Using Trace Formulae for Security Protocol Design

Ricardo Corin[1], Antonio Durante[2], Sandro Etalle[1,3] and Pieter Hartel[1]

[1] Faculty of Computer Science,
University of Twente, P.O.Box 217, 7500AE Enschede, The Netherlands
[2] Università di Roma "La Sapienza", Via Salaria 113, Rome 00198, Italy
[3] CWI, Center for Mathematics and Computer Science Amsterdam
{corin,etalle,pieter}@cs.utwente.nl, durante@dsi.uniroma1.it

Abstract. We report on work-in-progress on a new *trace* logic for describing security properties. This trace logic can be used *directly* in the design process, and it can be incorporated in a natural way into a protocol specification. Then, verification can be performed using previously developed tools by the authors [4, 7]. As a result, traces of protocol executions can be tested for situations which violate the specification. We exemplify our method describing the design of a "toy" protocol for secure database query.

1 Introduction

Cryptographic protocols are the essential means for the exchange of confidential information and for authentication. Their correctness and robustness are crucial for guaranteeing that a hostile intruder can not get hold of secret information (e.g. a private key) or to force unjust authentication. Unfortunately, the design of cryptographic protocols appears to be rather error-prone: a great deal of published protocols has later been shown to contain errors prejudicing their safety. This stimulated research on formal verification of security protocols (see e.g. [6, 3, 10, 12, 9, 14, ?,13, 5, 7, 4]). In this setting most approaches are based on Dolev and Yao's [6], where it is proposed to test a protocol explicitly against a hostile intruder who has complete control over the network, and who can intercept and forge messages.

Security protocols are specified for accomplishing certain security goals. Unfortunately, in some cases there exist no standard definitions for the properties one needs to enforce. For example, a precise notion of authentication is still a topic of research.

In this paper we present an approach we are currently developing for the engineering of security protocols based on the idea of specifying at the same time both the protocols (i.e., the actions that it implies) and the properties that the protocol is expected to achieve. The latter are expressed using a simple yet effective *trace logic*. This way, the protocol designer does not need to refer to standard "hardwired" security notions, but can specify precisely what the protocol is supposed to achieve. Then, with the use of a protocol verifier such as those developed by the authors [4, 7], the designer can immediately check whether the design actually meets the specification, and modify it otherwise.

There are existing logics in the literature; e.g. the work of Adi et. al. [1], where it is shown how to express classical properties for security (e.g. secrecy and authentication) and e-commerce. The main difference with our approach consists in the way we apply the logic: in Adi et. al. [1], a formula is a different entity than the protocol. In our work trace formulae are part of the protocol specification itself: a protocol description is a *combination* of the (standard) protocol messages and trace formulae. This way, the designer can also determine the precise point in a protocol execution when a given formula is supposed to hold.

To the best of our knowledge, our method is unique in that it allows to integrate the design and the verification phases.

The paper is organized as follows. Preliminaries are given in Section 2. In Section 3 the protocol model is developed. A trace logic is introduced in Section 4, and then, in Section 5, it is combined with standard protocol messages to form *protocol specifications*. Also, a case study of a "toy" protocol for secure database query is presented. Finally, conclusions are drawn in Section 6.

2 Preliminaries

The reader is assumed to be familiar with the terminology and the basic results of logic programming [2, 8]. Moreover, we use Prolog notation for distinguishing variables (that start with an uppercase letter, or an underscore) from non-variable terms. The only exception to this rule is that we follow the standard security notation in which names of principals are denoted by uppercase letters (A, B, E).

Let $Agents = \{A_1, B_1, ..., Z_1, A_2, B_2, ...\}$ be a set of variables representing honest agents, and let ε be a constant representing the intruder s.t. $\varepsilon \notin Agents$.

Message Notation Messages are built according to the free algebra generated by the following operators from a finite set of constants:

- t_1, t_2 pairing.
- $pk(P)$ public key of P.
- $h(t)$ the hash of t.
- $\{t\}_k^{\leftrightarrow}$ term t encrypted with k using a symmetric key algorithm.
- $\{t\}_k$ term t encrypted with k using a public key algorithm.
- $sig_k(t)$ public key signature of t that is validated using key k. We assume that private keys of a key pair are never leaked, thus the attacker can only construct its own signatures.

The term algebra does not support operators with associative or commutative properties (e.g. *xor*); however, as stated by Millen and Shmatikov [11], this deficiency can be tackled by adding an equational theory to the algebra.

3 Protocol Model

A protocol step is usually specified using the standard notation $A \rightarrow B : M$. However, this notation is unsuitable for formal verification. In fact, in a protocol step, two different events take place: A sends message M, and B receives message M'. In presence of an intruder, M might not be equal to M'. Moreover, not even the identities of the correspondent communication parties may be the same (i.e., A sends to B' and B receives from A'). It is therefore convenient to take an approach that considers separately each agent's point of view; this is the idea of *protocol roles*.

Definition 1. A *protocol role* is a pair $\langle A, [M_1 \diamond B_1, ..., M_n \diamond B_n]\rangle$, where $\{A, B_1, ..., B_n\} \subset Agents$, $\diamond \in \{\triangleleft, \triangleright\}$ and $M_1, ..., M_n$ are messages. □

Given a protocol role $\langle A, [M_1 \diamond B_1, ..., M_n \diamond B_n]\rangle$, A is called the *identity* of the role, while elements $M_i \diamond B_i, i = 1..n$ are the *actions* of the role: $M \triangleright B$ is a *send* action (that is, introduce to the network message M with destination B), while $M \triangleleft B$ is a *receive* action.

Now consider a protocol written in standard notation. Protocol roles can be easily extracted, as shown in the following example:

Example 2. Consider the protocol:
Message 1. $A \rightarrow B \ : \ NA$
Message 2. $B \rightarrow A \ : \ \{NA\}_{pk(A)}$

The protocol roles *initiator* and *responder* are:
$initiator(A_1, B_1, NA_1) = \langle A_1, [NA_1 \triangleright B_1, \{NA_1\}_{pk(A_1)} \triangleleft B_1]\rangle$
$responder(A_2, B_2, NA_2) = \langle B_2, [NA_2 \triangleleft A_2, \{NA_2\}_{pk(A_2)} \triangleright A_2]\rangle$

Notice the different variables (e.g. for representing the NA, NA_1 and NA_2 are used) for each role. This is meant to underline that objects (i.e., messages or identities) may *not* necessarily be the same to each participant. For instance, a sent message may be intercepted by the intruder an replaced by other message, which then is received by the other party. In the rest of the paper, however, for the sake of simplicity we assume that variables in protocol roles are 'standardized apart'; that is, there is no relationship between an X in one protocol role and another X in another role.

Protocol roles in a security protocol often receive self-explanatory names as *initiator, responder* and *server*.

By appropriately (partially) instantiating a number of protocol roles, one obtains a *system scenario*.

Definition 3. A *system scenario* is a finite multiset of instantiated protocol roles. □

In the system scenario one typically determines how many sessions are present and which agents play which roles.

Example 4. Consider the protocol of Example 2.

- The typical system scenario would be $\{initiator(a, B, na), responder(A, b, NA)\}$, in which we have an initiator played by the honest agent a, generating nonce na and communicating with an unknown party B. This is reflected by the fact that a and na are atoms, while B is a variable. The responder role is played by honest agent b communicating with another unknown initiator A. Notice that, since b does not generate the nonce, it is also represented as a variable NA.
- One could specify a scenario in which there are two sessions and in which agent a is the initiator of one session and the responder in the other one: $\{init(a, B, na), resp(A, b, NA), init(c, D, nc), resp(E, a, NE)\}$.
- Finally, one could also specify a useless system scenario, such as $init(a, b, na), resp(c, a, nc)$. In this setting, the two roles can not communicate with each other.

A system scenario is called (less intuitively) *semibundle* in previous work [11, 4].

Operational Semantics We begin the description of the executions of system scenarios by using a *concrete* semantics, in which we assume that the given scenario is ground (i.e. does not have free variables). Later we generalize this concept to the case of non-ground system scenarios. Executions are described using *traces*, which are in turn composed of *events*, i.e. single actions performed by an agent.

Definition 5. An *event* is a pair $\langle A : M \diamond B \rangle$ where $A, B \in Agents$, $\diamond \in \{\triangleleft, \triangleright\}$ and M is a message. □

The event $\langle A : M \triangleright B \rangle$ should be read as "agent A sends message M with *intended destination* B". On the other hand, $\langle B : M \triangleleft A \rangle$ stands for "agent B receives message M *apparently from* A".

Definition 6. A *trace* is a finite sequence of ground events. □

To analyze the protocol, we combine the system scenario with the usual Dolev-Yao intruder, who can perform the following actions:

- intercept and learn any sent message;
- introduce into the system new messages forged using information he knows.

The information obtained by the intruder is stored in a set of terms K called the *intruder's knowledge*. The execution of a (ground) system scenario is described by the notion of *run*, introduced below.

Definition 7. Let S be a ground system scenario, and K be the intruder's initial knowledge, consisting of constants representing agents identities and their public keys. Let tr be an initial empty trace. A *run* is a trace obtained by a reiterated sequence of the following steps:

1. a non-empty role in S is chosen nondeterministically, and its first action p is removed from it. Let a be the identity of the chosen role.
2. if $p = t \triangleright y$, then:
   - t is added to the knowledge of the intruder, $K := K \cup \{t\}$
   - event $e = \langle a : p \rangle$ is added to tr, $tr := tr \cdot e$
3. if $a = t \triangleleft y$, then:
   - it is checked if the attacker ε can generate t using the knowledge K, if so, then event $e = \langle a : p \rangle$ is added to the trace: $tr := tr \cdot e$.
   - If ε cannot generate such a message, then the run stops.

□

Consider now a non-ground system scenario. Then, the set of runs associated with a non-ground system scenario S is given by the union of the runs of the ground instances of S. Clearly this set may be infinite. For instance, suppose we have a system scenario with a variable NA representing a nonce, and at least one constant na. Then we can instantiate NA with an infinite number of terms, using only the pairing operator: na,$\{na, na\}$,... However, thanks to the symbolic nature of the analyzer [4], given a non-ground scenario we always only have to check a finite number of (symbolic) traces.

4 A Trace Logic

In this section we introduce a trace logic language for describing security properties. The first step is to plan *what* our logic should be able to express. Some examples of properties are:

- "message m is not known by the intruder ε".
- "there is some event e recorded in trace tr".
- "nonce N is fresh".

Notice that these examples are only useful for having a rough idea of what are we aiming at; the exact ingredients of the logic (that is, the constructors and operators) must be carefully devised.

Keeping these examples in mind, we define the trace logic language. The formulae we are interested in will be evaluated at a specific point during protocol execution, we can think that at that specific state of execution there is a *current trace*, that we want to refer to, so it can be used in our predicate. Thus, we distinguish this current trace with a special constant, called *ctr*.

Definition 8. A trace logic formula is generated according to the following grammar:

$$F ::= \texttt{true} \mid \texttt{false} \mid F_1 \wedge F_2 \mid F_1 \rightarrow F_2 \mid \neg F \mid \forall e \in tr \ : F \mid \exists e \in tr \ : F \mid$$
$$generable(tr, m) \mid m_1 \not\preceq m_2 \mid e_1 = e_2$$

where e, e_1 and e_2 are events, tr is a trace and m, m_1, and m_2 are messages. □

Now we define the precise meaning of a formula.

Definition 9. Suppose $[F]$ is the set of well-formed trace logic formulae. Then the trace logic semantic function $[\![\cdot]\!] : [F] \rightarrow \{true, false\}$ is the following:

$[\![\texttt{true}]\!]$	$= true$	$[\![\texttt{false}]\!]$	$= false$
$[\![F_1 \wedge F_2]\!]$	$= true$ iff $[\![F_1]\!] = true$ and $[\![F_2]\!] = true$	$[\![F_1 \rightarrow F_2]\!]$	$= true$ iff $[\![F_1]\!]$ implies $[\![F_2]\!]$
$[\![\forall x \in tr \ : F]\!]$	$= true$ iff, for *each* e of tr, $[\![F[^e/_x]]\!] = true$	$[\![\neg F]\!]$	$= true$ iff $[\![F]\!] = false$
$[\![\exists x \in tr \ : F]\!]$	$= true$ iff, for *some* e of tr, $[\![F[^e/_x]]\!] = true$	$[\![generable(tr, m)]\!]$	$= true$ iff ε can generate m
$[\![m_1 \not\preceq m_2]\!]$	$= true$ iff m_1 is not a subterm of m_2	$[\![e_1 = e_2]\!]$	$= true$ iff e_1 is equal to e_2

□

Informally, predicate $generable(tr, m)$ is used for checking secrecy. A secrecy check can be performed by *directly* asking the intruder if he is able to *output* message m. However, notice that the capacity of the intruder to generate m depends on his specific knowledge K at the time of the check. In previous work[11, 4], a constraint solving approach is used for dealing with this, where *constraints* of the form $m : K$ are created and checked for satisfiability. Due to space constraints, in this paper we do not expand further; for a more formal approach the interested reader can consult the work of Millen and Shmatikov [11].

Recall that messages are represented as terms generated by a free algebra. Because of this, relations $\preceq$ and $=$ perform the comparisons *literally* on the terms. For instance, $h(t_1) \not\preceq h(t_2)$ iff $t_1 \not\preceq t_2$.

5 Protocol Specifications

In this section we combine the definition of protocol roles and trace logic formulae to obtain *extended protocol roles* and *extended system scenarios*.

Definition 10. An *extended protocol role* is a triple $\langle A, [M_1 \diamond B_1 : F_1, ..., M_n \diamond B_n : F_n], F_{inv} \rangle$, where $\{A, B_1, ..., B_n\} \subset Agents$, $M_1, ..., M_n$ are messages, $\diamond \in \{\triangleleft, \triangleright\}$ and $F_{inv}, F_1, ..., F_n$ are trace logic formulae. □

Intuitively, adding a formula F_i after a protocol role action means that F_i must hold after execution of the action. Moreover, invariant F_{inv} must hold after *each* action of the role.

Similarly, we define an *extended* system scenario.

Definition 11. An *extended system scenario* is finite multiset of instantiated extended protocol roles. □

In an extended system scenario we have gathered all the necessary ingredients needed to proceed with the verification.

Let us summarize the typical protocol design procedure:

1. identify the application requirements
2. translate the requirements into specifications of security properties (described by trace logic formulae)
3. provide a prototype of the protocol messages
4. append the trace logic formulae into the protocol messages and form the protocol specification
5. proceed with verification of the protocol specification

After Step 5 -if the verification phase points out a flaw- one needs to go back to Step 3, and propose a new prototype of the protocol messages. At this point, the designer should try to follow some *rules* of protocol design; even if the verification points out protocol flaws, a lot of time could be saved by the designer if she follows simple rules like the following:

- *never* sign something you do not know
- ensure freshness
- be explicit: do not over optimize a protocol by leaving out important pieces like names or nonces.

Having stated the abstract protocol design methodology, many questions arise (e.g. what properties a *refinement* of the protocol specification still meets?) that are out of the scope of this paper and are regarded as future work.

5.1 An example: A secure database query protocol

We apply this design technique to a case study. In this example, the designer intends to develop a protocol for secure querying of a database server.

Step 1. Identifying the application requirements. In our application, a client issues a query Q to the server and expects a result, modelled as $h(Q)$. The protocol should guarantee that Q and $h(Q)$ remain secret and that the client may be confident that she is talking to the proper server.

Step 2. Translation into trace logic formulae. The formulae for checking secrecy of Q and $h(Q)$ are easy to translate, since we have the predicate $generable(\cdot, \cdot)$ especially for this purpose:

- secrecy of Q: $\neg generable(ctr, Q)$
- secrecy of $h(Q)$: $\neg generable(ctr, h(Q))$

These formulae are expressing that the intruder does not know Q and $h(Q)$, respectively, in the current run ctr.

The second requirement, authentication of S to A, is slightly more difficult. We need to write a formula that states that if A received a message M from S, then it was really sent by S. We still do not know the protocol messages (they are going to be developed in the next step) so we leave for now the variable M undefined.

– authentication of S to A yields: $\langle A : M \triangleleft S \rangle$ `in` $ctr \rightarrow \langle S : M \triangleright A \rangle$ `in` ctr
where e `in` tr is an abbreviation of $\exists x \in tr : x = e$.

This formula is expressing that, if we receive message M *apparently* from S, then it should be recorded in the current trace that S sent it with intended destination A.

Step 3. Prototype of protocol messages. We start with a first naive version of the protocol:

Message 1. $A \rightarrow S \ : \ A, \{NA, Q\}_{pk(S)}$
Message 2. $S \rightarrow A \ : \ \{NA, h(Q)\}_{pk(A)}$

The protocol roles are:
$initiator(A, S, NA, Q) = \langle A, [A, \{NA, Q\}_{pk(S)} \triangleright S : F_1, \{NA, h(Q)\}_{pk(A)} \triangleleft S : F_2], F_{inv1} \rangle$
$server(A, S, NA, Q) = \langle S, [A, \{NA, Q\}_{pk(S)} \triangleleft A : F_3, \{NA, h(Q)\}_{pk(A)} \triangleright A : F_4], F_{inv2} \rangle$

Recall that variables in each role are 'standardized apart'. Also notice that we still have to define the trace logic formulae $F_1, F_2, F_3, F_4, F_{inv1}$ and F_{inv2} in the next step. Intuitively, the protocol works as follows: the client sends the first message containing her name in plain, and the challenge and query encrypted with the public key of the server. The server replies with a message encrypted with the client public key containing the challenge and the result of the query. The client then checks that the nonce she sent matches with the one she received; in this case the server is authenticated to the client and thus she can be sure about the authenticity of the query result.

Step 4. Final protocol specification. Now we have to append the formulae of *Step 2* into the protocol messages. For each formula, we have to decide where is its right place; that is, after execution of what protocol message the message should hold.

For checking secrecy, this is simple: the formula should always hold, since we want to know that a secret was revealed as soon as it happened. Thus, we set them as invariants:

$$F_{inv1} = \neg generable(ctr, Q) \text{ and } F_{inv2} = \neg generable(ctr, h(Q))$$

Notice that Q need not be quantified in the formula definition, since we can be confident that at evaluation time it is *already* instantiated to a constant (recall that the considered scenario is ground). Also note that Q is checked for secrecy in the initiator role, and not elsewhere: this is so because Q is generated by A.

On the other hand, we also want to check authentication of the server to the client. Agent A (in the initiator role) authenticates S after receiving the second message, so at that point we set our formula:

$$F_2 = \langle S : \{NA, h(Q)\}_{pk(A)} \triangleright A \rangle \text{ in } ctr$$

Notice that this formula is not exactly equal to the given in *Step 2*. In that formula we had an implication; in this formula we do not need the condition (because we know that $\{NA, h(Q)\}_{pk(A)}$ is in ctr: in fact, it is the last message received by A) and thus we only specify the consequent.

Finally, notice that we did not use all of the formulae of the protocol specification, only F_2, F_{inv1} and F_{inv2}. We therefore set the unused formulae: $F_1 = F_3 = F_4 = true$.

Step 5. Verifying the protocol. To verify the protocol, we can use the approaches of Durante et. al. [7] and Corin and Etalle [4]. We first specify a system scenario with only one client role, played by a, and one server role played by s:

$$\{initiator(a, S, na, q), server(A, s, NA, Q)\}$$

Notice that even if the verification phase does not expose any vulnerability, we cannot be sure that the protocol is secure for the general case; it is only secure for the system scenario tested. It can be the case that a bigger system scenario (i.e. with more participants) exposes a protocol vulnerability. This drawback is shared by all model-checking based approaches.

We ran the verification tools over the protocol specification, and obtained execution traces. Then, we checked[1] the validity of the trace logic formulae over these traces. As expected, the protocol is incorrect. A trace that violates formula $F_{inv2} = secrecy(h(q))$ is the following:
$\langle a : a, \{na, q\}_{pk(s)} \triangleright s\rangle, \langle \varepsilon : \varepsilon, \{na, q\}_{pk(s)} \triangleright s\rangle, \langle s : \varepsilon, \{na, q\}_{pk(s)} \triangleleft \varepsilon\rangle, \langle s : \{na, h(q)\}_{pk(\varepsilon)} \triangleright \varepsilon\rangle,$
$\langle \varepsilon : \{na, h(q)\}_{pk(\varepsilon)} \triangleleft s\rangle$
This trace corresponds a man-in-the-middle attack that can be shown in standard notation as follows (we use α and β to identify the different sessions):

$\alpha.1.\ a \rightarrow \varepsilon(s) : a, \{na, q\}_{pk(s)}$
$\beta.1.\ \varepsilon \rightarrow s : \varepsilon, \{na, q\}_{pk(s)}$
$\beta.2.\ s \rightarrow \varepsilon : \{na, h(q)\}_{pk(\varepsilon)}$

Here, $\varepsilon(s)$ is the standard notation for describing that intruder $\varepsilon$is impersonating honest agent s (in fact, in this attack εnever continues session α; he only intercepts the message and uses it in session β). Also notice that *only* the secrecy of $h(q)$ is violated in this attack (q remains secret). Since na is also compromised, the same trace can be extended to violate $F_2 = \langle s : \{na, h(q)\}_{pk(a)} \triangleright a\rangle$ `in` ctr. The attack is the following:

$\alpha.1.\ a \rightarrow \varepsilon(s) : a, \{na, q\}_{pk(s)}$
$\beta.1.\ \varepsilon \rightarrow s : \varepsilon, \{na, q\}_{pk(s)}$
$\beta.2.\ s \rightarrow \varepsilon : \{na, h(q)\}_{pk(\varepsilon)}$
$\alpha.2.\ \varepsilon(s) \rightarrow a : \{na, h(q')\}_{pk(a)}$

A simple modification of the previous protocol consists in also encrypting the identity of the client in the first message:

Message 1. $A \rightarrow S : \{A, NA, Q\}_{pk(S)}$

Now the intruder is unable to split identity A and nonce NA in Message 1. Note that this modification is natural given the previously exposed vulnerabilities.

Proceeding with verification, in this case we observe that traces do not violate the trace logic formulae. Thus, we can regard the protocol to be secure for the system scenario we tested; of course, bigger scenarios can be tested to increase confidency about the protocol security.

A new requirement The execution of a complex query can be a time-consuming task for the server. Thus, a (fairly realistic) additional requirement can be that the server should reject replayed queries.

We can translate this requirement by redefining formula F_3 in our protocol specification to express *freshness* as follows:

$$F_3 = \forall e \in ctr : \neg last\_event(ctr, e)\ \wedge\ (\pi_0(e) = S) \rightarrow (NA\ \not\preceq \pi_{msg}(e))$$

where, given an event $e = \langle A : M \diamond B\rangle$, then π_0 and π_{msg} are the projections s.t. $\pi_0(e) = A$ and $\pi_{msg}(e) = M$. Predicate $last\_event(tr, e)$ is true iff e is the last event of trace tr.

F_3 is expressing that, according to S's point of view, NA is fresh.

Results of Verification When we test this new added formula with the protocol messages unmodified, we expect a vulnerability, since the original prototype was designed *without* having in mind this requirement. Indeed, a trace that violates the specification was found. This trace exposes a trivial replay attack on the protocol, in which the intruder can replay the first message and thus violate the freshness of NA for the server S:

$\alpha.1.\ a \rightarrow s : \{a, na, q\}_{pk(s)}$
$\alpha.2.\ s \rightarrow a : \{na, h(q)\}_{pk(a)}$
$\beta.1.\ \varepsilon(a) \rightarrow s : \{a, na, q\}_{pk(s)}$

[1] At this moment, manually. An extension of the systems is work-on-progress.

But this first approximation gives us a hint for "patching" the protocol: the server should also generate a nonce which will be used by A in his query message. We modify the protocol adding the following messages:

Message 1. $A \rightarrow S : A$
Message 2. $S \rightarrow A : NS$
Message 3. $A \rightarrow S : \{A, NA, NS, Q\}_{pk(S)}$
Message 4. $S \rightarrow A : \{NA, h(Q)\}_{pk(A)}$

Verifying this protocol we did not find any violating trace for the chosen system scenario. Thus, we can regard that our protocol is secure: that is, it never violates formulae F_2 (authentication of S to A), F_3 (avoid replay attacks) and finally F_{inv1} and F_{inv2} (secrecy of the query and result of the query).

6 Conclusions

We have developed a *trace logic* for expressing security properties. Using this trace logic, the protocol designer can specify *precisely* what goals the protocol has to accomplish. Of course, what can be specified in the protocol goals is limited by the expresiveness of our trace logic language; fortunately, the language can be easily extended to cover new situations.

Our main novelty is that, by combining trace logic formulae and protocol messages, we form *protocol specifications* that include *all* the necessary information needed by the verifiers; no property is "hard-wired" into the protocol verifiers. We are currently extending the systems developed earlier [4, 7] to have a totally automated verification process; in the present situation, traces have to be checked manually.

References

1. K. Adi, M. Debbabi, and M. Mejri. A new logic for electronic commerce protocols. In N. Heintze and J. Wing, editors, *8th International Conference on Algebraic Methodology and Software Technology AMAST2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 499–513. Springer, 2000.
2. K. R. Apt. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice Hall, 1997.
3. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
4. R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. To appear in Proc. 2002 Static Analysis Symposium. Springer-Verlag, LNCS, 2002.
5. G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In A. Pettorossi, editor, *Proc. Eleventh International Workshop on Logic Program Synthesis and Transformation – LOPSTR 2001*, LNCS, pages 76–91. Springer-Verlag, 2002.
6. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
7. Antonio Durante, Riccardo Focardi, and Roberto Gorrieri. A compiler for analyzing cryptographic protocols using noninterference. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):488–528, 2000.
8. J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation – Artificial Intelligence. Springer-Verlag, Berlin, 1987. Second edition.
9. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW '97)*, pages 18–30. IEEE, 1997.
10. C. Meadows. Formal verification of cryptographic protocols: A survey. In J. Pieprzyk and R. Safavi-Naini, editors, *Advances in Cryptology – ASIACRYPT ' 94*, LNCS, pages 133–150. Springer-Verlag, 1995.
11. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. 2001 ACM Conference on Computer and Communication Security*, pages 166 – 175. ACM press, 2001.
12. J. K. Millen, S. C. Clark, and S. B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987. Special Issue on Computer Security and Privacy.
13. J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using murϕ. In *Proceedings of the 1997 Conference on Security and Privacy*, pages 141–153. IEEE Press, 1997.
14. A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *IEEE Symposium on Foundations of Secure Systems*, 1995.
15. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1999.