Lecture Notes:
Automated Analysis of Cryptographic Protocols using Murphi,
Mitchell, Mitchell, and Stern, S&P 1997.

By Marie Nesfield

What does this paper do?

This paper presents a model checking tool (murphi) to analyze cryptographic protocols.

Discussion:

1. What does model checking do?
    a. Explore finite state models.
    b. Guarantee properties hold in model.
2. What sorts of things do cryptographic protocols do?
3. Are they finite state?

Protocol phases of development and associated bugs:

1. Requirements (I assume that some of these somehow translate into security properties we'd like to verify).
2. Specification (Bugs will be covered in this paper).
3. Implementation (e.g., Heartbleed, bugs about improperly implementing the specification's state machine).

This paper looks at the specification phase. The murphi tool allows one to enter a specification in the murphi language to be model checked.

We're going to look at three protocols: Needham-Schroeder, TMN, and Kerberos.

What they do:

1. Formulate protocol
2. Add adversary to system
3. State correctness condition
4. Run the protocol
5. Change things and do again?

Power of the adversary: assume that the adversary can only decrypt things for which it has the key. Cryptanalysis is out of scope (e.g. factoring RSA keys, RC4 weakness [1]). Probabilistic behavior is out of scope (e.g. guessing nonces or keys). Unclear how we would allow these in a finite state system.

Needham-Schroeder (public key)
Agents: Initiator A attempting to establish session with responder B.
Goal: achieve mutually authenticated session.

3. $A \rightarrow B : \{N_{a,} A\}_{K_B}$
6. $B \rightarrow A : \{N_a, N_b\}_{K_a}$
7. $A \rightarrow B : \{N_b\}_{K_b}$ B should be assured that he is talking to A

Problem:

1.3 $A \rightarrow I : \{N_{a,} A\}_{K_i}$
2.3 $I(A) \rightarrow B : \{N_{a,} A\}_{K_b}$
2.6 $B \rightarrow I(A) : \{N_a, N_b\}_{K_a}$ With proposed change: $B \rightarrow I(A) : \{B, N_a, N_b\}_{K_a}$
1.6. $I \rightarrow A : \{N_a, N_b\}_{K_a}$
1.7. $A \rightarrow I : \{N_b\}_{K_i}$
2.7. $I(A) \rightarrow B : \{N_b\}_{K_b}$

Solution proposed by Lowe:

3. $A \rightarrow B : \{N_{a,} A\}_{K_B}$
6. $B \rightarrow A : \{B, N_a, N_b\}_{K_a}$
7. $A \rightarrow B : \{N_b\}_{K_b}$

Modeling:
Initiator data structures:
Number of initiators variable set for verification
Initiator id (scalarset, symmetry reduction)
Where the messages are going

This is simplified modeling code for initiators:

$\forall$ $InitiatorID$: //this deals with the scalarset symmetry
    $\forall$ $AgentID$: //also deals with the scalarset symmetry
        $send\ initial\ message\ to\ agent,\ wait$
$\forall$ $InitiatorID\ i$: //this deals with the scalarset symmetry
    $\forall$ $messages\ for\ i\ on\ network$: //also deals with the scalarset symmetry
        $process\ response$
        $send\ final\ message$

Process response involves:

1. Remove message from network

2. Decrypt, check if message type is correct, ensure nounce checks out.
3. If valid send response.

Correctness invariant: Initiator and responder must be correctly authenticated.

Adversary: stores all messages and has 3 rules for messages on the network (one intruder is sufficient, as has total network control):

1. Overhearing
2. Replay
3. Generating messages using known nounces (not guessed)

Discussion points:

1. What attacks won't this rule out?
2. What are we actually verifying here?

TMN Protocol:
Goal: key distribution, obtain shared secret for use as a session key
Actors: initiator, responder, trusted server
Keys: public

1. $A \rightarrow S : B,\ \{N_a\}_{K_S}$ //A shares PK and encrypted nonce with server
2. $S \rightarrow B : A$ //Server shares A's PK with B
3. $B \rightarrow S : A,\ \{N_b\}_{K_S}$ //B sends A's PK and N_b
4. $S \rightarrow A : B,\ \{N_b\}_{N_A}$ // N_b should be able to be used as a session key

Attack 1 (impersonation):

1. $A \rightarrow S : B,\ \{N_a\}_{K_S}$
2. $S \rightarrow B : A$
3. $I(B) \rightarrow S : A,\ \{N_i\}_{K_S}$
4. $S \rightarrow A : B,\ \{N_i\}_{N_A}$

This is detected using their model in a few seconds.
Fix (in model): enforce source field matches up.

Attack 2 (eavesdropping):
Assume fix from above.

1.1. $I \rightarrow S :\ I,\ \{N_i\}_{K_S}$ //I want to be more open with myself

1.2. $S \rightarrow I : A$ //Here you are!

2.1. $A \rightarrow S : B,\ \{N_a\}_{K_S}$

2.2. $S \rightarrow B : A$

2.3. $B \rightarrow S : A,\ \{N_b\}_{K_S}$

1.3. $I \rightarrow S : I,\ \{N_B\}_{K_S}$ //Replay $\{N_B\}_{K_S}$

1.4. $S \rightarrow I : I,\ \{N_b\}_{N_i}$

2.4 $S \rightarrow A : B,\ \{N_i\}_{N_A}$

Solution: make S keep track of previously used session keys.

Other attack: RSA blinding:
Fact (if the key is RSA): $\{m\}_K \{n\}_K = \{mn\}_K$
Just multiply the last thing by something else. (Figure out what it should be.)
Note: they simply use a single bit to indicate that their message is blinded in model checking.

How could we do that? What state would need to be added to stuff? We'd need to model check for different values of m. Or just change the model to have an adversary controlled field inside the encrypted portion of the message.

Simplified Kerberos (if time):
Agents: KDS (key dist server), TGS (ticket granting server), client, server
Goal: authenticate client to server
Requirement: to authenticate self to server, client requires ticket from TGS. To authenticate self to TGS, requires ticket from KDS.
Major difference: here the keys are symmetric. This means that all traffic can be decrypted/encrypted with the same key.

Note: K_{tgs}, K_c are secret keys.

Protocol:
1. $C \rightarrow KDC :\ C,\ TGS$ //client requests ticket for TGS from KDC
2. $KDC \rightarrow C :\ \{K_{S1}\}_{K_C},\ \{C,\ K_{S1}\}_{K_{TGS}}$
   $\{C,\ K_{S1}\}_{K_{TGS}}$ is the ticket for the TGS.
   K_{S1} is a new session key generated by KDC to be used for communication between C and TGS
3. $C \rightarrow TGS :\ \{C\}_{K_{S1}},\ \{C,\ K_{S1}\}_{K_{TGS}},\ S$
   $\{C\}_{K_{S1}}$ is C's identity encrypted with K_{S1} (C and TGS's shared secret).

S is the identity of the server with which C hopes to communicate.

4. $TGS \rightarrow C: \ \{K_{S2}\}_{K_{S1}}, \ \{C, \ K_{S2}\}_{K_S}$

   $\{C, \ K_{S2}\}_{K_S}$ is the ticket for S.

5. $C \rightarrow S: \ \{C\}_{K_{S2}}, \ \{C, \ K_{S2}\}_{K_S}$

   $\{C\}_{K_{S2}}$ is C identifying itself to S

Problem:

3. $C \rightarrow TGS: \ \{C\}_{K_{S1}}, \ \{C, \ K_{S1}\}_{K_{TGS}}, \ S$

The server name is sent unencrypted. An attacker can change the name to S', and redirect client traffic to S'.

This attack is not allowed in the actual protocol, just the simplification used in their model.

Challenges formatting adversary:

Formalize knowledge of adversary.

Select finite set of possible actions at any point in protocol run.

Paper outline:

Challenges:

1. State explosion
2. Adversary formulation (how does proverif do this?)
3. How to model the crypto (e.g. malleability properties)

I found the following quite insightful, from [2]:

Thus the verification question is actually: Does the model of the protocol in a fixed configuration contain an error under the verification assumptions used? If it does, then clearly the protocol is flawed. If it doesn't, then nothing can directly be said about the protocol correctness for an arbitrary number of participants and concurrent protocol runs. Also there might be some cryptanalysis, and implementation detail attacks left out of the verification model. In a way finite-state verification methods can be seen as a computerized debugging aid to find a subset of the possible attacks on the protocol analyzed.

The main problem in modelling cryptographic protocols with finite-state systems is the model of the intruder. The intruder is usually modelled as a state-machine which can remove, add, and fake messages. The intruder model remembers and can alter all the messages sent as plaintext, and also remembers the encrypted forms of all message parts which have been sent encrypted. It can then fake new messages by using parts of the messages it has already seen, and some initial data it has i.e. intruders own nonces and keys. Creating such an intruder model is straightforward, but can be nonetheless subtle. The intruder is modelled as a very

nondeterministic state-machine which tries all the combinations of message parts it can create to deceive the protocol participants. Most of the message sequences generated by such an intruder model will fail, but because the search is fully automated, it is usually less time consuming than creating a smarter intruder with human modelling effort.

The problem with the intruder modelling is that in finite-state models the intruder can only have a finite amount of memory. This effectively limits the length of the protocol runs that can be analyzed, and also the number of protocol runs and participants that can be analyzed. Also when the amount of information stored by the intruder grows larger the state-explosion problem which is seen in normal protocol verification, becomes harder to handle.

One of the issues that needs to be solved is how the encryption and decryption functions should be modelled. The usual assumption used in finite-state modelling is that the used encryption functions are ideal, i.e. nothing can be said about the contents of an encrypted message without having the key needed to decrypt the message. When using such idealizations the protocol participants, and the intruder model do not actually perform any encryption or decryption functions. The message format is used to specify which fields the intruder is able to remove (and alter), and which fields have been encrypted, so their contents is not available to the intruder for use and modification. Thus the encryption doesn't change the messages passed around in the protocol (all message fields are actually sent in cleartext), but only the way in which the intruder can learn and modify the contents of each message field.

Only safety properties (nothing bad will happen) of cryptographic protocols have been verified so far with finite-state analysis. However also liveness properties (something good will eventually happen) which talk about the cyclic behavior of the system could in principle be also verified. The basic problem with verifying liveness properties is the intruder model. The intruder knowledge monotonically increases as messages are sent, and thus there are no cycles in the reachability graph of the system model. The only way to overcome this problem is to get rid of this monotonicity by allowing the intruder to also forget messages it has learned when its' memory capacity is exceeded. This is quite a radical change in the intruder model, but we think it's the only possibility when verifying liveness properties.

However even this change is not enough to verify liveness properties. The problem is that the intruder can choose to cut-off all communication between the protocol participants. Clearly if this is the case then the participants won't be able to finish the protocol. What this calls for is again a limitation on the intruder model. We could for example assume that the intruder won't be able to intercept all messages if a protocol participant keeps on sending a message. A property like this can be verified using standard finite-state verification methodology by using fairness assumptions. The liveness property verification could be useful in analyzing denial-of-service attacks, under the assumption that the intruder has only partial control over the communication network used.

The following explanations of Scalarsets and Multisets come from [3]:

- ***Scalarsets** are **enumerated types**. They are similar to enum types in C++. For example, type declaration AgentId:scalarset(5) says that variables of type AgentId can take one of 5 (unnamed) values. Scalarset types are simply finite, unordered sets of values.*
- *If you write ruleset i: ScalarType where ScalarType is a Scalarset type, then the enclosed rule will be executed once for every possible value of ScalarType. For each execution, the current value can be accessed as i.*
- ***Multisets** are **data structures**. A multiset is a set in which the same element may be included more than once. Multisets are similar to arrays, except that they are unordered.*
- *If you write multisetcount(m:mset,P) where mset is declared as a multiset and P is a predicate (i.e., a function returning true or false), then P will be applied to every element of the multiset mset, and the return value of multisetcount will be the number of elements of mset on which P evaluates to true. Within the body of P, the current element (i.e., the element to which the predicate P is being applied) can be accessed as mset[m].*

Resources for protocols:

Lowe's attack on Needham-Schroeder Public-Key [4] details the first protocol covered in the paper. This is a better explanation, the paper [4] is very short and approachable.

In the past I've found this play [5] helpful for understanding Kerberos and the sort of give and take in security protocol designs.

[1] E. Tews , R-P. Weinmann , A. Pyshkin, Breaking 104 Bit WEP in less than 60 seconds, Proceedings of the 8th international conference on Information security applications, August 27-29, 2007, Korea

[2] K. Heljanko. Can finite-state system verification methods help cryptographic protocol analysis? Technical report, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Finland, 1998.

[3] Design and Analysis of Security Protocols. Accessed April 11, 2017. http://www.cs.utexas.edu/~shmat/courses/cs395t_fall04/cs395t_murphi.html.

[4] An Attack on the Needham-Schroeder Public-Key Authentication Protocol, Gavin Lowe, Information Processing Letters, 1995.

[5] Designing an Authentication System: a Dialogue in Four Scenes. Front Cover. Bill Bryant. M.I.T., Project Athena, 1988 - 26 pages.