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

James P. Delgrande and Aaron Hunter
School of Computing Science,
Simon Fraser University,
Burnaby, B.C.,
Canada V5A 1S6.
{jim,hunter}@cs.sfu.ca

Torsten Grote
Institut für Informatik,
Universität Potsdam,
D–14482 Potsdam, Germany
Torsten.Grote@uni-potsdam.de

***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 a theory of action, specifically the situation calculus. Our thesis is that all aspects of a protocol must be explicitly specified. 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.**

I. Introduction

A cryptographic protocol is a formalised sequence of messages exchanged between agents, where parts of the messages are protected using cryptographic functions such as encryption. These protocols are used for many purposes, including the secure exchange of information, carrying out a transaction, authenticating an agent, etc. 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 [1]. Part of the problem is that there is no clear agreement on what exactly constitutes an attack [2], 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. The framework makes explicit background assumptions, protocol goals, agent's capabilities, and the message passing environment. 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.

978-1-4244-7550-6/10/$26.00 ©2010 IEEE

The next section motivates our approach, while the following section presents an axiomatisation of an instance of the approach in the situation calculus. We then provide an outline of our prototype software for protocol verification. We conclude by comparing our approach with related work, and discussing future directions.

II. Background and Motivation

The standard intruder model used in cryptographic protocol verification is the so-called Dolev-Yao intruder [3]. 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 [5], strand spaces [6], [7], multi-set re-writing formalisms [8] and logic programs under the stable model semantics [2].

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". Consider the following exchange:

Another Attack on the Challenge-Response Protocol

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

In this case A sends the unencrypted nonce to the intruder. This of course is outlandish, but it does represent a logically possible compromise of the protocol. The problem is that, much like the qualification problem in planning, there is an assumption that "nothing untoward happens" in a protocol execution. However, it may well be that there are "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.

We argue that in order to provide a robust demonstration of the security and correctness of a protocol, all of the above points need to be addressed. 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.

A. The Situation Calculus

In this section, we briefly introduce the situation calculus (SitCalc). We provide only a superficial introduction here; see [9] for details. The *situation calculus* is a multi-sortal language that contains, in addition to the usual connectives and quantifiers of first-order logic, at least the following elements.

1) A sort *action* for actions, with variables a, a', etc.
2) A sort *situation* for situations, with variables s, s', etc.
3) A sort *objects* for everything else.
4) A function $do : action \times situation \rightarrow situation$.
5) A distinguished constant $S_0 \in situation$.
6) A binary predicate $Poss : action \times situation$, where $Poss(a, s)$ is intended to indicate that action a is possible in situation s.

Thus, the terms in a SitCalc action theory range over a domain that includes *situations* and *actions*. A *fluent* is a predicate that takes a situation as the final argument. A SitCalc action theory includes an *action precondition* axiom for each action symbol, a *successor state axiom* for each fluent symbol, as well as the foundational axioms of the SitCalc. Informally, a situation represents the state of the world, along with a complete history of actions that have been executed. The distinguished constant S_0 represents the initial situation, and the distinguished function symbol *do* represents the execution of an action. Every situation can be written as follows:

$$do(A_n, do(A_{n-1}, \ldots, do(A_1, S_0) \ldots).$$

To simplify the notation, we will abbreviate this situation as $do([A_1, \ldots, A_n], S_0)$. Hence, to state that it is possible to break an object in a situation iff it is fragile, and that the result of breaking an object is that it is broken, can be expressed as follows:

$$Poss(break(x), s) \equiv Fragile(x, s)$$

$$Broken(x, do(break(x), s))$$

III. Approach

We present a formalization for cryptographic protocols, using the Challenge-Response protocol as an example. While we don't cover all points raised in the previous section, given space constraints, it should be clear that omissions are easily addressable.

A. Vocabulary

There are four main sorts of objects (beyond actions and situations): *agents, keys, messages* and *nonces*.

Agents: The term *agent* refers to both honest agents and to the malicious intruder. Variables a, a_1, ... range over agents. The constant $intr$ denotes the intruder. Unary predicates $Agent$ and $Intruder$ have their obvious meanings.

Fluent $Alive(a, s)$ indicates that a is alive in situation s. It is a precondition for executing any action; for brevity however we omit it in action preconditions. $Has(a, x, s)$ means that a has access to x in situation s, where the variable x ranges over messages, keys and nonces. This is can be seen as a kind of knowledge, but we use an epistemically neutral term and interpret the meaning in terms of "access" to information. We use $Bel(a, f, s)$ to indicate that a believes that f is true in situation s. The semantics of Bel can be defined using the treatment of belief in [10] (where they use epistemic fluent $Knows$ for Bel).

Messages: Communication in our framework involves the exchange of messages. Variables m, m_1, ... range over messages. Unary predicate Msg is true of messages. Messages are considered to be atemporal, and so are not indexed by a situation. Messages are composed of a finite sequence of parts, which may be nonces, agent names, or keys; each part may be encrypted. We assume an appropriate situation calculus axiomatization of lists, including the constructor $list(p_1, \ldots, p_n)$ and selectors $first(m)$, $second(m)$, etc. A useful axiom is that if an agent Has a message, then it has the message parts, for example:

$$Has(a, m, S_0) \wedge Msg(m) \supset Has(a, first(m), S_0).$$

Keys: Variables k, k_1, ... range over keys. Predicate $Key(k)$ indicates that k is a key, while $SymKey(k)$ and $AsymKey(k_1, k_2)$ have their expected meaning for symmetric and asymmetric keys respectively. $ShKey(a_1, a_2, k)$ indicates that k is a shared (symmetric) key for agents a_1, a_2. $PubKey(a, k)$ and $PrivKey(a, k)$ give public and private keys, respectively, of an agent.

Three functions are associated with keys: $encKey(x)$ is the key which has been used to encrypt x. The value of $enc(x, k)$ is the message that results when x is encrypted with k, and the value of $dec(x, k)$ is the corresponding decrypted message. We have the following relation between enc and $encKey$:

$$m_1 = enc(m_2, k) \supset k = encKey(m_1).$$

Nonces: Variables n, n_1, ... range over nonces. Nonces must be *freshly generated* during the current protocol run. The fluent $IsFresh(n, s)$ is intended to be true if and only if the nonce n has been generated "recently" with respect to the situation s. To this end, the functional fluent $fresh(s)$ is used to model the generation of new nonces during a protocol run using the axiom:

$$fresh(s) = fresh(s') \supset s = s'.$$

Actions: We complete our description of the vocabulary by specifying the set of action terms. These include actions for encryption and decryption, sending and receiving messages, and composing messages. To ease readability we omit sort predicates; instead we use the variable conventions given above to specify the sort of each variable. Free variables are implicitly universally quantified.

1) $encrypt(a, m, k)$ – Agent a encrypts m using key k.
   Precondition:
   $Poss(encrypt(a, m, k), s) \equiv (Has(a, m, s) \wedge (Has(a, k, s) \vee \exists a' PublicKey(a', k)))$
   Effect: $Has(a, enc(m, k), do(encrypt(a, m, k), s))$
2) $decrypt(a, m, k)$ – Agent a decrypts m using key k.
   Precondition:
   $Poss(decrypt(a, m, k), s) \equiv (Has(a, m, s) \wedge Has(a, k, s) \wedge [(SymKey(k) \wedge k = encKey(m)) \vee (AsymKey(k, k') \wedge k' = encKey(m))])$
   Effect: $Has(a, dec(m, k), do(decrypt(a, m, k), s))$
3) $send(a_1, a_2, m)$ – Agent a_1 sends m intended for a_2. The intruder can masquerade as a principal. Fluent $Sent$ indicates that a message is in some fashion "posted", that is can be received by an agent.
   Precondition:
   $Poss(send(a_1, a_2, m), s) \equiv ((Has(a_1, m, s) \wedge a_1 \neq a_2) \vee Has(intr, m, s))$
   Effect: $Sent(m, a_1, a_2, do(send(a_1, a_2, m), s))$
4) $receive(a_1, a_2, m)$ – a_1 receives message m from a_2. The intruder can intercept messages. $\neg Sent$ indicates that the message is no longer available to be received. $Recd$ records information concerning receipt of the message.
   Precondition:
   $Poss(receive(a_1, a_2, m), s) \equiv (Sent(m, a_2, a_1, s) \vee (a_1 = intr \wedge \exists a' Sent(m, a_2, a', s)))$
   Effect:
   $Has(a, m, do(receive(a_1, a_2, m), s)) \wedge \neg Sent(a_2, a_1, m, do(receive(a_1, a_2, m), s)) \wedge Recd(a_1, a_2, m, do(receive(a_1, a_2, m), s))$
5) $compose(a, m, x_1, x_2, \ldots)$ – Agent a composes message m having parts x_1, x_2, ... Thus, $compose$ is in fact a set of actions, one for each possible number of arguments. This presents no issues, since any protocol will have fixed, known, message lengths.
   Precondition:
   $Poss(compose(a, m, list(x_1, \ldots x_n), s) \equiv (Has(a, x_1, s) \wedge \ldots Has(a, x_n, s))$
   Effect:
   $Has(a, i, do(compose(a, m, list(x_1, \ldots x_n), s))) \wedge Msg(m) \wedge first(m) = x_1 \wedge second(m) = x_2 \wedge \ldots$

B. Initial Situation

The initial situation contains information about the number of agents, their keys, etc. Since the details are straightforward, we just outline what is required. For example, using the $Agent$ fluent, a finite set of principals is specified. This set includes the individual $intr$, whom we have called the intruder. As

well, for each agent, we specify a combination of private, public, and shared keys. For brevity we consider just shared (symmetric) keys here.

As well, in the initial situation we specify additional constraints; these are propagated, as usual in a basic action theory, to successor situations. For proving properties about protocols, some *epistemic* constraints are useful, for example, an agent will know what actions it carried out. In the Challenge-Response protocol we use the following:

$$Sent(a_1, a_2, m, S_0) \supset Bel(a_1, Sent(a_1, a_2, m), S_0)$$
$$Recd(a_1, a_2, m, S_0) \supset Bel(a_1, Recd(a_1, a_2, m), S_0)$$

We can then state that if an agent a_1 sends a fresh nonce encrypted in the key it shares with a_2, and gets the unencrypted nonce back, then a_1 believes that a_2 is alive:

$$(Bel(a_1, Sent(a_1, a_2, en), S_0) \ \wedge \ en = enc(n, k) \wedge \\ Fresh(n) \ \wedge \ ShKey(a_1, a_2, k) \wedge \\ Recd(a_1, x, n, S_0)) \ \supset \ Bel(a_1, Alive(a_2), S_0)$$

C. Adding Control

Parallelism is effected by allowing (concurrent) interleaving of actions. We model a Dolev-Yao intruder through the following scheme, which allows the intruder to perform an arbitrary number of actions before an honest agent can act:

```
loop {
  Intruder executes some actions;
  A principal executes one action
}
```

This is implemented in our action theory as follows. A new fluent OkP is introduced to state that a principal may execute an action. Basic actions are modified as follows:

- For a principal: Each precondition of the form $Poss(a, s) \equiv \phi$ is modified to $Poss(a, s) \equiv (\phi \wedge OkP(s))$. Each effect axiom $\psi(do(a, s))$ is replaced by $\psi(do(a, s)) \wedge \neg OkP(do(a, s))$.
- The intruder can make $OkP(s)$ true. A new action $onOkP$ is introduced with precondition $Poss(onOkP(a), s) \equiv a = intr$ and effect $OkP(do(onOkP, s))$.

An advantage of this framework is that other models of agent concurrency can be easily expressed. For example, in some applications it might be reasonable to limit the intruder to perform one action following each principal action, whereas in other applications this might be too restrictive. At present, we are interested in modelling a Dolev-Yao intruder, but we stress that our underlying framework is flexible enough to allow other options.

D. Representing a Protocol in an Action Theory

Our goal is to completely and explicitly specify a theory of agent communication involving encryption, freshly generated nonces, and a hostile intruder. A protocol then is regarded as a high-level description of prescribed agent actions designed to achieve some end, or goal, in a dynamic, unpredicatable, hostile environment. There are two things that remain to be specified:

1) how the protocol corresponds to sets of agent actions, and
2) the goal of the protocol.

Encoding the Actions: In the approach, lines of a protocol are compiled into new, protocol-specific actions. (The alternative is to compile lines of a protocol into a sequence of our already-defined primitive actions. While more perspicuous, we don't take this option, as it proved to be much less efficient in the implementation.) Each line of a protocol is implicitly made up of two complex actions, the first corresponding to the composition and sending of a message, and the second corresponding to the receiving and decrypting of the message. Thus in the first line of the Challenge-Response protocol, the intent is that A compose a message and send it, followed by B receiving it and decrypting it. However, note that for every line in a protocol except the last, the implicit $receive$ of that line can be combined with the $send$ of the next. Hence a n-line protocol can compile into $n+1$ protocol-specific actions. For the Challenge-Response protocol, the protocol compiles into three new protocol-specific actions:

$CR.1.send$:
: Agent a_1 composes a message with a fresh nonce, encrypts it in the key shared with a_2, and sends it to a_2.

$CR.1.rec.2.send$:
: a_2 receives the message, decrypts it, and sends a message with the nonce to a_1.

$CR.2.rec$:
: a_1 receives the unencrypted nonce from a_2.

We introduce the following constants and fluents: $\langle pid \rangle$ is an identifier inserted by the compiler giving the protocol type and instance of the run. (We also use pid without angle brackets as a variable.) Predicate $Type$ extracts the protocol type from its argument; here $Type(pid) =$ "CR". Fluent $Expect$ expresses control knowledge, for instance that after initiating the protocol, a_1 expects at some later point to receive a message from a_2 comprising the second step in this instance of the protocol. In this way, multiple instances of multiple protocols may concurrently be executed. Fluent $Completed$ with argument pid indicates that as far as the first agent is concerned, the protocol has completed.

We have the following action preconditions and effects:

$CR.1.send$:
Precondition:
$$Poss(CR.1.send(a_1, a_2, m, k, n), s) \equiv \\ ShKey(a_1, a_2, k) \ \wedge \ n = fresh(s) \wedge \\ m = list(\langle pid \rangle, enc(n, k))$$
Effect: Let $s' = do(CR.1.send(a_1, a_2, m, k, n), s)$.
$$Sent(a_1, a_2, m, s') \ \wedge \ Has(a_1, n, s') \wedge \\ Has(a_1, enc(n, k), s') \ \wedge \ Expect(a_1, a_2, pid, 2, s')$$

$CR.1.rec.2.send$:
Precondition:
$$Poss(CR.1.rec.2.send(a_2, a_1, m, m'), s) \equiv \\ Sent(a_1, a_2, m, s) \ \wedge \ Type(first(m)) = \text{“}CR\text{”} \wedge \\ Has(a_2, encKey(m), s) \ \wedge \ first(m') = first(m) \wedge$$

$second(m') = dec(second(m), encKey(m)) \wedge$
$Has(a_2, encKey(m), s)$

The precondition is cumbersome, reflecting the fact that two composite actions (a receive and send) are combined here into one protocol-specific action.

Effect: Let $s' = do(CR.1.rec.2.send(a_2, a_1, m, m'), s)$.

$Recd(a_2, a_1, m, s') \wedge Has(a_2, m, s') \wedge$
$Has(a_2, first(m), s') \wedge Has(a_2, second(m), s') \wedge$
$Has(a_2, dec(second(m), encKey(m)), s') \wedge$
$\neg Sent(a_1, a_2, m, s') \wedge Sent(a_2, a_1, m', s') \wedge$
$first(m') = pid \wedge second(m') = dec(second(m)))$

The effect is likewise cumbersome: a_2 has the message and all its parts; the original message is marked as unavailable; and a new message is sent to a_1.

$CR.2.rec$:

Precondition:

$Poss(CR.2.rec(a_1, a_2, m), s) \equiv$
$Sent(a_2, a_1, m, s) \wedge Type(first(m)) =$ "*CR*" $\wedge$
$Expect(a_1, a_2, first(m), 2, s)$

Effect: Let $s' = do(CR.2.rec(a_1, a_2, m), s)$.

$Recd(a_1, a_2, m, s') \wedge Has(a_1, m, s') \wedge$
$Has(a_1, first(m), s') \wedge Has(a_1, second(m), s') \wedge$
$Completed(first(m), s')$

E. Expressing the Goal of a Protocol

The goal of a protocol will often have epistemic components. For the Challenge-Response protocol, the overall goal is that the initiating agent will *know* the responding agent is alive following a successful run. That is, the initiating agent will not believe that the responding agent is alive unless it is in fact alive.

$(Completed(pid', s) \wedge pid' = \langle \text{pid} \rangle) \supset$
$(Bel(a_1, Alive(a_2), s) \equiv Alive(a_2, s))$

One may want to prove further protocol properties as well. For instance, if the intruder carries out no actions, then the protocol should be guaranteed to succeed. At a more basic level, one might want to prove that there is a successful run:

$$\exists s, m.\ (Completed(pid', s) \wedge pid' = \langle \text{pid} \rangle)$$

That is, a protocol that can never complete will vacuously never be compromised, but is of no use.

F. The Attack on the CR Protocol

We illustrate the approach by describing the attack on the Challenge-Response protocol:[1]

1) Agent a_1 initiates a round of the protocol:
   $CR.1.send(a_1, a_2, (\text{"}CR\text{"}, enc(n, k)), k, n)$
   One effect is $Sent(a_1, a_2, (\text{"}CR\text{"}, enc(n, k)))$
2) The intruder intercepts the sent message:
   $receive(intr, a_2, (\text{"}CR\text{"}, enc(n, k)))$
3) The intruder sends a message to a_1, masquerading as a_2:
   $send(a_1, a_2, (\text{"}CR\text{"}, enc(n, k)))$
4) The message is received by a_1 who understands it as an initiation of a new round of the CR protocol by a_2:
   $CR.1.rec.2.send(a_1, a_2, (\text{"}CR\text{"}, enc(n, k)), (\text{"}CR\text{"}, n))$
   This has effect $Sent(a_1, a_2, (\text{"}CR\text{"}, n))$.
5) The intruder intercepts this message:
   $receive(intr, a_1, (\text{"}CR\text{"}, n))$.
   This has effects $Has(intr, (\text{"}CR\text{"}, n))$ and $Has(intr, n)$.
6) The intruder sends the nonce to a_1, masquerading as a_2:
   $send(a_2, a_1, (\text{"}CR\text{"}, n))$
7) The message is received by a_1:
   $CR.2.rec(a_1, a_2, (\text{"}CR\text{"}, n))$
   a_1 understands it as the completion of the original protocol; thus a_1 believes a_2 alive in the resulting situation.

Thus a_1 executes appropriate steps in the protocol, while the intruder executes a plan to compromise the protocol's goal.

IV. Implementation

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 implementation.[2]

The basic actions in our Golog program are specified in the following format:

```
primitive_action(encrypt(A,M,K)).
```

In addition to all of the actions in the SitCalc vocabulary, two additional actions are also required: `makeNonce` and `makePID`.

Translating successor state axioms into Golog is straightforward. For example, for the fluent `sent`, we have expressions of the following form:

```
sent(A,B,M, do(ACT,S)) :- ACT=send(A,B,M).
```

Action preconditions are easy to express in Golog, as is the initial state:

```
agent(intr,s0).
intruder(intr,s0).
key(k-bob,s0).
```

Our implementation makes particular use of a special functional fluent `next` that starts with a value of 1 in S_0 and increases whenever a new object (message, nonce, or PID) is constructed. For instance, if the intruder makes a new nonce, the current value of `next` becomes a nonce and `next` is incremented. In this way, `next` offers an infinite supply of objects. Note that our usage of `next` essentially makes it impossible for the value of a nonce to be guessed; this is consistent with the standard "symbolic" approach to the verification of cryptographic protocols.

Finding Attacks

We represent the notion that a fluent `F` is believed to be true by introducing a fluent `belF`. For example, the following code is used to indicate when an agent believes a given message has been sent in the Challenge-Response protocol.

[1] We suppress situation arguments in fluents for readability.

[2] Available at http://www.cs.sfu.ca/~cl/software/software.htm.

```
belSent(A,B,M, do(ACT,S)) :-
  (ACT = send(A,B,M));
  (ACT = cr1s(A,B), next(M,S));
  (ACT = cr1r2s(A,B,_), next(M,S));
  belSent(A,B,M,S).
```

The goal of the entire protocol is that, once the protocol is complete, A will believe B is alive if and only if B is actually alive. This is coded as follows.

```
goal(S) :- belAlive(A,B,S), alive(B,S).
```

Given the Golog protocol specification and goal, we use Reiter's iterative deepener to find attacks on protocols.[3] The iterative deepener is started by calling the planbf goal with a maximum depth:

```
?- planbf(10).
```

Our implementation discovered all attacks that we are aware of, including those that appear in the literature, and the aforementioned "stupidity attack".

Performance Considerations

The Golog implementation of a protocol is easily defined, and it is human readable. Much of the encoding is a direct translation of the SitCalc action theory. Unfortunately, the system suffers from relatively slow performance. For the Challenge-Response protocol, the known attack is discovered in under a day running on a standard desktop computer. However, to find the attack on the well-known Needham Schroeder protocol it would take much longer on the same platform. For this reason, we are currently developing a second implementation using the Scheme encoding of the SitCalc. Initial experiments suggest that the search times improve by two orders of magnitude.

V. Automatic Translation

The overall goal of the project is to find attacks on protocols based solely on a simple specification, without requiring the user to have specialised knowledge about our SitCalc encoding. In our software, we use the following grammar as a protocol specification language:

protocol = {*transaction*}
transaction = *agent*, "−>", *agent*, ":", *message*
message = *basic-message*, {",", *basic-message*}
basic-message = *enc-message* | *agent* | *nonce* | *key*
enc-message = "{", *message*, "}", *key*
agent = *STRING NOT STARTING WITH 'N' OR 'K'*
nonce = *STRING STARTING WITH 'N'*
key = *sym-key* | *public-key* | *private-key*
sym-key = "K-", *agent*, "/", *agent*
public-key = "K-", *agent*
private-key = "KP-", *agent*

Our software automatically translates protocols specified in this grammar into Golog programs. The translator is a recursive descent parser for the input grammar, written in Java.

[3]Available at http://www.cs.toronto.edu/cogrobo/kia/wspbf.

Note that our protocol grammar is based on the typical "arrows and colons" specification, and it therefore does not include the goal of the protocol. At present, we still need to hand-code the goal of a protocol and add it to the output before we can search for attacks.

VI. 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 specifying an ad hoc protocol 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 [11]. 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.

VII. Discussion

This paper has introduced a declarative, logical approach for the representation and analysis of cryptographic protocols. Our thesis is that all aspects of a protocol need to be explicitly specified. That is, aspects of message passing, intruder capabilities, agent actions (along with their preconditions and effects) all need to be specified in a declarative, readable form. Thus for example, the fact that the intruder may intercept or redirect a message is explicitly stated as part of the problem specification. There are several advantages of such an explicit form: It is perspicuous, in that all information that is relevant is explicitly stated. As well, having all aspects of a protocol given in a declarative specification means that one can reason about all aspects of a protocol, and so ideally there are no "hidden assumptions" masked by the representation. The goal is that, ideally, a software engineer will be able to read off all aspects of a protocol given such a logical specification. The downside to this level of generality is that computational issues are a significant challenge. Indeed, while our software is still at the prototype stage, we are able to handle only the simplest of protocols in a reasonable amount of time.

Our theory for protocol specification is expressed in terms of a situation calculus theory. The framework makes explicit background assumptions, protocol goals, agent's capabilities, and the message passing environment. In alternative logic-based approaches to protocol verification, such features are often hard-coded or implicit in the definition of a fixed

logic. In the approach, a protocol is first *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. A valid protocol then is one which is secure, which may complete, and in which at completion the goal is provably established. The approach is flexible, and significantly more general than previous approaches. As well, it is elaboration tolerant, in that in principle it is straightforward to modify a specification. For example, it would be easy to encode the topology of a particular network or modify agent capabilities in our framework. In contrast, in extant logical approaches to protocol verification, it is not straightforward to modify the model for a specific application.

In the development of our approach, we observed that 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. Similarly, ours is the first approach to encode a declarative specification of the goals of a protocol; such a specification gives new insight into the precise meaning of protocol correctness.

In addition to the theoretical advantages gained by using the situation calculus 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 expressions in the grammar into Golog programs that encode situation calculus action theories. In principle, this means that users of our software need not have specific knowledge of the situation calculus in order to analyse the security of a protocol.

There are several directions for future work. First, it would be interesting to explore a wider range of protocols, such as non-repudiation and fair exchange. Towards this end, we are currently using the situation calculus to design a formal approach to analyse the secure exchange of digitally signed forms. One desirable improvement is to include the goals of a protocol in the input specification, thereby eliminating the need for a user to know the situation calculus. Another direction is to improve the performance of the existing implementation. in order to address longer and more complex protocols. We are currently working on these improvements.

REFERENCES

[1] S. Brackin, C. Meadows, and J. Millen, "CAPSL interface for the NRL protocol analyzer," in *Proceedings of ASSET 99*. IEEE Computer Society Press, March 1999.

[2] L. Aiello and F. Massacci, "Verifying security protocols as planning in logic programming," *ACM Transactions on Computational Logic*, vol. 2, no. 4, pp. 542–580, 2001.

[3] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Trans. on Inf. Theory*, vol. 2, no. 29, pp. 198–208, 1983.

[4] M. Burrows, M. Abadi, and R. Needham, "A logic of authentication," *ACM Trans. on Computer Systems*, vol. 8, no. 1, pp. 18–36, 1990.

[5] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning about Knowledge*. Cambridge, Massachusetts: The MIT Press, 1995.

[6] F. J. Thayer, J. C. Herzog, and J. D. Guttman, "Strand spaces: Proving security protocols correct," *Journal of Computer Security*, vol. 7, no. 1, 1999.

[7] J. Y. Halpern and R. Pucella, "On the relationship between strand spaces and multi-agent systems," *CoRR*, vol. cs.CR/0306107, 2003.

[8] A. Armando, L. Compagna, and Y. Lierler, "Automatic compilation of protocol insecurity problems into logic programming," in *Proceedings of JELIA*, 2004, pp. 617–627.

[9] H. Levesque, F. Pirri, and R. Reiter, "Foundations for the situation calculus," *Linköping Electronic Articles in Computer and Information Science*, vol. 3, no. 18, 1998.

[10] R. Scherl and H. Levesque, "Knowledge, action, and the frame problem," *Artificial Intelligence*, vol. 144, no. 1-2, pp. 1–39, 2003.

[11] J. Hernández-Orallo and J. Pinto, "Formal modelling of cryptographic protocols in situation calculus," 1997, (Published in Spanish as: Especificación formal de protocolos criptográficos en Cálculo de Situaciones, *Novatica*, 143, pp. 57-63, 2000).