On the Representation and Verification of Cryptographic Protocols in a Theory of Action

No Author Given

No Institute Given

Abstract Cryptographic protocols are usually specified in an informal, ad hoc language, with crucial elements, such as the protocol goal, left implicit. We suggest that this is one reason that such protocols are difficult to analyse, and are subject to subtle and nonintuitive attacks. We present an approach for formalising and analysing cryptographic protocols in the situation calculus. We provide a declarative specification of underlying assumptions and capabilities in the situation calculus. A protocol is translated into a sequence of actions to be executed by the principals, and a successful attack is an executable plan by an intruder that compromises the specified goal. Our prototype verification software takes a protocol specification, translates it into a high-level situation calculus (Golog) program, and outputs any attacks that can be found. We describe the structure and operation of our prototype software, and discuss performance issues.

1 Introduction

A cryptographic protocol is a formalised sequence of messages exchanged between agents, where parts of the messages are protected using encryption. These protocols are used for many purposes, including authentication and secure information exchange. Protocols are typically specified in a quasi-formal language, as illustrated in the following example:

The Challenge-Response Protocol

1. $A \rightarrow B : \{N_A\}_{K_{AB}}$
2. $B \rightarrow A : N_A$

The goal is for agent A to determine whether B is alive on the network. The first step is for A to send B the message N_A encrypted with a shared key K_{AB}. N_A is a *nonce*, a random number assumed to be new to the network. The second step is for B to send A the message N_A unencrypted. The claim is that since only A and B have K_{AB}, and assuming that K_{AB} is indeed secure, then N_A could only have been decrypted by B, and so B must be alive. However, the protocol is flawed; here is an attack in which an intruder I masquerades as B and initiates a round of the protocol with A, thereby obtaining the decrypted nonce:

An Attack on the Challenge-Response Protocol

1. $A \rightarrow I_B : \{N_A\}_{K_{AB}}$
1.1 $I_B \rightarrow A : \{N_A\}_{K_{AB}}$
1.2 $A \rightarrow I_B : N_A$
2. $I_B \rightarrow A : N_A$

This example is simplistic, but it illustrates the type of problems that arise in protocol

verification. Even though protocols are generally short, they are notoriously difficult to prove correct. As a result, many different formal approaches have been developed for protocol verification. However, often these approaches are difficult to apply for anyone other than the original developers [3]. Part of the problem is that there is no clear agreement on what exactly constitutes an attack [1], which leaves one with considerable ambiguity about the status of a protocol when no attack is found. Moreover, as we later discuss, the language for specifying a protocol is highly ambiguous, and much information is left implicit.

Our thesis is that all aspects of a protocol need to be explicitly specified. The main contribution of this paper is the introduction of a declarative theory for protocol specification, including message passing between agents on a possibly compromised network, expressed in terms of a situation calculus (SitCalc) theory. A protocol is *compiled* into a set sequence of actions for agents to execute. These actions may be interleaved with others, and indeed the framework allows simultaneous runnings of multiple protocols. The intention of an intruder is to construct a *plan* such that the goal of the protocol, in a precise sense, is thwarted. A protocol is secure when no such plan is possible. The approach is flexible, and significantly more general than previous approaches. We have used this approach to implement a verification tool that finds attacks by translating protocol specifications into Golog programs.

2 Motivation

2.1 Background

The standard intruder model used in cryptographic protocol verification is the so-called Dolev-Yao intruder [5]. Informally, the intruder can read, block, intercept, or forward any message sent by an honest agent. Verification consists of encoding the structure of a protocol in an appropriate formalism, and then finding attacks that a Dolev-Yao intruder is able to perform. Many different formalisms have been explored, including epistemic logics [4], multi-agent systems [6], strand spaces [12,7], multi-set re-writing formalisms [2] and logic programs under the stable model semantics [1].

Most existing work in protocol verification relies on the same semi-formal specification of protocols that we used in the introduction. Consider the Challenge-Response protocol and the attack described previously. Several points may be noted about the protocol specification. First, while the intent of the protocol and the attack are intuitively clear, the meaning of the exchanges in the protocol are ambiguous. Consider the first line of the protocol: it cannot mean that A sends a message to B, since this may not be the case, as the attack illustrates. Nor can it mean that A *intends* to send a message to B, because in the attack it certainly isn't A's intention to send the message to the intruder! As well, in the first line of the protocol there is more than one action taking place, since in some fashion A is involved in sending a message and B is involved in the receipt of a message. Hence, the specification language is imprecise and ambiguous; notions of agent communication should be made explicit.

The specification leaves crucial notions unstated, including: the goal of the protocol, the fact that N_A is a freshly generated nonce, and the capabilities of the intruder.

Moreover, the specification does not take into account the broader context in which the protocol is to be executed. This context might contain other agents, interleaved protocol runs, and even constraints on appropriate behaviour for honest agents. For example, it is quite possible that a protocol could fail via what might be called a "stupidity attack", in which A simply sends the unencrypted nonce to the intruder. Certainly this would not be an expected outcome, but it is a logically possible compromise of the protocol. It may well be that there are other "untoward happenings" significantly more subtle than the stupidity attack; consequently, it is desirable to have a framework for specifying protocols in a general enough fashion in which such possibilities may be explicitly taken into account.

2.2 A Declarative Approach

We argue that in order to provide a robust demonstration of the security and correctness of a protocol, all aspects of a protocol exchange need to be specified. We suggest that an explicit, logical formalisation in the SitCalc provides a suitable framework. Our primary aim is to clearly formalize exactly what is going on in a cryptographic protocol in a declarative action formalism; such a formalization will provide a more nuanced and flexible model of agent communication.

We proceed as follows. The first step is to develop a suitable formal model the message passing environment. In past work, we have used a simple transition system representation [9]. While this is sufficient for a the high-level analysis of a protocol in terms of the beliefs of the agents, it is not expressive enough to precisely encode all aspects of a message passing system. In§3, we develop a SitCalc model of the message passing environment in which agents, messages, and keys are all first-class objects that are composed, manipulated and exchanged through a small set of actions.

Our goal is to translate the standard "arrows and colons" representation of a protocol directly into an executable GoLog program that corresponds to our SitCalc theory of message passing. As such, in §4, we demonstrate how a simple syntactic variation on a standard protocol specification can be translated into a GoLog procedure that precisely encodes the messages that are sent and received in a protocol specification. Roughly, the idea is to iterate through all components of each message and translate each message exchange into a detailed procedure for composing, sending, receiving, and decomposing each message.

In §5, we present a high-level version of our verification algorithm. We use an iterative deepening approach to consider progressively longer sequences of message exchanges that encode a complete protocol run. We assume that every message is first received and decomposed by the intruder, who then has all components of the message for use in an attack. Although our model is highly expressive and the intruder space of possible actions for the intruder grows quickly, we find that attacks are still found in a reasonable amount of time for many standard test protocols.

3 The Message Passing Domain

While our approach is to translate protocol specifications directly into GoLog, underlying our approach is a formal SitCalc model of the message passing domain. In this

section, we briefly outline the details of our model using the Challenge-Response protocol as an example. We assume familiarity with the situation calculus [10].

3.1 Vocabulary

In our model, there are four primitive sorts of objects (beyond actions and situations): *agents, nonces, keys,* and *encrypted data*.

- **Agents**: Agent predicates include **Agent(a)**, **Intruder(a)** and **Principal(a)**. By default, the domain only has the lone intruder *intr* and two principals: *Alice*, and *Bob*.
- **Nonces:** For nonces, there is the candidacy predicate **Nonce(n)** and the functional fluent **fresh(s)** that generates a new unique nonce.
- **Keys: Key(k)** is true for any key, and there are predicates to distinguish symmetric keys, public keys, and private keys: **SymKey(k)**, **PubKey(k)**, and **PrivKey(k)**. There are also functional fluents to encode the relationship between key pairs.
- **Encrypted Data** is text encoded with a specific key, fulfilling **Encrypted(enc)**. The *decKey* function can be applied as follows **decKey(**$\{Bob\}_{K_{Alice}}$**)** to garner the correct key: KP_{Alice}. **encKey** is similar, except it extracts the key used for encryption rather than decryption. Another extraction fluent is **dec(encrypted,k)**, which returns internal contents of *encrypted*, assuming that *k* is the correct key. To convert raw data and a key into the correct encoded object, use the function **enc(text,k)**.

Any list of the above objects is called *text*, designated by the corresponding candidacy predicate **Text(t)**. There are several functions for working with text, such as the length function **Length(text)** and the extraction functions **First(lst)**,**Second(lst)**, **Third(lst)**, and so on.

A *message* is composed of text contents, a sender address, and a recipient address. For message fluents, there is the basic candidacy predicate **Msg(m)** along with the extraction functions **SenderAddr(m)**, **RecipAddr(m)**, and **Contents(m)**. To determine whether or not a message has been posted for access, use the fluent **Sent(msg, s)**. If true, then it is possible for another agent to receive this message. To check whether a message was received by an agent, evaluate **Recd(a, msg, s)**. This fluent is true only if the agent has received that message at least once. In the domain, we assume that any sent message can only be received once. Implicitly, this means that if a message *msg* is sent *n* times, that message can be received at most *n* times.

The notion of *having* a message or a key is fundamental in protocol verification. An agent's ownership of data fundamentally changes what text agents can send or encrypt, and this is represented in our model through the ownership fluent **Has(agent,object,s)**. We will see that the truth of such expressions changes as actions are executed. The **Has** fluent also has an expression for defining the ownership of lists. For example, if $[l_1, l_2, \ldots]$ is some ordered list, then:

$$Has(agent, [l_1, l_2, \ldots], s) \leftrightarrow Has(agent, l_1, s) \wedge Has(agent, l_2, s) \wedge \ldots$$

Equivalently, if an agent has a list of objects then it owns each object in the list and an agent owns a list of objects if it has each of the individual objects in the list.

3.2 The Initial Situation

In our model, it is necessary to explicitly specify which keys and text strings are owned by each agent at the outset. Consider a domain with three agents: the intruder *intr*, and two principals *Alice* and *Bob*. We can model a situation where all agents have a shared key as follows:

Agent	Owned Agent IDs	Owned Shared Keys
Alice	Alice, Bob, intr	$K_{Alice/Bob}$, $K_{Alice/intr}$
Bob	Alice, Bob, intr	$K_{Alice/Bob}$, $K_{Bob/intr}$
intr	Alice, Bob, intr	$K_{Alice/intr}$, $K_{Bob/intr}$

This table indicates that all pairs of agents share a key, and each agent is able to compose messages that involve the names of the other agents. As agents compose and exchange messages, each agent will acquire ownership of more information. It is also possible to encode non-standard initial situations; for example, it is straightforward to specify that a certain key has been compromised before a protocol run begins. This flexibility in changing the initial ownership of information is an advantage of our approach.

3.3 Message Passing Actions

Using the constructed initial situation and the language of logical fluents, we can then define both the preconditions and postconditions of performing nonce generation, encryption, composition, and message sends/receive. For simplicity, general type predicates, such as **Agent(a)**, are always part of the preconditions but are not listed in the following definitions:

1. **makeNonce(a,n,s)**
   PRECONDITIONS: IsFresh(n,s)
   EFFECT: Has(n,s)
2. **encrypt(a,text,k,s)**
   PRECONDITIONS: Has(a,text,s) $\wedge$ ($\exists$ k Pubkey(k,s) $\vee$ Has(a,k,s))
   EFFECT: Has(a, enc(text,k),s)
3. **decrypt(a, enc-text, k,s)**
   PRECONDITIONS:
   Has(a,enc-text,s) $\wedge$ Has(a,k,s)
   $\wedge$ ((SymKey(k, s) $\wedge$ (k = encKey(enc-text))) $\vee$ (AsymKey(k, encKey(enc-text),s)))
   EFFECT: Has(a,dec(enc-text,k),s)
4. **compose(a,m,se,re,t)**
   This action allows for an agent to package text into a message with a **se**nder and **re**cipient.
   PRECONDITIONS: Has(a,t,s)
   EFFECT: Has(make-message(se,re,t),s)
5. **send(a,m)**
   PRECONDITIONS:
   Has(a,m,s) $\wedge$ ((a = intr) $\vee$ ((a = SenderAddr(m)) $\wedge$ (a $\neq$ RecipAddr(m))
   The precondition ensures that principals can only send messages where they are the *Sender*.
   EFFECT: Sent(m,s)

6. **receive(a,m)**
   PRECONDITIONS:
   Sent(m,s) $\wedge$ ((= a intr) $\vee$ (= a RecipAddr(m))))
   Principals can only receive a message if it is addressed to them.
   EFFECTS: Has(a,Contents(m),s), Recd(a,m), a Sent(msg,s) is cancelled.

These simple definitions removes many ambiguities associated with a message passing domain, such as whether the intruder can intercept messages (it can) or if encryption can be broken (it cannot). The strength of this model is that a list of six actions and their preconditions/effects sufficiently define all valid agent interactions.

4 Golog Encoding

In order to automatically find attacks on a protocol, we encode the SitCalc representation as a Golog program. In this section, we sketch the details of our encoding.

4.1 Protocol Representations

The standard "arrows and colons" representation of protocols is standard in the protocol verification literature. As such, we use a similar representation based on a formal grammar. The following table illustrates how our grammar can be used to encode the Challenge Response Protocol:

Standard Representation	**Grammar Representation**
$A \rightarrow B : \{N_a\}_{K_{A/B}}$	`A->B: {Na}K-A/B`
$B \rightarrow A : N_a$	`B->A: Na`

Our representation of a protocol also includes a specification of initial key ownership, based on the following grammar:

```
"KEYS:"
{key-line}
key-line = role ":" key-list
key-list = key
key-list = key "," key-list
```

The key-list after each role indicates which keys an agent owns in the initial situation. In order to facilitate the representation of standard protocols, if no key ownership properties are set then the default is to assign a shared key to each pair of agents, as well as appropriate public-private key pairs for assymetric cryptography. Adding key-ownership details, the Challenge Response Protocol is encoded as follows:

```
A->B: {Na}K-A/B
B->A: Na
KEYS:
A: K-A/B
B: K-A/B
```

For any "arrows and colons" representation of a protocol, it is easy to produce this grammatical specification as input to the GoProVe verifier.

4.2 Compiling to GoLog: Motivation

Actions in a protocol must be compiled into multiple-step procedures in GoLog. For example, in the standard specification, one line encodes the act of sending a fresh nonce. A simple GoLog procedure for sending a fresh nonce is the following:

```
Agent-Nonce-Send(A,B,s):
    let Na = fresh(s)
    act(makeNonce(A,Na))
    let send-text = list(Na)
    let send-message = make-message(A, B, send-text)
    compose(A, B, send-text, send-message)
    send(A,send-message)
```

This procedure can be initiated by an agent A to generate a random number, compose a message including this number, and finally send the message to B. When B responds, A will need to perform conditional checks to see if the received message has the proper format and data. The main purpose of the compiler is to produce a series of procedures that send valid protocol messages, and verify that received messages match the transaction format. The GoLog representation of a protocol must also include constraints about protocol advancement, so that honest agents only continue a protocol if they believe that all previous sends/receives have been legitimate. In the next section, we describe the actions and checks that are required.

4.3 The Compilation Algorithm

Each step in a protocol involves one agent sending a message to another. The compilation algorithm translates each send action to a pair of procedures: one procedure to encode the sending action, and one procedure to encode the receiving action. In this section, we focus on the sending action.

Consider a message-send action of following form:

$$A \to B : \{M_1, \ldots, M_{j_1}\}_{K_1}, \ldots, \{M_1, \ldots, M_{j_n}\}_{K_n}.$$

The algorithm for translating this step into a GoLog procedure is as follows:

1. Write GoLog commands to declar procedure.
2. Initialize a list enc of encrypted components.
3. For each $\{M_1, \ldots, M_{j_i}\}_{K_i}$ with $1 \leq i \leq n$:
   (a) Initialize a list $data_i$ of (unencrypted) data components.
   (b) For each M_k with $1 \leq k \leq j_i$:
      i. Write GoLog commands to get the data M_k.
      ii. Append M_k to the list $data_i$.
   (c) Write GoLog commands to encrypt the list $data$ with the key K_i.
   (d) Append $data_i$ to the list enc.
4. Write GoLog commands to compose and send message constaining items in enc.
5. Write GoLog commands to advance to the next protocol step.

To illustrate the process, we show how the first message of the Challenge-Response Protocol is encoded as a GoLog program. In this case, there is just one encrypted component $\{N_a\}_{K_{A/B}}$ and one data component N_a. The algorithm therefore simplies to a four step procedure:

I. Write GoLog commands to declare procedure.
II. Write GoLog commands to get data corresponding to N_a.
III. Write GoLog commands to encrypt N_a with $K_{A/B}$.
IV. Write GoLog commands to compose and send the message.
V. Write GoLog commands to advance to the next protocol step.

For step (I), we give the procedure a name and check that the participants are distinct.

```
CR-A-1-proc(A, B, protocol-states, s):
    test(neq(A, B))
```

For step (II), we use the functional fluent *fresh* to generate a new nonce and the *makeNonce* action to specify it is created by A:

```
    let Na = fresh(s)
    act(makeNonce(A, Na))
```

In this particular case, there is only one item of text data to send. It is then passed to step (III), where we specify the key to be used for encryption and apply it to the list of text items using the *encrypt* action:

```
    let K-A/B = get-shared-key(A, B)
    let text-to-encrypt = list(Na)
    let enc-text = enc(text-to-encrypt, K-A/B)
    act(encrypt(A, text-to-encrypt, K-A/B))
```

Note that this step is essentially identical if multiple items of text are encrypted with the same key. In the general case, the output of this step is also a list, where each item of the list is a text component encrypted with a different key. For step (IV), we compose a message that consists of all encrypted components

```
    let send-text = list(enc-text)
    let send-message = make-message(A, B, send-text)
    compose(A, B, send-text, send-message)
    send(A, send-message)
```

Finally, for step (V.), we advance the state of the protocol:

```
    new-protocol-state(A, send-message, (A, B), CR-A-1)
```

This completes the specification of the GoLog procedure for sending the first message of the Challenge Response Protocol. The procedure generated to receive this message is similar. The main difference is that we need to consider previously sent messages. This is simply a matter of additional checks on variable bindings.

In the interest of space, we do not include the complete encoding of the Challenge-Response protocol here. We simply remark that each message exchanged in the protocol is encoded through a send procedure and a receive procedure, as illustrated above.

5 The GoProVe Verifier

5.1 Protocol Simulation

Protocol simulation is the process of testing protocols runs where all agents, including the intruder, are honest. The process of protocol simulation involves the following sub-procedures:

1. **Initiate protocol**. Select a pair of agents, and assign them the roles in the protocol.
2. **Simulation control**. Check if a message has been sent. If a message was sent, then run a procedure to receive and react to that message. If no message has been sent, then check if there is a message available to be sent. If there is no message to send, the session is complete.
3. **Output**. Determine if a successful run has been completed for some agent pair.

This simple algorithm detects if a protocol run between any pair of agents can complete successfully. This is important for verification, because we need to be able to distinguish between a badly formed protocol with no successful runs, and an insecure protocol that is vulnerable to attack.

5.2 Protocol Verfification

For protocol verification, the simulation algorithm is extended by introducing an intruder. The intruder has special recieve/send procedures that allow the protocol to advance even if messages are intercepted or forged. Since a protocol run does not have a fixed (or unbounded) length in this context, the search for a run must use iterative deepening. The GoProVe verifier uses the following procedure to find attacks.

Initialize: Set message ownership for all agents.
Start: Choose a principal A to initiate protocol run with B. Set depth=0.
While(true)
    Execute protocol action: Send or receive message.
    Set depth = depth + 1.
    If (message was sent)
        Intruder receives message, decrypts, composes all possible messages.
    If(attack occurred)
        Return attack.
    If(depth $\geq$ max depth)
        Backtrack.
    Advance protocol: Do one of the following:
        Principal receives a sent message.
        Intruder sends message.

There are many ways that the search space can be expanded indefinitely, but we have limited the behaviour of the intruder in a way that makes this less of a problem. For example, the set of nonces held by the intruder is fixed, it is not possible for the

intruder to expand the search space by continually generating nonces. In order to optimize performance, we also use a **Greedy Intruder Channel Model**. In this model, the intruder immediately decrypts, parses and analyzes every received message immediately. While this might appear to slow down run times due to unecessary processing, it actually protects the verifier from combinatorial explosion. Since the intruder does all processing immediately, the pseudocode algorithm above can actually be understoond to progress rather than simply stall as an intruder loops through internal operations.

On finding a compromised goal, Go ProVe outputs a description of any attacks, information about agent beliefs regarding sessions, in the following format:

```
Protocol-ID: CR
     Status: completed
     Viewer: Alice
     Role B: Bob
     Role A: Alice

Messages:
     (Alice,Bob,[{N2}K-Alice/Bob])
     (Bob, Alice, [N2])
```

In the case of the Challenge-Response protocol, two different runs are output: one where she started a session with Bob and another where she thought that Bob was challenging her. This is an indication of an attack.

5.3 Performance

The table below gives a list of the GoProVe Verifier's runtimes for Challenge Response, Challenge Response with Servers, Needham-Schroeder, and Needham-Schroeder-Lowe [11].

Protocol	**Max-Depth**	**attack?**	**time**
CR	3	Y	78 ms
CR-Server	7	Y	9 min
NS	4	Y	2 s
NS	4	Y	5 s
NSL	5	N	7 min

These results are promising, as protocol verification is a challenging task even for simple protocols.

6 Related Work

Most logic-based approaches to protocol verification are influenced to some degree by the pioneering BAN logic of [4]. This approach has been highly influential because it reduces protocol verification to reasoning about knowledge in a formal logic. However, BAN logic itself consists of an ad hoc set of rules of inference with no formal semantics.

In this respect, our approach differs from the BAN tradition. Rather than defining a new protocol-specific logic, we encode protocols in a flexible, general purpose action formalism.

Hernández and Pinto propose an approach similar to ours, notably due to the fact that they also use the SitCalc [8]. However, they focus on producing proofs of correctness based on the actions of honest agents. By contrast, we explicitly model the actions of an intruder, and we view protocol verification as the process of "planning an attack." Our treatment of communication is also different: while Hernández and Pinto define an unreliable broadcast channel, we define a direct channel that allows the intruder the first opportunity to receive a message. As such, our approach is best viewed as an alternative to the Hernández-Pinto approach, rather than a continuation.

7 Discussion

We have described a declarative approach for the representation and analysis of cryptographic protocols. In our model, the effects of actions and the properties of the environment are explicitly specified in a Golog program, based on an underlying SitCalc action theory. Our approach is distinguished by the explicit representation of all aspects of a protocol, including the capabilities of the intruder that are often hard-coded. As a result, our approach is more flexible and more elaboration tolerant than alternative approaches.

Specific advantages of our declarative model can be seen by considering simple variations on the standard Dolev-Yao intruder. For example, we have already discussed the fact that it is possible to modify the key-ownership settings. Similarly, it would be possible to constrain procedures with respect to the topology of a particular network. In contrast, in extant logical approaches to protocol verification, it is not straightforward to modify the model for a specific application.

Many proofs of protocol correctness rely on the assumption that honest agents do not perform actions that compromise secret information; however, it is not always clear which actions are likely to do so. In our framework, we can discover these undesirable actions and we can formally specify axioms that restrict honest agents from performing them. To the best of our knowledge, this problem has not been addressed in related formalisms.

In addition to the theoretical advantages gained by using the SitCalc for encoding protocols, we also gain the practical advantage that it is relatively easy to implement a prototype verification system in Golog. Our software uses a formal grammar to represent the structure of a protocol, and translates protocols directly into Golog programs that encode SitCalc action theories. In principle, users need not have detailed knowledge of our SitCalc encoding in order to analyse the security of a protocol.

There are several directions for future work. One direction would be to explore a wider range of protocols, such as those designed for non-repudiation and fair exchange. Another improvement would be to reduce the running time, in order to address longer and more complex protocols. However, as a proof of concept, the current run times demonstrate that the we can use a flexible model of protocol execution based on a rigorous action formalism, while still finding attacks in a reasonable time.

References

1. L. Aiello and F. Massacci. Planning attacks to security protocols: Case studies in logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, 2001.
2. A. Armando, L. Compagna and Y. Lierler. Automatic Compilation of Protocol Insecurity Problems into Logic Programming In *Proceedings of JELIA*, 617-627, 2004.
3. S. Brackin, C. Meadows, and J. Millen. CAPSL Interface for the NRL Protocol Analyzer In *Proceedings of ASSET 99*, IEEE Press, 1999.
4. M. Burrows, M Abadi, and R. Needham. A logic of authentication. In *ACM Transactions on Computer Systems* 8(1):18–36, 1990.
5. D. Dolev, and A.C. Yao. On the Security of Public Key Protocols. *IEEE Trans. on Inf. Theory*, 2(29), 198-208, 1983.
6. R. Fagin, J.Y. Halpern, Y. Moses and M.Y. Vardi. Reasoning about Knowledge. The MIT Press. 1995.
7. J.Y. Halpern R. Pucella. In *On the Relationship between Strand Spaces and Multi-Agent Systems* CoRR, cs.CR/0306107, 2003.
8. J. Hernández-Orallo, and J. Pinto. Especificación formal de protocolos criptográficos en Cálculo de Situaciones. *Novatica*, 143, 57-63, 2000.
9. A. Hunter and J. Delgrande. Belief Change and Cryptographic Protocol Verification. In *Proceedings of AAAI*, 2007.
10. H.J. Levesque, F. Pirri and R. Reiter. Foundations for the Situation Calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998.
11. G. Lowe, G. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proceedings of TACAS*, 147-166, 1996.
12. J. Thayer, J. Herzog, and J. Guttman. Strand Spaces: Proving Security Protocols Correct. *Journal of Computer Security*, 7(2-3), 191-230, 1999.