Experimenting with STA, a tool for automatic analysis of security protocols*

Michele Boreale
Dipartimento di Sistemi e Informatica
Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
boreale@dsi.unifi.it

Maria Grazia Buscemi
Dipartimento di Matematica e Informatica
Università di Catania
Viale Doria 6, 95125 Catania, Italy
buscemi@dmi.unict.it

ABSTRACT

We present STA (*Symbolic Trace Analyzer*), a tool for the analysis of security protocols. STA relies on symbolic techniques that avoid explicit construction of the whole, possibly infinite, state-space of protocols. This results in accurate protocol modeling, increased efficiency and more direct formalization, when compared to finite-state techniques. We illustrate the use of STA by analyzing two well-known protocols, asymmetric Needham Schroeder and Kerberos. We discuss the results of this analysis, and contrast them with previous work based on finite-state model checking.

1. INTRODUCTION

In recent years, formal methods have proven useful to analyze security protocols, often revealing previously unknown attacks.

One of the most successful approaches is based on model checking [14, 16, 17]. Within a model checker, both the honest participants and the adversary are modeled as communicating processes, described in some appropriate language, such as CSP, and the whole system is just the parallel composition of all these processes. A finite-state operational model for this system is then explicitly generated and explored, to check whether any insecure state can ever be reached. Properties like secrecy and authentication can be analyzed in this way. In order to keep the model finite and use standard model checking, two simplifying requirements are necessary: (a) there is a fixed number of participants and, (b) there is a bound on the number of possible messages the adversary can generate at any moment. Discarding one of these two requirements leads to infinite models. Moreover, the chosen bounds have to be very tight to avoid state-explosion. In general, while it is known that discarding requirement (a) leads to undecidability of protocol analysis, even under very mild hypotheses (see e.g. [8]), it is not clear to what extent requirement (b) can be relaxed, while preserving decidability and effectiveness.

In this paper, we present STA (*Symbolic Trace Analyzer*), an ML-based tool for the analysis of security protocols. What distinguishes STA from finite-state model checkers is its use of symbolic techniques that avoid explicit construction of the whole state-space of the protocol. This results in:

- More accurate protocol models. In particular, while keeping requirement (a), we discard requirement (b). The whole, possibly infinite, protocol model can be searched for attacks.
- Increased efficiency, both in terms of memory occupation and execution time. In particular, state-explosion induced by message exchange between participants is avoided.
- More straightforward protocol formalization. In particular, there is no need for carefully crafted message types or bounds, and no explicit description of the adversary must be provided.

*Research partly supported by the Italian MURST Project TOSCA *(Teoria della Concorrenza, Linguaggi di Ordine Superiore e Strutture di Tipi).*

The theory underlying STA is developed and explained in full detail in [4, 5]. In the present paper, we mainly intend to illustrate the use of STA and the significance of the above listed features in practice. To do so, we analyze two well-known protocols: the asymmetric Needham-Schroeder protocol [18] and a simplified version of Kerberos [12]. Examining these protocols gives us a chance of explaining the STA's model and specification method, commenting on the results of the analysis and contrasting them with those obtained using finite-state model checkers.

The rest of the paper is organized as follows. In Section 2, we explain the model underlying STA. The syntax of STA's specifications is briefly outlined in Section 3. In Sections 4 and 5 we report our results of the STA analysis of the Needham-Schroeder and Kerberos protocols. Section 6 contains a discussion on STA's main features, contrasted with finite-state model-checkers. A few concluding remarks and a comparison with related work on symbolic analysis are given in Section 7.

2. OVERVIEW OF THE MODEL

Akin to many others (e.g., [14, 17, 16]), our formal model is close in spirit to the Dolev-Yao model of security protocols [7]. Informally, agents executing the protocol are viewed as concurrent processes that communicate through an insecure network. It is assumed that an adversary has total control over the network. Sending a message just means handing the message to the adversary; conversely, receiving a message just means accepting any message among those the adversary can produce. The adversary records all messages that transit over the network, and can produce a message by either replaying an old one, or by combining old messages (e.g. by pairing, encryption and decryption) and/or by generating fresh quantities. Both the honest agents and the adversary obey the rules of *perfect encryption* (by which, e.g., secret keys cannot be guessed; see [1]).

2.1 Configurations and transition relation

As noted, a protocol is modeled as a system of concurrent processes. A state of the system is a pair $\langle s, P\rangle$, called *configuration*. Here, the *trace* s is a sequence of I/O events (*actions*), and represents the current adversary's knowledge; P is a process term, describing the intended behavior of honest participants. The language

$$(\textsc{Inp})\quad \langle s,\ \mathsf{a}(x).P\rangle \longrightarrow \langle s\cdot \mathsf{a}\langle M\rangle,\ P[^{M}/_{x}]\rangle \qquad s\vdash M,\ M \text{ closed}$$

$$(\textsc{Out})\quad \langle s,\ \overline{\mathsf{a}}\langle M\rangle.P\rangle \longrightarrow \langle s\cdot \overline{\mathsf{a}}\langle M\rangle,\ P\rangle$$

$$(\textsc{Case})\quad \langle s,\ \mathsf{case}\,\{M\}_N\,\mathsf{of}\,\{y\}_N\,\mathsf{in}\,P\rangle \longrightarrow \langle s,\ P[^{M}/_{y}]\rangle$$

$$(\textsc{Select})\quad \langle s,\ \mathsf{pair}\,\langle M,N\rangle\,\mathsf{of}\,\langle x,y\rangle\,\mathsf{in}\,P\rangle \longrightarrow \langle s,\ P[^{M}/_{x},{}^{N}/_{y}]\rangle$$

$$(\textsc{Match})\quad \langle s,\ [M=M]P\rangle \longrightarrow \langle s,\ P\rangle$$

$$(\textsc{Par})\quad \frac{\langle s,\ P\rangle \longrightarrow \langle s',\ P'\rangle}{\langle s,\ P\,\|\,Q\rangle \longrightarrow \langle s',\ P'\,\|\,Q\rangle}$$

plus symmetric version of (Par).

Table 1: Transition relation ($\longrightarrow$)

used for process description is a dialect of the spi-calculus [1]. The set of all configurations is denoted by $\mathcal{C}$. The dynamics of configurations is given by a transition relation $\longrightarrow \subseteq \mathcal{C}\times\mathcal{C}$, that describes elementary steps of computations. In Table 1 we report the operational semantics for the case of shared-key encryption. Rules for the other cryptographic primitives can be easily added. Rules (Inp) and (Out) concern sending and receiving messages, respectively. In these rules, a represents a user-defined label. Labels are attached to I/O actions for ease of reference, and are useful when specifying protocol properties, as we shall see below. Since sending a message just means handing the message to the adversary, in rule (Out), after an output action $\overline{\mathsf{a}}\langle M\rangle$ is fired by a process, it is recorded in the adversary's current knowledge s. Conversely, receiving a message just means accepting any message among those the adversary can produce. Therefore, in rule (Inp) the variable x can be replaced by any message M (this is the meaning of $[^{M}/_{x}]$), nondeterministically chosen among those the adversary can synthesize from its current knowledge s. The synthesis of a message M from a set of known messages S is formalized by a deduction relation $\vdash$. Here is a sample of deduction rules defining $\vdash$ (see [4]):

$$\frac{M\in S}{S\vdash M}\qquad \frac{S\vdash M \quad S\vdash k}{S\vdash \{M\}_k}\qquad \frac{S\vdash \{M\}_k \quad S\vdash k}{S\vdash M}$$

Basically, the full knowledge of the adversary consists of the messages exchanged over the network plus anything that can be obtained by pairing, projection, encryption and decryption of known messages, provided the right key is itself part of the knowledge.

The other operational rules in Table 1 govern how a process decrypts a message ($\mathsf{case}\,M\,\mathsf{of}\,\{y\}_N\,\mathsf{in}\,A$), compares two messages for equality ($[M=N]A$), splits a pair ($\mathsf{pair}\,\langle M,N\rangle\,\mathsf{of}\,\langle x,y\rangle\,\mathsf{in}\,A$) and interleaved execution of parallel threads ($A\,\|\,B$).

It is worthwhile to point out that there is no need for an explicit description of the adversary's behavior, as the latter is wholly determined by its current knowledge – the s in $\langle s,P\rangle$ – and by the deduction relation $\vdash$. This is somehow in contrast with other proposals [14, 17], where the adversary must be explicitly described.

2.2 Properties

Given a configuration $\langle s,\ P\rangle$ and a trace s', we say that $\langle s,\ P\rangle$ *generates* s' if $\langle s,\ P\rangle \longrightarrow^* \langle s',\ P'\rangle$ for some P' ($\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$, i.e. zero or more steps of $\longrightarrow$). We express properties of the protocol in terms of the traces it generates. In particular, we focus on correspondence assertions of the kind

> for every generated trace, if action β occurs in the trace, then action α must have occurred at some previous point in the trace

that is concisely written as $\alpha \hookleftarrow \beta$. More accurately, we allow α and β to contain free variables, that may be instantiated to ground values. Thus $\alpha\hookleftarrow\beta$ actually means that *every instance* of β must be preceded by the corresponding instance of α, for every generated trace. We write $\langle s,\ P\rangle \models \alpha\hookleftarrow\beta$ if the configuration $\langle s,\ P\rangle$ satisfies this requirement. This kind of assertions is flexible enough to express interesting secrecy and authentication properties. As an example, the final step of many key-establishment protocols consists in A's sending a message of the form $\{N\}_K$ to B, where N is some authentication information, and K the newly established key. A typical property one wants to verify is that any message encrypted with K that is accepted by B at the final step should actually originate from A (this ensures B he is really talking to A, and that K is authentic). If we call $\mathsf{final_A}$ and $\mathsf{final_B}$ the labels attached to A's and B's final action, respectively, then the property might be expressed by $\overline{\mathsf{final_A}}\langle\{x\}_K\rangle \hookleftarrow \mathsf{final_B}\langle\{x\}_K\rangle$, for x a variable. In practice, all forms of authentication in Lowe's hierarchy [15] are captured by the scheme $\alpha\hookleftarrow\beta$, except the most demanding one that requires one-to-one bijection between α's and β's. However, our scheme can be easily adjusted to include this stronger form.

The scheme also permits expressing secrecy, in the style of [2]. To this purpose, it is convenient to fix a conventional 'absurd' action $\perp$ that is nowhere used in agent expressions. Thus the formula $\perp\hookleftarrow\alpha$ expresses that no instance of action α should ever take place. The fact that a protocol P does not leak a sensible datum d can be expressed also by saying that the adversary will never be capable of synthesizing d. This can be formalized by considering an extended protocol that also includes a 'guardian' that can at any time pick up one message from the network, $P\,\|\,\mathsf{g}(x).\mathbf{0}$, and then requiring that this guardian will never receive d, that is: $\langle\varepsilon,\ P\,\|\,\mathsf{g}(x).\mathbf{0}\rangle \models \perp\hookleftarrow \mathsf{g}\langle d\rangle$, where ε is the empty trace.

2.3 Symbolic execution

When synthesizing new messages, the adversary can apply operations like pairing, encryption generation of fresh names, an arbitrary number of times. Thus the set of messages the adversary can synthesize at any time is actually infinite. Any such message can be non-deterministically chosen by the adversary and sent to a participant willing to receive it, therefore every model based on Dolev and Yao's is in principle infinite. Our model makes no exception: in rule (Inp) the set of M s.t. $s\vdash M$ is always infinite, and this makes the model infinitely-branching. This can be regarded as a state explosion problem induced by message exchange.

To overcome this problem, STA implements a verification method based on a notion of symbolic execution. A new transition relation (written $\longrightarrow_{\mathrm{s}}$, below) is introduced in order to condense the infinitely many transitions that arise from an input action (rule (Inp) in Table 1) into a single, *symbolic* transition. The received message is now represented simply by a free variable, whose possible values are gradually constrained as the execution proceeds. Technically, a constraint takes the form of *most general unifier* (mgu), i.e., the most general substitution that makes two expressions equal. The set of traces generated using the symbolic transition relation constitutes the *symbolic model* of the protocol. Differently from the standard model given by $\longrightarrow$, the symbolic model is finite, because each input action just gives rise to one symbolic transition and agents cannot loop.

The rules of the symbolic transition relation $\longrightarrow_{\mathrm{s}}$ are reported in [4, 5]. For a flavor of how symbolic execution works, let us consider an example focusing on shared-key encryption. Suppose that agent P, after receiving a message, tries decryption of this message using key k; if this succeeds and y is the result, the agent checks whether y equals b and, if so, proceeds like P'. This is written as $P \stackrel{\text{def}}{=} \mathsf{a}(x).\,\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,[y=b]P'$, for y fresh. Let us explain how the symbolic execution proceeds, starting from the initial configuration $\langle \varepsilon,\, P\rangle$. After the first input step, in the second step the decryption $\mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\cdots$ is resolved by unifying x and $\{y\}_k$, which results in the substitution $[\{y\}_k/x]$. In the third step, the equality test $[y=b]$ is in turn resolved by unifying y and b, that results in $[b/y]$. Formally,

$$\begin{array}{lll} \langle \varepsilon,\, P\rangle & \longrightarrow_{\mathrm{s}} & \langle \mathsf{a}\langle x\rangle,\, \mathsf{case}\,x\,\mathsf{of}\,\{y\}_k\,\mathsf{in}\,[y=b]P'\rangle \\ & \longrightarrow_{\mathrm{s}} & \langle \mathsf{a}\langle \{y\}_k\rangle,\, [y=b]P'[\{y\}_k/x]\rangle \\ & \longrightarrow_{\mathrm{s}} & \langle \mathsf{a}\langle \{b\}_k\rangle,\, P'[\{y\}_k/x][b/y]\rangle. \end{array}$$

In [4, 5] the method based on symbolic execution is proven sound and complete, in the sense that every attack detected in the symbolic model (relation $\longrightarrow_{\mathrm{s}}$) corresponds to some attack in the standard model (relation $\longrightarrow$), and vice-versa. In other words, the symbolic model captures all and only the attacks of the standard model. For instance, the method detects type-dependent attacks, which usually escape finite-state analysis. In this kind of attacks, the adversary cheats on the type of some messages, e.g. by inserting a nonce where a key is expected according to the protocol description.

3. STA SPECIFICATIONS

Protocol specification in STA follows closely the syntax and semantics of the formal model, with a few minor differences. Concerning cryptographic operations, shared-key encryption of `X` with `K` is written `{X}K`. Asymmetric encryption is written `(X)^+K`. Here, `+K` is the encryption key; the corresponding decryption key is written `-K`. Asymmetric encryption can be used to model both public-key cryptography (if `+K` is public and `-K` is kept private) and digital signature (if `+K` is kept private and `-K` is public). Hashing of `X` is written as `H(X)`. For process syntax, we have:

- output actions are written as `a!M`;
- input actions are written as `a?M`. Here, `M` can be a generic message pattern, with variables. This means input of any adversary-generated message whose form matches up `M`.
- sequencing of actions is written `>>`;
- testing for equality of two messages `M` and `N` is written `(M is N)`. Operationally `(M is N) >> P` means 'unify `M` and `N`, if possible, and then proceed like `P`, otherwise stop'. Test for syntactic equality of two messages is a special case of this operation.
- parallel composition of `P` and `Q` is written `P||Q`.

There are a few more process operations (non-deterministic choice and a facility for generating fresh names), but those listed above are enough to illustrate our approach. Configurations are written as `(L @ P)`, where `L` is a list of input/output actions, which typically provides the adversary with its initial knowledge. Finally, properties are written like `A <-- B`, with `A` and `B` being actions.

It is worthwhile to note that, in the formal model, an input pattern can be rendered as an input action followed by an appropriate sequence of decryptions and/or equality tests. For example, reception of a piece of information encrypted with `+K` (i.e., reception of anything that can be correctly decrypted using the key `-K`) is written in the formal model as $\mathsf{a}(y).\mathsf{case}\,y\,\mathsf{of}\,\{x\}_{-K}\,\mathsf{in}\,P$. In STA, this can be simply written as `a?(x)^+K >> P`.

4. EXAMPLE: NEEDHAM-SCHROEDER

We consider here two versions of the asymmetric Needham-Schroeder protocol. The first one is a simplified version (the same considered e.g. in [14, 17]), that leaves out the initial steps necessary to distribute the participants' public keys. The second version considers these steps too. The extended version is interesting because it exhibits a type-dependent flaw (which was well-known, though).

4.1 Basic Needham-Schroeder

Below, we give the informal description of the protocol. A acts as the initiator and B as the responder. The notation $\{X\}_{+KY}$ denotes encryption of X with the public key of Y. As the protocol begins, $+KA$ and $+KB$ are assumed to be already known by any participant.

1.	$A \longrightarrow B:$	$\{NA,A\}_{+KB}$	(NA fresh nonce)
2.	$B \longrightarrow A:$	$\{NA,NB\}_{+KA}$	(NB fresh nonce)
3.	$A \longrightarrow B:$	$\{NB\}_{+KB}$	

There is a well-known attack on this protocol (see [14]) in case A may also run the protocol with a malicious insider I, i.e. a principal that has disclosed its keys to the adversary. In Table 2 we give the complete STA script for the above version of the protocol. (By convention, names begin by a capital letter, variables begin by one of letters `u`, `x`, `y`, `w` or `z`; the rest are labels. The name of each variable is meant to remind its expected value.) `InA` plays the role of A and consists of two parallel threads, one for talking to B and one for talking to I. In both threads, `InA` sends a nonce challenge and then expects a pair of encrypted nonces in response, the first of which must be the one `InA` previously issued. The third step of both these threads consists in sending out the second nonce, encrypted with the responder's public key. `InA` uses two different nonces (`NA` and `N'A`) for its two threads. `ReB` plays a role somehow specular to `InA`'s first thread. The insider I is not explicitly described, because its role is implicitly impersonated by the adversary, which knows I's keys. In other words, the behavior of I is subsumed by the behavior of the adversary. Thus, the whole system is the parallel composition of initiator and responder, `(InA || ReB)`. The initial configuration, `NS`, consists of a list containing one action plus the system. The `disclose!` action supplies the environment with the appropriate initial knowledge: identities of the participants (`A,B,I`), public keys of A (`+KA`) and B (`+KB`), and the 'seed' `KI` to synthesize both the public (`+KI`) and the private (`-KI`) key of I. The property `AuthAtoB` means that *any* message accepted by B at step 3 should indeed originate from A, since A is supposedly the only initiator: this means that B is really talking to A. `AuthBtoA` can be explained similarly.

When required to check whether `NS` satisfies `AuthAtoB`, STA finds a trace of `NS`, reported below, that violates the property `AuthAtoB`. The adversary, which intercepts all messages, re-uses the nonce `N'A` (issued by A when interacting with I) when impersonating the role of A talking to B. Then, the adversary induces A to decrypt message `(N'A,NB)^+KA`, thus getting `NB` (actions `a'2` and `a'3`). This attack was found after examining 26 symbolic configurations, which took slightly more than half a second. (This and the following examples were run on a PC with a 350 MHz Pentium III and 64Mb RAM.) The actual output of STA is given below. Note that the symbolic execution causes `yNA` to be instantiated to `N'A`.

```
An attack was found:
   disclose!(KI,+KA,+KB,A,B,I). a'1!(N'A,A)^+KI.
   a1!(NA,A)^+KB. b1?(N'A,A)^+KB. b2!(N'A,NB)^+KA.
   a'2?(N'A,NB)^+KA. a'3!(NB)^+KI. b3?(NB)^+KB.
26 symbolic configurations reached.
```

```
(* Initiator and Responder *)
val InA = a1!(NA,A)^+KB >> a2?(NA,xNB)^+KA >>
          a3!(xNB)^+KB >> stop
         ||
          a'1!(N'A,A)^+KI >> a'2?(N'A,xNI)^+KA >>
          a'3!(xNI)^+KI >> stop;
val ReB = b1?(yNA,A)^+KB >> b2!(yNA,NB)^+KA >>
          b3?(NB)^+KB >> stop;
(* The initial configuration *)
val NS  = ( [ disclose!(KI,+KA,+KB,A,B,I) ]
          @ (InA || ReB) ) ;
(* Prop.1:  A is correctly authenticated to B *)
val AuthAtoB = ( a3!u <-- b3?u);
(* Prop.2:  B is correctly authenticated to A *)
val AuthBtoA = ( b2!u <-- a2?u);
```

Table 2: STA script for Needham Schroeder protocol

After repairing the flaw as suggested in [14], i.e. by inserting explicit identities inside each encrypted message, STA finds no additional attack, either on property AuthAtoB, or on AuthBtoA. In both cases, the exploration reached all the 60 configurations that constitute the complete symbolic state-space of the protocol, and this took again slightly more than half a second. We also tried other instances of the protocol (see Table 4), without finding any attack.

4.2 A type flaw

We now consider a more accurate version of Needham-Schroeder, in which both A and B also communicate with a trusted server S to get a signed certificate of the other party's public key. Below, $Sig_Y(X)$ denotes X digitally signed by agent Y.

1.	$A \longrightarrow S:$	B	
2.	$S \longrightarrow A:$	$Sig_S(+KB,B)$	
3.	$A \longrightarrow B:$	$\{NA,A\}_{+KB}$	(NA fresh nonce)
4.	$B \longrightarrow S:$	A	
5.	$S \longrightarrow B:$	$Sig_S(+KA,A)$	
6.	$B \longrightarrow A:$	$\{NA,NB,B\}_{+KA}$	(NB fresh nonce)
7.	$A \longrightarrow B:$	$\{NB\}_{+KB}$	

This example is interesting because it exhibits a type-dependent flaw. This kind of flaws is normally not detected using traditional finite-state techniques, unless the specification is tailored towards finding specific, hence, *a priori* known bugs (see, e.g., [6]).

We analyze two parallel runs of the protocol: A acts as responder and as initiator, respectively, while B acts only as responder. The interaction of S with A and B can be interleaved. This version of the protocol can be specified in STA by modifying the scripts for the previous example. E.g., the specification of the responder B is:

```
val ReB = b1?(yNA,A)^+KB >> b2!A >>
    b3?(ypkA,A)^+SigS >> b4!(yNA,NB,B)^(ypkA) >>
    b5?(NB)^+KB >> stop;
```

Note that, in action b3, ReB implicitly uses -SigS as a verification key for messages signed by S (the signing key is +SigS). We asked STA to check the property a5!u <-- b5?u, being a5 the label of InA's final action (when acting as an initiator). Such property guarantees the initiator A is correctly authenticated to B. After searching 16,275 configurations, STA found a trace violating the property, reported below. In this trace, InA is involved in two parallel runs. In the first of these runs, where A acts as the initiator, the adversary gets the b4 message (yNA,NB,B)^+KA. In the second run, where A acts as the responder, the adversary replays this message to A (action a'1). A interprets the pair (NB,B) as an agent's name and sends it to the server (a'2). The adversary can then intercept (NB,B) and reply the nonce NB to the nonce challenge (b5). Note that in the trace below the variable yNA can be instantiated to any value.

```
disclose!(A,B,-SigS,+KA,+KB). a1!B. b1?(yNA,A)^+KB.
b2!A.s'1?A. s'2!(+KA,A)^+SigS. b3?(+KA,A)^+SigS.
b4!(yNA,NB,B)^+KA. s1?B. s2!(+KB,B)^+SigS.
a'1?(yNA,NB,B)^+KA. a'2!(NB,B). b5?(NB)^+KB
```

This attack can be prevented by simply changing $\{NA,A\}_{+KB}$ in step 3 of the protocol into $\{NA,(\mathrm{id},A)\}_{+KB}$, for id an arbitrary tag indicating its intended type. This tag avoids any confusion between messages in step 3 and 6 of different runs.

5. EXAMPLE 2: KERBEROS

Kerberos [12] is a protocol based on shared-key cryptography for mutual authentication between a client C and a server S. In order to authenticate itself, C needs a ticket issued by a Ticket Granting Server (TGS) and, to get this ticket, another ticket is required from a Key Distribution Center (KDC). We analyze the simplified version considered in [17], that leaves out timestamping and nonces. First, C requests from the KDC a ticket for the TGS. Then, the KDC sends to C a newly generated session key for the communication between C and the TGS, and a ticket for the TGS (encrypted with a symmetric key K_{KT}, shared between KDC and TGS). C decrypts the session key, generates a so-called authenticator and sends it to the TGS, together with the received ticket and a request of a ticket for S. C is authenticated to the TGS, since C encrypts the authenticator under the session key that is contained in the ticket for the TGS. In a corresponding fashion, C is authenticated to the server in the further two steps. K_{TS} is the symmetric key shared between TGS and S. K_{s2} is the session key, shared between C and S at the completion of the protocol. The informal description of the protocol is reported below.

1.	C	$\longrightarrow$	KDC	:	C,TGS
2.	KDC	$\longrightarrow$	C	:	$\{K_{s1}\}_{K_c}, \{C,K_{s1}\}_{K_{KT}}$
3.	C	$\longrightarrow$	TGS	:	$\{C\}_{K_{s1}}, \{C,K_{s1}\}_{K_{KT}}, S$
4.	TGS	$\longrightarrow$	C	:	$\{K_{s2}\}_{K_{s1}}, \{C,K_{s2}\}_{K_{TS}}$
5.	C	$\longrightarrow$	S	:	$\{C\}_{K_{s2}}, \{C,K_{s2}\}_{K_{TS}}$

The present version of the protocol authenticates C only. A further step is required for S's authentication. We split our STA analysis of Kerberos into two parts, client authentication and server authentication.

5.1 Client authentication

We begin by analyzing a run that has one participant for each role. The STA script for this version of the protocol is reported in Table 3. Declarations of labels, variables and names are omitted. As for the case of Needham-Schroeder protocol, we adopt input patterns to get a compact specification. In fact, certain input actions would take more than one step in the formal model (and in real protocol's implementations too). For instance, the execution of the server's action s1 implies decrypting the second component ({zC,zKs2}KTS) under KTS; extracting the key (zKs2); decrypting with zKs2 the first component ({zC}zKs2); getting the client's ID therein (zC); comparing zC to the ID found in the second component. The property AuthC requires that for each value of u and u', if s1?(u,u') occurs in a trace, there is some value of u'' s.t. c5!(u,u'') occurred at some previous point in that trace. (In general, variables that appear only to the left of <--, like u'' in this example, are existentially quantified.) In other words, the first

```
val Client = c1!(C,TGS) >>
             c2?({xKs1}Kc, xTkt) >>
             c3!({C}xKs1,xTkt,S) >>
             c4?({xKs2}xKs1,xTktS) >>
             c5!({C}xKs2,xTktS) >> stop;

val KDC    = kdc1?(C,TGS) >>
             kdc2!({Ks1}Kc,{C,Ks1}KKT) >>
             stop;

val TGS    = t1?({wC}wKs1,{wC,wKs1}KKT,S) >>
             t2!({Ks2}wKs1,{wC,Ks2}KTS) >>
             stop;

val Server = s1?({zC}zKs2,{zC,zKs2}KTS) >>
             stop;

val System = Client || KDC || TGS || Server;

val KERB   = ([disclose!(C,C',S)] @ System);

val AuthC  = (c5!(u,u'') <-- s1?(u,u'));
```

Table 3: STA script for the Kerberos protocol

component of the message accepted by S at step `s1` is required to originate from the C's message at step `c5`. This implies that, at the protocol's completion, S is really talking to C, and they agree on the newly established key too. The property has been verified by STA and the result is as follows:

```
No attack was found.  307 symbolic
configurations reached.
```

Note that there is no guarantee that C and S agree on the second component of their final messages, bound to `xTktS` and to `{zC,zKs2}KTS`, respectively: in fact, these two expressions may assume different values in some run of the protocol, even though there is agreement on the first components, `{C}xKs2` and `{zC}zKs2` respectively. The reason is that here C just forwards the value bound to `xTktS`, but cannot inspect it. Thus, while the adversary provides S with the right TGS ticket, `xTktS` might actually be anything different. STA pointed out this anomaly when required to verify the property `(c5!u <-- s1?u)`, i.e. an agreement on the whole final message. It is dubious whether it should be regarded as a flaw. A similar anomaly, i.e., the lack of connection between the first two components of `c3`, is at the basis of a more interesting attack on the server's authentication, as explained in the next Subsection.

This version of the protocol is also subject to a (well-known) attack based on the use of compromised session keys and tickets – not surprisingly, due to the absence of timestamps and nonces. This attack shows up in case the initial adversary's knowledge is augmented with the compromised information. Here `KOld` is an old session key and `{C,KOld}KTS` the corresponding TGS ticket: it is assumed that these two quantities have been issued by the TGS at some time in the past and `KOld` has been somehow disclosed to the adversary.

```
val KERB =  ([disclose!(C,S, KOld, {C,KOld}KTS )]
              @ System);
```

When fed with this example, STA almost immediately found an attack, reporting the trace:

```
disclose!(C,S,KOld,{C,KOld}KTS). c1!(C,TGS).
s1?({C}KOld,{C,KOld}KTS).
```

The attack arises from the absence of a check on the freshness of the TGS ticket. Traditionally, timestamps are adopted to repair this flaw. However, presently, timestamps are not handled within our symbolic method.

5.2 Server authentication

We consider now extending the above protocol with a sixth step, which lets S authenticate itself to C:

$$6. \quad S \longrightarrow C: \quad \{C,S\}_{K_{s2}}.$$

In STA, this corresponds to adding one more action to both `Server` and `Client` in the specification in Table 3:

```
val Server = ··· >> s2!{zC,S}zKs2 >> stop;
val Client = ··· >> c6?{C,S}xKs2 >> stop;
```

The property `AuthServer = s2!u <-- c6?u` expresses that the final message accepted by C indeed originates from S. If so, C is really talking to S, and the two parties agree on their reciprocal identities and on the session key. When asked to verify `AuthServer`, STA found an attack after exploring 298 symbolic configurations. The relevant attack's actions are as follow (the trace does not contain any `s2`-action):

```
··· c3!({C}Ks1, xTkt,S). ··· c4?({C}Ks1,xTktS).
 c5!({C}C,xTktS).c6?{C,S}C.
```

Note that the free variables in the trace can be instantiated to any value. The point here is that, in action `c4`, C accepts anything encrypted with `Ks1` as first component of the received message, including `{C}Ks1`, issued by C itself in `c3`. Thus, it mistakes its own ID, `C`, for the new session key. Of course, `C` is known to the adversary, who can, therefore, build a message `{C,S}C` and hand it to C, no matter if S is involved in the run. Again, this is the case of a type flaw, which we were not aware of before checking the property. This anomaly can be repaired by simply changing in step 4 $\{K_{s2}\}_{K_{s1}}$ into $\{0,K_{s2}\}_{K_{s1}}$: the addition of 0 avoids messages with the same format in steps 3 and 4. After this adjustment, STA found no additional attack. The whole symbolic state-space of 622 configurations has been explored in about 3 seconds.

Next, we considered a run of the corrected protocol with two servers, S and S', and a client. Again no attack was found by STA, after exploring 7,516 symbolic configurations. The authentication step 6 prevents the attack described in [17], based on the adversary's replacing the ID S in step 3 with a different S'. In fact, in the final step, each server explicitly declares its identity.

6. DISCUSSION

We try an assessment of our approach, by discussing what we think are its main benefits, and by contrasting our work with those of finite-state methods (e.g. [14, 16, 17]).

6.1 Accuracy of the model

Finite-state model checking of security protocol relies on finite approximations of the actual model obtained, e.g., by imposing some fixed typing to the adversary-generated messages [17]. For example, it might be assumed that, at a given stage, the adversary can only send messages that fit in the format {nonce}key. The number of such possible messages is finite, as long as the number of distinct nonces and keys is finite. These typing assumptions may be sensible under certain circumstances (see [11]), but they cannot

Initiators	Responders	Murφ	STA
1	1	1706	60
2	1	17,277	411
2	2	514,550	24,655

Table 4: Number of reachable states for the correct version of Needham-Schroeder protocol.

be established automatically: it seems that some a priori knowledge is required of how the protocol works. This makes the whole analysis process potentially error-prone.

On the contrary, our approach makes no assumption on typing, or on the number of messages the adversary may generate. In fact, as a consequence of the completeness result, the symbolic model that STA explores makes no approximation with respect to the infinite-state standard model.

6.2 Efficiency

Symbolic analysis does not suffer from any state explosion problem depending on message exchange. Conversely, in finite-state model checkers, even if clever assumptions may reduce the state space to explore (see e.g. [20]), branching factor still remains. For instance, if in some state the adversary can only send messages of type $\{\mathsf{nonce},\mathsf{nonce}\}\mathsf{key}$, then at least $n \times n \times k$ transitions will branch from that state (being n and k the number of possible nonces and keys). As the number of participants, and consequently of possible data values, increases, the size of the model is expected to increase dramatically.

In view of these considerations, STA seems to have some advantage over finite-state model checkers. This is reflected on those instances of Needham-Schroeder analyzed here and in [17]. Some figures on the number of states are reported in Table 4. Execution times heavily depend on the number of states, but also on a variety of technological factors, thus exact figures for them are less meaningful. Indicatively, our execution times are from 5 to 70 times shorter than those reported in [17]. Memory occupation is very well controlled in STA because a depth first search strategy is adopted.

It is well known that a form of state-explosion is also induced by interleaving. However, this is not specific to security protocols. There exist standard techniques to deal with this problem, but they have not been considered in our implementation.

The aspect of efficiency deserves, anyway, a comprehensive comparison study. Of course, the word 'efficiency' should be taken in a practical sense, here, rather than in the formal sense of 'worst-case complexity efficiency'. In fact, the problem is NP-hard [19], and pathological examples can always be exhibited, for STA like for other tools.

6.3 Usability

The user interface of STA is at the moment rather rudimentary, thus specifying a protocol requires a certain acquaintance with process algebras. However, no deep understanding of security is needed; in particular, no a priori knowledge of the protocol to analyze is required. Another point worth noticing is that no description of the adversary must be explicitly given, apart form its initial knowledge. For an experienced user, once the configuration to be tested has been chosen, translating the informal description of the protocol into the STA specification takes about half an hour, and then everything is automatic. The whole specification process could be made more user-friendly by developing some high-level user interface in the same fashion as Lowe's Casper [13].

7. CONCLUDING REMARKS

We have presented STA, a tool based on symbolic semantics for automatic verification of security protocols. We have tested our tool by analyzing a few properties of the Needham-Schroeder and Kerberos protocols. We have then compared our methods and results to those obtained by other authors with finite-state model checking.

STA can be currently considered little more than a prototype (it can be downloaded [3]). Future work includes development of a user-friendly interface, optimization of data structures and translation into C language.

The papers [4, 5] develop the theory underlying STA. Initial work on symbolic analysis is due to Huima [10]. Symbolic techniques are also exploited in [2, 9], but the algorithms they use are quite different from ours. In particular, some brute force instantiation of variables is still necessary to guarantee the completeness of their methods.

8. REFERENCES

[1] M. Abadi, A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1-70, 1999.

[2] R.M. Amadio, S. Lugiez. On the reachability problem in cryptographic protocols. In *Proc. of Concur'00*, LNCS 1877, Springer, 2000. Full version: RR 3915, INRIA Sophia Antipolis.

[3] STA: a tool for trace analysis of cryptographic protocols. ML object code and examples, 2001. Available at `http://www.dsi.unifi.it/~boreale/tool.html`.

[4] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proc. of ICALP'01*, LNCS 2076, Sringer, 2001.

[5] M. Boreale, M. Buscemi. A framework for the analysis of security protocols. Submitted. An abstract appears in *Proc. of WSDAAL 2001*, Como, Italy.

[6] B. Donovan, P. Norris, and G. Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proc. of Workshop on Formal Methods and Security Protocols*, Trento, 1999.

[7] D. Dolev, A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198-208, 1983.

[8] N. Durgin, P. Lincoln, J. Mitchell, A. Scedrov. Undecidability of bounded security protocols. In *Proc. of Workshop on Formal Methods and Security Protocols*, Trento, 1999.

[9] M.P. Fiore and M. Abadi. Computing Symbolic Models for Verifying Cryptographic Protocols. In *Proc. of 14th Computer Security Foundations Workshop*, IEEE Computer Society Press, 2001.

[10] A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. of Workshop on Formal Methods and Security Protocols*, Trento, 1999.

[11] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proc. of 13th Computer Security Foundations Workshop*, IEEE Computer Society Press, 2000.

[12] J. Kohl, B. Neuman. The Kerberos network authentication service (version 5). Internet Request For Comment RFC-1510, 1993.

[13] G. Lowe. Casper, a compiler for the analysis of security protocols. In *Proc. of 10th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, 1997.

[14] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proc. of TACAS'96*,

LNCS 1055, Springer, 1996.

[15] G. Lowe. A Hierarchy of Authentication Specifications. In *Proc. of 10th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, 1997.

[16] W. Marrero, E.M. Clarke, S. Jha. Model checking for security protocols. Technical Report TR-CMU-CS-97-139, Carnegie Mellon University, 1997.

[17] J.C. Mitchell, M. Mitchell, U. Stern. Automated Analysis of Cryptographic Protocols Using Murφ. In *Proc. of Symp. Security and Privacy*, IEEE Computer Society Press, 1997.

[18] R. Needham, M. Schroeder. Using encryption for authentication in large networks of computers. *Communication of the ACM*, 21(12):993-9, 1978.

[19] M. Rusinowitch, M. Turuani. Protocol Insecurity with Finite Number of Sessions in NP-Complete. In *Proc. of 14th Computer Security Foundations Workshop*, IEEE Computer Society Press, 2001.

[20] V. Shmatikov and U. Stern. Efficient finite-state analysis for large security protocols. In *Proc. of 11th Computer Security Foundations Workshop*, IEEE Computer Society Press, 1998.