Applying Formal Methods to Analysis of Security Protocols

The principles are quite straightforward:

- Model the participants in the protocol, including the Intruder, as processes in a reactive concurrent system.
- Formulate the security goals as a specification the systems should satisfy.
- Apply available verification techniques, i.e., model checking and deductive verification, to establish that the protocol meets its goals or, alternately, to identify counter-examples which should be interpreted as successful attacks on the protocol.

This approach is based on the reduction of the security problem to a general verification problem of reactive systems.

As is often the case, success largely depends on the details of the reduction.
In View of this Reduction

We have as many different suggestions for formal analysis of security as there are different models for concurrency, specifications of reactive systems, and different verification methods.

Among these approaches, we find methods based on CSP, CCS, Petri-Nets, LTL, CTL, IO-Automata, ASM, Logics of Knowledge, Linear Logic.

We will rely in part on a textbook:

**Modelling and Analysis of Security Protocols**

by P. Ryan and S. Schneider from Addison Wesley. This book is based on the concurrency model of CSP language and its implementation by the FDR model checker.

Following the organization of this text, we will try to develop our own version, based on appropriate extensions of SPL and the TLV model checker.
Example: The Yahalom Protocol

Consider the following protocol:

\[ A \cdot N_a \]
\[ A \cdot N_a \]
\[ B \cdot \{ A \cdot N_a \cdot N_b \}_{K_{BJ}} \]
\[ B \cdot \{ A \cdot N_a \cdot N_b \}_{K_{BJ}} \]

\[ \{ B \cdot K \cdot N_a \cdot N_b \}_{K_{AJ}} \cdot \{ A \cdot K \}_{K_{BJ}} \]
\[ \{ B \cdot K \cdot N_a \cdot N_b \}_{K_{AJ}} \cdot \{ A \cdot K \}_{K_{BJ}} \]

\[ \{ A \cdot K \}_{K_{BJ}} \cdot \{ N_b \}_K \]
\[ \{ A \cdot K \}_{K_{BJ}} \cdot \{ N_b \}_K \]

The protocol is believed to be correct, but its verification is far from trivial. The hardest part is to prove that \( N_b \) remains secret.
Message Passing in SPL

In addition to communication by shared variables, SPL also provides the alternate mechanism of message passing. We allow three types of channels, distinguished by their declarations:

\[ \langle \text{identifier} \rangle: \text{channel of } \langle \text{type} \rangle \quad \text{— Synchronous message passing.} \]
\[ \text{Zero buffering. Both sender and receiver are blocked until both are ready.} \]

\[ \langle \text{identifier} \rangle: \text{channel}[1..N] \text{ of } \langle \text{type} \rangle \quad \text{— Asynchronous message passing.} \]
\[ \text{Bounded buffering up to size } N. \text{ Sender is blocked on a full buffer.} \]
\[ \text{Receiver is blocked on an empty buffer.} \]

\[ \langle \text{identifier} \rangle: \text{channel}[1..] \text{ of } \langle \text{type} \rangle \quad \text{— Asynchronous message passing with unbounded buffering. Sender is never blocked.} \]
\[ \text{Receiver is blocked on an empty buffer.} \]

Commands:

\[ \alpha \leftarrow e \quad \text{— Send the value of expression } e \text{ on channel } \alpha. \]

\[ \alpha \Rightarrow x \quad \text{— Receive a message from channel } \alpha \text{ and store it in variable } x. \]
Example: Mutual Exclusion with Message Passing

\[ P_1 :: \begin{align*}
\text{loop forever do} & \\
\ell_0: \text{noncritical} & \\
\ell_1: \alpha \Rightarrow & \\
\ell_2: \text{critical} & \\
\ell_3: \beta \Leftarrow 1 & 
\end{align*} \]

\[ A :: \begin{align*}
\text{loop forever do} & \\
k_0: \alpha \Leftarrow 1 & \\
k_1: \beta \Rightarrow & 
\end{align*} \]

\[ P_2 :: \begin{align*}
\text{loop forever do} & \\
m_0: \text{noncritical} & \\
m_1: \alpha \Rightarrow & \\
m_2: \text{critical} & \\
m_3: \beta \Leftarrow 1 & 
\end{align*} \]

\[
\alpha, \beta : \text{channel[1..]} \text{ of boolean}
\]
The FDS Semantics of Message Passing

Consider first the semantics of asynchronous message passing, where the channel has a buffer bound $N$, $0 < N \leq \infty$. We assume that channel $\alpha$ is implemented as a list of messages.

- The statement $S = [\alpha \Rightarrow x]$ in process $P_i$ contributes the disjunct

$$
\pi_i = \text{pre}(S) \land \pi'_i = \text{post}(S) \land |\alpha| > 0 \land (x', \alpha') = (\text{hd}(\alpha), \text{tl}(\alpha)) \land \text{pres}(V - \{\pi_i, x, \alpha\})
$$

to the transition relation, and the compassion requirement

$$(\pi_i = \text{pre}(S) \land |\alpha| > 0, \pi_i = \text{post}(S))$$

- The statement $S = [\alpha \Leftarrow e]$ in process $P_i$ contributes the disjunct

$$
\pi_i = \text{pre}(S) \land \pi'_i = \text{post}(S) \land |\alpha| < N \land \alpha' = \alpha \bullet e \land \text{pres}(V - \{\pi_i, \alpha\})
$$

to the transition relation, and the compassion requirement

$$(\pi_i = \text{pre}(S) \land |\alpha| < N, \pi_i = \text{post}(S))$$
Synchronous Message Passing

Let $\alpha$ be a synchronous channel, i.e. declared as $\alpha: \text{channel of} \ldots$

Every pair of statements $S_i = [\alpha \leftarrow e]$ in process $P_i$ and $S_j = [\alpha \Rightarrow x]$ in process $P_j$, where $i \neq j$ contributes the disjunct

$$\pi_i = \text{pre}(S_i) \land \pi_j = \text{pre}(S_j) \land \pi'_i = \text{post}(S_i) \land \pi'_j = \text{post}(S_j)$$
$$\land x' = e \land \text{pres}(V - \{\pi_i, \pi_j, x\})$$

to the transition relation, and the compassion requirement

$$(\pi_i = \text{pre}(S_i) \land \pi_j = \text{pre}(S_j), \; \pi_i = \text{post}(S_i) \land \pi_j = \text{post}(S_j))$$

to the compassion set.
The Needham Schroeder Public Key Protocol

Following is a diagram for the NSPK protocol.

\[
\begin{align*}
\{N_a \cdot A\}_{K_B} & \quad A \\
B \cdot \{N_a \cdot N_b\}_{K_A} & \quad B \\
\{N_b\}_{K_B} & \quad B
\end{align*}
\]
Representation in Casper

The system endorsed in

Modelling and Analysis of Security Protocols

by P. Ryan and S. Schneider from Addison Wesley. Consists of two components:

1. A compiler, called Casper which accepts as input an abstract (yet more detailed) description of the protocol and participants, and translates it into the CSP language.

2. The FDR2 (Failures-Divergence Refinement Ver. 2) model checker which checks whether one finite-state CSP system refines another.

We found the Casper input language a very useful vehicle for describing security protocols. We will illustrate its use on the NSPK protocol.
NSPK Protocol in Casper

#Free variables

A, B : Agent  
na, nb : Nonce  
PK : Agent -> PublicKey  
SK : Agent -> SecretKey  
InverseKeys = (PK, SK)

#Processes

INITIATOR(A,na) knows PK, SK(A)  
RESPONDER(B,nb) knows PK, SK(B)

#Protocol description

0.   -> A : B
1.   A -> B : {na, A}{PK(B)}
2.   B -> A : {na, nb}{PK(A)}
3.   A -> B : {nb}{PK(B)}

#Specification

Secret(A, na, [B])  
Secret(B, nb, [A])  
Agreement(A,B,[na,nb])  
Agreement(B,A,[na,nb])
#Actual variables

Alice, Bob, Mallory : Agent  
Na, Nb, Nm : Nonce

#Functions

symbolic PK, SK

#System

INITIATOR(Alice, Na)  
RESPONDER(Bob, Nb)

#Intruder Information

Intruder = Mallory  
IntruderKnowledge = {Alice, Bob, Mallory, Nm, PK, SK(Mallory)}
The Meaning of Casper Security Goal Specifications: Secrecy

The above description contained several security goal requirements under the Specification paragraph. They are as follows:

\[ \text{Secret}(A, s, [B_1, \ldots, B_n]) \]

Specifies that, in any run of the protocol (completed or partial), the only agents who know the value of variable \( s \) are \( A, B_1, \ldots, B_n \). For the case that \( A, B_1, \ldots, B_n \) comprise all the legitimate agents in the system, this is equivalent to the temporal specification

\[ a : A, m : \text{Intruder} \rightarrow \square (a.s \notin m.pool) \]
Casper Security Goal Specifications: Agreement and Authentication

Agreement\((A, B, [v_1, \ldots, v_n])\)

Specifies that \(A\) is correctly authenticated to \(B\) and the two agents agree on the values of \(v_1, \ldots, v_n\). More precisely, if \(B\) thinks it has successfully completed a run of the protocol with \(A\), then \(A\) has been running the protocol with \(A\). For the case of NSPK, \(\text{Agreement}(A, B, [\text{na}, \text{nb}])\) can be formalized as

\[
a : A, \ b : B \rightarrow \left( (b)\_\text{received\_at}3 \land \Box (b)\_\text{sent\_to}(a)\_\text{at}2 \Rightarrow \begin{align*}
a.\text{na} &= b.\text{na} \land a.\text{nb} = b.\text{nb} \land \\
\Diamond (a)\_\text{sent\_to}(b)\_\text{at}3
\end{align*} \right)
\]

where, \((b)\_\text{received\_at}3\) means that agent \(b\) received a message at a location corresponding to line 3 in the protocol. \((b)\_\text{sent\_to}(a)\_\text{at}2\) means that (previously) agent \(b\) sent a message intended to agent \(a\) at line 2. This also identifies that agent \(b\) believes it is communicating with agent \(a\). Similarly, \((a)\_\text{sent\_to}(b)\_\text{at}3\) provides evidence that agent \(a\) believes to be communicating with agent \(b\).
Idealized Encoding of the NSPK Protocol

\[
\begin{align*}
B_{\text{in}} & : \text{array}[1..] \text{ of } \text{channel}[1..] \text{ of message initially } \Lambda \\
A_{\text{in}} & : \text{channel}[1..] \text{ of message initially } \Lambda
\end{align*}
\]

\[
A \parallel \left( \parallel_{j>0} B[j] \right)
\]

where the descriptions of \( A \), and \( B[j] \) are given in the following slides.
Process $A[i]$

\[
A \leftrightarrow \{ N_a \cdot A \} _{K_B} \\
B \leftrightarrow B \cdot \{ N_a \cdot N_b \} _{K_A} \\
\{ N_b \} _{K_B}
\]

Process $A$ is given by:

\[
A :: \\
\begin{align*}
\ell_0 & : b := \text{random}(1..) \\
\ell_1 & : na := \text{draw}() \\
\ell_2 & : \text{mess\_out} := \text{encr}(na \cdot a, PK(b)) \\
\ell_3 & : B_{in}[b] \leftarrow \text{mess\_out} \\
\ell_4 & : A_{in} \Rightarrow \text{mess\_in} \\
\ell_5 & : p := \text{decr}(\text{mess\_in}, SK[a]) \\
\ell_6 & : (q_1, nb) := \text{brk}_2(p) \\
\ell_7 & : \textbf{await} \ q_1 = na \\
\ell_8 & : \text{mess\_out} := \text{encr}(nb, PK(b)) \\
\ell_9 & : B_{in}[b] \leftarrow \text{mess\_out} \\
\ell_{10} & : \text{Session}(a, b, nb)
\end{align*}
\]
Process $B[j]$

\[ A \xrightarrow{\{N_A \cdot A\}^B_K} B \]

\[ B \cdot \{N_A \cdot N_B\}^{K_A} \xrightarrow{\{N_B\}^B_K} \]

Process $B[j]$ is given by:

\[
B[j] ::
\begin{align*}
  m_0 &: B_{in}[j] \Rightarrow mess_{in} \\
  m_1 &: (na, a) := \text{brk}_2(\text{decr}(mess_{in}, SK(j))) \\
  m_2 &: nb := \text{draw}() \\
  m_3 &: mess_{out} := \text{encr}(na \cdot nb, PK(a)) \\
  m_4 &: A_{in} \leftarrow mess_{out} \\
  m_5 &: B_{in}[j] \Rightarrow mess_{in} \\
  m_6 &: p := \text{decr}(mess_{in}, SK(j)) \\
  m_7 &: \textbf{await} p = nb \\
  m_8 &: \text{Session}(j, a, nb)
\end{align*}
\]
The Non-Idealized Situation

Since the intruder can get access to all ongoing messages, we remove the identification of individual channels. All messages go in and out of a single channel called media.

In addition, we have to model the intruder itself. A possible program for the intruder can be given as follows:

\[
\text{Intruder} ::
\]

\[
\begin{align*}
p, q, r & : \text{message} \\
pool & : \text{set of message where } pool = \{a, b, SK(m)\}
\end{align*}
\]

\[
\text{loop forever do}
\]

\[
\begin{align*}
p & \in pool \\
or & q \in pool \\
or & r \in pool \\
or & pool := pool \cup \{p, q, r\} \\
or & media \Rightarrow p \\
or & media \Leftarrow p \\
or & p := q \cdot r \\
or & p := encr(q, r) \\
or & p := decr(q, r) \\
or & (p, q) := brk_2(r)
\end{align*}
\]

Thus, the intruder can non-deterministically construct new terms, read and write to the common channel, encrypt messages with all available public keys, and decrypt messages for which it currently has the secret key.