Verifying Security Protocols as Planning in Logic Programming

LUIGIA CARLUCCI AIELLO
Università di Roma "La Sapienza"
and
FABIO MASSACCI
Università di Siena

We illustrate $\mathcal{AL}_{SP}$ (Action Language for Security Protocol), a declarative executable specification language for planning attacks to security protocols. $\mathcal{AL}_{SP}$ is based on logic programming with negation as failure, and with stable model semantics. In $\mathcal{AL}_{SP}$ we can give a declarative specification of a protocol with the natural semantics of send and receive actions which can be performed in parallel. By viewing a protocol trace as a plan to achieve a goal, attacks are (possibly parallel) plans achieving goals that correspond to security violations. Building on results from logic programming and planning, we map the existence of an attack into the existence of a model for the protocol that satisfies the specification of an attack. We show that our liberal model of parallel actions can adequately represent the traditional Dolev-Yao trace-based model used in the formal analysis of security protocols. Specifications in $\mathcal{AL}_{SP}$ are executable, as we can automatically search for attacks via an efficient model generator (`smodels`), implementing the stable model semantics of normal logic programs.

Categories and Subject Descriptors: C.2.0 [**Computer and Communication Networks**]: General—*Security and Protection*; C.2.2 [**Computer and Communication Networks**]: Network Protocols—*Protocol verification*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; *Model Checking*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*; *Specification Techniques*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Logic Programming*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representation Languages*

General Terms: Design, Security, Theory, Verification

Additional Key Words and Phrases: Security protocols, specification language, logic programming, AI planning

This work is partly supported by ASI, CNR, and MURST grants. F. Massacci acknowledges the support of a CNR Short Term Mobility fellowship at Koblenz-Landau Univ.
Authors' addresses: L. Carlucci Aiello, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, 00198 Roma, Italy; email: aiello@dis.uniroma1.it; F. Massacci, Dipartimento di Ingegneria dell'Informazione, Università di Siena, via Roma 56, 53100 Siena, Italy; email: massacci@dii.unisi.it.

1. INTRODUCTION

Large open networks where trusted and untrusted parties coexist and where messages transit through potentially "curious" if not hostile providers pose new challenges to the designers of communication protocols.

Properties such as authenticity, confidentiality, proof of identity, proof of delivery, or receipt are difficult to assure in this scenario. Security protocols, communications protocols with an essential use of cryptographic primitives, aim at solving this problem [Schneier 1994, Chapters 2-4]. By a suitable use of shared and public key cryptography, random numbers, hash functions, encrypted and plain messages, a security protocol may assure an agent that the "invisible" responder at the other side of the network really is who he claims to be.

Not surprisingly, security protocols are very difficult to design by hand, as errors may creep in by combining protocols actions in ways not foreseen by the designer [Burrows et al. 1990]. Indeed, a protocol may be discovered flawed after many years [Lowe 1996], and even when one proves it correct, shortcuts to obtain export licenses can make it flawed [Ryan and Schneider 1998]. This situation is further complicated by the often ambiguous definition of the goals of a security protocol, which makes it difficult to assess what really counts as a flaw [Burrows et al. 1990; Gollmann 1996; Lowe 1996; Syverson and Meadows 1996].

Thus, the formal verification of security protocols became the subject of intense research in the last decade, after the seminal paper by Burrows, Abadi, and Needham [Burrows et al. 1990]. For instance, methods based on beliefs logics [Burrows et al. 1990; Syverson and van Oorschot 1994], model checking and verification in process algebras [Lowe 1996; Schneider 1998], theorem proving with induction [Paulson 1998; Bella and Paulson 1998], and state exploration methods [Mitchell et al. 1997; Meadows 1994] have been successfully used to verify and debug security protocols.

However, it became apparent that the formalization process itself was a serious bottleneck in the design process. At first, formalizing a protocol was hardly doable by somebody different from the proposer of the formal method itself. Second, the ambiguity of the goals of the protocol made it possible to find "attacks" with formal methods that security analysts will never regard as such (see for instance the debate between Lowe [1996] and Gollmann [1996]). Indeed, a security analyst can easily define a "security violation" in terms of sent, received, and missing messages, while the language gap between the analyst and the formal method makes it difficult to formally and exactly capture what is needed.

To bridge the gap, a number of intermediate languages for the description of protocols and protocol specifications have been proposed in the literature: for example CASPER [Lowe 1998], CASPL [Brackin et al. 1999], and NAPTRL [Syverson and Meadows 1996]. The protocol description is close to the operational description used in the security literature, and a compiler takes care of mapping it into the target formal language. Nevertheless, this situation is still unsatisfactory, as the security analyst must still buy, stock and barrel, the definitions of security goals and other properties (such as properties of private and public keys, the abilities of an intruder, etc.) which are provided by the formal

analyst. For instance, it is often not possible for the security analyst to "customize" the capabilities of a potential intruder (e.g., the intruder can intercept all messages coming from Alice but not those coming from Bob, etc.) without hacking into the formal language.

Thus, one would like to further extend the possibilities of specification languages, to allow for a flexible description of protocol actions, protocol goals, and protocol environment.

1.1 The Contribution of this Paper

In this paper we illustrate $\mathcal{AL}_{SP}$ (Action Language for Security Protocols [Carlucci Aiello and Massacci 2000]), an executable specification language for representing security protocols, and checking the possibility of attacks.

Our goal in the design of $\mathcal{AL}_{SP}$ is to provide the designer of security protocols with a formal environment where the logical formalization of a protocol and the behavior of an intruder are as close as possible to the operational specification in terms of message passing as one can find in published descriptions of protocols. An attack to a protocol should be easily specifiable in terms of the operations that one would or would not like to happen, without worrying about initiators, responders, or sessions. Properties of messages, keys, etc., should be easily formalized in the language.

Finally, the logical formalization should be executable, at least for model checking: there should be a system that takes the specification and outputs a model (if any) which shows an attack.

$\mathcal{AL}_{SP}$ is based on logic programming with the stable model semantics ($\mathcal{LP}_{SM}$), and is executable via a model finder, namely `smodels` [Niemelä 1999].

The choice of $\mathcal{LP}_{SM}$ [Apt 1990; Gelfond and Lifschitz 1988] is motivated by three properties of stable models:

—if a fact is true in a stable model there is a justification for it, and no circular justification is allowed;

—if something is not explicitly said, it is false by default;

—it is possible to say that some facts *must* be true in a stable model, and other facts *may* be true in it.

As we shall see, this is particularly appropriate in dynamic domains representing actions and changes, such as security protocols.

The innovative aspect of our proposal is the way we use logic programs to formalize protocols. The point of view we take comes from robotic planning. Planning, if seen as an artificial intelligence problem, is the task of automatically generating the sequence of actions a robot should perform to achieve its goal. The logic approach to planning sees plan generation as a task of automated reasoning: out of a declarative specification of the world, the (constructive) proof of the existence of a goal state can be easily transformed into a plan (i.e., a sequence of actions) to achieve it. Conversely, the nonexistence of a plan can be checked as an unsatisfiability problem: if no model for a goal state can be found, then we have proven that there is no plan that achieves it.

A plan is not different from a trace of a protocol. As in Paulson's approach for verifying security protocols [Paulson 1998; Bella and Paulson 1998], we describe each protocol run as a trace of send and receive actions. Paulson relies on inductive theorem proving for proving a protocol correct; we rely on stable-model generation for model-checking the protocol and finding traces corresponding to attacks.

Thus, our method, based on a logical specification for model-checking security protocols, offers an alternative to process algebras [Lowe 1996; 1998] and state exploration methods [Meadows 1994; Mitchell et al. 1997], and complements Paulson's inductive approach [Paulson 1998].

1.2 Plan of the Paper

In the rest of the paper we introduce security protocols by means of a simple running example (see Section 2). Then we briefly give some background on logic programs, stable models and `smodels` (Section 3), and on the logic approach to planning (Section 4). We then present the language $\mathcal{AL}_{SP}$ (Section 5), and show how to use it for modeling protocols (Sections 6 and 7) and attacks (Section 8). Finally, we show how to use them in practice (Section 9) and then discuss the adequacy of $\mathcal{AL}_{SP}$ specifications w.r.t. the standard trace-based semantics of security protocols (Section 10). A discussion on related works concludes the paper (Section 11). We refer instead to Carlucci Aiello and Massacci [2001] for a detailed presentation of some case studies: the Needham-Schroeder protocol, and the Aziz-Diffie key agreement protocol for mobile communication.

2. SECURITY PROTOCOLS: AN INTRODUCTION

As we anticipated in the introduction, a security protocol is a particular type of communication protocol in which cryptographic primitives play a key role. Here we sketch some features of security protocols and refer to Schneier [1994] for further details and references.

At first, when reasoning about the correctness of security protocols, there is a simplifying assumption about cryptographic primitives: they are almost perfect[1] so that cryptographic attacks are out of scope.

The basic setting of a security protocol, which has been first proposed by Dolev and Yao [1983], is constituted by two honest agents (say Alice and Bob), who want to communicate over a network owned by a potentially hostile agent (Charlie), and who use the services of one trusted server (Sam). Charlie may intercept, read, and fake any message in transit, but cannot break, nor alter, encrypted components (unless he knows the decryption keys). In more sophisticated protocols, such as an e-commerce protocol, we may have a trusted third party (Trent), who intervenes in case of disputes or who provides unrepudiable evidence that certain transactions took place. There are of course variations on this theme: there might be more agents or servers with different tasks; Alice may not entirely trust Bob, etc.

[1]For instance, hash functions are injective; correlation attacks on different cipher-texts are not possible; etc.

The goals of the particular security protocol that Alice and Bob run are application dependent, but may be loosely classified as follows:

—*aliveness* is sought when Alice follows the protocol and has some assurance that Bob is alive on the network (even if talking with somebody else);

—*authenticity* means that whenever Alice follows the protocol and gets a message allegedly from Bob, then Bob actually sent the message (possibly to Alice, depending on how strong we want authentication to be);

—*confidentiality* is obtained when Alice gets some secret information from Bob that is only shared by her, Bob, and possibly by some other entity trusted by her;

—*freshness* (or timeliness) means that Alice is assured that the message was sent recently, after she has done a certain action or after a given time;

—*proof of identity* is obtained when Alice is convinced that Bob is really the entity she is communicating with;

—*proof of delivery* means that Alice gets some message that convinces her that Bob received (and read) some crucial information from her;

—the wording *nonrepudiation* usually refers to a mixture of the above in which the evidence in the hands of Alice must be sufficient to convince somebody else (typically Trent).

Most protocols target more than one goal at the same time, or more complex goals such as distributed computation. For instance, in a key agreement protocol, the task of Alice is to get hold of a secret key (confidentiality) so that all subsequent messages she receives encrypted with that key could only come from herself or Bob (authenticity). In a protocol which certifies a public key, the Certification Authority Trent may want a proof of identity: if Alice asks for the public key K_A to be certified as hers, then Trent may ask Alice to sign some random number with the corresponding private key, to show that she really knows it.

We use the notation of the classical BAN paper [Burrows et al. 1990] to represent protocols. The basic building blocks of security protocols are agents' identifiers, keys, and (possibly random) numbers. Agents, and agents' identifiers, are denoted by capital letters: A, B, etc.

According to a notation that is standard in the computer security literature, we call *nonces* numbers used *once*, i.e., we assume the availability to each agent of a generator of random numbers ready to provide a new random number whenever needed. We denote nonces by capital N, possibly with an index to specify the agent that produces it. Hence, N_A indicates that N is a nonce generated by agent A.

Security protocols often make use of (at least) two types of *encryption keys*: *symmetric keys* (sometimes also called shared or secret keys) and *asymmetric keys* (sometimes also called public/private keys). Basically, we use symmetric keys when our crypto-algorithm can use the same key for both encryption and decryption. Thus, it is essential that the key is kept secret and only shared between those allowed to read the encrypted messages (hence the names "shared" or "secret" key).

We use asymmetric keys when the crypto-algorithm uses a key for encryption and another key for decryption, and it is difficult (unless NP = P or other similarly hard problems turns out easy) to obtain one from the other. With asymmetric keys, Alice can "publish" the encryption key (thus called her "public key") so that everybody can send her secure messages that only she can read by using the decryption key which she carefully guards (hence the name "private key"). Asymmetric keys are also used for signature schemes: when Alice wants to sign a message M she encrypts it with her (private) key. She is the only one who can do that. Using the public key, everybody else can recover the original message, and check that the signature is valid. See Schneider [1994] for further details.

A key K is decorated by indices that denote the agents that know that key: K_A is a key known (only) to agent A; K_{AB} is a key shared by agents A and B; etc. Public and private keys are sometimes prefixed by the letters "p" or "s". For instance, $pK(A)$ is A's public key, and $sK(A)$ is A's secret key.

To compose messages, we can use (at least) concatenation, hashing, and encryption. We denote the concatenation of message m_1 and m_2 as $m_1 \parallel m_2$. The hashing of a messagge m is denoted by $h(m)$.

Following the standard practice of security verification literature, the encryption of message m with key k is denoted by $\{m\}_k$ for both symmetric and asymmetric encryption. The type of encryption is determined by the type of the key.

To exemplify our specification methodology in $\mathcal{AL}_{SP}$, we focus here on a simple example: a variant of the ISO Challenge-Response protocol. More complex examples such as the classical Needham-Schroeder protocol and the Aziz-Diffie key agreement protocol are discussed by Carlucci Aiello and Massacci [2001]. Further examples, and in particular the whole Clarke-Jacob's library of authentication protocols [Clark and Jacob 1997], which is the current benchmark for verification of security protocols, has also been encoded in $\mathcal{AL}_{SP}$ [Lorenzon 2000].

Using BAN notation, the Challenge-Response protocol is the following:

$$\begin{aligned} A &\rightarrow B : A \parallel \{N_A\}_{K_{AB}} \\ B &\rightarrow A : B \parallel N_A \end{aligned}$$

The goal of the Challenge-Response protocol is to assure Alice that Bob is alive on the network. To accomplish this goal, Alice generates a nonce N_A and encrypts it with a secret key which is only shared by her and Bob. Then she sends the encrypted data to Bob. Bob decrypts the data and sends it in clear. When Alice receives N_A back, which she knows she has just generated in order to send it to Bob, and thus it is fresh and unguessable by others, she is sure that only Bob could have sent it by decrypting her challenge, and thus Bob is alive.

Here, each agent participanting in the protocol can play any of the two roles: challenger and responder. These roles can also be played in parallel. For instance, Alice may challenge Bob and respond to Charlie, who is challenging Sam, who is responding to Bob's challenge, etc. All these parallel exchanges of messages can be seen as a single protocol run.

However, if the protocol is run by both Alice and Bob in the attempt to provide a mutual assurance of aliveness, it has a serious flaw. Indeed, there is an attack (in the security jargon, *a mirror attack*) where Alice successfully completes a run of the protocol, allegedly with Bob, but Bob neither sent nor received any message.

Basically, the intruder intercepts the message from Alice to Bob and feeds it back to Alice (pretending it comes from Bob). When Alice dutifully replies with the challenge (her original challenge indeed), the intruder intercepts it and feeds it back again to Alice claiming to be Bob. The attack is automatically found by `smodels` in few seconds, when checking the $\mathcal{AL}_{SP}$ specification of the protocol. See Figure 1 (Section 9) for a sample output session of `smodels`.

3. LOGIC PROGRAMS WITH NAF, STABLE MODELS, AND SMODELS

In this section we give the basics of logic programming needed for our modeling tasks, and refer to Apt [1990] for further details.

It is well known that negation can be accommodated into logic programming as *negation as failure* (NAF). This is done by extending the language of Horn Logic by allowing negated literals in the body of clauses. In their simplest form, programs with negated literals (called *normal logic programs*) are sets of rules of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$$

where $m, n \geq 0$ and a, b_i, c_j are *literals*. Literals prefixed by the *not* operator are called *NAF literals* and are to be read as follows: *not c* is true if all attempts to prove c failed.

One of the most prominent semantics for normal logic programs is the *stable model* semantics proposed by Gelfond and Lifschitz [1988] (see Apt [1990] for a discussion). The intuition is to interpret the rules of a program P as constraints on a solution set S for the program itself. S is a set of atoms, and a rule of the form

$$a \leftarrow b_1, \ldots, b_m, not\ c_1, \ldots, not\ c_n$$

is a constraint on S stating that if $b_1, \ldots,$ and b_m are in S and none of $c_1, \ldots,$ and c_n are in it, then a must be in S.

We now consider ground rules, i.e., rules where atoms do not contain variables (but may contain arbitrary terms).

Definition 3.1 [*Gelfond and Lifschitz 1988*]. The *reduct* P^S of a ground logic program P with respect to a set of atoms S is the definite program obtained from P by deleting

(1) each rule that has a negative literal *not c* in its body with $c \in S$;
(2) each negative literal in the bodies of the remaining rules.

The reduct P^S is a definite logic program. Let $\mathcal{M}(P^S) = \mathcal{M}_{P^S}$ be the semantics of the definite logic program P^S, i.e. its minimal model.

Definition 3.2 [*Gelfond and Lifschitz 1988*]. A set of atoms M is a stable model of a normal logic program P iff $M = \mathcal{M}(P^M)$.

A normal logic program can have none, one, or many stable models.

Example. The logic program P defined by the rules

$$p \leftarrow not\ q, r \qquad q \leftarrow not\ p, s$$
$$r \leftarrow not\ s \qquad s \leftarrow not\ p$$

has a stable model $S = \{r, p\}$ because the reduct P^S is $\{p \leftarrow r.\ \ r \leftarrow\}$ and S is the deductive closure of P^S. P has another stable model, namely $\{s, q\}$.

The set $S' = \{s, p\}$ is not a stable model of P, as the reduct of the program $P^{S'}$ with respect to S' is the rule $p \leftarrow r$ and its deductive closure is the empty set.

Example. The logic program $\{p' \leftarrow not\ p',\ p.\ \ p \leftarrow\}$ has no stable model. In fact, if it had one, say S, then $p \in S$. If $p' \in S$, then the reduct is just $p \leftarrow$ and p' does not derive from it; hence $p' \notin S$, a contradiction. If $p' \notin S$, then the reduct is $\{p' \leftarrow p.\ \ p \leftarrow\}$ and p' derives from it; hence $p' \in S$, a contradiction.

If we remove the second rule, and the logic program becomes $\{p' \leftarrow not\ p',\ p\}$ then it has a stable model, namely the empty set (i.e., everything is false).

The definition of stable models captures the two key properties of solution sets of logic programs.

(1) Stable models are minimal: a proper subset of a stable model is not a stable model.

(2) Stable models are grounded: each atom in a stable model has a justification in terms of the program, i.e., it is derivable from the reduct of the program with respect to the model.

Minimality and groundedness make logic programming with stable model semantics ($\mathcal{LP}_{SM}$) particularly suited to model actions and change, in particular in security problems, where we want to model exactly what happened (i.e., we do not want to leave room for unwanted models), and where everything has a justification in the model. For example, if an intruder has got a secret key, there is an explanation in the model in terms of actions that the intruder has performed; it cannot have happened for other reasons not captured by the stable model itself. These properties of $\mathcal{LP}_{SM}$ make it a good choice for representing dynamic situations, such as planning and security problems, which are nonmonotonic. Hence, $\mathcal{LP}_{SM}$ proves to be better for application to security than other proposed logic formalisms based on (monotonic) modal logics (such as the BAN logic [Burrows et al. 1990] or the SVO logics [Syverson and van Oorschot 1994]), where together with the intended model we get unwanted ones, and there is no possibility of ruling them out.

The definition of stable models in terms of fix points is nonconstructive. Nevertheless, given a set of atoms it can be checked in linear time whether or not it is a stable model [Apt 1990]. Even though computing stable models has been proved to be NP-complete, now there are systems that can cope with ground programs having tens of thousands of rules. The system `smodels` [Niemelä 1999; Niemelä and Simmons 1997], is one of them. In order to introduce it, we present some more notions.

Logic programs with variables can be given a semantics in terms of stable models.

Definition 3.3. [*Apt 1990*] The stable models of a normal logic program P with variables are those of its ground instantiations P_H with respect to its Herbrand universe.

If logic programs are function free, then an upper bound on the number of instantiations is rc^v, where r is the number of rules, c the number of the constants, and v the upper bound on the number of distinct variables in each rule.

Definition 3.4 [*Niemelä 1999*]. A logic program P is *domain restricted* for a set of predicates D if, for each rule in P, every variable of the rule appears also in a positive body literal of the rule and the predicate of the literal is from D.

Programs where variables are sorted, i.e., where each variable occurring in a rule also occurs as the argument of a sort predicate[2] in the body of the same rule, are domain restricted to the domain of the sort predicates. This property holds for the logic programming language $\mathcal{AL}_{SP}$. Actually, we allow for functions in $\mathcal{AL}_{SP}$ programs, but we still keep the domain restrictedness because we impose that arguments of functions are sorted and range over finite domains.

Definition 3.5 [*Niemelä 1999*]. Let P be a logic program domain restricted for a set of predicates D. Let $\hat{D}$ be a set of ground instances of predicates in D, and $P_{\hat{D}}$ be the corresponding grounding of the program P. Then $\hat{D}$ is *complete* for P iff for each ground instance $\hat{d}$ of a predicate $d \in D$ it holds that (i) if $\hat{d}$ is in some stable model of P, then $\hat{d} \in \hat{D}$ and (ii) if $\hat{d}$ is in some stable model of $P_{\hat{D}}$, then $\hat{d} \in \hat{D}$.

The following theorem holds:

THEOREM 3.1. (*See Niemelä* [*1999*].) *Let P be a logic program domain restricted w.r.t. D. Let $\hat{D}$ be a set of ground instances of predicates in D, w.r.t. the Herbrand universe of P such that $\hat{D}$ is complete w.r.t. P. Then P and $P_{\hat{D}}$ have the same stable models.*

Domain restriction is a limitation that still leaves logic programs with expressive power to deal with interesting applications. At the same time the above theorem ensures that with this limitation the grounding problem and the search for stable models can be solved very efficiently, in particular if the domain is nonrecursive, i.e., D does not contain predicates that are recursively defined in P.

Again, $\mathcal{AL}_{SP}$ enjoys this property. In fact, the above theorem tells, that without losing completeness of the search, we may limit the ground instances in $P_{\hat{D}}$ to those where each literal with a predicate from D belongs to $\hat{D}$.

`smodels` [Niemelä and Simmons 1997; Niemelä 1999] is an implementation of $\mathcal{LP}_{SM}$, logic programming based on the stable model semantics for range-restricted function-free normal programs. It consists of two modules: the proper

[2]A positive literal with a monadic predicate.

`smodels`, which implements $\mathcal{LP}_{SM}$ for ground programs, and `lparse`, an efficient parsing module which works for range-restricted programs with nonrecursive domains. `lparse` automatically detects domain predicates, and deals with them very efficiently. In addition, it has some built-in arithmetic functions.

The implementation of `smodels` is a mixture of bottom-up and top-down backtracking search, where only the negative atoms in the program contribute to an increase of the search space; hence it is very efficient.

`smodels` offers the possibility of including a "choice" rule into logic programs:

$$\{c\} \leftarrow a, b$$

It reads as "if a and b are both true, then c *may* be in the stable model, but this is not mandatory." A program containing the choice rule can be translated into a normal program by a wise use of NAF. So, this rule is just a shorthand notation.

The language $\mathcal{AL}_{SP}$ borrows the choice rule from `smodels`, as it proves useful when representing security problems. For instance, it allows us to easily represent the fact that an intruder may intercept a message, but he is not compelled to.

4. LOGICAL APPROACH TO PLANNING

Planning is a research area in artificial intelligence aiming at the construction of algorithms (planners) that enable an agent to synthesize a course of actions that achieves its goals. Here we only present the definition of a planning problem and give enough background on the approach we take to represent the problem of verifying security protocols as a planning problem. We refer to Weld [1999] for a comprehensive survey.

A *planning problem* is a representation, in some formal language, of the following three aspects:

(1) a description of the initial state of the world;

(2) a description of the goal state the agent has to achieve;

(3) a description of the possible actions that can be performed by the agent. This is often called *domain theory* or *action theory*.

The solution of the problem (if one exists) is a *plan*: a sequence of actions that, when executed in a world satisfying the initial state description, achieves the goal.

In addition to the description of a planning problem, a planner may need a *background theory*, i.e., a description of generally known properties about the world.

A planning problem in the context of security protocols, where agents exchange messages and are subject to attacks by intruders, can be formulated as follows:

(1) the initial state is described in terms of the keys known to agents and the messages already exchanged (typically none), at the time the protocol starts;

(2) the goal state is an unwanted situation where some security violation has occurred;[3]

(3) actions are exchanges of messages among agents.

If a solution of the planning problem exists, then it is a sequence of actions leading to an unwanted situation, i.e., *a plan is an attack to the security of the protocol*.

The background theory, in this case, includes the description of how messages are composed and decrypted by agents, the properties of keys, how knowledge is attained by the agents participating in the protocol, etc.

In classical planning problems, a number of simplifying assumptions are made, some of them are absolutely not trivial to be removed: atomic time (or, equivalently, instantaneous duration of action execution), no exogenous events, deterministic action effects, etc. and are presently topics of active research.

We live with these assumptions, since they do not create problems with our application. In our application of planning to security, actions possibly performed by an intruder are not considered exogenous, as they are explicitly modeled. The intruder is one of the several agents in the scenario, typically more powerful than others.

A key problem is finding the adequate representation language: we look for a concise representation of dynamic situations (actions change the state of the world) that allows for an easy search for plans. Both specialized planning formalisms and logics have been proposed for representing and solving planning problems.

Usually, to represent predicates and functions whose value changes over time, *fluents* are introduced, i.e., predicates/functions with an extra parameter, typically the last one, representing the situation s or the time t where they hold. As an example: $\mathsf{got}(alice, m, s_1)$, and $\neg\mathsf{got}(alice, n, s_2)$ indicate that the agent *alice* got a message m in situation s_1 whereas she did not get it in situation s_2.

The term *situation* is borrowed from the *Situation Calculus* (SitCal), a "dialect" of first-order logic introduced by McCarthy and Hayes [1969], and recently revisited by Reiter [1991]. We borrow more nomenclature from the SitCal to further illustrate action theories and how security protocols can be accommodated into them.

In SitCal actions are denoted by function symbols; they can be performed in a situation, and the result is a new situation. In general, a situation is a term of the form $do(a_n, do(a_{n-1}, \ldots, do(a_1, s_0)\ldots))$ where do is a binary function symbol which, given an action a and a situation s, evaluates to its successor situation, and the constant symbol s_0 denotes the initial situation. Hence, situations can be seen as *histories*, or *traces*, of plan executions.

This description of actions is fairly close to Paulson's inductive presentation of security protocols which uses traces of atomic send and receive actions [Paulson 1998; Bella and Paulson 1998].

Actions modify the current status of the world. This is described by stating *causal laws*, i.e., how actions affect the values of the fluents, in the form of the

[3]For instance, A receives a message allegedly from B, who actually never sent it to A.

so-called *effect axioms*. For example, if a number N is part of another message M, then the action of an agent A sending the message M in a situation S causes the number N to be used. Thus, in all subsequent times, N has lost the property of being a freshly generated, unused number. Formally the effect axiom is the following:

$$\mathsf{part}(N, M) \supset \mathsf{used}(N, do(says(A, B, M), S))$$

The effect of an agent getting a message can be described as

$$\mathsf{got}(A, M, do(gets(A, M), S)).$$

Actions may have *preconditions*, i.e., requirements to be satisfied in order for the action to be executable. The predicate symbol *Poss* is used to indicate that all the preconditions are satisfied; hence an action is possible: $Poss(a, s)$ reads as "action a is possible in situation s." In SitCal, an axiom called *action precondition axiom* is associated with each action a. It has the form

$$Poss(a, S) \equiv \Pi_a(S)$$

where $\Pi_a(S)$ is a first-order formula with no occurrence of the function symbol *do*. Note here that we make a sort of "closure" assumption, i.e., all the necessary and sufficient conditions for the execution of action a are listed in $\Pi_a(S)$. In other words, we ignore all the other qualifications for the execution of a, by considering them irrelevant.[4] For example, we have

$$Poss(intercepts(spy, M), S) \equiv \exists A \exists B.\ \big(\mathsf{said}(A, B, M, S) \wedge \neg\mathsf{got}(B, M, S)\big)$$

meaning that the (only) preconditions for the capture of a message by the *spy* in the situation S is that the message has been sent by A to B and that the legitimate recipient has not got it already.

In addition to the specification of the effect axioms and the action precondition axioms, we have also to state the "*laws of inertia*" for our domain, i.e., actions change the value of some fluents; all the others are unaffected, so their truth value persists through the action execution.[5] As an example, intercepting a message does not change its content, the status of other messages already sent by the agents participating in the protocols, the encryption keys used for them, and so on.

In summary, a domain theory in SitCal consists of a set of actions and a precondition axiom for each action, and effect and inertial axioms for each fluent.

With a suitable definition of a domain theory and a rich background theory we would have had Paulson's inductive specification of security protocols, but for notational presentation: the value of each fluent would have been given by stating inductive definitions over traces of actions. Then we could have used Reiter's deductive approach to planning in SitCal [Reiter 1991] to obtain a plan (i.e., security attacks) out of a constructive proof of the formula $\exists S.Violation(S)$

[4]In the artificial intelligence literature, this is known as the *Qualification Problem*.
[5]This is known in the artificial intelligence literature as the *Frame Problem*.

from the domain theory and the properties of the initial situation. This would have given us an elegant duality between verification (by inductive theorem proving) and debugging (by finding attacks as plans).

Unfortunately, such an approach proved to be unworkable in practice: a number of preliminary experiments run with P. Baumgartner and U. Furbach from Koblenz University in October, 1999, have shown that state-of-the-art theorem provers (such as SPASS, FDPLL, or PROTEIN) could not cope even with our expository Challenge-Response protocol with the standard SitCal formulation.

The source of the problem can be identified, in our opinion, in the representation of parallel actions by their interleaving into a serial trace. If there are few possibilities for parallelism, this is not a major problem. If the domain allows for many unrelated parallel actions (and this is our case), the representation of interleavings blows up the search space. We illustrate this with an example on our Challenge-Response protocol. Alice may run the protocol with Bob, while Charlie runs it with Eve. Clearly, to find an attack on Alice's run, the outcome of Charlie's run is immaterial. However, the planner does not know it, and thus it will consider all possible interleavings of Alice and Charlie's runs. In particular, it will waste its time by considering the interleaving of Alice's correct runs with all possible runs by Charlie, whereas the attack is elsewhere.

Thus, we need to look at situations in a different way, which do not force parallel actions into serialized traces. We may consider a situation as a set of predicates that are true in it, each situation being distinguished by a timestamp, the time when it was generated by the execution of an action. By the close-world assumption, we read as false all predicates not occurring in the set.

Time arguments t are associated with values from an initial segment of the natural numbers; hence we can speak of a time $T_0 = 0$, of a total order among timestamps, of a successor time to t, namely $t + 1$, etc. The explicit representation of time has an added advantage when modeling security protocols which use timestamps (Kerberos is the paradigmatic example), as many protocols use digital certificates, and we may want to add checks on the expiration dates of certificates.

Fluents, instead of being decorated with a situation argument, have a time argument. We also decorate actions with a timestamp, so $do(a, s)$ now becomes an action predicate $a(t)$ which is true when action a is executed at time t yielding some results at time $t + 1$. Then, we can easily express parallelism: if $a(t)$ and $b(t)$ are both true then the actions a and b are executed in parallel at time t. This still leaves the possibility of considering plans as traces of action executions, with the proviso that (i) two different traces may lead to the same situation, (ii) we must add conditions to guarantee that parallel actions are serializable.

Causal laws and laws of inertia can be recast as constraints on the possible sets of predicates that are admissible for consecutive times t and $t + 1$. For instance, if the action predicate $says(A, B, M, t)$ is true, then $\mathsf{said}(A, B, M, t+1)$ must be true. The planning problem becomes the problem of finding a time t_g such that $Goal(t_g)$ holds (where $Goal(t_g)$ is the conjunction of the (relevant) formulas true in the goal state) and a set of actions (indexed by time instants)

that leads to t_g and respects the causal and inertial laws. In particular, since situations are no longer totally ordered by a sequence of *do*'s, and the time indexing imposes a temporal sequence only on some actions, parallelism among actions can be exploited to make plans shorter.

Following a proposal by Kautz and Selman [1992], we may solve a planning problem as model finding. The basic intuition is to limit the size of plans (by considering plans whose length l is $l \leq n$ for some fixed n) and then encode the planning problem as a satisfiability problem in propositional logic by encoding each causal and inertial constraint as a propositional formula. If we find a model for the goal and all the formulae, then we have a plan.

Plans can be also generated using logic programs. In particular, this has been done by Niemelä [1999] and Nebel and coworkers [Dimopoulos et al. 1997]. Both of them encode a planning problem as a ground logic program (written in $\mathcal{LP}_{SM}$) in such a way that the stable models of the encodings correspond to valid sequences of actions. Consequently, planning is the problem of finding a stable model that, for a given instant of time t, assigns true to all fluents that belong to the final situation. Action predicates that are true in the stable model and refer to time instants earlier than t constitute a plan that achieves the goal. Niemelä and Nebel both use `smodels` for their experiments.

We build on this idea, as `smodels` can be used for model generation of $\mathcal{AL}_{SP}$ specifications to verify whether plans to attack a protocol exist.

5. THE LANGUAGE $\mathcal{AL}_{SP}$

We start by introducing *basic sort predicates* to characterize the basic components of our language. Some of these predicates are common to many protocols:

—$\mathsf{ag}(A)$ denotes that A is an *agent*

—$\mathsf{nonce}(N)$ denotes that N is a *nonce*

—$\mathsf{key}(K)$ denotes that K is a *key*

Then we have *constructors for messages*. Some "classical" constructors are pairing, encryption, hashing, and exclusive-or, which we represent in BAN-like notation [Burrows et al. 1990]:

—$\{M\}_K$ is the encryption of M with the key K;

—$M_1 \parallel M_2$ is the concatenation of M_1 with M_2;

—$h(M_1)$ is the hash of message M_1;

—$M_1 \oplus M_2$ is the bitwise xor of M_1 and M_2.

Other functions may be added for particular protocols. From the standpoint of logic programs, they are just function symbols. For instance $\{M_1 \oplus M_2\}_K$ is just a pretty printing of the term `crypt(K,xor(M1,M2))`.

A *special sort predicate* is $\mathsf{msg}(M)$, which denotes that M is a valid (sub)*message* that may appear in a run of the protocol. The predicate $\mathsf{msg}(\cdot)$ specifies how messages are built with message constructors from basic

components. For sake of efficiency,[6] it is sometimes desirable to have a predicate protmsg(M) which distinguishes actual messages that can be sent/received in a protocol run from valid (sub)messages identified by msg(M).

We have predicates for defining *properties of messages*:

—part(M_1, M) denotes that M_1 is a submessage of M;

—invKey(K, K_I) denotes that K_I is the inverse of K;

—symKey(K) denotes that K is a symmetric key;

—sharedKey(K, A, B) denotes that K is a (symmetric) key shared between A and B;

—asymKeyPair($Kpriv$, $Kpub$) denotes that $Kpriv$ and $Kpub$ constitute an asymmetric key pair.

Other predicates may be introduced on demand.

Next, we introduce fluents corresponding to actions and properties, as we discussed in Section 4.

First, we have predicates for *knowledge and ability to compose messages*. From now on we must introduce *time* as an additional argument.

—*knows*(A, M, T) denotes that agent A knows message M at time T;

—synth(A, M, T) denotes that A can construct message M at time T.

Then we have predicates for *actions*:

—*says*(A, B, M, T) denotes the attempt[7] by A to send message M to B at time T;

—*gets*(B, M, T) denotes the receipt[8] of message M by B at time T;

—*notes*(A, M, T) denotes the storage of message M by A at time T.

These actions are basically present in the inductive theory of traces by Paulson and Bella [Paulson 1998; Bella and Paulson 1998]. Together with the predicate *knows*(A, M, T), they are the only predicates typeset in italics, as they are the only ones whose truth value we need to know for extracting attacks from stable models.

We also use the predicates said(A, B, M, T), got(B, M, T), and noted(A, M, T), with the obvious meaning that they are true when the corresponding action happened some time before T. We prefer this solution w.r.t. the explicit temporal operators, as for instance proposed by Syverson and Meadows [1996], because it leads to simpler semantics and gives us the flexibility to explicitly axiomatize when and how information about past runs of the protocol carries on into the current run. For instance, if we want to model agents with bounded memory, we may have a *Lost* action which can make noted(A, M, T) no longer true.

[6]In particular to make the ground representation of $\mathcal{AL}_{SP}$ specifications smaller.

[7]Attempt, because the *spy* might intercept the message, and the intended recipient might never see it.

[8]We only specify the recipient in the "get" action, as the sender is unreliable. See also Bella and Paulson [1998], or Paulson [1998], for a discussion of this modeling choice.

6. MODELING MESSAGES, KNOWLEDGE, AND ACTIONS IN $\mathcal{AL}_{SP}$

The starting point is the *background theory* for modeling the ability of agents to manipulate messages.

First, we specify which messages are valid. The direct approach would be using $\mathsf{msg}(\cdot)$ and defining messages inductively with constructors. For instance

$$\mathsf{msg}(M_1 \parallel M_2) \leftarrow \mathsf{msg}(M_1), \mathsf{msg}(M_2)$$

could be a rule for inductively defining message concatenation. Unfortunately, inductively defined predicates with function symbols have infinitely many ground instances.

For our particular application, we do not need inductive definitions for $\mathsf{msg}(\cdot)$: it is sufficient to use only messages that may occur as (sub)messages in a possible run of a protocol. For instance, the concatenation of thousands of nonces will never appear in the Needham-Schroeder public key protocol, and if it does it will be ignored by all honest agents. In most protocols, even complex ones, the format and number of valid messages is fixed[9] and can be expressed by few applications of the constructors to elements of the basic types.

Therefore we impose two constraints:

Definition 6.1. A basic sort predicate is admissible for $\mathcal{AL}_{SP}$ if it is not recursively defined by logic programming rules.

Definition 6.2. A logic programming rule defining message constructor(s) with the special sort predicate $\mathsf{msg}(\cdot)$ in the head of the rule is admissible for $\mathcal{AL}_{SP}$ only if basic sort predicates alone occur in the body of the rule.

Obviously if we introduce the additional predicate $\mathsf{protmsg}(\cdot)$, which is just a special case of $\mathsf{msg}(\cdot)$, then Definition 6.2 must be extended to it.

This is the only part of $\mathcal{AL}_{SP}$ specifications in which we forbid inductively defined predicates. The gain is substantial: if we have finitely many basic entities (agents, nonces, etc.), then we have finitely many messages in $\mathcal{AL}_{SP}$, and therefore we have only finite models. The price to pay is that the rules defining $\mathsf{msg}(\cdot)$ depend on the particular protocol we are analyzing. We must define each submessage in terms of the atomic components.

For our running example this boils down to

$\mathsf{msg}(A \parallel \{N\}_K) \leftarrow \mathsf{ag}(A), \mathsf{nonce}(N), \mathsf{key}(K)$ %actual protocol message
$\mathsf{msg}(B \parallel N) \leftarrow \mathsf{ag}(B), \mathsf{nonce}(N)$ %actual protocol message
$\mathsf{msg}(\{N\}_K) \leftarrow \mathsf{nonce}(N), \mathsf{key}(K)$ %submessage
$\mathsf{msg}(N) \leftarrow \mathsf{nonce}(N)$ %submessage

This step is entirely mechanical. Thus a translator has been implemented [Lorenzon 2000] for generating such $\mathcal{AL}_{SP}$-rules from protocol descriptions in CASPER [Lowe 1998].

Next, we need rules to model the ability of agents of manipulating messages. Lowe [1996] and Schneider [1998] use one relation to model the abilities to

[9]The recursive protocol analyzed by Paulson [1998] and Ryan and Schneider [1998] is an exception.

compose and decompose messages. Paulson [1998] treats these abilities separately with two relations. We prefer the latter approach: the presentation is simpler, and rules are easier to write.

We start by inductively defining the parts of a message:

$$\begin{aligned}
&\mathsf{part}(M, M) \leftarrow \mathsf{msg}(M)\\
&\mathsf{part}(M, M_1 \parallel M_2) \leftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{msg}(M_2), \mathsf{part}(M, M_1)\\
&\mathsf{part}(M, M_1 \parallel M_2) \leftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{msg}(M_2), \mathsf{part}(M, M_2)\\
&\mathsf{part}(M, \{M_1\}_K) \leftarrow \mathsf{msg}(M), \mathsf{msg}(M_1), \mathsf{key}(K), \mathsf{part}(M, M_1)
\end{aligned}$$

Some of our basic objects, typically keys, might have particular properties that we want to model using logic programming rules. This can be done, provided the resulting rules are admissible according to Definition 6.1.

To model private and public keys (see again Section 2) we need to state that a public key is the inverse of the private key and vice versa. To model the fact that asymmetric keys come in pairs we use a binary predicate $\mathsf{asymKeyPair}(Ks, Kp)$ in which the first argument is the secret key and the second is the public one.

The following modeling is admissible in $\mathcal{AL}_{SP}$ provided that $\mathsf{key}(\cdot)$ is a basic sort predicate.

$$\begin{aligned}
&\mathsf{isPubKey}(Kp) \leftarrow \mathsf{asymKeyPair}(Ks, Kp)\\
&\mathsf{isPrivKey}(Ks) \leftarrow \mathsf{asymKeyPair}(Ks, Kp)\\
&\mathsf{invKey}(Ks, Kp) \leftarrow \mathsf{key}(Ks), \mathsf{key}(Kp), \mathsf{asymKeyPair}(Ks, Kp)\\
&\mathsf{invKey}(Kp, Ks) \leftarrow \mathsf{key}(Ks), \mathsf{key}(Kp), \mathsf{asymKeyPair}(Ks, Kp)
\end{aligned}$$

In most protocols, each agent has only one such pair (Alice's private key, Bob private keys, etc.), and thus it is desirable to associate each agent A with a function which yields the agent's own private key $sK(A)$, and a function which yields its public key $pK(A)$. These functions must have the appropriate sort $\mathsf{key}(\cdot)$. This must be done with care as it may destroy the admissibility of the $\mathsf{key}(\cdot)$ sort predicate, or the finiteness of the ground representation of the domain of $\mathsf{key}(\cdot)$. However, the domain of the function is restricted to be a basic object (an agent in this case). If we have finitely many basic entities, then we have also a finite ground representation.

We code this new requirement as follows:

$$\begin{aligned}
&\mathsf{asymKeyPair}(sK(A), pK(A)) \leftarrow \mathsf{ag}(A)\\
&\mathsf{key}(sK(A)) \leftarrow \mathsf{ag}(A)\\
&\mathsf{key}(pK(A)) \leftarrow \mathsf{ag}(A)
\end{aligned}$$

If $\mathsf{ag}(\cdot)$ is not mutually defined in terms of $\mathsf{key}(\cdot)$, which is usually the case, as agents are not keys (although an agent's name may be used as a key or to build keys), $\mathsf{key}(\cdot)$ is still a basic predicate.

The notation is a bit heavy due to the need of using sort predicates to have domain-restricted programs. In the sequel, for sake of readability, *we omit all*

sort predicates and use the convention that A, B, C, etc. stand for agents, N stands for nonces, T stands for time, K stands for keys, and M stands for messages.

With this "omission" in mind, shared keys can be modeled with admissible predicates as asymmetric keys:

$$\begin{aligned}
&\mathsf{invKey}(K, K) \leftarrow \mathsf{symKey}(K)\\
&\mathsf{symKey}(K) \leftarrow \mathsf{sharedKey}(K, A, B), \mathsf{ag}(A), \mathsf{ag}(B)\\
&\mathsf{sharedKey}(shK(A, B), A, B) \leftarrow \mathsf{ag}(A), \mathsf{ag}(B)\\
&\mathsf{sharedKey}(shK(B, A), A, B) \leftarrow \mathsf{ag}(A), \mathsf{ag}(B)
\end{aligned}$$

The function $shK(B, A)$ should be of the appropriate sort predicate $\mathsf{key}(\cdot)$.

Next, we define what an agent can infer from messages and how the agent can construct messages. In the initial proposals by Lowe [1996], Schneider [1998], or Paulson [1998], knowledge is captured by complex operations or indirectly. In the NRL Protocol Analyzer [Meadows 1994], one explicitly represents actions that change the knowledge of agents, such as decrypting messages.

We build upon Paulson's approach, as extended by Bella and Paulson [1998], who make an explicit use of a predicate *knows* defined over agents, messages, and traces.

Since the word "knows" has many meaning in the AI and in the security literature (see for instance the substantial difference between Bella and Paulson [1998], Syverson and van Oorshot [1994], and Marrero et al. [Clarke et al. 1998]), it is important to clarify its meaning: by $knows(A, M, T)$ we mean that agent A can get hold of M by breaking down messages that were sent, received, or somehow got. We use the predicate $knows(A, M, T)$ to represent the knowledge that an agent can gather by observing the network traffic directed to himself (or to other agents, if he is the intruder) and by performing cryptographic operations for which he has the right decryption keys. Obviously, this knowledge can then be used to compose other messages, but to this end we will introduce another predicate.

As a consequence of sending and receiving actions, the knowledge of agents will change. However, we do not change state when reasoning about knowledge. Reasoning about knowledge of messages is "instantaneous" and deterministic: as soon as an agent receives the correct bits of information that agent immediately derives all consequences by breaking and decrypting all messages for which he has a key.

With the above interpretation of knows, we can write the first two rules about knowledge: you know something because you either got it or said it to somebody.

$$\begin{aligned}
&knows(A, M, T) \leftarrow \mathsf{got}(A, M, T)\\
&knows(A, M, T) \leftarrow \mathsf{said}(A, B, M, T)
\end{aligned}$$

As we said in Section 4, this is exactly the way in which model-based planning works: an action happening at a given time (saying a message) implies its preconditions (knowing its content); equally, an action happening at a given time (getting a message) implies its effects at some later time (knowing the message content). See Kautz and Selman [1992] for a further discussion on the adequacy of this specification. Since we decided to make reasoning about

knowledge instantaneous, both things happen at the same time. To avoid frame axioms for knowledge, we used $\mathsf{got}(A, M, T)$ instead of $gets(A, M, T)$, and similarly for sending messages.

Then we need rules to peel constructors off messages:

$$\begin{aligned} knows(A, M_1, T) &\leftarrow knows(A, M_1 \parallel M_2, T) \\ knows(A, M_2, T) &\leftarrow knows(A, M_1 \parallel M_2, T) \\ knows(A, M, T) &\leftarrow knows(A, \{M\}_K, T), \\ &\qquad knows(A, K_I, T), \mathsf{invKey}(K, K_I) \end{aligned}$$

An observation is needed on the rule for decryption. Even if it is conceptually identical to the corresponding rules in other formal verification papers [Burrows et al. 1990; Lowe 1996; Paulson 1998; Schneider 1998] it is not satisfactory from an epistemological point of view: in fact, it is not enough for A to know K_I and for K_I to be the inverse of K; A should also know that K_I is the inverse of K. In practical implementations of security protocols, the information about the key to use may not be specified.[10] The rationale is that whereas legitimate parties in a correct run surely know what key to use, one may not want to make this information available to potential intruders. Here we consider it implicit, and leave the explicit representation of these epistemic notions to future work.

We can also exploit the properties of logic programs with negation as failure: what is not explicitly said is false by default. So, if we do not give a rule for hashing, then we cannot conclude anything about it, i.e., we can not reconstruct a message out of its hash (as expected).

Using the stable model semantics of logic programs has another advantage: we can rule out unwanted models introduced by ungrounded circular reasoning about cryptographic properties; models that are difficult to eliminate with other (monotonic) logics. For example, we can model the exclusive-or operation in $\mathcal{AL}_{SP}$:

$$\begin{aligned} knows(A, M_1, T) &\leftarrow knows(A, M_1 \oplus M_2, T), \\ &\qquad knows(A, M_2, T) \\ knows(A, M_2, T) &\leftarrow knows(A, M_1 \oplus M_2, T), \\ &\qquad knows(A, M_1, T) \end{aligned}$$

Now suppose that with these rules we want to model the knowledge relative to a pay-TV channel. In a pay-TV channel the *video* signal is xored together with a *key* and then transmitted on the air. So, if you are a legitimate subscriber and know the *key*, once you received the encrypted signal $video \oplus key$ you should be able to recover the *video*.

So, let P be the logic program with the above two rules only, and opportunely grounded over the domain *alice*, *bob*, *video*, and *key*. If we add to P the two ground facts $knows(bob, key, 1)$ and $knows(bob, video \oplus key, 1)$ then we can correctly infer that in every stable model of the augmented program it is true that $knows(bob, video, 1)$.

[10]The ISO ASN.1 standard only specifies that encrypted messages have the type opaque. Further details on the key to use must be explicitly concatenated to the message.

However, if *alice* is a hacker who just got hold of the encrypted signal on the air, i.e., we add to P the ground fact $knows(alice, video \oplus key, 1)$, the correct interpretation would be that *alice* does not know anything else. In first-order logic, or any other monotonic belief logic, we would have to take on board also the model in which A knows M_1 and M_2. This knowledge will be self-sustained: we could use the first rule to derive that M_1 is there because M_2 is there, and the second rule to conclude that M_2 is there because M_1 is there. Stable models forbid this (wrong) circular reasoning in $\mathcal{AL}_{SP}$: $knows(alice, video, 1)$ and $knows(alice, key, 1)$ are not grounded in the premise $knows(alice, key \oplus video, 1)$.

To express the knowledge derived by the ability of composing messages we use another predicate:

$$
\begin{aligned}
&\mathsf{synth}(A, M, T) \leftarrow knows(A, M, T)\\
&\mathsf{synth}(A, \{M\}_K, T) \leftarrow \mathsf{synth}(A, M, T),\\
&\qquad\qquad\qquad\qquad\quad knows(A, K, T)\\
&\mathsf{synth}(A, M_1 \parallel M_2, T) \leftarrow \mathsf{synth}(A, M_1, T),\\
&\qquad\qquad\qquad\qquad\quad \mathsf{synth}(A, M_2, T)\\
&\mathsf{synth}(A, h(M), T) \leftarrow \mathsf{synth}(A, M, T)\\
&\mathsf{synth}(A, M_1 \oplus M_2, T) \leftarrow \mathsf{synth}(A, M_1, T)\\
&\qquad\qquad\qquad\qquad\quad \mathsf{synth}(A, M_2, T)
\end{aligned}
$$

Composition, decomposition, decryption, and encryption are, in a sense, instantaneous if you have the right key. It is possible to add explicit enciphering and deciphering actions, but this would make our formalization less declarative and our planned attacks longer (and thus harder to find).

We can now build the first part of the *action theory* in $\mathcal{AL}_{SP}$. The following *successor state axioms* are intuitive:

$$
\begin{aligned}
&\mathsf{said}(A, B, M, T+1) \leftarrow says(A, B, M, T)\\
&\mathsf{got}(B, M, T+1) \leftarrow gets(B, M, T)\\
&\mathsf{said}(A, B, M, T+1) \leftarrow \mathsf{said}(A, B, M, T)\\
&\mathsf{got}(B, M, T+1) \leftarrow \mathsf{got}(B, M, T)
\end{aligned}
$$

We need similar axioms for the pair $notes(A, M, T)$ and $\mathsf{noted}(A, M, T)$. The first two rules model a causal law (saying something now causes it to be said afterward), and the remaining two rules model the law of inertia.

We have not found the need for forgetful agents in the protocols described and verified so far in the security literature [Clark and Jacob 1997; Schneier 1994], though there might be esoteric protocols for which we may need to modify the "law of inertia."

Reasoning about *freshness* of messages is a difficult issue in almost all formalisms, as freshness guarantees are the pillars on which every proof of authentication or security rests (see our discussion of the Challenge-Response example in Section 2 or Schneier [1994], Burrows et al. [1990], and Paulson [1998]). Basically, freshness is guaranteed by the usage of nonces and timestamps, and one needs to model in one's own formalism the following intuition [Burrows et al. 1990]:

> If you've sent Joe a number that you have never used for this purpose before and if you subsequently receive from Joe something that depends on knowing that number, then you ought to believe that Joe's message originated recently—in fact after yours.

In other words, we need to model an endless supply of fresh numbers which, once used by somebody in some message, lose the qualification of being fresh. The occurrence of a "once fresh," "never used by anyone before," nonce in a message is the proof that the message is relatively fresh, i.e., it has been composed after the nonce has been firstly generated and used.

To model freshness in $\mathcal{AL}_{SP}$ we first introduce a fluent which specifies when something is not surely fresh: $\mathsf{used}(M, T)$ is true when message M has been used by somebody before time T. This is typically used for nonces. The axiomatization is simple:

$$\mathsf{used}(M, T+1) \leftarrow \mathsf{used}(M, T)$$
$$\mathsf{used}(M, T+1) \leftarrow \mathit{says}(A, B, M_1, T), \mathsf{part}(M, M_1)$$

Since we have parallel actions, these rules are not sufficient to avoid bad surprises when reasoning about freshness of nonces: we must be sure that two agents are not trying to use the same nonce in parallel, which might well be fresh at time T (as none of them used it), but might yield an inconsistent state at $T+1$ if both of them used it as fresh.

We can rule out this event with the following rules:[11]

$$\begin{aligned}\mathsf{usedPar}(M, T) \leftarrow\ & \mathit{says}(A_1, B_1, M_1, T),\\ & \mathit{says}(A_2, B_2, M_2, T),\\ & \mathsf{part}(M, M_1), \mathsf{part}(M, M_2),\\ & (A_1 \neq A_2 \mid B_1 \neq B_2 \mid M_1 \neq M_2)\end{aligned}$$
$$\mathsf{fresh}(M, T) \leftarrow \mathit{not}\ \mathsf{used}(M, T), \mathit{not}\ \mathsf{usedPar}(M, T)$$

Now we have the machinery to write the preconditions of a protocol step which requires to generate a new nonce N or a new session key K: simply say that $\mathsf{fresh}(N, T)$ or $\mathsf{fresh}(K, T)$. This allows us to decouple the problem of specifying that a protocol is using something fresh, from the problem of specifying how many "potentially fresh" things we have. This is sharply in contrast with the difficulties faced by the CSP-based model-checking approaches [Lowe 1997] where to write the specification of the protocol one must know the available nonces, keys, and agents beforehand. This happens because "freshness" is modeled by assigning a set of nonces to each agent and by duplicating CSP processes, according to which nonce is used in the protocol. In other words, if agent A can use nonce N_{A1} and N_{A2} then agent A is modeled by writing two processes: one that uses N_{A1} and one that uses N_{A2}. This hard-to-manage modeling capability is also one of the reasons behind the development of the intermediate language CASPER [Lowe 1998].

If we allow for infinitely many nonces, session keys, or whatever needs to be freshly generated, we would capture precisely the domain. For practicality, we

[11] We use $h \leftarrow a, b, (c_1 \mid c_2 \mid c_3)$ as a short-hand notation for the conjunction of the three rules $h \leftarrow a, b, c_i$ with $i = 1 \ldots 3$.

limit ourselves to finitely many ground instances of the basic sorts and thus only finitely many fresh things.

So far we have applied the fresh(M, T) rule to all possible messages in a protocol. However, the size of the grounded $\mathcal{AL}_{SP}$ specification grows quadratically with the number of agents and messages. In some cases it is convenient to adapt the rule to the protocol and limit M to be a nonce (or key, if fresh sessions keys are distributed by the protocol), and constrain M_1 and M_2 to have the form of the actual messages where such nonces or keys are first introduced. Already the use of protmsg(M_i) substantially simplifies the size of the ground representation. These restrictions are common in other model-checking approaches, either state-based [Mitchell et al. 1997] or process-based [Lowe 1996].

Now, we define the preconditions for getting and receiving messages that are independent of the protocol that we want to analyze. As in Bella and Paulson's approach [Bella and Paulson 1998], modeling message reception is simple:

$$\{gets(B, M, T)\} \leftarrow says(A, B, M, T)$$

We use the choice rule (see Section 3) to specify that if A attempts to send a message M to B at time T, then B *may* receive it. There are stable models where the message is delivered (the normal execution of the protocol) and stable models where B does not receive the message. In these latter models, the intruder has intercepted the message. Thus, we do not need to explicitly model the action of message interception, as done by Meadows [1994] or Syverson and Meadows [1996]. We can also model delayed delivery by replacing "says" with "said" in the body.

Modeling the intruder, along the classical Dolev and Yao model (see Section 2), is simple:

$$\{gets(spy, M, T)\} \leftarrow says(A, B, M, T)$$
$$\{says(spy, B, M, T)\} \leftarrow \mathsf{synth}(spy, M, T)$$

The intruder may get any message in transit and may say any message (but in both cases needs not to). There will be stable models of the protocol where the intruder does nothing (the correct runs) and stable models where he is busy.

A remark is in order. The rule describing message creation by the intruder can be finitely grounded. For conciseness sake, here we have omitted the domain predicates for messages and agents, but they are present in the actual rule, and with Definition 6.2 they make the rules domain restricted.

Of course, other specifications of the intruder's actions are possible, according to one's own application domain. For instance, the intruder may only read messages but not intercept them; he may only work along some communication lines and not others (e.g., may only grab Alice's messages but not Bob's ones), etc.

7. MODELING PROTOCOLS IN $\mathcal{AL}_{SP}$

For planning attacks, we need to complete our action theory with the preconditions for the actions of the protocol we want to analyze. This is the part the protocol analyst has to write, as this is also the place where she may wish to

make explicit one or more checks she believes should be made before a message is sent or accepted.

To write these rules we simply *look at the protocol as specified*. Assume that the protocol looks something like

$$
\begin{array}{ll}
A \rightarrow B_{i_1} & : m_{i_1}\% \text{ first message } A \text{ must send} \\
\ldots & \\
B_{j_1} \rightarrow A & : m_{j_1}\% \text{ first message } A \text{ must receive} \\
\ldots & \\
A \rightarrow B_{i_s} & : m_{i_s}\% \text{ last message } A \text{ must send before } m \\
\ldots & \\
B_{i_r} \rightarrow A & : m_{j_r}\% \text{ last message } A \text{ must receive before } m \\
\ldots & \\
A \rightarrow B & : m \\
\ldots &
\end{array}
$$

Then, to write down the axiom precondition saying that agent A may send message m to B, we identify the messages that should have been received and sent before m in a correct run, and we write the following choice rule:

$$
\begin{array}{rl}
\{says(A, B, m, T)\} \leftarrow & \mathsf{said}(A, B_{i_1}, m_{i_1}, T), \ldots, \mathsf{said}(A, B_{i_s}, m_{i_s}, T), \\
& \mathsf{got}(A, m_{j_1}, T), \ldots, \mathsf{got}(A, m_{j_r}, T), \\
& \text{protocol-dependent literals}
\end{array}
$$

Usually, the preconditions for sending message m_i by agent A are the actions that agent A has performed up to step i according to his view of the protocol run, plus the additional conditions. The protocol-dependent conditions are typically freshness of nonces or timestamps or additional checks that the protocol analyst wants enforced.

The protocol analyst may also weaken the preconditions, w.r.t. those recommended here. For instance, she may state that for sending a message agent A may just look at the last received message. This is sensible if we want to test whether the protocol is secure when using stateless but fast implementations. The choice rule would then become

$$
\begin{array}{rl}
\{says(A, B, m, T)\} \leftarrow & \mathsf{got}(A, m_{j_r}, T), \\
& \text{protocol-dependent literals}
\end{array}
$$

In our Challenge-Response protocol, the rules would be

$$
\begin{array}{rl}
\{says(A, B, A \,\|\, \{N\}_K, T)\} \leftarrow & \mathsf{sharedKey}(K, A, B), \\
& \mathsf{fresh}(N, T), A \neq B \\
\{says(B, A, B \,\|\, N, T)\} \leftarrow & \mathsf{got}(B, A \,\|\, \{N\}_K, T), \\
& \mathsf{sharedKey}(K, A, B), A \neq B
\end{array}
$$

8. SPECIFYING AN ATTACK

The declarative specification of attacks is also the task of the protocol analyst, as different analysts may disagree on the security properties a protocol should provide (see Gollmann [1996] versus Lowe [1996]).

The protocol analyst must add a rule for a predicate *attack*, specifying the send or receive events which must be present or absent in a protocol run to

constitute a security violation. Specifying "what an attack is" is easy for a security analyst, who knows which security property the protocol claims to achieve. In contrast, finding "how an attack is actually done" is a hard task even for security specialists, and here the usage of automated reasoning tools pays off.

For instance, in Challenge-Response protocols, if the challenger receives the response, he can be assured that the responder actually sent it. So, an attack is simply a protocol run in which the challenger sent the challenge and received the response but in which the responder did not actually respond. Finding such a run, with parallel and interleaved combinations of valid protocol steps, is the difficult part of the analysis and is left to the planner. In a certification protocol, if the certification authority certifies a public key as belonging to an individual, say Alice, then the protocol ought to guarantee that Alice does own the corresponding private key. Again, an attack is simply a protocol run in which the certification authority issued Alice's certificate but Alice did not know the corresponding private key.

The procedure for specifiying attacks is simple and only requires us to look at the description of the protocol. We list below some of the most common cases.

As for *authentication attacks*, we first list the sequence of messages received by an agent A and the message he sent and received up to the point we want to analyze, i.e., the run relative to A of a correct run. Then we pick up one corresponding send (or receive) action from the "right" agent and state that this action is missing.

Consider again the protocol skeleton presented above, and suppose the run correctly completed for A: she started by sending m_{i_1}, at last correctly received m_{j_r}, allegedly from B_{j_r}, and sent m to B. Suppose, that according to the analyst understanding of the security goals of the protocol, there is an authentication attack if a certain intermediate message m_j was received by A but was not sent by the appropriate agent B_j. Then we write down the rule

$$
\begin{array}{lll}
attack \leftarrow & \mathsf{said}(A, B_{i_1}, m_{i_1}, T), \ldots, \mathsf{said}(A, B_{i_s}, m_{i_s}, T), & \%\text{messages correctly sent} \\
& \mathsf{got}(A, m_{j_1}, T), \ldots, \mathsf{got}(A, m_{j_r}, T), & \%\text{and correctly received} \\
& \mathsf{got}(A, m_j, T), & \%\text{a message looks correct} \\
& \mathit{not}\ \mathsf{said}(B_j, A, m_j, T), & \%\text{in reality from the } \mathit{spy} \\
& \text{protocol-dependent literals} & \%
\end{array}
$$

This is an authentication attack from the viewpoint of A: the protocol apparently completed up to the ith step, and yet B_j did not send the required message to A.

It is the task of the planner to find the attack (if it exists) by finding a stable model of the protocol specifications (background theory and action theory) where *attack* is true.

Among the protocol-dependent conditions, we may add $A \neq spy$, $B_j \neq spy$, and $B_{i_h} \neq spy$ for all i_h as needed. These additional conditions may be weakened if our protocol is supposed to be secure also when some participants are not trusted, as it might be the case for e-commerce protocols.

The specification of the attack is a good blend of operational and declarative specification. We say which bits of a run constitute an attack in terms

of a natural operational semantics; still we are purely declarative. The protocol analyst must only specify *what* she considers an attack, and she needs not know beforehand whether the potential attack she worries about actually exists.[12] Finding *how* the attack must be carried out, if it exists, is the task of the planner, which must unroll normal protocol actions into a valid plan.

In our running example we have

$$\begin{aligned} \mathit{attack} \leftarrow\ & \mathsf{said}(A, B, A \parallel \{N\}_K, T), \mathsf{got}(A, B \parallel N, T), \\ & \mathit{not}\ \mathsf{said}(B, A, B \parallel N, T), \\ & \mathsf{sharedKey}(K, A, B), A \neq \mathit{spy}, B \neq \mathit{spy} \end{aligned}$$

To check for *confidentiality attacks*, we add the literal $\mathit{knows}(\mathit{spy}, m, T)$ to the goal (if m is the message which is supposed to remain secret) and write the rule

$$\begin{aligned} \mathit{attack} \leftarrow\ & \mathsf{said}(A, B_{i_1}, m_{i_1}, T), \ldots, \mathsf{said}(A, B_{i_s}, m_{i_s}, T), \\ & \mathsf{got}(A, m_{j_1}, T), \ldots, \mathsf{got}(A, m_{j_r}, T) \\ & \mathit{knows}(\mathit{spy}, m, T), \\ & \text{protocol-dependent literals} \end{aligned}$$

In some cases, to correctly model authentication and confidentiality attacks, it is necessary to add an *oops-rule*, where the intruder gets hold of past confidential data [Paulson 1998]. Typically, one introduces the oops-rule to check the strength of the protocol w.r.t. compromised past session keys or nonces. This is essential to reason correctly for protocols such as Needham-Schroeder.

The typical format of the oops-rule is

$$\begin{aligned} \{\mathit{notes}(\mathit{spy}, m, T)\} \leftarrow\ & \mathsf{said}(A, B_{i_1}, m_{i_1}, T), \ldots, \mathsf{said}(A, B_{i_s}, m_{i_s}, T), \\ & \mathsf{got}(A, m_{j_1}, T), \ldots, \mathsf{got}(A, m_{j_r}, T) \\ & \text{protocol-dependent literals} \end{aligned}$$

where m is the concatenation of all nonces and session keys exchanged up to m_{j_r}. We typically only need one oops-rule for the whole protocol, so that m_{j_r} is also the last message exchanged in a correct protocol run.

Once we have the oops-rule, we need to slightly change the rules to model attacks by excluding that the oops-rule takes place for the current run:

$$\begin{aligned} \mathit{attack} \leftarrow\ & \mathsf{said}(A, B_{i_1}, m_{j_1}, T), \ldots, \mathsf{said}(A, B_{i_s}, m_{j_n}, T), \\ & \mathsf{got}(A, m_{j_1}, T), \ldots, \mathsf{got}(A, m_{j_r}, T), \\ & \mathit{not}\ \mathsf{noted}(\mathit{spy}, m, T), \\ & \mathsf{got}(A, m_j, T), \mathit{not}\ \mathsf{said}(B_j, A, m_j, T), \\ & \text{protocol-dependent literals} \end{aligned}$$

This rule just says that the allegedly correct run has not been directly compromised by an oops-rule, but does not say anything about past runs which might have been compromised by an oops-event.

[12] If, conversely, our method could be used only for checking whether known attacks exist, it would be of little practical interest.

9. MODEL-CHECKING AND VERIFICATION

To summarize, for practical verification of security protocol specifications in $\mathcal{AL}_{SP}$, we can use the `smodels` system:

—we use the $\mathcal{AL}_{SP}$ specification of the protocol-independent background and action theories;

—we write the $\mathcal{AL}_{SP}$ specification of the protocol-dependent part with choice rules for representing the correct execution of the protocol;

—we define a rule for the security property (attack) we want to check;

—we merge the three specifications, set the maximum execution time of the protocol to t_{max}, and a bound on the number of basic objects (agents, nonces, etc.);

—we use `lparse` to obtain the finite ground representation of $\mathcal{AL}_{SP}$ specifications;

—we use `smodels` to look for a stable model of the ground system;

—if no stable model exists, then the attack does not exist for all (possibly parallel) interleaved runs of the protocol up to t_{max};

—if there is a stable model, then we look for the atoms representing actions ($says(A, B, M, T)$, $gets(B, M, T)$, $notes(A, M, T)$) that are true in the model, as they give us the sequence of (possibly parallel) actions that constitute the attack.

If we are looking for confidentiality attacks, then we must also gather the atoms $knows(spy, M, t_{max})$ that are true in the model. They represent the knowledge of the intruder at the end of the protocol.

To speed up the search, we may add extra constraints on the rules. For instance, in the action preconditions $\{says(A, B, M, T)\} \leftarrow \ldots$ we may add $A \neq B$ in the body of the rule or *not* $\mathsf{said}(A, B, M, T)$. These additional facts substantially reduce the size of the ground program which we use for model-checking and do not usually change the possibility of finding attacks. For instance, for $A \neq B$, in no protocol does an agent knowingly send a message to himself. He might be fooled into running the protocol with himself (a "mirror attack"), but this typically happens because he is running two protocol instances in parallel, one instance as initiator and one instance as responder. It is the intruder who intercepts the messages and feeds them back to the unsuspecting victim. These attacks are not prevented by this optimization.

In Figure 1 we give an example of a verification session with the Challenge-Response protocol.[13] `smodels` found the mirror attack that we have mentioned in Section 2. The number at the right indicates the time instant at which the action takes place.

In this case, the attack that is found is not minimal, in the sense that some steps can be removed from it. Indeed at time 4 we only need the spy to send the nonce `n1` to `A` for the attack to take place. It is interesting to note that the attack is found with practically no search.

[13]The $\mathcal{AL}_{SP}$ encoding of the protocol is available in the electronic appendix of this paper. For sake of readability, in the encoding we removed the name of the agents from the messages.

```
[massacci@goldrake Chall-Resp]$ ../nice-filter.sh example-lean.lp 2
****** Model Checking example-lean.lp up to 2 steps **********
 - Grounding with lparse
    Original program has 31 rules
    Ground program has 25577 rules
 - Searching for attacks with smodels
******* NO attack found in  1.550 second (after 0 choices)*******
[massacci@goldrake Chall-Resp]$ ../nice-filter.sh example-lean.lp 4
****** Model Checking example-lean.lp up to 4 steps **********
 - Grounding with lparse
    Original program has 31 rules
    Ground program has 50849 rules
 - Searching for attacks with smodels
******* ATTACK found in  3.450 second ******
with 11 choices of which 0 are wrong ones *******
1.  A ---> B : c(shK(A,B),n1)
1.         -> spy : c(shK(A,B),n1)
2.  spy ---> A : c(shK(A,B),n1)
2.         -> A : c(shK(A,B),n1)
3.  A ---> B : n1
3.         -> spy : n1
4.  spy ---> A : c(shK(A,B),n1)
4.         -> A : c(shK(A,B),n1)
4.  A ---> B : n1
4.  spy ---> A : n1
4.         -> A : n1
4.         -> spy : n1
******* The SPY Learned *******
```

Fig. 1. A sample verification session with `smodels`.

With a larger number of steps the attack is still found, but the number of spurious messages may increase. This depends on the search strategy of `smodels`: `smodels` does not search for a minimal stable model, but only for any stable model, so some messages that are not necessary may be inserted. A different ordering of rules in the $\mathcal{AL}_{SP}$ specification leads to different spurious messages.

It is important to notice that the presence of spurious messages does not hinder the ability of finding attacks; it just makes it slightly difficult for the human reader to winnow the chaff and get a minimal error trace. This is a common problem of all model-checking tools.

This minimization process can also be partly automated by rerunning `smodels` with the addition of some constraints that exclude some messages to see whether the attacks continue to materialize or not. In some cases it is possible to impose general constraints (such as the spy should not send a message to an agent if this agent does not receive it, etc.) with the caveat that they may cut also attacks.

10. ADEQUACY OF $\mathcal{AL}_{SP}$ SPECIFICATIONS

An interesting question at this stage is to know how accurate is the $\mathcal{AL}_{SP}$ description of this challenging domain.

Obviously, it is impossible to "prove" that $\mathcal{AL}_{SP}$ specifications are correct descriptions of implemented security protocols. The validity of a modeling and

verification methodology can only be established w.r.t. its usability and its ability to help designers in producing better designs.

Still, it is possible to prove the correctness of specifications w.r.t. other formal models of the same application domain. Here, we focus on the classical Dolev-Yao model of security protocols [Dolev and Yao 1983] which describes protocols as traces of send and receive actions over a network under the control of an active intruder. This is a widely used underlying semantic model adopted by researchers in formal verification [Abadi and Tuttle 1991; Meadows 1994; Lowe 1996; Paulson 1998; Schneider 1998].

We focus here on a class of protocols generalizing *authentication protocols* [Clark and Jacob 1997], which are the current benchmarks of analysis[14] for formal methods in security protocols.

We call them *monotonic* (*parallel-composable*) *protocols*: the intuition is, that to run the protocol, an agent reacts on the basis of what he has received and sent. No action requires him to check that in the course of its many parallel instances he has not sent or not received some other messages. Protocols are monotonic because of practicality: agents (machines) listen to communication ports over the net, and each time a new connection is attempted, a new child process is spawned which runs the protocol over the newly established connection. To boost parallelism and performance, each process minds its own business, and, each time an action is done and acknowledged, it moves to the next state until the protocol is completed.

All protocols in the 1997 reference survey of Clark and Jacob [1997] are of this kind and, to the best of our knowledge, all security protocols analyzed with formal methods up to now. In the sequel, when referring to protocols we always mean a monotonic (parallel-composable) protocol.

10.1 Correctness without an Intruder

At first, we focus on a simple theory without the intruder and show that the inductive modeling of protocols as traces has a tight correspondence with the $\mathcal{AL}_{SP}$ semantical models. In this preliminary setting, everybody follows the protocol.[15]

We assume a common *Herbrand domain of messages* as sketched in Section 6. Here functions are injective, and thus equality boils down to syntactic equality.

We assume a *domain of agent names* and a distinguished domain of *traces of actions* built by the concatenation of the primitive actions $\underline{says}(A, B, M)$, $\underline{gets}(B, M)$, and $\underline{notes}(A, M)$ where A, B are agents and M is a message. We use the underlined version $\underline{says}(A, B, M)$ to denote the action occurring within a trace and $says(A, B, M, \overline{T})$ to denote the fluent in the $\mathcal{AL}_{SP}$ specifications. Many protocols require reasoning about time, and thus we assume a function $\underline{now}(\cdot)$ defined over a trace that identifies what time it is at any point of the trace. We also allow for natural numbers (time) and the primitive relation

[14]The analysis of e-commerce and anonymity protocols is still at a preliminary stage, and it is not clear even what constitutes a good specification [Bella et al. 2000; Meadows and Syverson 1998].
[15]Attacks are still possible because nobody forces the participants to act, and collusions in multi-party protocols are always possible.

"less-than" which allows us to compare numbers with the current time. We assume that time is monotone and discrete. For instance, if $\mathcal{T}$ is a trace and # the concatenation operator on traces, we have that $\underline{now}(\underline{says}(a, b, m)\#\mathcal{T}) = \underline{now}(\mathcal{T}) + 1$, and similarly for the other actions. It is possible to make time continuous, but this complicates the proofs without really improving the modeling capabilities.

Now we are ready to define the protocol model.

Definition 10.1. A *protocol* $\mathcal{P}$ is an inductively defined set of traces, starting from the empty trace, such that all inductive rules are instances of the following schema:
"Let act_{next} be an action; let $\mathcal{A}$ be a set of actions; let $\mathcal{M}$ be the set of (fresh) messages; if the following conditions are satisfied

—$\mathcal{T} \in \mathcal{P}$ and

—for all $act \in \mathcal{A}$, act occurs in $\mathcal{T}$ and

—for all $m \in \mathcal{M}$, m does not occur in $\mathcal{T}$ (i.e. it is fresh) and

—a conjunction of monadic predicates (or time predicates) over $\mathcal{T}$, the objects occuring in the actions in $\mathcal{A}$ and in the action act_{next};

then $act_{next}\#\mathcal{T} \in \mathcal{P}$."

Equality or monadic predicates may specify roles of agents, such as servers or clients, and temporal relations less-than, greater-than may be used for timestamps.

Being inductively defined, traces are finite.

We abuse notation and write $p \leftarrow \wedge_i q_i$ for the rule $p \leftarrow q_1, q_2 \ldots q_n$.

Definition 10.2. Let $\mathcal{P}$ be a protocol. The $\mathcal{AL}_{SP}$ *specification* P *is adequate for* $\mathcal{P}$ if for every rule of the protocol's inductive definition there is a corresponding $\mathcal{AL}_{SP}$ rule of the form

$$\begin{aligned}\{says(a, b, m, T)\} \leftarrow\ & \textstyle\bigwedge_{\underline{says}(a,b_i,m_i)\in\mathcal{A}} \mathsf{said}(a, b_i, m_i, T),\\ & \textstyle\bigwedge_{\underline{gets}(a,m_j)\in\mathcal{A}} \mathsf{got}(a, m_j, T),\\ & \textstyle\bigwedge_{\underline{notes}(a,m_k)\in\mathcal{A}} \mathsf{noted}(a, m_k, T),\\ & \textstyle\bigwedge_{m_l\in\mathcal{M}} \mathsf{fresh}(m_l, T)\\ & \text{other predicates in the inductive rule}\end{aligned}$$

and similarly for $\underline{gets}(a, m)$ and $\underline{notes}(a, m)$. The laws of inertia and effect axioms for $\mathsf{said}(A, B, M, T)$, $\mathsf{got}(A, M, T)$, $\mathsf{noted}(A, M, T)$ are the only other rules for these fluents in P.

Obviously, P includes the rules for defining $\mathsf{fresh}(M, T)$, but we do not need anything else (for instance, $knows(A, M, T)$ is not necessary at this stage).

Then we can prove the following lemma:

LEMMA 10.1. *Let* $\mathcal{P}$ *be a protocol over a ground domain* $\hat{D}$*, and let* P *be an adequate* $\mathcal{AL}_{SP}$ *specification of* $\mathcal{P}$*. Then, for every trace* $\mathcal{T} \in \mathcal{P}$ *there is a stable model of* $P \cup \hat{D}$ *such that for every action* $\underline{says}(a, b, m)$ *in* $\mathcal{T}$ (*respectively,* $\underline{gets}(a, m)$, $\underline{notes}(a, m)$)

(1) *there is a* t *such that* $says(a, b, m, t)$ (*respectively,* $gets(a, m, t)$, $notes(a, m, t)$) *is in the stable model of* P;

(2) *if* $\underline{says}(a', b', m')\#\mathcal{T} \in \mathcal{P}$ (*respectively,* $\underline{gets}(a', m')$, $\underline{notes}(a', m')$) *then there is a* $t' \geq t$ *such that* $says(a', b', m', t')$ (*respectively,* $gets(a', m', t')$, $notes(a', m', t')$) *is in the stable model of* P.

PROOF. The proof is by induction on the length of the trace. Basically, the trace itself is the stable model, and nothing else happens.

It is trivially verified for a trace of length zero: the model in which nothing is sent and nothing is received is always a valid stable model of P (all "says" and "notes" are optional, and all "gets" require a "says").

Suppose that the claim holds for a trace of length l. We take the corresponding stable model, which runs up to time t_l, and add the $(l+1)$th action tagged with $t_l + 1$ by suitably using the fresh constants not yet appearing in any "says" submessage up to time t_l (included). □

The opposite direction is more complex because of parallelism. In practice we need to prove that we can serialize the parallel actions of the protocol while respecting the ordering constraints.

Remark 10.1. If timestamps and lifetimes are used in the protocol, there are obvious cases where the parallel execution of the protocol cannot be serialized in a trace if the granularity of time is 1.

For instance, suppose that we get a "ticket for services" (such as in Kerberos [Schneier 1994]) with a lifetime of 10 units of time. If we are able to make 20 parallel requests of services at each time, we can have 200 requests before the ticket expires. However, once we serialize everything, we can only ask for 10 requests before the expiration time. Thus, we assume that time is discrete, but its granularity can be as small as needed.

LEMMA 10.2. *Let* $\mathcal{P}$ *be a protocol over a finite ground domain* $\hat{D}$*, and let* P *be an adequate* $\mathcal{AL}_{SP}$ *specification of* P*. Then, for every stable model of* $P \cup \hat{D}$ *there is a trace* $\mathcal{T} \in \mathcal{P}$ *such that for every action fluent* $says(a, b, m, t)$ (*respectively,* $gets(a, m, t)$, $notes(a, m, t)$) *true in the stable model of* P *one has*

(1) $\underline{says}(a, b, m)$ (*respectively,* $\underline{gets}(a, m)$, $\underline{notes}(a, m)$) *is in* $\mathcal{T}$;

(2) *if* $says(a', b', m', t')$ *is in the stable model of* P *for some* $t' > t$ (*respectively,* $gets(a', m', t')$ *or* $notes(a', m', t')$) *then* $\mathcal{T}$ *can be decomposed as the concatenation* $\mathcal{T}'\#\underline{says}(a', b', m')\#\mathcal{T}''\#\underline{says}(a, b, m)\#\mathcal{T}'''$ (*respectively,* $\underline{gets}(a', m')$ *or* $\underline{notes}(a', m)$*, etc.*).

provided the $\underline{now}()$ *function is discrete but not necessarily with a unit step.*

PROOF. The proof is constructive: we construct the trace $\mathcal{T}$ out of the stable model. If no "says" action is present in the stable model (and thus no "gets"), we simply take the empty trace.

Suppose that we can construct a valid trace out of a stable model using the actions labeled with the time t, we show how to construct a trace by adding the parallel actions performed at time $t + 1$.

Let $\mathcal{T}_t$ be the trace corresponding to time t. We simply take an arbitrary action true at time $t+1$, for instance $says(a, b, m, t+1)$, and create the trace $\underline{says}(a, b, m)\#\mathcal{T}$. Since the model is stable, there must be some rule to justify the presence of $says(a, b, m, t)$. Thus its preconditions in terms of what is said, noted, and got must be also true in the stable model. For each of those fluents to be true in the stable model, there must be an action predicate also true at a previous time, e.g., if $\mathsf{said}(a, b_i, m_i, t+1)$ is in the stable model there must be an action $says(a, b_i, m_i, t')$ with $t' < t+1$ in the stable model.

By inductive hypothesis there is the corresponding $\underline{says}(a, b_i, m_i)$ already in the trace constructed so far. A similar reasoning can be done for freshness constraints: they are not present in any element of the trace encompassing actions performed before time $t+1$. Then all the preconditions of the inductive rule are satisfied, and the newly composed trace is a valid trace or $\mathcal{P}$.

Then we repeat the process of picking the next action true at time $t+1$ and concatenate it to the trace resulting from the previous step. The reasoning is the same except that now we observe that a fresh object cannot be used in parallel at the same time. Without this additional constraint, we would not be able to "serialize" the actions performed in parallel at time $t+1$: the first "says" action added to the trace might have already used up the wrong objects.

We also need the hypothesis of monotonic (parallel composable) protocols. Consider the simple case of two actions done in parallel at time $t+1$. From the viewpoint of either action the other action was not done yet. After we serialize them, when the "second" action is added to the trace, the "first" action has been already done. With monotonic preconditions this is not a problem, as we have no precondition on what is *not* in a trace.

We must now construct the $\underline{now}(.)$ function. Here we simply construct a function where the increase step is 1 divided by the number of parallel actions at each instant. This is the only place where we need the ground domain to be finite. □

Even without an intruder, we can already define the notion of authentication properties. The definition is borrowed from Schneider [1998] and is substantially equal to Paulson's properties [Bella and Paulson 1998; Paulson 1998].

Definition 10.3. An *authentication property* over a set of primitive actions is a pair of finite sets of actions $\mathcal{C}$ (for Causes) and $\mathcal{E}$ (for Effects). A *protocol* $\mathcal{P}$ *satisfies the authentication property* $\langle \mathcal{C}, \mathcal{E} \rangle$ iff whenever all actions in $\mathcal{E}$ occur in a trace $\mathcal{T} \in \mathcal{P}$ at least one action in $\mathcal{C}$ occurs in $\mathcal{T}$.

Definition 10.4. Let $\langle \mathcal{C}, \mathcal{E} \rangle$ be an authentication property over a domain D and a protocol $\mathcal{P}$, and an $\mathcal{AL}_{SP}$ specification P. The *definition of attack is adequate for* $\mathcal{AL}_{SP}$ if for all time instants t' there is a time $t \geq t'$ in P and a rule of the form

$$\begin{aligned} attack \leftarrow & \bigwedge_{\underline{says}(a,b,m)\in\mathcal{E}} \mathsf{said}(a, b, m, t) \\ & \bigwedge_{\underline{gets}(a,m)\in\mathcal{E}} \mathsf{got}(a, m, t) \\ & \bigwedge_{\underline{notes}(a,m)\in\mathcal{E}} \mathsf{noted}(a, m, t) \\ & \bigwedge_{\underline{gets}(a,m)\in\mathcal{C}} not\ \mathsf{got}(a, m, t) \end{aligned}$$

$$\bigwedge_{\underline{says}(a,b,m)\in\mathcal{C}} not\ \mathsf{said}(a,b,m,t)$$
$$\bigwedge_{\underline{notes}(a,m)\in\mathcal{C}} not\ \mathsf{noted}(a,m,t).$$

No other rule for *attack* is present.

The condition on the time is essential for the final equivalence result (attacks in the protocol models correspond to stable models where *attack* is true) to hold. Otherwise there might be stable models where an attack is indeed present, but, because it took longer than expected, the *attack* atom is not present in the stable model. If time is finite, we only need one rule (for the $t_{max}+1$ instant). If time is infinite we need infinitely many rules.

From the previous two lemmas we get the following theorem:

THEOREM 10.3 (SOUNDNESS). *Let $\mathcal{P}$ be a protocol and $\langle \mathcal{C}, \mathcal{E}\rangle$ an authentication property over a ground domain $\hat{D}$. Let P be an admissible $\mathcal{AL}_{SP}$ specification such that the ground instance of $P \cup \hat{D}$ is adequate for $\mathcal{P}$ and $\langle \mathcal{C}, \mathcal{E}\rangle$. If there is a stable model for the ground instance of $P \cup \hat{D}$ which satisfies the atom attack then there is a trace of the protocol $\mathcal{P}$ that violates the authentication property.*

PROOF. If time is finite and we only have one rule for *attack* (for the largest possible time instant) we use Lemma 10.2 to obtain the trace of $\mathcal{P}$ corresponding to the stable model. It clearly violates the authentication property because the preconditions in the body of the *attack* rule are satisfied.

However, if the model is infinite or if we have more than one rule for *attack* at different time instants, we cannot use Lemma 10.2 to obtain the trace of $\mathcal{P}$: once we unroll the model in a trace it might be, that along the trace, after the attack has taken place, we add all causes, and thus, on the final trace, the authentication property is satisfied. We need to "shorten" the stable model at the right instant.

Since the *attack* atom is true in the model, there is an *attack* rule that fires. Let t be the first time instant such that the rule fires. We eliminate from the stable model of P all actions labeled with a time instant equal to or greater than t. As we have remarked, this can always be done, as we are never forced to act, in $\mathcal{AL}_{SP}$ specifications.

Then, by using a standard compactness argument, we obtain the minimal, finite set of ground rules which is necessary to derive the atom *attack*. From this set of ground rules we obtain a finite subset $\hat{D}_f$ of $\hat{D}$ and use $\hat{D}_f$ for grounding P. Then we still have a stable model of the specification, and to this stable model we can apply Lemma 10.2. □

The opposite direction can also be proven:

THEOREM 10.4 (BOUNDED COMPLETENESS). *Let $\mathcal{P}$ be a protocol and $\langle \mathcal{C}, \mathcal{E}\rangle$ an authentication property over a finite ground domain $\hat{D}$. Let P be an admissible $\mathcal{AL}_{SP}$ specification such that all its ground instances for the time instances $t = 1, 2, 3\ldots$ are adequate for $\mathcal{P}$. If there are (possibly parallel) runs of the protocol which violate the authentication properties of $\mathcal{P}$ with length at most t_{max} then there is a stable model of the ground instance for $t = t_{max}+1$.*

10.2 Formal Analysis with the Intruder

In the Dolev-Yao model, the intruder is able to read traffic not directed to him and to send messages without respecting the protocol. Therefore we must be able to reason about its abilities to manipulate messages (the "knows" and "synth" predicates from Section 6).

As for *operators on messages*, we only need Paulson's inductively defined operators on set of messages: $\underline{analz}$, $\underline{synth}$, and $\underline{parts}$ [Paulson 1998]. Schneider's and Lowe's operator $\vdash$ for their CSP handling of messages can be reconstructed in terms of those three basic operators. Thus we need to show how these inductive operators can be reconstructed by $\mathcal{AL}_{SP}$ specifications:

Definition 10.5. Let $\underline{pSet}(\vec{t})$ be an inductively defined set of messages for some elements $\vec{t}$ of the domain $\hat{D}$ such that

—for the base case $m \in \underline{pSet}(\vec{t})$ if the condition on m and $\vec{t}$ is a conjunction of monadic predicates, equality, or predicates not depending on $\underline{pSet}(\vec{t})$;

—for the inductive cases all inductive rules have the form "if $m_1 \in \underline{pSet}(\vec{t})$ and $\ldots$ and $m_n \in \underline{pSet}(\vec{t})$ then $m \in \underline{pSet}(\vec{t})$."

Then the following *$\mathcal{AL}_{SP}$ specification is adequate for* $\underline{pSet}(\vec{t})$ if

—for the base case there is a rule of the form

$$\mathsf{pSet}(\vec{t}, m, T) \leftarrow \mathsf{time}(T), \mathsf{msg}(m), \text{monadic and inequality predicates defining } \underline{pSet}(\ ), \text{domain predicates for } \vec{t}$$

—for every inductive rule there is an $\mathcal{AL}_{SP}$ rule of the form

$$\mathsf{pSet}(\vec{t}, m, T) \leftarrow \mathsf{time}(T), \mathsf{msg}(m), \mathsf{msg}(m_1), \ldots \mathsf{msg}(m_n), \mathsf{pSet}(\vec{t}, m_1, T), \ldots, \mathsf{pSet}(\vec{t}, m_n, T), \text{domain predicates for } \vec{t}.$$

—There is no other rule for $\mathsf{pSet}(\vec{t}, m, T)$ which recursively depends on $\mathsf{pSet}(\vec{t}, m', T)$ for some m'.

LEMMA 10.5. *Let P be an $\mathcal{AL}_{SP}$ specification adequate for $\underline{pSet}(\vec{t})$ over a domain $\hat{D}$, $\mathcal{M}$ a set of messages over $\hat{D}$ satisfying the base condition for $\underline{pSet}(\ )$. For every stable model of P, elements $\vec{t}$, and time t_0, if $m \in \mathcal{M}$ iff $\mathsf{pSet}(\vec{t}, m, t_0)$ is in the stable model then $m' \in \underline{pSet}(\vec{t})$ iff $\mathsf{pSet}(\vec{t}, m', t_0)$ is in the stable model.*

In one direction the proof is by simple induction on the construction of $\underline{pSet}(\vec{t})$. For the other direction we exploit the fact that $\mathcal{AL}_{SP}$ models are stable and that there is no other rule for getting $\mathsf{pSet}(\vec{t}, m, t_0)$ in the model.

As for *operators on traces*, we need to define an adequate representation of inductively defined operators of traces of atomic messages:

Definition 10.6. Let $\underline{pTrace}(\mathcal{T})$ be an inductively defined set of messages for some elements $\vec{t}$ of the domain $\hat{D}$ and a trace $\mathcal{T}$ of atomic actions such that

$$\underline{pTrace}(\vec{t},[]) = \underline{pSet}_0(\vec{t})$$

$$\underline{pTrace}(\vec{t},\underline{says}(a,b,m)\#\mathcal{T}) = \begin{cases} \underline{pSet}_1(\vec{t},m)\cup\underline{pTrace}(\vec{t},\mathcal{T}) & \text{if } c_1(a,b,m)\\ \vdots & \\ \underline{pSet}_k(\vec{t},m)\cup\underline{pTrace}(\vec{t},\mathcal{T}) & \text{if } c_k(a,b,m)\\ \underline{pTrace}(\vec{t},\mathcal{T}) & \text{otherwise}\end{cases}$$

where all $\underline{pSet}_i(\vec{t})$ are inductively defined set-predicates according to Definition 10.5 and $c_i(a,b,m)$ is a conjunction of (in)equality predicates or other set-predicates, and similarly for $\underline{notes}(a,m)$ and $\underline{gets}(a,m)$.

The *$\mathcal{AL}_{SP}$ specification is adequate for* $\underline{pTrace}(\vec{t},\cdot)$ if it is adequate for $\underline{pSet}_i(\vec{t})$ for $i=0,1\ldots$ and for every rule for $\underline{pTrace}(\vec{t})$ there is an $\mathcal{AL}_{SP}$ rule of the form

$$\begin{array}{l}\mathsf{pTrace}(\vec{t},m',0)\leftarrow \mathsf{pSet}_0(m',\vec{t})\\ \mathsf{pTrace}(\vec{t},m',T+1)\leftarrow says(a,b,m,T),c_1(a,b,m),\mathsf{pSet}_1(\vec{t},m,m',T)\\ \ldots\\ \mathsf{pTrace}(\vec{t},m',T+1)\leftarrow says(a,b,m,T),c_k(a,b,m),\mathsf{pSet}_k(\vec{t},m,m',T)\\ \mathsf{pTrace}(\vec{t},m',T+1)\leftarrow \mathsf{pTrace}(\vec{t},m,m',T)\end{array}$$

where domain predicates for all variables are omitted for readability. This occurs similarly for $\underline{notes}(a,m)$ and $\underline{gets}(a,m)$. Moreover, there is no other rule for $\mathsf{pTrace}(\vec{t},m',T)$.

Typically one has $p^1=\ldots=p^k$. For example, Paulson uses $init(A)$ as the set of messages initially known to agent A and $\underline{parts}(M)$ as the set of submessages of message m and then defines the predicate $\underline{used}$ to be the following:

$$\begin{array}{ll}\underline{used}([]) & = \cup_{A\in Agents} init(A)\\ \underline{used}(\underline{says}(a,b,m)\#\mathcal{T}) & = \underline{parts}(m)\cup\underline{used}(\mathcal{T})\\ \underline{used}(\underline{gets}(a,m)\#\mathcal{T}) & = \underline{parts}(m)\cup\underline{used}(\mathcal{T})\\ \underline{used}(\underline{notes}(a,m)\#\mathcal{T}) & = \underline{parts}(m)\cup\underline{used}(\mathcal{T})\end{array}$$

Bella's inductive operator $\underline{knows}$ is the following:

$$\underline{knows}(A,[]) = init(A)$$

$$\underline{knows}(A,\underline{says}(a,b,m)\#\mathcal{T}) = \begin{cases}\{m\}\cup\underline{knows}(A,\mathcal{T}) & \text{if } A=a\\ \{m\}\cup\underline{knows}(A,\mathcal{T}) & \text{if } A=spy\\ \underline{knows}(A,\mathcal{T}) & \text{otherwise}\end{cases}$$

$$\underline{knows}(A,\underline{gets}(a,m)\#\mathcal{T}) = \begin{cases}\{m\}\cup\underline{knows}(A,\mathcal{T}) & \text{if } A=a\\ \underline{knows}(A,\mathcal{T}) & \text{otherwise}\end{cases}$$

$$\underline{knows}(A,\underline{notes}(a,m)\#\mathcal{T}) = \begin{cases}\{m\}\cup\underline{knows}(A,\mathcal{T}) & \text{if } A=a\\ \{m\}\cup\underline{knows}(A,\mathcal{T}) & \text{if } compromised(a)\\ & \text{and } A=spy\\ \underline{knows}(A,\mathcal{T}) & \text{otherwise}\end{cases}$$

The "knowledge of the intruder" is then defined by the combination of the predicates $\underline{synth}(\underline{analz}(\underline{knows}(spy,\mathcal{T})))$. This defines the set of messages that the spy can compose after decrypting all messages she has intercepted in the traffic. These combinations can be simply captured by the obvious concatenation of rules.

The corresponding adequate $\mathcal{AL}_{SP}$ specifications are given in Section 6 for the action predicates $knows(A, M, T)$ and $\mathsf{used}(M, T)$. In our case, since we have no independent use of $\underline{analz}$, but we (and indeed any formal approach to security protocols) only use it in combination with $\underline{knows}$, we have merged both of them into our only predicate $knows(A, M, T)$.

Finally we can upgrade the definition of protocols given in Definition 10.1 by allowing two additional inductive rules:

—if $\mathcal{T} \in \mathcal{P}$ and $m \in \underline{pSet}(\underline{pTrace}(spy, \mathcal{T}))$ then $\underline{says}(spy, a, m)\#\mathcal{T} \in \mathcal{P}$.
—if $\mathcal{T} \in \mathcal{P}$ and $\underline{says}(a, b, m)$ occurs in $\mathcal{T}$ then $\underline{gets}(spy, m)\#\mathcal{T} \in \mathcal{P}$.

Obviously $\underline{pSet}(\cdot)$ and $\underline{pTrace}(\cdot)$ depend on the particular model and abilities of the intruder. Almost all models incorporate syntactic variants of the combination of the predicates $\underline{synth}(\underline{analz}(\underline{knows}(spy, \mathcal{T})))$.

We can now prove the equivalent version of Lemma 10.5 for the trace property $\underline{pTrace}(\cdot)$. The proof is again by induction on the construction of $\mathcal{T}$ where the base case exploits that $\underline{pSet}_0(\cdot)$'s encoding in $\mathcal{AL}_{SP}$ is adequate. Notice that for this proof to go through we must define each $\mathsf{pSet}_i(\cdot)$ separately for each $\mathsf{pTrace}(\cdot)$. It is obviously possible to "recycle" the same definition of $\mathsf{pSet}_i(\cdot)$ for different trace functions, but then the proof must be redone on a case-by-case basis.

Then, one can prove the equivalent of Lemma 10.2 and Lemma 10.1 for the upgraded protocol model with the intruder. These results can be combined, with the same compactness argument used in Theorem 10.3, yielding the final theorems:

THEOREM 10.6 (SOUNDNESS WITH INTRUDERS). *Let $\mathcal{PI}$ be a protocol augmented with an intruder and $\langle \mathcal{C}, \mathcal{E} \rangle$ an authentication property over a ground domain $\hat{D}$. Let P be an admissible $\mathcal{AL}_{SP}$ specification such that the ground instance of $P \cup \hat{D}$ is adequate for $\mathcal{P}$ and $\langle \mathcal{C}, \mathcal{E} \rangle$. If there is a stable model for the ground instance of $P \cup \hat{D}$ which satisfies the atom attack then there is a trace of the protocol $\mathcal{PI}$ that violates the authentication property.*

Clearly the actions which lead to the attack are identified by the action predicates ($says(A, B, M, T)$, $gets(B, M, T)$, $notes(A, M, T)$) that are true in the stable model. This is also true when the stable model is infinite, e.g., if we allow for an infinite number of agents and nonces or do not set a bound on the maximum length of possible runs.

THEOREM 10.7 (BOUNDED COMPLETENESS WITH INTRUDERS). *Let $\mathcal{PI}$ be a protocol augmented with an intruder and $\langle \mathcal{C}, \mathcal{E} \rangle$ an authentication property over a finite ground domain $\hat{D}$. Let P be an admissible $\mathcal{AL}_{SP}$ specification such that all its ground instances for the time instances $t = 1, 2, 3 \ldots$ are adequate for $\mathcal{P}$. If there are (possibly parallel) runs of the protocol which violates the authentication properties of $\mathcal{P}$ of length at most t_{max} then there is a stable model of the ground instance for $t = t_{max} + 1$ which contains the atom attack*

Remark 10.2. For e-commerce and fair-exchange protocols the presence of explicit parallel actions, rather than their interleaving semantics, makes a difference. The so called "optimistic protocols" introduced by N. Asokan et al.

[1997] are subject to parallel attacks that can be modeled in $\mathcal{AL}_{SP}$ but not with a trace-based model.

11. RELATED WORKS AND CONCLUSIONS

Throughout the paper we have referred to the differences with some of the state-of-the-art approaches for protocol verification which have been automated. Here we just summarize the main differences.

The NRL Protocol Analyzer (NPA) by Meadows [1994] shares with us the choice of the programming paradigm, as we both use logic programs. A key difference is that we use the logic programming language $\mathcal{AL}_{SP}$ as specification language, whereas Prolog is used as the implementation language for NPA [Meadows 1994]. The protocol description and the specifications for NPA are based on state variables and rules for changing state variables with an explicit modeling of the words learned by the intruder. This aspect of NPA is closer to state exploration tools such as Murphi [Mitchell et al. 1997].

Security specifications, whose violation may lead to an attack, must be written in a different language either with temporal operators as done by Syverson and Meadows [1996] or by using the CAPSL intermediate language [Brackin et al. 1999]. Such specifications are declarative but not executable [Syverson and Meadows 1996].

Our current formalization does not cope with an infinite search space, which can be treated by NPA by letting the user introduce some inductive lemmas. Infinite state space (such as an infinite number of agents or nonces) can be modeled in our approach by minor modifications (just add a rule also for agents and nonces indexed by time instants), but the price to pay is that we would lose the finiteness theorem. Another possibility would be to use iterative deepening on t_{max} and the number of basic objects, as this allows us to retain the benefits of the bounded model-checking completeness. We believe, that w.r.t. other model-checking approaches, the use of a declarative specification language greatly simplifies the presentation of actions and events.

The work on multiset rewriting proposed by Cervesato et al. [1999] extends the semantics of NPA by defining protocols in terms of rewriting rules over sets of messages known by the participating agents. Multiset rewriting turned out to be a useful tool for theoretical analysis for relating various formalisms [Cervesato et al. 2000b; Cervesato et al. 2000a]. However, as a specification language, multiset rewriting turned out to be unfeasible, and the authors had to revert to an intermediate specification such as CIL and then ultimately to CAPSL [Brackin et al. 1999].

As for the approaches based on theorem proving, we have already pointed out connections between our proposal and Paulson's inductive method [Paulson 1998; Bella and Paulson 1998]. Indeed, we have in common the operational semantics for the specification of protocols. In the inductive method one models a protocol as a set of traces and then uses interactive theorem proving to prove that the protocol is secure, i.e., prove that *all* traces satisfy a desired guarantee. Paulson's theory is richer than ours (as he uses set theory, functions, inductive definitions, etc.), as his objective is theorem proving about all traces. The limit

is that inductive theorem proving is interactive and requires expert knowledge, even if current tools substantially help in shortening the verification efforts. Our approach is based on model finding, and thus we look for *one* trace that satisfies a given property, i.e., a security violation. Thus, by combining techniques from logic programming and planning, we can fully automatize the search for attacks.

$\mathcal{AL}_{SP}$ combines these various approaches into a coherent framework. Our formal language allows for a declarative description of the properties of protocols and their operational behavior in term of traces. The use of stable model semantics makes it possible to use powerful tools such as `smodels` for the automatic verification of the protocol.

$\mathcal{AL}_{SP}$ is a good blending of the three contrasting needs: being close to the description of protocols as specified in the security literature, specifying security properties at a high level of abstraction, automating the analysis of the protocols, and the search for bugs (i.e., security attacks). Gollmann [1996, page 53] writes

> High level definitions of entity authentication may obscure the precise goals an authentication protocol should achieve. On the other hand, a low level description of the cryptographic mechanisms employed in the protocol may obscure their intended purpose.

We believe that our specification language $\mathcal{AL}_{SP}$ is a step toward making these ends meet.

To ease comparison and integration with other approaches, a translator from CASPER specifications [Lowe 1998] into $\mathcal{AL}_{SP}$ specifications has also been implemented [Lorenzon 2000]. This makes it possible to obtain $\mathcal{AL}_{SP}$ specifications for all protocols described in CASPER in an almost[16] automatic way.

We refer to Carlucci Aiello and Massacci [2001] for a detailed presentation of case studies in $\mathcal{AL}_{SP}$: the Needham-Schroeder protocol, and the Aziz-Diffie key agreement protocol for mobile communication. We plan to extend our verification methodology to more complex protocols such as SET by Visa and MasterCard and test to what extent, in terms of the size of specifications, we can use only general-purpose tools such as `smodels` for verifying specifications written in $\mathcal{AL}_{SP}$.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/tocl/2001-2-4/p542-carlucci-aiello`.

ACKNOWLEDGMENT

We dedicate this paper to Bob Kowalski, on his 60th birthday, for his seminal contributions to computational logic.

We thank P. Baumgartner and U. Furbach for many useful comments on an earlier formalization of this work, I. Niemelä and T. Syrjänen for support

[16]The wording almost is due because some CASPER specifications allow for the insertion of CSP code in the specifications. These CSP hacks must of course be translated by hand into $\mathcal{AL}_{SP}$ hacks.

in using smodels. We are grateful to K. Apt and the anonymous referees for suggestions that improved the presentation.

REFERENCES

ABADI, M. AND TUTTLE, M. 1991. A semantics for a logic of authentication. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC'91)*. ACM Press and Addison Wesley, 201–216.

APT, K. 1990. Logic programming. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier Science Publishers (North-Holland), Amsterdam.

ASOKAN, N., SCHUNTER, M., AND M., W. 1997. Optimistic protocols for fair exchange. In *Proceedings of the 4th ACM Conference on Communications and Computer Security (CCS'97)*. ACM Press and Addison Wesley, 7–17.

BELLA, G., MASSACCI, F., PAULSON, L., AND PIERO, T. 2000. Formal verification of Card-Holder Registration in SET. In *Proceedings of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, F. Cuppens, Y. Deswarte, and D. Gollmann, Eds. Lecture Notes in Computer Science, vol. 1895. Springer-Verlag, 159–174.

BELLA, G. AND PAULSON, L. C. 1998. Kerberos version IV: Inductive analysis of the secrecy goals. In *Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS'98)*. Lecture Notes in Computer Science, vol. 1485. Springer-Verlag, 361–375.

BRACKIN, S., MEADOWS, C., AND MILLEN, J. 1999. CAPSL interface for the NRL Protocol Analyzer. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET-99)*. IEEE Computer Society Press.

BURROWS, M., ABADI, M., AND NEEDHAM, R. 1990. A logic for authentication. *ACM Trans. Comput. Syst. 8,* 1, 18–36.

CARLUCCI AIELLO, L. AND MASSACCI, F. 2000. An executable specification language for planning attacks to security protocols. In *IEEE Computer Security Foundation Workshop*, P. Syverson, Ed. IEEE Computer Society Press, 88–103.

CARLUCCI AIELLO, L. AND MASSACCI, F. 2001. Planning attacks to security protocols: case studies in logic programming. To appear in F. Sadri and A. Kakas eds. *Computational Logic: From Logic Programming into the Future*, Springer Verlag.

CERVESATO, I., DURGIN, N., KANOVICH, M., AND SCEDROV, A. 2000. Interpreting strands in linear logic. In *Workshop on Formal Methods and Computer Security*, E. Clarke and N. Heinze, Eds. CAV-2000 Satellite Workshop, Chicago.

CERVESATO, I., DURGIN, N., LINCOLN, P., MITCHELL, J., AND SCEDROV, A. 1999. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW 1999)*, P. Syverson, Ed. IEEE Computer Society Press, 55–69.

CERVESATO, I., DURGIN, N., LINCOLN, P., MITCHELL, J., AND SCEDROV, A. 2000. Relating strands and multiset rewriting for security protocol analysis. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW 2000)*, P. Syverson, Ed. IEEE Computer Society Press, 35–51.

CLARK, J. AND JACOB, J. 1997. A survey of authentication protocol literature: Version 1.0. Tech. Rep., University of York, Department of Computer Science. November. Available on the Web at http://www-users.cs.york.ac.uk/~jac/. A complete specification of the Clark-Jacob library in CAPSL [Brackin et al. 1999] is available at http://www.cs.sri.com/~millen/capsl/.

CLARKE, E., JHA, S., AND MARRERO, W. 1998. A machine checkable logic of knowledge for specifying security properties of electronic commerce protocols. In *LICS Workshop on Formal Methods and Security protocols*. LICS. Also available as Technical Report CMU-SCS-97-139, Carnegie Mellon University, May 1997.

DIMOPOULOS, Y., NEBEL, B., AND KOEHLER, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning (ECP'97)*, S. Steel and R. Alami, Eds. Lecture Notes in Artificial Intelligence, vol. 1348. Springer-Verlag, 169–181.

DOLEV, D. AND YAO, A. 1983. On the security of public key protocols. *IEEE Trans. Inf. Theory IT-30,* 2, 198–208.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP'88)*, R. Kowalski and K. Bowen, Eds. MIT-Press, 1070–1080.

GOLLMANN, D. 1996. What do we mean by entity authentication? In *Proceedings of the 15th IEEE Symposium on Security and Privacy (SSP'96)*. IEEE Computer Society Press, 46–54.

KAUTZ, H. AND SELMAN, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)*. John Wiley & Sons, 359–363.

LORENZON, L. 2000. Un traduttore da CASPER ad $\mathcal{AL}_{SP}$. M.S. Thesis, Facoltà di Ingegneria, Univ. di Roma I "La Sapienza".

LOWE, G. 1996. Some new attacks upon security protocols. In *Proceedings of the Ninth IEEE Computer Security Foundations Workshop (CSFW'96)*. IEEE Computer Society Press, 162–169.

LOWE, G. 1997. A hierarchy of authentication specifications. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW'96)*. IEEE Computer Society Press, 31–43.

LOWE, G. 1998. Casper: A compiler for the analysis of security protocols. *J. Comput. Security 6,* 18-30, 53–84.

MCCARTHY, J. AND HAYES, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, B. Meltzer and D. Michie, Eds. Vol. 4. Edinburgh University Press, Edinburgh, Scotland, 463–502.

MEADOWS, C. 1994. The NRL Protocol Analyzer: An overview. *J. Logic Program. 26,* 2, 113–131.

MEADOWS, C. AND SYVERSON, P. 1998. A formal specification of requirements for payment transactions in the SET protocol. In *Proceedings of Financial Cryptography 98*, R. Hirschfeld, Ed. Lecture Notes in Computer Science, vol. 1465. Springer-Verlag.

MITCHELL, J., MITCHELL, M., AND STERN, U. 1997. Automated analysis of cryptographic protocols using Murphi. In *Proceedings of the 16th IEEE Symposium on Security and Privacy (SSP'97)*. IEEE Computer Society Press, 141–151.

NIEMELÄ, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals Math. Artif. Intell. 25,* 3-4, 241–273.

NIEMELÄ, I. AND SIMMONS, P. 1997. Smodels – an implementation of stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*. Lecture Notes in Artificial Intelligence, vol. 1265. Springer-Verlag, 420–429.

PAULSON, L. C. 1998. The inductive approach to verifying cryptographic protocols. *J. Comput. Security 6*, 85–128.

REITER, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. Academic Press, 359–380.

RYAN, P. AND SCHNEIDER, S. 1998. An attack on a recursive authentication protocol: A cautionary tale. *Inf. Proces.s Lett. 65,* 15, 7–16.

SCHNEIDER, S. 1998. Verifying authentication protocols in CSP. *IEEE Trans. Softw. Eng. 24,* 9, 741–758.

SCHNEIER, B. 1994. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons.

SYVERSON, P. AND MEADOWS, C. 1996. A formal language for cryptographic protocol requirement. *Designs, Codes Cryptogr. 7*, 27–59.

SYVERSON, P. F. AND VAN OORSCHOT, P. C. 1994. On unifying some cryptographic protocols logics. In *Proceedings of the 13th IEEE Symposium on Security and Privacy (SSP'94)*. IEEE Computer Society Press.

WELD, D. S. 1999. Recent advances in AI planning. *Artif. Intell. Mag. 20,* 2, Summer 1999, 93–123.

Received August 2000; revised January 2001 and April 2001; accepted April 2001