Lazy Infinite-State Analysis of Security Protocols

David Basin

Institut für Informatik,
Universität Freiburg, Germany
basin@informatik.uni-freiburg.de

Abstract. Security protocols are used to exchange information in a distributed system with the aim of providing security guarantees. We present an approach to modeling security protocols using lazy data types in a higher-order functional programming language. Our approach supports the formalization of protocol models in a natural and high-level way, and the automated analysis of safety properties using infinite-state model checking, where the model is explicitly constructed in a demand-driven manner. We illustrate these ideas with an extended example: modeling and checking the Needham-Schroeder public-key authentication protocol.

1 Introduction

The increasing popularity of distributed computing and applications like internet banking and electronic commerce has created both tremendous risks and opportunities. Many of the risks stem from security breaches, which can be ruinously expensive. One of the cornerstones of security is the use of *security* (or *cryptographic*) *protocols* in which information is exchanged that is intended to provide security guarantees such as authentication or atomicity of cash/goods transactions. Although such protocols typically involve each agent sending only a few messages, they are extremely difficult to get right. Designing correct protocols has been likened to "programming Satan's computer" [1] as the protocol should work in the presence of a hostile, powerful, opponent who can read and alter messages at will.

In response to this challenge, various formal methods have been proposed for analyzing security protocols. Many of them are based on either model checking or interactive verification. In model checking[1], systems are modeled as (finite-state) transition systems, where states model protocol events or agent knowledge [7, 12, 16, 17]; the state space can then, at least in theory, be completely analyzed to determine if a desired property holds. Model checking methods are powerful and automatic, but their use as decision procedures often requires strong assumptions to bound the information that is analyzed. This is problematic because there are

[1] We use the term *model checking* in a general way that includes approaches based on explicit state enumeration, e.g. [4], as well as temporal logics and automata theory.

infinitely many messages that attackers can possibly send. An alternative is to develop specialized logics [2, 11] or approaches for reasoning about protocols that do not bound the possible messages sent. For example, one can model protocols as sets of possible communication traces, where messages of unbounded size can be sent [15]. The disadvantage is that this results in an undecidable formalism and verification requires interactive theorem proving, which demands considerable effort.

In this paper, we present a new approach to protocol analysis that combines complementary aspects of model checking and verification. We show how, in the appropriate setting, the kinds of formal models that are used for interactive verification can also be used for automatic, infinite-state, model checking.

The key idea is that we use *lazy data types* to model the infinite state-space associated with a protocol. A lazy data type is one where data-type constructors (e.g., *cons* for building lists, or *node* for building trees) build data types without evaluating their arguments [6]; this allows us to represent and compute with infinite data (e.g., streams or infinite trees), generating as much of the data as is needed on demand. We use lazy data types to build a model and to compute with it afterwards. We formalize a protocol and attacker model (a description of the powers of attackers) as an infinite tree and perform infinite-state model checking using standard search algorithms combined with heuristics that prune and reorder the infinite tree in a demand driven fashion.

The semantic formalism we use for modeling protocols and attackers is a trace-based interleaving semantics, motivated by, and closely following, the account given by Paulson in [15]. Paulson models a protocol as an inductively defined sets of traces, where each trace is consistent with the protocol and the chosen attacker model. He uses these models for verification: he interactively proves, by induction, that violations of security properties (i.e., some bad situation, such as a spy learning the key of an honest agent) cannot occur in any trace. In our work, rather than formalizing protocols as inductively defined sets, we formalize them as infinite trees. The nodes of the tree are traces and children correspond to trace extensions by a step of (some run of) the protocol or an action by an attacker. Hence, a protocol, along with an attacker model, defines an infinite tree and a security property is a property of nodes in the tree. Violations of security properties are found by a kind of infinite-state model checking, which is performed by searching the infinite tree.

Formalizing inductively defined sets as lazy trees makes it easy to represent the search space in a structured way. During search we explore the infinite tree, constructing finite prefixes of it "on the fly". The result is a formalism, with a clear semantic foundation, that we can directly use for automatic property checking. Moreover, the use of lazy trees makes it easy to incorporate heuristics into model checking: we use heuristics to lazily reorder the infinite tree, producing dramatic speedups in subsequent search.

The remainder of this paper is organized as follows. In section 2 we provide an overview of the semantic formalism used in this paper and some background on lazy data types and Haskell, a lazy language that we use for our work. We

$$
\begin{array}{lll}
s_1) & A \to B: & \{A, N_A\}_{K(B)} \\
s_2) & B \to A: & \{N_A, N_B\}_{K(A)} \\
s_3) & A \to B: & \{N_B\}_{K(B)}
\end{array}
$$

Fig. 1. Needham-Schroeder public-key authentication protocol

also introduce our running example: the Needham-Schroeder public-key authentication protocol. In section 3 we explain how we formalize models of protocols and in section 4 we show how to carry out model checking with them. We also discuss heuristics and present experimental results. Finally, in section 5, we draw conclusions.

2 Background

2.1 From protocols to traces

A protocol is a recipe that describes how *agents* (or *principles*) should act to achieve some goal. Protocols are often described using informal notation, for example as a sequence of instructions explaining the actions taken by the agents. Figure 1 is a typical textbook account of a protocol, in this case, a version of an authentication protocol proposed by Needham and Schroeder [14]. The protocol consists of three steps (s_1–s_3) in which two agents, A and B, exchange messages in order to mutually authenticate each other.

Each step describes an *event* $A \to B: X$, which states that A exchanges the message X with B. Messages consist of atoms, like agent names and nonces (randomly generated strings), and are composed by tupling. Moreover, messages may be encrypted using keys of agents. Here, in the first step, A identifies himself and sends a nonce N_A to B. The entire message is encrypted with B's public key. In the second step, B sends N_A back to A, along with his own nonce N_B, encrypted with A's public key. Sending N_A back authenticates B: for only B could return N_A, at least if we assume (1) perfect encryption, (2) that only B knows his private key, and (3) that other agents cannot simply guess the nonce N_A. B also sends along his own challenge, the nonce N_B, which A returns in the third step, thus demonstrating that she is really A.

Although security protocols are small and appear intuitive, this appearance is deceptive. The above protocol was proposed in 1978 and twenty years went by before [10] discovered that, contrary to what was believed, the final step of the protocol does not authenticate A; it is possible for B to finish a run of the protocol with an agent who is other than she claimed to be in the first step. These kinds of errors, which can be exceedingly subtle, motivate the need for formal analysis. By giving a protocol a formal semantics we can then make meaningful statements about whether it has desired properties. Moreover, a formal semantics is the basis for rigorous analysis based on verification or, in our case, model checking.

$$\frac{}{\langle\rangle \in P}\ empty \qquad \frac{t \in P \qquad N_A \notin used\ t}{t,\ \mathbf{A} \rightarrow \mathbf{B} : \{\mathbf{A}, \mathbf{N_A}\}_{\mathbf{K_B}} \in P}\ s_1$$

$$\frac{t \in P \qquad N_B \notin used\ t \qquad \mathbf{A}' \rightarrow \mathbf{B} : \{\mathbf{A}, \mathbf{N_A}\}_{\mathbf{K_B}} \in t}{t,\ \mathbf{B} \rightarrow \mathbf{A} : \{\mathbf{N_A}, \mathbf{N_B}\}_{\mathbf{K_A}} \in P}\ s_2$$

$$\frac{t \in P \qquad \mathbf{A} \rightarrow \mathbf{B} : \{\mathbf{A}, \mathbf{N_A}\}_{\mathbf{K_B}} \in t \qquad \mathbf{B}' \rightarrow \mathbf{A} : \{\mathbf{N_A}, \mathbf{N_B}\}_{\mathbf{K_A}} \in t}{t,\ \mathbf{A} \rightarrow \mathbf{B} : \{\mathbf{N_B}\}_{\mathbf{K_B}} \in P}\ s_3$$

$$\frac{t \in P \qquad X \in synthesize\ (analyze\ (sees\ t))}{t,\ Spy \rightarrow B : X \in P}\ attacker$$

Fig. 2. Protocol as inductively defined set

One way to model a security protocol, which is also popular in modeling other kinds of protocols, is to abstract away any possible implementation of the steps and instead to focus on communication between agents: one describes, like in figure 1, the externally observable events that take place and the order in which they occur. No further assumptions are made such as when an event occurs, or how the events of different runs of the protocol between different agents in a network are temporally ordered. Hence, a natural model of a protocol is the set of all possible *traces*, that is, the event sequences that can result from any interleaving of (possibly partial) runs.

It is a relatively straightforward to build a formal model based on this idea. This has been done by Paulson, who completely formalizes such a model in higher-order logic, which he uses for machine supported verification. In his work, a protocol plus an attacker model corresponds to an inductively defined set of traces. Such an inductively defined set can be presented by a collection of rules and is the smallest set closed under application of the rules.

Figure 2 contains the rules Paulson uses to formalize the Needham-Schroeder protocol. Explaining this in detail would take us too far off topic, but the idea is simple enough. The rules define a set of traces P, which constitute the semantics of the protocol. The rules should be read that when the premises (above the line) hold, then the conclusion (below the line) also holds. They formalize how, when certain conditions hold, traces can be extended with new events.

The rule *empty* starts off the inductive definition: the empty trace always belongs to P. This models a system where no communication has (yet) taken place. The rules s_1–s_3 formalize the identically named steps of the protocol. For example, s_1 formalizes that any trace $t \in P$ can be extended ("," is used to extend the trace t by an event) by the event $A \rightarrow B : \{A, N_A\}_{K_B}$. That is, independent of what events have come before, any agent may start a run of the protocol with any other (here terms like A and B are variables that range over all agents). The premise $N_A \notin used\ t$ formalizes that the nonce N_A is fresh, i.e., it doesn't appear in any previous event in the trace t. We have used **boldface**

font to highlight the similarity with the steps in figure 1. The rules s_2 and s_3 explain how, provided the previous steps have occurred (i.e., are contained in the trace), the next step can occur. Note that in s_2 (and similarly in s_3), for B to say $\{N_A, N_B\}_{K_A}$ to A, we require that B received $\{A, N_A\}_{K_B}$, but not necessarily from A. It could be any agent, masquerading as A. Said another way, B cannot, from this message alone, determine who sent it — this is the whole point of authentication! The rule *attacker* formalizes a commonly used attacker model due to Dolev and Yao [5]: the spy can say (and hence augment any trace t) anything that he can synthesize from analyzable parts of messages that he can see. The auxiliary functions *synthesize*, *analyze* and *see* are defined over traces and sets of messages, and we will shortly give our own definitions of them.

To summarize, the rules formalize how traces can be extended with new events in a way consistent with the protocol and together they define the set of all possible communications where protocol runs can be interleaved with each other as well as attacker broadcasts. Hence a protocol corresponds to a set of traces and this provides a basis for formal analysis. A protocol then has [or lacks] some property, precisely when the property holds [or fails] for every [some] trace in the corresponding trace set.

2.2 Lazy Data Types and Haskell

In verification, we can formalize and reason about infinite sets. However, the kinds of constructs available for doing so (e.g., inductive definitions or set comprehension) are usually not available in programming languages where expressions should be computable. However, programming languages offer other possibilities for representing infinite objects: we can write functions that enumerate the elements of infinite sets, at least until we run out of memory or patience. In this sense, a function can represent an infinite set or infinitely large data.

Lazy data types provide a principled way of representing infinite data using functions that can generate arbitrarily large finite prefixes on demand. The term *lazy* derives from the mechanism of lazy evaluation, which ensures that expressions or components of structures are expanded in a demand driven way and are not evaluated more than is necessary to provide a value at the "top level". In a lazy functional programming language this allows us to write recursive definitions that represent infinite data. For example, if ":" is the "cons" constructor for building lists, the two equations

```
nat = from 0
from n = n : from (n+1)
```

define `nat` to be the infinite list `[0,1,2,3,4, ...]`. The cons constructor can be viewed as a function that does not evaluate its arguments until they are required. Hence the elements of this list are generated on demand; only when we ask for the head of the list is the definition of `from` unfolded to generate the first element 0. Evaluation of the remainder (the tail) is again delayed, until computation forces further evaluation.

Lazy data types are extremely useful. We use them to specify the infinite trace sets associated with protocols. We execute these specifications and functions over them using Haskell [8], which is a lazy, higher-order, polymorphically typed, functional programming language. The equations for `nat` given above, and indeed all text in this paper in `typewriter font`, are Haskell programs.

An introduction to Haskell is outside the scope of this paper. However most aspects of Haskell should be clear from the examples given, at least if the reader is familiar with modern functional programming languages. We briefly mention one feature though, which turned out to be very useful for describing protocols. Haskell supports a notation for specifying (possibly infinite) sets using *list comprehension*, which is analogous to *set comprehension*. Sets in Haskell are represented by lists. For example we can represent the set

$$\{2 \times x \mid 1 \leq x \leq 10 \wedge x \; mod \; 2 = 0\}$$

in Haskell as

```
[2 * x | x <- [1 .. 10], x `mod` 2 == 0]
```

which is equal to $\{4, 8, 12, 16, 20\}$. The notation `[c | t <- xs, p]` represents a set where for each `t` in the list `xs`, for which the predicate `p` holds, we add an element `c` to the result set. In general, there can be more than one *generator* (expressions like `t <- xs`) as well as zero or more predicates. Moreover, in a generator, the term `t` can be a composite term, called a *pattern*, which is matched against each element of `xs`. Due to technical reasons, patterns in Haskell must be *linear*, which means that each variable in `t` can occur just once. One can often work around this by renaming common variables apart and adding predicates that equate them, e.g., translating the generator `[x,x] <- xs` (which selects all doubleton lists from `xs` whose elements are identical) to `[x,x1] <- xs, x == x1`. We will see applications of this shortly.

3 Model building

In the previous section we have described two ideas: how a protocol can be modeled as an infinite set of traces and how to represent and compute with infinite data. We now put these ideas together and show how to formalize protocol models using lazy data types. For concreteness, we return to our running example, the Needham-Schroeder protocol; the ideas are more general.

As observed in the introduction, our formalism is based on Paulson's, except we formalize inductive definitions in a way that we can directly compute with. Moreover, rather than formalizing an infinite set, we formalize an infinite (trace) tree. The difference is slight: each node in the tree is labeled with a trace in the inductively defined set of traces and the layers of the tree correspond to stages of the inductive definition. The advantage over a formalization based on implementing infinite sets as lazy lists is that it is easier to compute the closure of a set represented as a tree since, for a monotone closure operator, the elements

```
s1 t = [(Says a b (Crypt b (Pair (Agent a) (mkNonce t) )))
          | a <- [Alice,Bob,Spy],  b <- [Alice,Bob,Spy], a /= b]

s2 t = [Says b a (Crypt a (Pair (Nonce na) (mkNonce t))) |
          Says _ b (Crypt b1 (Pair (Agent a) (Nonce na) )) <- t,
          b == b1]

s3 t = [Says a b (Crypt b (Nonce nb)) |
          Says a b (Crypt b1 (Pair (Agent a1) (Nonce na) )) <- t,
          a == a1, b == b1,
          Says _ a' (Crypt a1' (Pair (Nonce na') (Nonce nb))) <-t,
          a == a', a' == a1', na == na']

attacker t = [Says Spy a msg | a <- [Alice,Bob],
           msg <- synth(analz Spy (sees Spy t))]
```

Fig. 3. Functions for generating the trace set

introduced at each iteration correspond to the next ply of the tree. This makes it easy to generate the set, without checking for repetition, and, as we will later see, to introduce search heuristics.

The central data type in our work is that of a tree labeled with elements of some type `a` (`a` is a type variable, so we can have trees labeled by elements of arbitrary types). To allow for arbitrarily many children, each node contains an element of type `a` and a list of zero or more trees of type `a`.

```
data Tree a = Node a [Tree a]
```

For example `Node 1 [Node 2 [], Node 17+18 []]` belongs to the type `Tree Int`. Note that `Node` is a lazy data constructor: its arguments (e.g., `17+18`) are evaluated later, only when required for computation.

We use data types to model agents, messages, and events. For the Needham-Schroeder protocol, we formalize three agents, `Alice`, `Bob`, and a `Spy`. Messages may be agent names, nonces, pairs of messages, or encrypted messages. Events are broadcasts of a message from one agent to another.

```
data Agent = Alice | Bob | Spy
data Msg = Agent Agent | Nonce Int | Pair Msg Msg | Crypt Agent Msg
data Event = Says Agent Agent Msg
```

Note that we abstract away from the details of cryptography and identify the public key of an agent with his name (and we will assume that only that agent has the inverse private key). Also, nonces are named by integers.

Figure 3 contains our formalization of the steps of the protocol and the attacker model, based on the inductive definition of figure 2. The functions correspond to the identically named rules: each function specifies how a trace `t` can be extended. For example, `s1` uses set comprehension to iterate over all pairs of agents `a` and `b` (`a` and `b` are variables ranging over agents) and states that

```
analz a hs =
  let inj  xs = xs
      fst  xs = [x | Pair x y <- xs]
      snd  xs = [y | Pair x y <- xs]
      decrypt xs = [x | Crypt ag x <- xs, ag == a]
  in closure (inj `or` fst `or` snd `or` decrypt) hs

synth hs =
  let inj  xs = xs
      agent _ = [Agent a | a <- [Alice,Bob,Spy]]
      crypt xs = [Crypt a x | x <- xs, a <- [Alice,Bob]]
      pair xs = [Pair x y | x <- xs, y <- xs]
  in (inj `or` agent `or` crypt `or` pair) hs

sees a t = foldr (\x r -> (sees1 x) `union` r) emptyset tr
  where sees1 (Says _ b x) = if a == b || a == Spy then [x] else []

or f g = \xs -> (f xs) `union` (g xs)
```

Fig. 4. Formalizing the attacker's capabilities

the event `Says a b (Crypt b (Pair (Agent a) (mkNonce t)))` can be used to extend any trace. The auxiliary function `mkNonce` generates a fresh nonce (not occurring in the trace `t`); hence this formalizes $A \to B : \{A, N_A\}_{K_B}$. The function `s2` (and similarly `s3`) formalizes that an extension corresponding to the second step of the protocol is allowed only when the trace contains the first step.

The last function, `attacker` formalizes the attacker model. This uses the auxiliary functions given in Figure 4.[2] The function `analz` decomposes messages into their analyzable parts, e.g., the parts of pairs. Looking into an encrypted messages requires the corresponding key (so we assume perfect cryptography). We use `synth` to build messages from known parts and `sees` formalizes that the spy can see all communication that has taken place (i.e., is in the trace).

To formalize the model itself, we employ a function, which given (1) an initial state, `init`, and (2) a function, `extension`, mapping states to lists of successor states, formalizes an infinite tree.

```
build_tree extension init
```

[2] In addition, `closure` takes a function `f` and a set `s` and applies `f` to `s` until a fixedpoint is reached. Not that our formalization differs here slightly with respect to Paulson's. He computes the closure in both `synth` and `analz`, whereas we only compute the closure in `analz`; the former is always infinite and the latter is always finite. Formalizing an infinite set is not a problem in a lazy setting; we could do this with a stream. However, we have avoided this as it would result in an infinitely branching tree (instead of a finite branching tree with infinite length branches), which would complicate the application of reordering heuristics. Instead, we perform just one synthesis step, and spread the closure computation down (instead of across) the infinite tree.

```
   = Node init (map (build_tree extension) (extension init))
```

The root of the tree is labeled by `init` and, at each ply, `extension` is applied to generate the successors, upon which `build_tree` is recursively applied. Depending on the extension function, `build_tree` formalizes a tree of finite or infinite depth, which is finitely or infinitely branching.

We complete our specification by applying the above function to formalize `p`, the model itself. The initial state consists of the empty trace and the extension function computes successor states by extending traces using the functions in figure 3.

```
ext t =  map (\x -> ins x t) extensions
          where extensions = ((s1 `or` s2 `or` s3 `or` attacker) t)

p =  build_tree ext []
```

The specification of the model is direct and concise. The corresponding Haskell code takes just over a page.

4 Model-checking

We have formalized an infinite-state model as a lazy tree. Here we show how to perform infinite-state model checking based on lazy state-enumeration: we can search the state space for attacks, constructing parts of the tree on demand. For protocols/attacker models that produce infinite trees, this constitutes a semi-decision procedure. It is a decision procedure when the trees are finite.

If there is an attack, it will be present in a trace located at some node in the tree `p`. Finding this node, however, may not be easy. The tree is not only infinitely deep, it has a large branching factor. For example, the first ply has 12 nodes; each node contains a singleton trace and together the 12 traces represent all the ways that any two distinct agents can start a run of the protocol as well as all the messages that the spy can send.

```
[Says Alice Bob (Crypt Bob (Pair (Agent Alice) (Nonce 0)))]
[Says Alice Spy (Crypt Spy (Pair (Agent Alice) (Nonce 0)))]
[Says Bob Alice (Crypt Alice (Pair (Agent Bob) (Nonce 0)))]
[Says Bob Spy (Crypt Spy (Pair (Agent Bob) (Nonce 0)))]
[Says Spy Alice (Crypt Alice (Pair (Agent Spy) (Nonce 0)))]
[Says Spy Bob (Crypt Bob (Pair (Agent Spy) (Nonce 0)))]
[Says Spy Alice (Agent Alice)]
[Says Spy Alice (Agent Bob)]
[Says Spy Alice (Agent Spy)]
[Says Spy Bob (Agent Alice)]
[Says Spy Bob (Agent Bob)]
[Says Spy Bob (Agent Spy)]
```

(Note that for the empty trace, the spy is unable to say much of interest — this quickly changes as the traces grow.) The second ply has 314 nodes and the

```
sortPly f t = mapBranches f t
    where mapBranches f (Node a l) = Node a (map (mapBranches f) (f l))

p' =  sortPly f (prune p)
  where f = sortBy cmp
        cmp (Node t1 _) (Node t2 _) =
               if w1 < w2 then LT
               else if w1 == w2 then EQ else GT
                      where w1 = weight (head t1); w2 = weight (head t2)
        weight e = (protStep e) + (fromSpy e)
        protStep (Says _ x (Crypt x1 (Pair (Agent _) (Nonce n) ))) =
               if x == x1 then 1 else 10
        protStep (Says _ a (Crypt a1 (Pair (Nonce na) (Nonce nb)))) =
               if a == a1 then 2 else 10
        protStep (Says _ b (Crypt b1 (Nonce nb))) =
               if b == b1 then 3 else 10
        protStep _ = 10
        fromSpy (Says a1 a2 _) =
               if (a1 == Spy || a2 == Spy) then 0 else 4
        prune t = filterBranches (\x -> protStep (head x) <= 3) t
```

Fig. 5. Heuristics for pruning and reordering the tree

third 17,529. Such an exponential branching factor means that standard search algorithms are unlikely to succeed in finding even relatively simple attacks that might reside at shallow depths. Therefore, we use heuristics both to prune the search tree as well as to reorder the way it is searched.

Figure 5 displays the heuristics we use, which are based on two simple ideas. First, the protocol specifies events of a particular format: agents taking part in the protocol only send certain kinds of messages. The spy, however, has considerably more freedom and, as a result, many traces contain events that could not have resulted from some step of the protocol and have no consequences as they will not provoke any response from honest protocol participants. The first heuristic is to prune these traces and their successors. Second, we assign priorities to events and give the highest priority to those that could arise from the first step of the protocol followed by the second and then the third. Moreover, we give events a higher priority when they involve the spy. This priority is input to a function, `sortPly`, which sorts, according to priority, the nodes on a ply.

Applying the heuristics to `p` yields `p'`. The use of lazy data types here allows us to separate search from pruning and ordering heuristics: conceptually, the heuristics map the infinite tree `p` into another, `p'`, which we can later search. In reality, lazy evaluation applies the heuristics in a demand driven way during search to explore first parts of the tree that contain potentially interesting dialogs with the spy.

We model check the improved search space using standard search algorithms. We found most useful an implementation of iterative deepening search, **ids**, that iterates bounded depth first search with bounds 0, 1, 2, ...

```
ids pred t = flatten [ idsn n pred t | n <- [0 ..]]

idsn n pred (Node a l) =
 if   n == 0 then (if pred a then [a] else [])
 else if pred a then (a:rest) else rest
      where rest = flatten (map (idsn (n-1) pred) l)
```

The result returned by `ids` is a stream (lazy list) of states for which `pred` holds.

Formalizing an appropriate instance of `pred` that characterizes security violations turned out to be surprisingly difficult. Attacks manifest themselves as traces where the spy learns secrets, or takes on identities, which he shouldn't and it is non-trivial to characterize such traces in a general, high-level, way. In our case, we encoded specialized predicates over traces to formalize violations of particular security properties. For example, we can formalize "after a run of the protocol, B fails to authenticate A" as: agent B gets a nonce from (an agent claiming to be) A in format of step 3 (so the run is complete) that he sent to a different agent C in format of step 2, which the spy can analyze.

```
attack [] = False
attack (Says a b (Crypt b1 (Nonce nb)):tr) =
 if b == b1 then (sent tr && analyze) else attack tr
 where sent [] = False
       sent (Says b2 c (Crypt c1 (Pair (Nonce _) (Nonce nb1))):esrest) =
            (b1 == b2 && nb == nb1 && c == c1 && c /= a && c /= b)
                || sent esrest
       sent (_:esrest) = sent esrest
       analyze = elem (Nonce nb) (analz a (sees Spy tr))
attack (_:tr) = attack tr
```

With all the pieces in hand, we can now model-check. The command

```
head(ids attack p')
```

applies `ids` to the optimized tree, and `head` forces the computation of the first element of the stream, which is returned as the result. The result of this search is a trace that represents a version of the "man in the middle" attack on the Needham-Schroeder protocol, first identified by [10].

```
[Says Alice Spy (Crypt Spy (Pair (Agent Alice) (Nonce 0))),
Says Spy Bob (Crypt Bob (Pair (Agent Alice) (Nonce 0))),
Says Bob Alice (Crypt Alice (Pair (Nonce 0) (Nonce 2))),
Says Alice Spy (Crypt Spy (Nonce 2)),
Says Spy Bob (Crypt Bob (Nonce 2))]
```

The trace, read top down, lists the events that constitute the attack. In this attack, Alice starts the protocol (first step) with the Spy. The Spy takes advantage of this by starting another run of the protocol with some other agent, here Bob, passing on Alice's message. When Bob responds with his own nonce (third step), Alice assumes that it came from the spy.[3] She then sends back the nonce to the Spy (fourth step) who uses it to convince Bob that he is Alice (fifth step). This attack shows that if Alice talks to someone who has the powers of an attacker, then this can be used to take on her identity. Hence, this protocol is too weak to actually authenticate the initiator.

This attack, which involves a trace of length 5, is found on the fifth ply of `p'`; the search requires 13 CPU seconds on a 400 Megaherz PC. Model checking with pruning, but no reordering, finds a solution in just over a minute. Without pruning, our computing resources were inadequate to find an attack.

5 Conclusions

Semantic models provide a foundation for interactive verification. We have shown how such models, in the context of a lazy programming language, can be used to formalize and automate the analysis of security protocols. Our empirical results are encouraging and provide evidence for the suitability of our approach.

There are a number of directions for future work. First, we would like to carry out more ambitious case studies. To support this, it would help to be able to generate formal models from high-level descriptions (as in figure 1) of security protocols and attacker models; furthermore, it should be possible to use these descriptions to generate automatically search heuristics like those we considered. Second, our formalization of the `attack` predicate in section 4 is ad hoc. We would like to develop more principled, high-level, ways of specifying security properties. Perhaps using predicates about agent belief, like those in the BAN logic [2], could help, although work is required to give such predicates meaning with respect to the kinds of models we used. Finally, we have shown that it is possible to take models from verification and recast them in a setting where they become more computable. The result, of course, is still completely formal. Hence, it would be interesting to use our models for both model checking and inductive theorem proving.

References

1. R. Anderson and R. Needham. Programming Satan's computer. In *Commputer Science Today*, volume 1000 of *LNCS*, pages 426–441. Springer-Verlag, 1995.

[3] For those familiar with Lowe's attack, our third step corresponds to two steps in his attack: (1) Bob sends his message to the Spy and (2) the Spy sends the message on to Alice. One can check that under our modeling, these two steps can indeed be merged: Alice cannot tell who sends her the message in step three, so it does not matter whether it comes from Bob or the Spy. All that matters to Alice is that it is in the format of step two for her run with the Spy; since it is, she answers with step three.

2. Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions in Computer Systems*, 8(1):18–36, 1990.
3. John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0. Available at the URL `http://www.cs.york.ac.uk/~jac/`.
4. G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols*, pages 939–944, Indianapolis, Indiana, June 1998.
5. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.
6. Dan P. Friedman and David S. Wise. Cons should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.
7. Nevin Heintze, J.D. Tygar, Jeannette M. Wing, and Hao-Chi Wong. Model checking electronic commerce protocols. In *Proceedings of the USENIX 1996 Workshop on Electronic Commerce*. 1996.
8. P. Hudak, S. Peyton Jones, and P. Wadler (Editors). Report on the programming language Haskell: A non-strict, purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), 1992.
9. Richard Kemmerer, Catherine Meadows, and Jonathan Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.
10. Gawin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS'96*, LNCS 1055, pages 147–166. Springer, Berlin, 1996.
11. Wenbo Mao. An augmentation of BAN-like logics. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 44–56. IEEE Computer Society Press, 1995.
12. W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols. In *Proceedings of the DIMACS Workshop on Design and Verification of Security Protocols*. 1997.
13. Jonathan K. Millen, S.C. Clark, and S.B. Freedman. The Interrogator: Protocol security analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, 1987.
14. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
15. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
16. A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings of 1995 IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1995.
17. Steve Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.