A General Approach to the Verification of Cryptographic Protocols using Answer Set Programming

James P. Delgrande, Torsten Grote, and Aaron Hunter

School of Computing Science,
Simon Fraser University,
Burnaby, B.C.,
Canada V5A 1S6.
{jim, tga14, hunter}@cs.sfu.ca

Abstract. We introduce a general approach to cryptographic protocol verification based on answer set programming. In our approach, cryptographic protocols are represented as extended logic programs where the answer sets correspond to traces of protocol runs. Using queries, we can find attacks on a protocol by finding the answer sets for the corresponding logic program. Our encoding is modular, with different modules representing the message passing environment, the protocol structure and the intruder model. We can easily tailor each module to suit a specific application, while keeping the rest of the encoding constant. As such, our approach is more flexible and elaboration tolerant than related formalizations. The present system is intended as a first step towards the development of a compiler from protocol specifications to executable programs; such a compiler would make verification a completely automated process. This work is also part of a larger project in which we are exploring the advantages of explicit, declarative representations of protocol verification problems.

1 Introduction

A cryptographic protocol is a sequence of encrypted messages that is used to exchange information and achieve communicative goals over an insecure network. Proving that a cryptographic protocol is secure is difficult, because many attacks are subtle and difficult to find. The problem is further complicated by the fact that protocol goals are often specified imprecisely, which makes it difficult to know exactly what might constitute an attack. In this paper, we use Answer Set Programming (ASP) to encode relevant information involved in a cryptographic protocol, including agent capabilities and the message passing environment, and to automatically detect attacks.

A wide range of methods have been previously employed for the verification of cryptographic protocols, including encodings in many different formal logics. Our work is distinguished from most of these methods in that we use a declarative formalism for reasoning about action effects. We specify the steps of a cryptographic protocol in a logic program where the answer sets correspond to sequences of messages exchanged between agents. After translating a given protocol into our encoding, it is straightforward to automatically generate attacks by using an answer set solver.

This paper makes several contributions to work on cryptographic protocol verification. First, given our declarative approach, all aspects of protocol specification, including agent capabilities, the message-passing environment, and the goals of the protocol, are explicitly specified. As a result also, our model of message passing is more general and more flexible than other models used in protocol verification. For example, in our approach it is straightforward to modify the capabilities of the intruder for a specific application. Second, our work provides a methodology for compiling an arbitrary cryptographic protocol into a formal encoding suitable for automated analysis. This is a significant improvement over purely logical representations of cryptographic protocols, where the translation from formalism to implementation may be non-trivial.

We proceed as follows. In §2, we provide an illustrative example, related work, and motivation for our approach. In §3, we present the details of our formal model. In §4, we discuss the advantages of our approach as compared with related formalisms. We then offer some concluding remarks and directions for future work.

2 Background and Motivation

2.1 Illustrative Example: The Needham Schroeder Protocol

We introduce the well-known Needham Schroeder protocol. This is an *authentication protocol* involving two agents A and B, which is to say that the goal is for each agent to establish that they are communicating with the other. In the protocol specification, the notation $\{M\}_K$ is used to denote the message M encrypted with the key K. In the given protocol, N_A and N_B denote random numbers, called *nonces*, generated by A and B respectively. The key K_A is the public key for A and the key K_B is the public key for B. Using this notation, the protocol is specified as follows.

The Needham Schroeder Protocol

1. $A \rightarrow B : \{N_A, A\}_{K_B}$
2. $B \rightarrow A : \{N_A, N_B\}_{K_A}$
3. $A \rightarrow B : \{N_B\}_{K_B}$

Each line in the specification indicates that a message is sent from one agent to the other, as indicated by the arrow. Hence, the first step in the protocol is for A to send B the nonce N_A as well as an identifier for A, both encrypted with B's public key. In the second step, B responds with the nonce N_A as well as a new nonce, both encrypted with A's public key. Intuitively, the idea is that A can conclude that B received the first message, because only B has the private key for decrypting this message. The protocol concludes with A responding to B's nonce, appropriately encrypted.

An attack on the Needham Schroeder protocol was discovered by Lowe [14]. Following convention, we describe the attack using the same "arrow" notation used for specifying protocols. The symbol I denotes a dishonest agent called the *intruder*. In order to clarify the attack, we use the notation I_A in situations where I is pretending to be A, either by intercepting a message intended for A or by sending a message that A would be expected to send. In the attack, A initiates a run of the protocol with I, but

then I uses the nonce generated by A to start a new run with B.

Needham Schroeder Protocol Attack

1. $A \rightarrow I : \{N_A, A\}_{K_I}$
1′. $I_A \rightarrow B : \{N_A, A\}_{K_B}$
2′. $B \rightarrow I_A : \{N_A, N_B\}_{K_A}$
2. $I \rightarrow A : \{N_A, N_B\}_{K_A}$
3. $A \rightarrow I : \{N_B\}_{K_I}$
3′. $I_A \rightarrow B : \{N_B\}_{K_B}$

By interweaving protocol runs, in this attack I is able to get A to decrypt the challenge nonce sent by B. Hence, at the conclusion of this trace, B believes that it has been communicating with A, when it has not.

2.2 Logic-Based Verification

Logical methods have been used to prove the correctness of cryptographic protocols. The basic idea is to encode the steps of a protocol as formulas in a logic, and also to formalize the goals of the protocol in the same logic. A protocol is then shown to be secure by proving that the goal is logically entailed by the formulas that correspond to the steps of the protocol. The first logic defined explicitly for protocol verification was the BAN logic of Burrows, Abadi and Needham [3]. This work has been highly influential for two main reasons. First, it illustrates that protocol verification can be reduced to formal reasoning in a logic. Second, it demonstrates the significance of knowledge and belief when proving protocol correctness. Although BAN logic itself has several known flaws, it has lead directly to the development of several more sophisticated protocol logics. In many cases, such logics are based on the well-known "runs and systems" model of message passing [6].

Virtually all logic-based approaches to protocol verification use the *Dolev-Yao intruder model*, which specifies that an intruder can read, intercept and forward the messages sent between honest agents [5]. In this model, messages are received with no proof of authorship, which means that a message recipient is never aware of the sender, except possibly via the contents of the sent message. Encryption is assumed to be unbreakable, so an intruder can never decrypt an encrypted message without the appropriate key.

In contrast to the bulk of the work in protocol verification, our approach emphasises a declarative representation. One declarative formalism that has been employed for protocol verification in the past is the situation calculus [13]. The first work in this area consisted of a direct encoding of message passing in the presence of a Dolev-Yao intruder [12]. In this model, message interception and forwarding was modeled by means of non-deterministic effects for message sending. More recently, in a companion to the present paper, we have introduced a new situation calculus model of cryptographic protocols [4]. In this approach, we introduce an explicit intruder model, in which message interception and forwarding are actions executed by an intruder. This approach is discussed further in the final section.

As well, there has been previous work in using ASP for protocol verification. Two distinct, yet similar, ASP models have been proposed in [1, 15]. The basic idea in these

approaches is to provide a template for defining a logic program that represents a specific protocol. Answer sets correspond to sequences of exchanged messages, which can then be analysed with respect to a set of explicitly defined attacks. Since attacks are specified in advance, this approach cannot readily be used to find "new" or "unexpected" attacks on a protocol. In contrast, in our approach, we define attacks through user-specified queries over the set of protocol runs. Another distinguishing feature of our approach is that we present modular definitions of message passing, intruder capabilities, and protocol structure. Hence, we define a formalism that is more flexible and elaboration tolerant.

2.3 Motivation

The standard specification of protocols, as illustrated by the Needham-Schroeder Protocol above, is both incomplete and imprecise. It is incomplete, in part, because the goal of the protocol is not explicitly given. Without a precise goal for a protocol, it is difficult to determine if a given trace constitutes an attack. The specification of protocols is ambiguous because it is not clear exactly what $A \rightarrow B$ actually means; clearly it does not mean that A successfully sends the message to B. Instead, it seems to conflate two things:

1. A *intends* to send the message to B.
2. A actually sends the message *to someone*.

This interpretation suggests that the intentions of a sender are relevant for protocol verification. Nevertheless, intentions are often left implicit in many approaches to protocol verification. One of the main motivations of this paper is to provide a more precise and explicit description of what is intended in a cryptographic protocol specification. An advantage of our approach is that we specify protocols and protocol goals in a declarative manner, which allows us to critically examine exactly what constitutes an attack.

One problem with existing formal models for protocol verification is that they implicitly assume that honest agents do not do anything irresponsible or dangerous. For example, honest agents do not send plain text messages that compromise secret keys. While this example is obvious, it may not be clear which other actions are irresponsible. By specifying attacks in terms of flexible queries on the set of traces satisfying a protocol, we are able to discover which actions should be avoided by an honest agent.

Lastly, we are interested in comparing alternative declarative paradigms for cryptographic protocol analysis, in particular, and multi-agent message passing, in general. Consequently we have been developing in parallel an ASP approach along with a situation calculus encoding. It is not immediately obvious, however, how to choose between ASP and a formalism such as the situation calculus. In the discussion, we briefly consider the relative merits of each approach.

3 Approach

Our aim is to define a logic program in which the answer sets correspond to sequences of exchanged messages between agents. We will refer to the honest agents as *principals*,

in contrast to the *intruder* whose goal is to disrupt communication between principals. The formalization consists of three essentially independent modules. One module includes generic information about message passing and protocols; one module includes a specification of the intruders capabilities; and the third includes the structure of a specific protocol. This approach makes the formalization very elaboration tolerant, as we are free to manipulate the message passing environment, the protocol structure, or the intruder to suit different applications. The ability to change the intruder is particularly novel in the protocol verification community, where the Dolev-Yao intruder is so entrenched. Nevertheless, there are practical examples where the power of the intruder may be altered by external issues, such as network topology.

We assume the reader is familiar with ASP, as described in [10], for example. We also make extensive use of constrained *choice rules* involving expressions of the form

```
i {p, q, r} j
```

Such expressions are understood to indicate that at least i of the enclosed atoms are true, but not more than j are true. Another kind of expression we make use of is conditions of the form `p(X) : q(X).` These expressions are used for instantiating the variable `X` with the domain predicate `q(X)` to collections of terms within a single rule. For example, if there are the facts `q(1)`, `q(2)` and `q(3)` in the program, then `i {p(X) : q(X)} j` will be grounded to `i { p(1), p(2), p(3) } j`.

3.1 Protocol Module

The protocol module sets up a general message passing framework, the principals' holdings and capabilities, as well as some high-level auxiliary predicates. It also specifies different types for variables.[1]

Keys and Nonces: Principals need keys for encryption. Our approach supports both asymmetric and symmetric key encryption. The protocol module is responsible for setting up the key infrastructure. It specifies keys for every agent and distributes the public or the shared keys to the appropriate principals. In order to guarantee the freshness of messages, certain protocols require nonces or timestamps in messages. Due to the propositional nature of our formalism, it is not possible to have every agent create an unlimited number of nonces on demand. Instead, a sufficient number n of nonces is assigned to each principal. For performance reasons, n defaults to 1, but can also be set as a command line parameter. In the initial state all nonces are fresh at the outset; and no messages have been sent or received.

Actions: Protocol analysis is focused on analyzing sequences of actions. Clearly, *send* and *receive* actions are central, although these need not be the only actions. Actions occur at some point in time, and the notion of time is abstracted into slices or time steps. In order to keep the search space manageable, internal actions related to message composition and encryption are implemented through state constraints. This means that

[1] Of course these declarations are eventually grounded, but it is useful to think of the variables as ranging over different classes of entities.

a single time step is used for one agent to receive a message, decrypt its contents, compose a reply, encrypt and send it. The reply may then be received by a recipient in the following time step. The maximum number of time steps is also set as a parameter. Sending and receiving of messages is modelled as events or actions that are triggered once all preconditions are satisfied. A message is sent as soon as a principal has the message and wants to send it.[2]

```
send(A, B, M, T) :- wants(A, send(A, B, M), T),
                    has(A, M, T).
```

If the message does not get intercepted by the intruder, it will be received in the next time step.

```
receive(A, B, M, T+1) :- send(B, A, M, T),
                         not intercept(M, T+1).
```

The sending action effectively acts as a precondition for receiving, while an effect of the receive action is that the recipient possesses the message afterwards.

```
has(A, M, T) :- receive(A, B, M, T).
```

If a principal receives a message as part of a particular protocol, then they want to send the appropriate response. The notion of "appropriateness" is modelled by the `fit` predicate that matches certain message components.

```
{ wants(A, send(A, B, M), T) } 1 :- receive(A, B, M2, T),
    fit(msg(J, B, A, M2), msg(J+1, A, B, M)).
```

Auxiliary predicates In order to be able to specify flexible goal and attack conditions in the instance module, we provide several auxiliary predicates. Some of these predicates are straightforward, such as the predicate `talked` to indicate that two agents have successfully communicated:

```
talked(A, B, T+1) :- send(A, B, M, T),
                     receive(B, A, M, T+1).
```

In contrast, the predicate `authenticated`, which is used as part of a goal specification, is much more complex. We say that A is *authenticated* with B only if A has sent a fresh "challenge nonce" encrypted with an appropriate key to B; B has to have replied to A's challenge with the same nonce, again encrypted with a key so that only A can decrypt it. As well, A received B's reply and all events happened in the right order. The predicate is given as follows:

```
authenticated(A, B, T) :-
    send(A, B, enc(M1, K1), T1),
    fresh(A, nonce(A, Na), T1),
    part_m(nonce(A, Na), M1),
```

[2] Note that this and following code examples may not be complete. On occasion, details, such as domain restrictions, are omitted for readability and brevity.

```
key_pair(K1, Kinv1), has(A, K1, T1),
has(B, Kinv1, T1),
not has(C, Kinv1, T1) : agent(C) : C != B,
send(B, A, enc(M2, K2), T2),
receive(A, B, enc(M2, K2), T),
part_m(nonce(A, Na), M2),
key_pair(K2, Kinv2), has(B, K2, T1),
has(A, Kinv2, T1),
not has(C, Kinv2, T1) : agent(C) : C != A,
T1 < T2, T2 < T.
```

Authentication is generally a difficult concept to define, and different notions of authentication may be required for different protocols. One advantage of our approach is that it is easy to change the definition of authentication to suit a particular application or a particular audience. Also, the protocol module can easily be extended by other auxiliary predicates that might be useful for a wide range of protocols.

3.2 Intruder Module

The intruder module specifies all aspects of the intruder. As noted, an advantage of our approach is that the intruder model is flexible and easy to modify. If our system is run without the intruder module, then it simply computes all valid protocol runs between honest principals within the given time bound. If an intruder module is given, then the system can be used to find attacks on the protocol that may be carried out by the intruder.

Holdings: By default, the intruder holds the public key of every agent. The intruder also holds a public-private key pair. It is important that the intruder has this pair, since it allows the intruder to "pretend" to be an honest agent that initiates protocol runs with principals. Since the intruder module can be modified, it is straightforward to model, for example, the situation in which an intruder has obtained a key belonging to someone else. This is an important improvement over existing approaches to verification, as it allows us to give customized proofs of security. For example, we may be able to prove that a certain protocol is secure, even if one of the participants has a compromised private or shared key.

Capabilities: The intruder module specifies the capabilities and limitations of the intruder. In general, we specify a Dolev-Yao type intruder, because this makes it easier to compare our approach with existing approaches. Hence, we specify that the intruder can intercept messages and that it receives the messages that it intercepts:

```
0 { intercept(M, T+1) } 1 :- send(A, B, M, T).
receive(I, A, M, T+1) :- send(A, B, M, T),
                         intercept(M, T+1).
```

The intruder can also send messages whenever it wants to. However, it can not send messages that it does not have:

```
:- send(I, A, M, T), not has(I, M, T).
```

Of course, the intruder can also fake the sender name of messages it sends:

```
1 { receive(A, B, M, T+1) : principal(B) } 1 :-
    send(I, A, M, T).
```

This will make the receive action of principal A look like it obtained the message from a principal B.

Again, the capabilities we have listed here are examples that are useful for many types of protocols. Using interface predicates from the protocol module and standard ASP syntax, it is possible to specify a wide range of different capabilities. For example, one might want to specify that an intruder can only eavesdrop on messages that arrive at or originate from one specific principal.

3.3 Instance Modules

In order to verify a particular protocol, we need to provide an encoding of that protocol in ASP. This encoding is done in the instance module. Currently, we have encoded two protocols: the Needham Schroeder Protocol and the Challenge Response Protocol. The structure of the ASP encoding is straightforward; indeed the next step in the project is to automate the process, so that, given a simple "arrows" specification like that for the Needham-Schroeder protocol, a corresponding instance module in ASP is produced. At present, however, the encodings are done by hand.

In the beginning of the instance module, certain options such as the maximum number of concurrent protocol runs have to be set. The principals participating have to be added, too. This is done by simply including several interface predicates of the form `principal(a)`. The instance module for a specific protocol also has to state which kind of encryption should be used. For example, public-key encryption is activated if the fact `set(pub_key_enc)` is added. All the details about the encryption are handled by the protocol module and is of no concern to the instance module.

The instance module has four main components: a specification of valid messages, completion conditions, the goal of the protocol, and attack conditions. We illustrate how each component is implemented, using the Needham Schroeder protocol as an example.

Messages: In order to model message passing in ASP, it is useful to bound the number of messages that can be exchanged. In practice, the set of messages is of course not bounded – given a composition operator and an encryption operation, it is possible to define an infinite set of messages. However, if we fix a specific protocol, then it is easy to specify the set of messages that can be used in a valid run. In the Needham Schroeder protocol, for example, there are three basic message forms that can be sent, one for each line of the protocol. However, each message can involve any nonce that is available and it can involve the public key of any agent on the network. In our framework there are only a finite number of nonces and agents, so it is easy to delimit the class of messages.

The protocol itself can then be described by a valid subset of possible messages:

```
msg(1,A,B,enc(m(nonce(C,Na),principal(A)),pub_key(B))).
msg(2,B,A,enc(m(nonce(C,Na),nonce(D,Nb)),pub_key(A))).
msg(3,A,B,enc(m(nonce(D,Nb)),pub_key(B))).
```

These rules correspond to the first, second and third message of the protocol.

Completion Conditions: Normally, a protocol run is completed if all of the messages have been sent and received. We define the completion conditions at the agent level, which means we need to specify when each participating agent *believes* the protocol is completed. To achieve this, we introduce a general `believes` predicate that is also used to represent other beliefs of the principals. If a principal has sent and received the appropriate messages in the right order, it will believe that it successfully completed a valid run of the protocol with another principal. Unfortunately, it was not possible to generalize this condition for every protocol in the protocol module. However a compiler could easily create this rule from the protocol specification.

Goals: As we saw, the protocol module includes a set of high-level meta-predicates that can be used to define specific goals. In the Needham Schroeder protocol instance, the goal is that both principals should believe that the other one is authenticated and that they actually are.

```
goal(A, B, T) :- authenticated(A, B, T),
                 believes(A, authenticated(A, B), T),
                 authenticated(B, A, T),
                 believes(B, authenticated(B, A), T).
```

This is essentially a definition of "mutual authentication", following the BAN tradition in which we only consider nested beliefs to depth two. As noted earlier, the concept of authentication is difficult to define and there might be disagreements about the definition used here. However, it is easy to change the definition of mutual authentication while leaving the rest of the encoding and the other modules unchanged.

Attacks: An attack on a protocol is a run of the protocol that causes an agent to believe the goal is true, when in fact the goal is not true. In the case of the Needham Schroeder protocol, an attack is specified in terms of a principal believing that it is authenticated, but is in fact not authenticated.

```
attack :- believes(A, authenticated(A, B), T),
          not authenticated(A, B, T).
```

In this case, the principal's belief is false and must have been created somehow. Our system can give an explanation of how this happened by showing the protocol trace that led to the false belief. This is done by including the integrity constraint `:- not attack` in the protocol instance. Then all answer sets of our system will represent an attack, because traces that are not an attack are excluded.

3.4 Protocol Verification

Given the three modules of our ASP encoding, an ASP solver can be used to find attacks on a given protocol. For our implementation, we ground our encoding using *GrinGo* [9] and compute the answer sets using *clasp* [8]. For simplicity, we use the hybrid program *Clingo*[3] to perform both tasks in one step. Also, in order to format the display of the output, we employ a Python script `output.py` that makes the results more readable. Therefore, at the command line, we can verify the Needham Schroeder protocol by executing the following command.

```
clingo protocol.lp intruder.lp needham_schroeder.lp \
    -c n=1 -c t=6 0 | output.py
```

Note that you have to specify the number of nonces n that each agents has initially and the maximum trace length t as parameters. We also provide run scripts, so users do not need to remember the entire command line. These scripts just take n and t as optional parameters that are initialised with reasonable default values.

The above command will output attacks on the Needham Schroeder protocol in the following format:

```
send(a,i,enc(m(nonce(a,1),principal(a)),pub_key(i)),0)

receive(i,a,enc(m(nonce(a,1),principal(a)),pub_key(i)),1)
send(i,b,enc(m(nonce(a,1),principal(a)),pub_key(b)),1)

receive(b,a,enc(m(nonce(a,1),principal(a)),pub_key(b)),2)
send(b,a,enc(m(nonce(a,1),nonce(b,1)),pub_key(a)),2)

receive(i,b,enc(m(nonce(a,1),nonce(b,1)),pub_key(a)),3)
send(i,a,enc(m(nonce(a,1),nonce(b,1)),pub_key(a)),3)

receive(a,i,enc(m(nonce(a,1),nonce(b,1)),pub_key(a)),4)
send(a,i,enc(m(nonce(b,1)),pub_key(i)),4)

receive(i,a,enc(m(nonce(b,1)),pub_key(i)),5)
send(i,b,enc(m(nonce(b,1)),pub_key(b)),5)

believes(b,authenticated(b,a),6)
believes(b,completed(b,a),6)
receive(b,a,enc(m(nonce(b,1)),pub_key(b)),6)
```

The attack can be read directly from this output. To improve readability, it would be straightforward to incorporate an additional script to transform this output into the usual "arrow" syntax enriched with the `believes` predicates.

Our system employs a minimize statement by default to always return the shortest attack. But it can easily be reconfigured to return all attacks that are possible in the given trace length.

[3] `http://potassco.sf.net`

It addition to Lowe's attack on the Needham Schroeder protocol, our system discovered several other attacks that become possible if the agents' capabilities are not circumscribed. The simplest such attack is what we have called the "Stupidity Attack" in which one principal sends an unencrypted nonce to the intruder. A more sophisticated attack we found was dubbed the "Bad Memory Attack", because it only works if the principals are unable to remember the current state of a protocol run.

Bad Memory Attack

1. $A \rightarrow B : \{N_A, A\}_{K_B}$
2. $B \rightarrow I_A : \{N_A, N_B\}_{K_A}$
2′. $I \rightarrow A : \{N_A, N_B\}_{K_A}$
3′. $A \rightarrow I : \{N_B\}_{K_I}$
3. $I_A \rightarrow B : \{N_B\}_{K_B}$

In this attack, A starts the protocol regularly with B. The second message is intercepted by the intruder who uses this message to send it as challenge to A in line with the second step of the protocol. A has a bad memory and does not realize that he did not initiate the protocol with I. So A replies to the challenge, effectively decrypting the nonce N_B for the intruder, who is then able to use it to fool B.

Our system produces attacks for the Challenge Response and Needham Schroeder protocol in less than two seconds. If each agent is given more than one nonce, the runtime increases significantly.[4] This seems to be partly caused by the grounding process. Nonces are involved in almost every ASP rule of our encoding and are responsible for the creation of exponentially many ground rules. Another reason for the increased overall runtime of our system when using more nonces or a larger plan length is the bigger search space. The option of minimizing the number of sent messages forces Clasp to almost traverse the entire search space to make sure that no smaller trace exists. Incremental grounding and solving as described in [7] might be useful for coping with the blowup caused by the plan length and the search for a minimal trace length of an attack. This is a topic for future work.

4 Discussion

4.1 Comparison with Related Formalisms

We have already mentioned previous work on declarative approaches to protocol verification in the situation calculus [12] and in ASP [1, 15]. Our approach is more flexible than these approaches in the sense that different aspects of the model can be modified, while maintaining a consistent overall framework. Our work is also distinguished by the fact that we can detect unexpected "attacks" on a protocol.

As well we mentioned a companion paper that formulates cryptographic protocol verification in terms of a situation calculus theory [4]. Both of our approaches are, or are in the process of, being implemented. The situation calculus approach allows a more

[4] We obtained the following times with the Needham Schroeder protocol: 3 nonces: 2 sec, 6 nonces: 13 sec, 9 nonces: 51 sec, 12 nonces: 190 sec.

procedural specification, in that agent actions are defined and "executed". As well, it allows a more refined specification of control and message passing. On the other hand, the situation calculus approach will require more work with regards to an implementation; we anticipate making use of more procedural constructs as found in ConGolog. For the ASP approach described here, implementation was more straightforward and a prototype was readily obtained[5]. The biggest issue in the ASP implementation was controlling the state explosion that resulted from grounding. Performance issues arise when the number of agents, nonces or concurrent protocol runs is increased.

Last, it is interesting to note that our original intention was to formulate our theory not directly in ASP, but rather in the action language $\mathcal{C}$ [11]. Unfortunately, $\mathcal{C}$ turned out to be problematic for several reasons, notably the fact that action effects must be Markovian. In reasoning about protocol exchanges, it is inevitably necessary to express effects due to a certain sequence of actions. Neither the description nor the query language of $\mathcal{C}$ support this. As a result, we found that it was easier and more straightforward to encode protocols directly in ASP.

4.2 Conclusion

We have described a formulation and implementation of cryptographic protocol analysis and verification in ASP. A primary advantage of using ASP is that it provides a declarative formalism for which efficient implementations exist. As a result we were able to specify a framework that is flexible, declarative, and elaboration tolerant. The approach provides for flexible models of the intruder and principals, and the specification of goals at an intensional level in terms of an agent's beliefs. The implementation discovered not just the standard known attacks, but also other possible attacks, among them the Bad Memory Attack and the Stupidity Attack. Both of these latter attacks are easily addressed; however the detection of such attacks for a simple protocol suggests that, in general, there may be other, more subtle attacks that a declarative formalism, with a full specification of agent actions, may be able to detect.

There have been challenges in the implementation, primarily in getting the implementation to scale up. Increasing the number of agents, the number of available nonces, and the suite of actions available to an agent all provide challenges. For future work, we will first continue to develop our implementations. Concurrently we are also working on a compiler, both for ASP and the situation calculus approaches, that will take an arbitrary protocol and produce an executable theory. Hence, in a broader potential contribution, this project will provide a testbed for comparing approaches implemented in a constraint formalism such as ASP, with a more planning-oriented approach such as the situation calculus.

References

1. Luigia Carlucci Aiello and Fabio Massacci. Verifying security protocols as planning in logic programming. *ACM Trans. Comput. Logic*, 2(4):542–580, 2001.

[5] The prototype is available for download from `http://www.cs.sfu.ca/~cl/software/ASP.SP/`.

2. C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.
3. Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, 1990.
4. James P. Delgrande, Aaron Hunter, and Torsten Grote. Modelling cryptographic protocols in a theory of action. In *IJCAI Workshop on Non-Monotonic Reasoning, Action and Change*, 2009.
5. D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.
6. Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. The MIT Press, Cambridge, Massachusetts, 1995.
7. M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.
8. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In Baral et al. [2], pages 260–265.
9. M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In Baral et al. [2], pages 266–271.
10. Michael Gelfond. Answer sets. In *Handbook of Knowledge Representation*, pages 285–316. Elsevier, 2007.
11. Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3, 1998.
12. J. Hernández-Orallo and J. Pinto. Formal modelling of cryptographic protocols in situation calculus. (Published in Spanish as: Especificación formal de protocolos criptográficos en Cálculo de Situaciones, *Novatica*, 143, pp. 57-63, 2000), 1997.
13. H.J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998.
14. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer-Verlag, 1996.
15. Shujing Wang and Yan Zhang. A logic programming based framework for security protocol verification. In Aijun An, Stan Matwin, Zbigniew W. Ras, and Dominik Slezak, editors, *ISMIS*, volume 4994 of *Lecture Notes in Computer Science*, pages 638–643. Springer, 2008.