A Compilation Method for the Verification of Temporal-Epistemic Properties of Cryptographic Protocols

I. Boureanu, M. Cohen, and A. Lomuscio
{ibourean,mcohen1,alessio}@doc.ic.ac.uk

Department of Computing
Imperial College London
London, UK

Abstract. We present a technique for automatically verifying cryptographic protocols specified in the mainstream specification language CAPSL. Our work is based on model checking multi-agent systems against properties given in AI logics. We present PC2IS, a compiler from CAPSL to ISPL, the input language of MCMAS, a symbolic model checker for MAS. The technique also reduces automatically the state space to be considered by the model checker, thereby maximising the number of protocols and sessions that can be verified. We evaluate the technique on protocols in the Clark-Jacobs library against custom secrecy and authentication requirements as well as against more advanced properties that are expressible in this epistemic-based approach.

1 Introduction

Several successes in AI have come from the application of core AI techniques to key technology-oriented areas, including robotics, vision, networking, scheduling, distributed applications, etc. In the case of cryptographic protocol analysis, AI has played a major role as a provider of several theorem proving techniques [1, 2]. However, it is apparent that a wealth of other AI concepts may usefully be adapted for fruitful application in this area. One of these is the computationally oriented uses of epistemic logic, or logic for knowledge [3].

Currently security protocols' specifications (authentication, non-repudiation, etc.) are most often given in terms of reachability only, or, in some more advanced approaches, in linear temporal logic [4]. It has long been recognised, including in much of the security literature [5], that richer specification languages would permit more powerful and easier to understand specifications. The seminal work in security on BAN logics [6], where the case for the use of logics for knowledge and related concepts was attempted, is a case in point. In fact, some security specifications, such as anonymity properties [7], seem most appropriately expressed only as epistemic properties. While the original BAN approach lacked a semantics, considerable progress in logic for AI has been made recently, on both the theory and the model checking of epistemic logic. This makes an epistemic-oriented verification of security protocols feasible, at least in principle.

Several well-founded *theoretical* approaches for the use of epistemic logic in security settings have been proposed recently [8]. However, there has been less concern so far about the *automatic verification* of security protocols in an epistemic setting. Of course model checking of security protocols is an active area of research in security verification, but only trace properties, typically reachability, are considered there. In this line of work, our aim is to provide fully-fledged formal techniques and tools for the automatic verification of epistemic properties for security protocols specified by well-founded AI-based concepts.

Model checking of epistemic security requirements has been proposed before [9]; however, no consideration for tackling the consequent explosion of the state-space in a principled way were given, thus making the approaches not viable for comprehensive deployment. Closer to our line of work is [10] where, an optimised, trace-based semantics for temporal epistemic logic based on interpreted systems [3] was put forward and a basic bounded model checking algorithm presented. However, [10] focuses on the bounded model checking case only, and no automated procedure is given to generate the models to be checked, thereby limiting its possible impact.

This paper presents a fully automatic methodology for checking cryptographic protocols specified by means of temporal-epistemic logic. Our key contribution is an automatic translation from protocol descriptions given in CAPSL (Common Authentication Protocol Specification Language) into ISPL (Interpreted Systems Programming Language), the input language for MCMAS [11], a BDD-based model checker for multi-agent systems supporting temporal epistemic specifications.

The translation employs advanced techniques (send/receive matching, initial states minimisation, etc.) to minimise the resulting state-space of the model, thereby making the produced ISPL code very optimised. Additionally we impose a variable limit on the number of sessions verified, thus minimising the model checking time without compromising the correctness of the results.

The rest of the paper is organised as follows. In Section 2, in attempt to make the paper self-contained, we present preliminary technical details and we give references for further details. Section 3 presents the core principles of the translation from security protocols descriptions into the semantics for temporal-epistemic logic that we employ here. In Section 4, we discuss the formalisation of the security requirements supported by temporal epistemic logic. In Section 5, we give details on the implementation of the methodology, and we evaluate the technique by discussing experimental results. We conclude in Section 6.

2 Preliminaries and Paper Overview

In this section, we recall basic notions of security protocol specification and concepts related to the interpreted systems framework. Furthermore, we outline some of the related and preliminary work to our approach.

2.1 High Level Description of Security Protocols

A security protocol is a communication protocol aiming to establish certain security-related goals (e.g., authentication, key establishment, anonymity etc). CAPSL (Common Authentication Protocol Specification Language) [12] is a high-level description language for describing features and requirements (goals) of security protocols. We give below an example of a CAPSL file:

```
PROTOCOL Needham-Schroeder Public Key;
VARIABLES
A, B: Node;
Na,Nb: Nonce;
Ka, Kb: Skey;
DENOTES
Ka = pk(A); Kb = pk(B);
ASSUMPTIONS
HOLDS A: Na; HOLDS B: Nb;
MESSAGES
1. A -> B: {A, Na}Kb;
2. B -> A: {Na, Nb}Ka;
3. A -> B: {Nb}Kb;
GOALS
PRECEDES A: B | Na; PRECEDES B: A | Nb;
AGREE A,B : Na,Nb,A;
END;
```

The above is a CAPSL *protocol description* for the well-known Needham-Schroeder Public Key (NSPK) protocol. The `VARIABLES` section denotes the data used in the communication together with their cryptographic type: Ka, Kb are keys, A and B are *principals* of the protocol and Na, Nb are nonces. The `DENOTES` and `ASSUMPTIONS` sections encode the initial knowledge of the principals. In the example above, A knows the nonce Na, B knows Nb, whilst both A knows the public key of B, Kb, and B knows the public key of A, Ka. The `MESSAGES` section specifies the *rules* of the protocol, i.e., the stream of the messages to be exchanged. The *initiator/sender* of each message is encoded, together with its intended *receiver* and the *step* in the protocol execution at which the communication should take place. In the above description, at step 1, A sends to B a message containing her identity and her nonce Na, all encrypted with B's public key. The section `GOALS` encodes the *security requirements*: for this protocol, A should hold Na before B learns it, B should hold Nb before A learns it and, eventually, both should agree upon these values.

2.2 Protocol Scenarios

In security protocol verification, the notions of *role* and *instantiation* are used to capture the (multisession) deployment of a protocol description.

A *(protocol) role* for a principal is given by the set of rules specifying that principal either as initiator or as receiver. In addition to this, and in line with the Dolev-Yao [13] model, we also consider an *intruder* role. The intruder can eavesdrop the communication, "jump" protocol steps, impersonate other agents, forge messages into the communication, usually with the purpose of subverting the security requirements. An

instantiation is a function mapping variables in the protocol description to concrete values over domains. Instantiations are homomorphically extended to messages, rules, and roles. We write $A\text{–}role$ to denote the role for an arbitrary principal A and $(alice, A\text{–}role)$ to denote an $A\text{–}role$ *instance*, where $alice$ is the mapping of A under the applied instantiation. The notation $(alice, A\text{–}role)$ intuitively represents one of the local processes of $alice$ executing an $A\text{–}role$, while $alice$ can have several running processes.
A *protocol scenario* is a protocol description together with a set of role instances. Protocol scenarios correspond to interleaved multisession executions of security protocols. To illustrate this, consider a scenario containing $(alice, A\text{–}role)$, $(bob, B\text{–}role)$, $(alice, B\text{–}role)$. Here, the same role (i.e., $B\text{–}role$) is instantiated more than once and the same name (i.e., $alice$) instantiates an $A\text{–}role$ as well as a $B\text{–}role$.

2.3 Interpreted Systems, CTLK and MCMAS

The *interpreted systems* (IS) formalism [3] describes a multiagent system as follows. We assume a set $A = \{1, \ldots, n\}$ of agents and a special agent called the *Environment*, abbreviated E. We associate to each agent $i \in A$, a set L_i of possible *local states*, a set Act_i of *local actions* and a *protocol function* $P_i : L_i \rightarrow 2^{Act_i}$, defining for each local state l_i the set of actions enabled at l_i. For the environment, we associate similar sets: L_E, Act_E and a protocol function P_E. The transition relation for agent i is defined by the *evolution function* $t_i : Act_1 \times \ldots \times Act_n \times Act_E \rightarrow 2^{L_i \times L_i}$. The *evolution function* t_E of the environment is defined in a similar way. To describe the system as a whole, we define a set of possible *global states* $G = \prod_{1 \leq i \leq n} L_i \times L_E$, *joint protocol* $\overline{P} = (P_1, \ldots, P_n, P_E)$, *joint actions* $Act = Act_1 \times \ldots \times Act_n \times Act_E$ and a *global evolution function* $\bar{t} = (t_1, \ldots, t_n, t_E)$, operating on global states. An interpreted system $\mathcal{I}$ is a tuple $\mathcal{I} = \langle G, \overline{P}, \bar{t}, I_0, V \rangle$, where $I_0 \subset G$ is a set of *initial global states* describing the initialisation of the system and $V : G \rightarrow 2^{AP}$ is a *valuation function* for a set of atomic propositions, AP.

The *CTLK* logic combines temporal logic CTL and epistemic logic $S5_n$ with K_i $(i = 1, \ldots, n)$, n knowledge operators and D_A epistemic operator for a group A of agents. Its BNF syntax is as follows:
$\varphi := p|\neg\varphi \,|\, \varphi \wedge \varphi \,|\, K_i\varphi \,|\, D_A\varphi \,|\, AX\varphi \,|\, AG\varphi \,|\, A(\varphi U \varphi)$.
K_i represents "agent i knows that φ", $D_A\varphi$ is read as "in group A, it is distributed knowledge that φ ", $AX\varphi$ stands for "on all possible paths, at each possible next step φ holds", $AG\varphi$ for "along on all possible paths, φ holds always" and, $A(\varphi U \psi)$ is read as "on all possible paths, at some point ψ holds true and before then, all along the path, φ held true". We refer to [3, 14] for details.

MCMAS [11] is an open-source, bdd-based symbolic model checker for verifying temporal-epistemic properties of interpreted systems specified in ISPL (Interpreted Systems Programming Language). It can handle state spaces of the order of 10^{30}. A self-explanatory example of an ISPL file, describing a single agent, is given below.

```
Agent Sample
```

```
Vars:
state : {val1, val2, val3, val4};
end Vars

Actions = {action1,action2};
Protocol:

(state=val1 or state=val4) : {action1};
(state=val2): {action2};

end Protocol
Evolution:

(state=val3) if (Action=action1) and  (state=val1) and
                (Env.Action=X); ...

end Evolution
end Agent
```

As it is apparent from the above, each ISPL program denotes an interpreted system.

3 Protocol Scenarios as Interpreted Systems

In this section, we present our formal approach for mapping a protocol scenario into an IS.

Our starting point for translating a protocol scenario into an IS is to map each role instance into a unique agent of the IS and the intruder role into the Environment agent. We use the notation ag_A to denote the mapping of an arbitrary A–$role$ instance into an agent of the IS. In order to give the description of the agent (i.e., local states, local protocol, local evolution function), we assume a typed signature and a (free) term algebra formalising the cryptographic data in a protocol description. Thus, the data describing a role is modelled by an ordered list of (typed) variables, called *store*. A self–explanatory example of the store for an A-role under the above NSPK description is: $(A\colon Node, B\colon Node, k_A\colon Skey, k_B\colon Skey, n_a\colon Nonce,$
$n_b\colon Nonce)$. However, moving from roles to role instances, we introduce the notion of *views*; a *view* is the order list of values obtained by the uniform application of instantiations on a store. An example of a view for the store above, under some instantiation, is $(alice, bob, pvk_{alice}, pbk_{bob}, r_1, \perp)$, where $\perp$ is a special value of “unassigned” variables. Thus, views capture the dynamics of a role execution: i.e., above, n_b is “viewed” by an A-role instance as $\perp$ (“unassigned”) at all steps prior to the one where it receives a message containing an actual value for n_b.
We now proceed to give the key elements of the IS defined by the mapping. We first give the mapping for the arbitrarily considered agent ag_A.

Local States of ag_A. A possible *local state* of agent ag_A is given by a pair $(nr, view_{ag_A})$, where nr is a counter for the number of protocol steps previously executed by the agent and $view_{ag_A}$ is a view as above.

Actions of ag_A. The *actions* available to agent ag_A are: $send\, M$, $receive\, M$ and action $empty$, where M is an instantiated message.

Local Protocol of ag_A. The *local protocol* of agent ag_A formalises the A-role. Thus, for each protocol step numbered nr specified for A, we have: $P_{ag_A}((nr, view_{ag_A})) = \{send\, M\}$, if "$nr.\, A \rightarrow Y : M$" is a rule in the instantiated A–$role$ and similarly for receiving actions[1], $P_{ag_A}((nr, view_{ag_A})) = \{empty\}$, if there is no rule with step nr for A in the protocol description.

Local evolution function for ag_A: setting and matching. The action $send\, M$ intuitively denotes ag_A composing M with values in her view and delivering this value-string to the network. The action $receive\, M$ intuitively denotes ag_A receiving the value-string M from the network, comparing its sub-strings to her local state (view) and, consequently, accepting or dropping M. We use the notation $M : view$ to symbolise this composition/validation of M with respect to a given view. We formalise $M : view$ by introducing three functional symbols: in_match, out_match and set. Thus, in_match regards the standalone consistency of an instantiated message (i.e., comparisons only between its sub-strings), out_match regards the consistency of an instantiated message with respect to a view (i.e., certain sub-strings have values compliant to the ones in the view), set implies assignments of values within the views according to values within an instantiated message. In the example below we give an intuition of the operational semantics of these symbols and thus of the local evolution function for ag_A.
As before, let the view of ag_A at step 1 in the protocol be $(alice, bob, pvk_{alice}, pbk_{bob}, r_1, \perp)$. In step 2, the agent ag_A receives some instantiated $t = \{n_a, n_b\}_{k_A}$. Upon receipt and decryption, ag_A performs $out\_match(n_a, t)$: she checks that the sub-string for n_a in t matches the value for n_a in her view. If satisfied, she performs $set(n_b, t)$: she sets the value for n_b in her local view to the sub-string for n_b in t (say, r_{bob}). Thus, her view after step 2 becomes $(alice, bob, pvk_{alice}, pbk_{bob}, r_1, r_{bob})$.

We proceed now to give the mapping for the Environment agent E, which —as we said— maps the intruder.

Local state of E. A possible local state of the Environment agent is given by a set X of instantiated messages together with a history H of instantiated actions, depicting the intruder's participation in the protocol execution.

Actions of E. The actions available to agent E are $intercept\, M$, $transmit\, M$, and $empty$.

[1] Y is an arbitrary principal in the description.

Local Protocol of E. At any local state, E can perform the action $intercept\, M$, implementing its continuous Dolev-Yao surveillance of the network. If the value-string M is composable at a local state, the action $transmit\, M$ is enabled. This captures the Dolev-Yao capability of composing messages and transmitting them to third parties.

Local Evolution Function of E. We formalise the descriptions above, by introducing the notation $X \vdash M$. It denotes that the value-string M is deducible from a set of instantiated messages X via cryptographic compositions/decompositions under the term algebra. In these terms, an example from the local evolution function of E is: $t_E((X,H),\tilde{a}) = (X \cup M \cup \{t |\, \{X \cup M\} \vdash t\}\,, H \cup ag_A.send\, M)$, if $\tilde{a}_{ag_A} = send\, M$ and $\tilde{a}_E = intercept\, M$, i.e., the Environment updates its local state by recording the intercepted message, $X \cup M$; deduces new values at his updated local state, $\vdash t$ and records the action taken by ag_A, $H \cup ag_A.send\, M$.

Initial states. The *initial states* of the system are given by instantiating stores into views, where certain variables can be initially left unassigned. In order to obtain smaller reachable state spaces, we initialise different MAS for a certain IS by circular permutations on the initial values of some variables (i.e., if B ranges over $bob1$ and $bob2$ and A over $alice$, we initialise a system where $alice$ first talks to $bob1$ and another system where she first talks to $bob2$). We thus obtain different MA systems correspondent to different "initial setups".

Thus, by defining all the mathematical objects of the IS above, we provided a model for mapping a protocol scenario in an interpreted system. In the following, we present certain CTLK specifications to be verified under this modelling and their role in security verification.

4 Temporal Epistemic Security Specifications

The `GOALS` ("secret", "agree", etc.) given in CAPSL-descriptions can be translated into CTL along standard lines (cf. [15]). However, in our CTLK specification language we can further check other, more complex properties of security protocols. Anonymity, in particular, has recently been analysed in terms of epistemic logic (cf. [16, 17, 9]). For example, in the dining cryptographer problem [7] the protocol ensures that after the announcements the cryptographers know whether or not one of them paid, but if one did they do not know who did. This can be very easily expressed by considering a temporal-epistemic specification; the case for 3 cryptographers is reported below but this can easily be generalised.

$$\begin{aligned}\neg paid_1 \rightarrow (AX((&K_1(\neg paid_1 \wedge \neg paid_2 \wedge \neg paid_3)) \vee \\ &(K_1(paid_2 \vee paid_3) \wedge \neg K_1 paid_2 \wedge \neg K_1 paid_3)))\end{aligned}$$

Indeed specifications of this kind can be effectively model checked if the models are directly implemented on temporal-epistemic model checkers such as [9, 11] as demonstrated in [**?**].

However in this paper we are mainly concerned with authentication where we believe the approach can also be useful.

Authentication. Even if authentication is traditionally expressed as reachability in the security literature, it naturally pertains to states of knowledge of the principals.

Several authentication properties have been described [18]; for instance, *non-injective agreement* between protocol roles A and B requires that whenever an agent playing role B has completed its local execution, the agent agrees on some local variables with some agent playing role A and the agent *knows* that it is so. It is easy to see that natural variations of these cannot be reduced to reachability only, even if neglecting the natural interpretation of the epistemic modality. Consider, for example, the weaker requirement stating that an agent i playing protocol role B *may* reach a point where it agrees on local variables with some agent j playing role A and where agent i *knows* that it is so:

$$\bigwedge_{i:B} EF\, K_i \bigvee_{j:A} agree(i,j) \tag{1}$$

where i and j range over agents instantiating B–$role$ and A–$role$ respectively. Specification (1) can be seen as a partial form of non-injective agreement, since it implies that agent i may come to know that it has achieved its agreement goal and perhaps that, from that point, it may safely continue. Unlike the standard non-injective agreement, (1) does not easily reduce to a reachability property.

Some authentication protocols intend authentication to be acknowledged, i.e., intend authentication itself to be authenticated. A possible specification for a partial form of acknowledged authentication is that every agent implementing role B may come to know that it agrees with some agent implementing role A who knows that it agrees with some agent implementing role B:

$$(\forall i : B)\, EF\, K_i\, ((\exists j : A)\, agree(i,j) \,\wedge\, K_j\, (\exists k : B) agree(j,k)) \tag{2}$$

Attack detection. Reachability based analysis of authentication protocols considers only the knowledge of individual agents in the run. However, examining the distributed knowledge of a group of agents may bring further insight into other properties of the protocol. Specifically, a group of agents belonging to the same principal instance, say $alice$, can pool their information together through inter-thread communication [19, 20], by which they might be able to detect an attack on one of them. Indeed, there are authentication protocols that include an attack detection mechanism based on such a pooling (cf. WMF protocol in [21]). An attack detection of this kind is naturally expressed in CTLK. For example, we can express whether $alice$ is able to infer she has been attacked by considering the formula:

$$AG\,(attack \rightarrow EF\, D_{alice}\, attack) \tag{3}$$

where D_{alice} is the distributed knowledge of all agents belonging to principal $alice$ (i.e., all role instances of the form $(alice, X$–$role)$ for any principal X), and $attack$

encodes that the authentication goal for such an agent is violated. If (3) fails, an attack undetectable by $alice$ exists even after pooling information together. It seems unlikely that (3) may be reduced to a plain reachability property.

5 Automatic Compilation of Protocol Scenarios into Interpreted Systems

In this section, we present an implementation of the mapping in Section 3. Specifically, we give details for a fully automatic compiler $PC2IS$ — Protocol Compilation to Interpreted Systems — that takes a protocol description given in CAPSL and returns a collection of interpreted systems given in ISPL.

Implementation. Given a protocol scenario, several ISPL files are generated, each file encoding the protocol deployment under a certain bound on the number of sessions. The CAPSL `GOALS` are translated in CTL. Additionally, simple CTLK specifications for checking authentication ((1) in Section 4) are also produced in ISPL. The platform then proceeds to input repeatedly the generated files to MCMAS for verification. MCMAS returns the calls either certifying that the specifications are satisfied or by providing sufficient data for $PC2IS$ to display a counterexample. The counterexample is a trace in which the specification is not satisfied, i.e., it constitutes a protocol attack. Furthermore, on a generated ISPL file, advanced CTLK security requirements such as (2) in Section 4 can be added by the user. $PC2IS$ is developed in JAVA; it is available at [22].

More in detail, $PC2IS$ comprises four different main modules: *utils*, *parser*, *unmarshaller* and *producer*. The module *utils* is composed by two submodules: the *description* submodule and the *scenario generator* submodule. The *description* submodule contains XML schemas that encode our protocol signature. The routines in the *scenario generator* submodule take as input a description file, plus a bound on the number of protocol sessions and generate several protocol scenarios. The *parser* module parses the CAPSL-description input and uses it to generate one collection of XML files for each of the scenarios generated above. These XML files are valid under the schemas above (i.e., all protocol data complies to our signature). The *unmarshaller* module then converts these XML files into JAVA objects and populates the data structures describing the interpreted system. Finally, the *producer* module processes these JAVA structures into several ISPL files, each encoding an interpreted system, as discussed.

In each ISPL file generated by $PC2IS$, agents' local variables represent the *views*. The agents' actions and local protocols contain instantiated *send* and *receive* actions. The agents' local evolutions contain appropriate matching preconditions, and postconditions for exchanging messages. Below, we report part of agent (playing role A in NSPK) coded in an ISPL file generated with $PC2IS$.

```
Vars: --ENCODING <VIEWS>
 A, B: {alice, bob,...};
 N_A, N_B: {n_1, n_2,...}; ...
end Vars
```

```
Protocol: --ENCODING <ROLES>
 A=alice,N_A=n1,step=1:{receive_enc_n1_X_pubkey_alice};
 --(an expression as above for every nonce X)
 ...
end Protocol

Evolution:
 N_B=X and step=step+1 --<SET> SEMANTICS
 if
 Action=receive_enc_n1_X_pubkey_alice and
 Env.Action=transmit_enc_n1_X_pubkey_alice and
 N_A=n1; --<OUT_MATCH> SEMANTICS
 --(a rule as above for every nonce X)
 ...
end Evolution
```

The local variables of the Environment agent encode all the actions that the intruder eavesdrops or executes, as well as the messages deduced by him. The Environment *evolution section* encodes this actual deduction.

```
Vars: -- ENCODE KNOWN VALUES

 knows_X:boolean;
 --(a line as above for every value X)

end Vars

Protocol: --ALL POSSIBLE DY COMPOSITIONS

 knows_X: {transmit_enc_alice_X_pubkey_bob};...
 --(a line as above for every nonce X)
 ...

end Protocol

Evolution: --ALL POSSIBLE DY DECOMPOSITIONS

 knows_X=true
 if
 Action=intercept_enc_alice_X_pubkey_intruder
 --(a rule as above for every nonce X)
 ...

end Evolution
```

For ease of understanding, in the examples above, we have simplified the actual code generated by $PC2IS$.

Experimental results. To evaluate the tool $PC2IS$ in a systematic way, we have run tests on protocol descriptions from a CAPSL-version of Clark-Jacob library [12]. The Clark-Jacob library is a standard, widely-used repository of authentication and key-exchange protocols. In Table 1, we summarise the results obtained for some of the protocols considered against CTL and authentication CTLK requirements directly inferred from the CAPSL goals.

Table 1. Verification CTL and simple CTLK authentication requirements on compiled interpreted systems

Protocol	Key Guessing	Attack Found	Nb. of MAS	Avg. Time
ISO1PUCCF	no	no	13	1
	yes	“PRECEDES” failed	3	1
ISO2PUCCF	no	no	21	2
	ye	no	9	2
	yes	“PRECEDES” failed	2	1
ISOSK2PU	no	no	14	3
	yes	“PRECEDES” failed	4	2
ISOSK3PM	no	no	23	6
	yes	one “PRECEDES” + “SECRET” failed	6	4
AndrewRPC	-	one“PRECEDES” + “SECRET” failed	5	6
NSPK	-	Lowe attack found	9	1

Table 1 presents some key results for the protocols reported here. The last column reports the weighted average time (in seconds) for the verification of a given protocol (or for an attack to be discovered) on a PC based on an Intel Core Duo 1.80Ghz, 1GB RAM, running Windows XP Pro and MCMAS under Cygwin. We report the weighted average time as the approach is based on checking repeatedly ISPL files corresponding to an increasing number of sessions up to a bound, thereby generating a number of IS (given in the 4th column). In addition we differentiate between scenarios in which the keys may or may not be compromised (2nd column).

The results obtained were entirely in line of what known in the literature. Without key learning the first 5 protocols were shown to be correct. If key guessing was allowed, the system identified an attack and displayed the original `GOAL` specification failing in the scenario (appropriately translated). In this case a trace describing the attack was also displayed by the tool (not shown in the table). The analysis of the AndrewRPC protocol identified known attacks and so did NSPK. In practice, verification of these protocols was next to instantaneous thereby reassuring us that the translation as defined produces models that are sufficiently optimised to be deployed on a very wide range of protocols.

Irrespectively of what appears to be more than satisfactory efficiency, the methodology focus on the verification of temporal-epistemic specifications. These are not considered in the Clark-Jacob library, so, instead, we report below an attack-detection result obtained on NSPK. Consider a principal instance *bob* running two B sessions; it turns out that if a Lowe-like attack [23] is made against *bob*, he can "pool together" the information from his two sessions and detect an attack on one of them. As discussed in Section 4, the "pooling together" is achieved by considering the distributed knowledge of *bob* over all the sessions he is running. More formally, if we can verify that the specification

$$AG(\neg auth \rightarrow EFD_{bob}\neg auth)$$

is satisfied in the protocol thereby guaranteeing that *bob* would be able to know he has been attacked.

6 Conclusions and Future Work

In this paper, we presented an automatic methodology for verifying cryptographic protocols against temporal-epistemic specifications. While not all the technical details could be presented in this short research note, the translation is based on formal principles. Among these we highlight the Dolev-Yao model of the intruder, the precise conditions on send/receive of messages, the efficient typing for the representation of messages, and the encoding of roles in IS. Indeed the map defined in Section 3 effectively amounts to an operational semantics of CAPSL programs in terms of IS. In this sense this paper is inspired by and extends previously advocated approaches such as [10]. Differently from [10] though, a less stringent definition of matched send-receive is adopted, and several other optimisations are implemented, including a minimisation of the number of initial states, and a bounded approach to the verification of multiple protocol sessions.

We believe the rationale behind the translation to have been validated by the results obtained with the toolkit $PC2IS$ presented in Section 5. $PC2IS$, when paired with MCMAS, could verify very efficiently all protocols we tested from the Clark-Jacob library. The approach does not seem to be significantly slower than mainstream security toolkits such as CASPER [24] or AVISPA [25], based on process algebras and constraint-solving respectively. However, the methodology presented here extends dramatically the specifications that may be checked to an AI-inspired temporal-epistemic language. As summarised in Section 4 and in the attack on NSPK described in the previous section, this opens the way for the verification of more complex specifications not normally tackled.

References

1. Cohen, E.: TAPS: A first-order verifier for cryptographic protocols. Proceedings of the 13th IEEE workshop on Computer Security Foundations (2000) 144
2. Blanchet, B.: An automatic security protocol verifier based on resolution theorem proving. In: 20th International Conference on Automated Deduction (CADE-20), Tallinn, Estonia (JULY 2005)
3. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press (1995)
4. Ricardo, C., Ari, S., Sandro, E.: PS-LTL for constraint-based security protocol analysis. Lecture Notes in Computer Science **3668** (2005)

5. Syverson, P., Stubblebine, S.: Group principals and the formalization of anonymity. In: World Congress on Formal Methods. (1999) 814–833
6. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. Technical report, DEC SRC (1990)
7. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Volume 1. (1988) 65–75
8. Halpern, J., Meyden, R.: A logical reconstruction of SPKI. Journal of Computer Security **11**(4) (2003)
9. Meyden, R., Gammie, P.: MCK: Model checking the logic of knowledge. In: CAV. (2004) 479–483
10. Lomuscio, A., Penczek, W.: LDYIS: a framework for model checking security protocols. Fundamenta Informaticae **85 (1-4)** (2008) 359–375
11. MCMAS: A model checker for multi-agent systems. http://dfn.dl.sourceforge.net/sourceforge/ist-contract /mcmas-0.9.6.tar.gz. (2009)
12. Millen, J.: Common Authentification Protocol Specification Language. (2001)
13. Dolev, D., Yao, A.: On the security of public-key protocols. IEEE Transactions on Information Theory 29 (1983) 198–208
14. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
15. De-qin, X., Huan-guo, Z.: Model checking electronic commerce security protocols based on ctl. Journal Wuhan University Journal of Natural Sciences (2004)
16. Halpern, J., O'Neill, K.: Anonymity and information hiding in multiagent systems. Journal of Computer Security **13**(3) (2005) 483–514
17. Hughes, D., Shmatikov, V.: Information hiding, anonymity and privacy: a modular approach. Journal of Computer Security **12**(1) (2004) 3–36
18. Lowe, G.: A hierarchy of authentication specifications. In Proceedings of the 10th IEEE workshop on Computer Security Foundations (1997)
19. Halpern, J., Pucella, R.: On the relationship between strand spaces and multi-agent systems. In: Proceedings of the 8th ACM conference on Computer and Communications Security. (2001) 106–115
20. Lowe, G.: A family of attacks upon authentication protocols. Technical report, Department of Mathematics and Computer Science, University of Leicester (1997)
21. ENS-Cachan: Security Protocols Open Repository. (2003)
22. PCtoIS: Protocol compiler to interpreted systems. https://sourceforge.net/projects/pc2is/ (2009)
23. Lowe, G.: An attack on the Needham-Schroeder Public-Key authentication protocol. Inf. Process. Lett. **56**(3) (1995) 131–133
24. Lowe, G.: Casper: A compiler for the analysis of security protocols. Journal of Computer Security **6**(1-2) (1998)
25. Vigano, L.: Automated security protocol analysis with the AVISPA tool. Proceedings of the XXI Mathematical Foundations of Programming Semantics (2006)