}(B)$ cannot, in general, be validated. It is only possible to increase our confidence in the validity of the hypothesis, by performing as many iterations of the `while` loop as possible and by systematically testing as many sequences of inputs/outputs as possible[10].

[9] Remember from Section 3 that $A||B||C \models \varphi$ iff $\mathit{Traces}([A||B||C||(\omega, Violate)], \emptyset) = \emptyset$.

[10] Under *regularity* and *uniformity* hypotheses on the data types [24], only a finite set of traces need to satisfy the trace-inclusion hypotheses, which becomes checkable.

```
function VerifyTestLearn(A, B, C, I_A, I_B, I_C, φ)
    inconclusive := true
    turn := A
    while inconclusive do
        inconclusive := false
        while A||C||B ⊨ φ and not done() do
            if turn = A then
              choose σ_A ∈ Traces(I_A) \ Traces(A)
              A := add-example(A, σ_A)
            else
              choose σ_B ∈ Traces(I_B) \ Traces(B)
              B := add-example(B, σ_B)
            endif
          turn := next(turn)
        endwhile
        if done() //by assumption, done() = true => A||C||B ⊨ φ
          return
          "Traces(I_A) ⊆ Traces(A) ∧ Traces(I_B) ⊆ Traces(B) ⇒ I_A||I_C||I_B ⊨ φ"
        else // implicitly, after while loop done() = false => A||C||B ⊭ φ
          choose(σ ∈ Traces(A||C||B, ∅) ∩ {σ|σ ⊭ φ})
          if is-executable(σ, I_A||I_C||I_B) return "I_A||I_C||I_B ⊭ φ''"
          else
            if turn = B
              A := remove-counterexample(A, σ_A)
            else
              B := remove-counterexample(B, σ_B)
            endif
          endif
          inconclusive := true
        endif
    endwhile
end.
```

Fig. 6. Our approach

- on the other hand, if, after a number of executions, $A||C||B \models \varphi$ does not hold any more, we obtain a trace σ, which is a sequence of interactions between the intruder and the honest agents, demonstrating that the security property is violated. There are again two cases:
  - if σ is *executable* on the implementation, that is, if it is possible to reproduce it on $I_A||I_C||I_B$ - where, e.g., the intruder's implementation I_C is replaced by a test execution engine - then the protocol's implementation also violates the property. Note that we have obtained an information about the protocol's *implementation* mostly by using informations obtained by *verifying* the protocol's *specification*. Of course, it is necessary to ensure that the trace σ obtained by verification is executable by the implementation; but we execute just *one* trace, obtained *via the capabilities of a model checker of exploring "all" the possible traces of a model.*

This is arguably, more efficient for finding errors than directly executing and monitoring the *implementation* $I_A||I_C||I_B$ (with all the execution traces induced by the intruder's implementation!) in the hope of detecting an error.

- otherwise, the trace σ found on the protocol's specification cannot be executed on the protocol's implementation, and we have a *spurious counterexample* in the sense given in Section 3.4. This means that the learning performed during the inner `while` loop was incorrect. Then, the `remove-counterexample` procedure just undoes the effect of the last `add-example` procedure, and proposes another way of including the last example. A marking mechanism is employed to ensure that a future attempt to add the same example trace gives a different result.

Example 7. Assume that we have performed one whole iteration of the inner `while` loop in the procedure described in Figure 6, and that, after the testing/learning phase, the passport, which plays the role of agent A, is that depicted in Figure 5 (right). This augmented specification of the passport, together with the intruder, and the initial specification of the terminal, i.e., that depicted in Figure 1 (left), satisfies the property represented by the observer depicted in Figure 2. Then, a new iteration of the inner `while` loop is started, and it is now the terminal's turn to "learn" new behaviours. Assume that in this phase we "learn" the transition depicted with a dashed line in Figure 5 (left). Then, as seen in Example 4 the protocol now violates the property, and we are presented with a counterexample showing how the property is violated. If the counterexample is executable on the global protocol's implementation, then, the procedure terminates with a conclusive (negative) answer. Otherwise, the *remove counterexample* procedure takes over and replaces the transition depicted with a dashed line in Figure 5 with the one depicted with a dotted line, and the process continues until, e.g., all the (finitely many) traces of the agent's specifications, satisfying adequate *uniformity* and *regularity* hypotheses, have been learned.

We now give the main theorem, stating the method's theoretical completeness. Inttuitively, it says that if the models of the agents in a protocol are sufficiently precise, then the results obtained on the models also hold on the implementation.

Theorem 1 (theoretical completeness). *Consider three IOSTS A, B, and C, such that C is an* intruder *as described in Section 3.2. Let φ be a safety property with the corresponding observer, $(\omega, violate)$, as described in Section 3.1, and assume that A, B, C, and ω are pairwise compatible IOSTS (cf. Definition 8). Let also I_A, I_B, I_C be the (unknown) models for the implementations of the honest agents and of the intruder, which are IOLTS such that $\Lambda^!_{I_A} = \Lambda^!_{[\![A]\!]}$, $\Lambda^?_{I_A} = \Lambda^?_{[\![A]\!]}$, $\Lambda^!_{I_B} = \Lambda^!_{[\![B]\!]}$, $\Lambda^?_{I_B} = \Lambda^?_{[\![B]\!]}$, $\Lambda^!_{I_C} = \Lambda^!_{[\![C]\!]}$, $\Lambda^?_{I_C} = \Lambda^?_{[\![C]\!]}$. Finally, assume that $A||C||B \models \varphi$ as defined in Section 3.3. Then, we have the valid implication $Traces(I_A) \subseteq Traces(A) \wedge Traces(I_B) \subseteq Traces(B) \Rightarrow I_A||I_C||I_B \models \varphi$.*

This theorem says that, when the procedure returns '$Traces(I_A) \subseteq Traces(A) \wedge Traces(I_B) \subseteq Traces(B) \Rightarrow I_A||I_C||I_B \models \varphi$" then this is really the case. As a

result, using standard conformance testing, one can ensure that the collaborating agents meet the required (security) properties. As the completeness result does not require that the composition of the agents is tested, a major advantage of our method is that the test effort can be distributed over several companies, i.e. each agent can be certified by a different company (or several). On the one hand, this reduces the required test effort per company and the amount of information that a company needs for testing, and, on the other hand, a distributed certification mechanism can have a significant positive impact on the trust one has in the agents and the system as a whole. Note that *soundness* - i.e., when the procedure says that the protocol's implementation violates a property, then this is really the case - is trivial by construction.

5 Conclusion and Future Work

We propose an approach for checking security properties (such as secrecy and authenticity properties), as well as other, general-purpose temporal logic properties, on black-box implementations of cryptographic protocols. The approach integrates ideas from verification, black-box testing, and learning.

Specifications of agents are given as IOSTS (Input-Output Symbolic Transition Systems), and the implementations of all the agents in the protocol are black boxes, assumed to have some unknown representations in terms of finite or infinite IOLTS (Input-Output Labelled Transition Systems). Security properties and other temporal logic properties are represented using *observers*, which are IOSTS equipped with a set of dedicated locations that are entered when the corresponding property is violated. The verification is then standard state-space exploration, and the learning consists in adding transitions to the honest agents' specifications by observing so-called *examples*, that is, traces of an agent's implementation that are not in the corresponding specification. Learning also consists in *removing counterexamples*, when a trace violating a property on the protocol's specification cannot be reproduced on the implementation.

The method is *sound*, as it only says that a property is violated by a protocol's implementation when such a violation has actually been found. It is also *theoretically complete*, in the sense that, if the learning is *exhaustive* (i.e., if the traces of the agent's implementations are included in the traces of the corresponding specifications obtained by learning) and if the property holds on the protocol's global specification, then the property also holds on the protocol's implementation. Of course, the trace-inclusion hypothesis between a black-box implementation and a white-box specification cannot be established in general, but confidence in it can be increased by increasing the amount of testing and learning. We are investigating connections with ideas from the area of testing of processes with data [24] where it is shown that if some *regularity* and *uniformity* hypotheses hold on the data, then a complete finite test suite can be given.

The method is *compositional* in the testing/learning parts: *adding examples* and *removing counterexamples* operate on each agent, not on the composition of the agents. The benefit of a compositional approach is the usual one (the

ability to deal with larger state spaces). From a security point of view, certifying a system by testing each agent in isolation has its benefits too: on the one hand, only isolated pieces of a (possibly proprietary) system have to be made available to a company for testing, and on the other hand, testing for security properties such as secrecy and authentication become a less ad-hoc activity.

We illustrate our method on the Basic Access Control protocol implemented in biometric passports. Some of the examples presented in the paper are simplified instances of actual experiments that we performed with actual passports [17]; the present paper formalises and enhances the empirical testing methodology that we used in [17].

Finally, our method is not limited, in principle, to cryptographic protocols or security properties. Cryptographic protocols are interesting here because they have "small" specifications, which are typically incomplete and for which it makes sense to attempt completion by means of learning techniques.

References

1. Burrows, M., Abadi, M., Needham, R.M.: A logic of authentication. ACM Trans. Comput. Syst. 8(1), 18–36 (1990)
2. Gong, L., Needham, R.M., Yahalom, R.: Reasoning about belief in cryptographic protocols. In: IEEE Symposium on Security and Privacy, pp. 234–248 (1990)
3. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. 148(1), 1–70 (1999)
4. Lowe, G.: Casper: A compiler for the analysis of security protocols. Journal of Computer Security 6(1-2), 53–84 (1998)
5. Armando, A., Basin, D.A., Boichut, Y., Chevalier, Y., Compagna, L., Cuéllar, J., Drielsma, P.H., Héam, P.-C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The avispa tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
6. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6(1-2), 85–128 (1998)
7. Hughes, J., Warnier, M.: The coinductive approach to verifying cryptographic protocols. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) Recent Trends in Algebraic Development Techniques. LNCS, vol. 2755, pp. 268–283. Springer, Heidelberg (2003)
8. Gunter, E.L., Felty, A.P. (eds.): TPHOLs 1997. LNCS, vol. 1275, pp. 19–22. Springer, Heidelberg (1997)
9. Denker, G., Millen, J.K.: Modeling group communication protocols using multiset term rewriting. Electr. Notes Theor. Comput. Sci., vol. 71 (2002)
10. Genet, T., Klay, F.: Rewriting for cryptographic protocol verification. In: McAlleste, D.A. (ed.) Automated Deduction - CADE-17. LNCS, vol. 1831, pp. 271–290. Springer, Heidelberg (2000)
11. Bozga, L., Lakhnech, Y., Périn, M.: Pattern-based abstraction for verifying secrecy in protocols. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 299–314. Springer, Heidelberg (2003)

12. Monniaux, D.: Abstracting cryptographic protocols with tree automata. Sci. Comput. Program. 47(2-3), 177–202 (2003)
13. ISO/IEC 9646. Conformance Testing Methodology and Framework (1992)
14. Jeffrey, A.S.A., Ley-Wild, R.: Dynamic model checking of C cryptographic protocol implementations. In: Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (fcs'06) (2006)
15. Goubault-Larrecq, J., Parrennes, F.: Cryptographic protocol analysis on real C code. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (2005)
16. Bhargavan, K.: Provable implementations of security protocols. In: IEEE Symposium on Logic in Computer Science (LICS 2006), pp. 345–346 (2006)
17. Breunesse, C.-B., Hubbers, E., Koopman, P., Mostowski, W., Oostdijk, M., Rusu, V., de Vries, R., van Weelden, A., Schreur, R.W., Willemse, T.: Testing the dutch e-passport, Technical report, Radboud University, Nijmegen, The Netherlands (2006)
18. Dolev, D., Yao, A.C.: On the security of public key protocols. In: Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science, pp. 350–357 (1981)
19. Technical advisory group on Machine-Readable travel documents. Pki for machine-readable travel documents, version 1.1. Technical report, International Civil Aviation Organization (October 2004)
20. Lynch, N., Tuttle, M.: Introduction to IO automata. CWI Quarterly, vol. 3(2) (1999)
21. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM 32(4), 444–458 (1989)
22. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design 19(3), 291–314 (2001)
23. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
24. Gaudel, M.-C., James, P.R.: Testing algebraic data types and processes: A unifying theory. Formal Asp. Comput. 10(5-6), 436–451 (1998)
25. Angluin, D.: Inference of reversible languages. Journal of the ACM 29(3), 741–765 (1982)