Automated Analysis of Cryptographic Protocols Using Murφ

John C. Mitchell Mark Mitchell Ulrich Stern
Dept Computer Science
Stanford University
Stanford, CA 94305

Abstract

A methodology is presented for using a general-purpose state enumeration tool, Murφ, to analyze cryptographic and security-related protocols. We illustrate the feasibility of the approach by analyzing the Needham-Schroeder protocol, finding a known bug in a few seconds of computation time, and analyzing variants of Kerberos and the faulty TMN protocol used in another comparative study. The efficiency of Murφ allows us to examine multiple runs of relatively short protocols, giving us the ability to detect replay attacks, or errors resulting from confusion between independent execution of a protocol by independent parties.

1 Introduction

Encouraged by the success of others in analyzing the Needham-Schroeder public-key authentication protocol using the FDR model checker for CSP [10, 11, 13, 14], we have carried out a feasibility study for a related, but somewhat different general tool called Murφ [1], pronounced "Mur-phi". In this paper, we outline our general methodology and summarize our investigation of three protocols. First, we repeat Lowe's analysis of the Needham-Schroeder protocol, finding a violation of the correctness condition in a simplified protocol, and then failing to find a violation in a repaired version of the protocol. Next, we analyze the TMN protocol [18], first finding a simple error also identified by two of the three tools described in a comparative study by Kemmerer, Meadows and Millen [7]. (These three tools appear to require more expert guidance than our brute force state exploration tool.) After modifying our system description to eliminate the first error, our system finds a second automatically. With some minor refinement of the cryptographic model, based on general principles we present in this paper, a third run also uncovers a related RSA-specific error that is explained in [18] and also discovered by the third tool in Kemmerer, Meadows and Millen's comparative study (but not the other two tools). We also investigate Kerberos, version 5, finding a failure in a simplified version based on documentation [9], and then "verifying" a repaired version that is closer to the full implementation given in RFC-1510 [8]. One interesting aspect of the Kerberos error is that it only occurs in a system configuration that includes more than the minimal number of participants.

In general, we believe that a general-purpose tool for analyzing finite-state systems may be useful for analyzing cryptographic or security-related protocols. The main challenges that arise are:

- State-space explosion, as with other tools,
- Subtleties involving formalization of the adversary or adversaries, and
- Subtleties involving properties of the encryption primitives, which may be modeled as completely secure black-box primitives, or primitives with other algebraic or "malleability" [3] properties.

One aspect of our approach that we believe will prove useful is that it is feasible to modify a Murφ system description to reflect a situation where one or more pieces of secret information have been compromised. For example, it is easy to modify our Kerberos description to give the adversary knowledge that two clients are using the same private key, without revealing the key to the adversary. The method is illustrated in our analysis of TMN to allow the adversary to generate an encryption of nm from an encryption of n, for

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891, and the National Science Foundation through grants CCR-9303099 and CCR-9629754. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, NASA, NSF or the US Government.

any numbers n and m, without allowing the adversary to decrypt any messages. The fact that an adversary can compute the RSA-encryption of one message from the RSA-encryption of another, without decrypting, is an example of "malleability" [3]. Since previous analyses tend to assume non-malleability, we expect that further insight into specific protocols may be gained by taking algebraic properties of specific cryptosystems into consideration.

Some promising future directions involve automatic translation of a higher-level protocol specification language such as CAPSL into Murφ, and combined analyses using both exhaustive finite-state analysis and formal logical methods. In particular, we hope to develop better techniques for using the results of state enumeration to simplify formal correctness proofs for potentially unbounded (or non-finite) systems, and to use formal proofs of invariants to narrow the search space for state enumeration. A larger limitation, to which we have not yet turned our attention, is that we have no way of incorporating probabilistic analysis. For example, we cannot outfit our adversary with an unbiased coin and compute the probability that a randomized attack will compromise a protocol.

Contents:

2 Outline of the methodology

Our general methodology is similar to the approach used in CSP model checking [10, 14] of cryptographic protocols. However, there are some differences between Murφ and FDR.

2.1 The Murφ verification system

Murφ [1] is a protocol verification tool that has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [2, 16, 19].

To use Murφ for verification, one has to model the protocol in the Murφ language and augment this model with a specification of the desired properties. The Murφ system automatically checks, by explicit state enumeration, if all reachable states of the model satisfy the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The Murφ language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The *state* of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by *rules*. Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e. the rule is enabled) and may change global variables. Most Murφ models are nondeterministic since states typically allow execution of more than one action. For example, in a model of a cryptographic protocol, the intruder usually has the nondeterministic choice of several messages to replay.

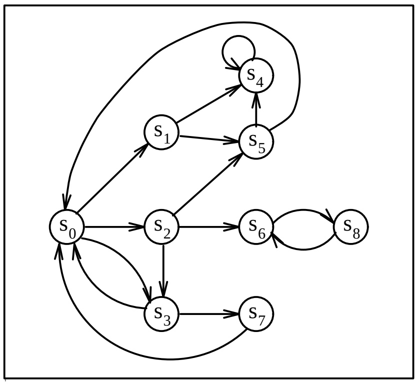

Figure 1. Sample state graph

Figure 1 shows a simple sample state graph with nine states ($s_0, \ldots, s_8$). The outgoing arcs in each state correspond to the rules that are enabled in that state. While a simulator chooses an outgoing arc at random, a verifier explores all reachable states from a given startstate (s_0).

Murφ has no explicit notion of *processes*. Nevertheless a process can be implicitly modeled by a set of related rules. The *parallel composition* of two processes in Murφ is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of *asynchronous, interleaving* concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murφ language supports *scalable* models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a down-scaled (finite state) version of the protocol. Murφ can only guarantee correctness of the down-scaled version of the protocol, but not correctness of the general protocol. For example, in the model of the Needham Schroeder protocol, the numbers of initiators and responders are scalable and defined by constants.

The Murφ verifier supports automatic *symmetry* reduction of models by special language constructs [4]. For example, in the Needham Schroeder protocol, if we have two initiators A_1 and A_2, the state where initiator A_1 has started the protocol and A_2 is idle is – for verification purposes – the same as the state where A_1 is idle and A_2 has started the protocol.

The desired properties of a protocol can be specified in Murφ by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murφ prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

There are two main differences between Murφ and FDR. First, while communication is supported in FDR by the CSP notions of channels and events, it is modeled by shared variables in Murφ. Second, Murφ currently implements a richer set of methods for increasing the size of the protocols that can be verified, including symmetry reduction [4], hash compaction [17], reversible rules [5], and repetition constructors [6]. In addition, there is a parallel version of the Murφ verifier [15]. Although available for internal use, the latter three techniques are not yet in the public Murφ release.

2.2 The methodology

In outline, we have analyzed protocols using the following sequence of steps:

1. *Formulate the protocol.* This generally involves simplifying the protocol by identifying the key steps and primitives. However, the Murφ formulation of a protocol is more detailed than the high-level descriptions often seen in the literature. The most significant issue is to decide exactly which messages will be accepted by each participant in the protocol (see Section 6 for further discussion). Since Murφ communication is based on shared variables, it is also necessary to define an explicit message format, as a Murφ type.

2. *Add an adversary to the system.* We generally assume that the adversary is a participant in the system, capable of initiating communication with an honest participant, for example. We also assume that the network is under control of the adversary and allow the adversary the following actions:

   - overhear every message, remember all parts of each message, and decrypt cyphertext when it has the key,
   - intercept (delete) messages,
   - generate messages using any combination of initial knowledge about the system and parts of overheard messages.

   Although it is simplest to formulate an adversary that nondeterministically chooses between all possible actions at every step of the protocol, it is more efficient to reduce the choices to those that actually have a chance of effecting other participants. This is discussed in more detail in Section 6.

3. *State the desired correctness condition.* We have generally found it easy to state correctness conditions, but we have no reason to believe that there are not other protocols where this step could prove subtle.

4. *Run the protocol* for some specific choice of system size parameters. Speaking very loosely, systems with 4 or 5 participants (including the adversary) and 3 to 5 intended steps in the original protocol (without adversary) are easily analyzed in minutes of computation time using a modest workstation. Doubling or tripling these numbers, however, may cause the system to run for many hours, or terminate inconclusively by exceeding available memory.

5. *Experiment with alternate formulations and repeat.* In our examples, which were known to be incorrect (except possibly Kerberos), we have repaired the detected error, either by strengthening the protocol or, where this did not seem feasible, redirecting the efforts of the adversary. In cases where a protocol appears correct, it also may be interesting to investigate the consequences of strengthening the adversary, possibly by providing some of the "secret" information.

Clearly there are many calls for creativity and good judgment; this is not in any way an automatic procedure that could be carried out routinely from a high-level description of a protocol. However, as we gain more experience with the method, we anticipate development of certain tools that will make the process easier to carry out.

3 Needham-Schroeder Public-Key Protocol

3.1 Overview of the protocol

The Needham-Schroeder Public-Key Protocol [12] aims at mutual authentication between an initiator A and a responder B, i.e. both initiator and responder want to be assured of the identity of the other.

As in [10], we only study a simplified version of the protocol, which can be described by the following three steps.

$$\begin{array}{rcl} A \rightarrow B & : & \{N_a, A\}_{K_b} \\ B \rightarrow A & : & \{N_a, N_b\}_{K_a} \\ A \rightarrow B & : & \{N_b\}_{K_b} \end{array}$$

The initiator A sends a nonce N_a (i.e. a newly generated random number) and its identifier to reponder B, both encrypted with B's public key K_b. Responder B decrypts the message and obtains knowledge of N_a. It then generates a nonce N_b itself and sends both nonces encrypted with A's public key to A. Initiator A decrypts the message and concludes that it is indeed talking to B, since only B was able to decrypt A's initial message containing nonce N_a; B is authenticated. In a corresponding fashion, A is authenticated after the third step of the protocol. (This is not entirely correct, though, as we shall see.)

3.2 Modeling the protocol

Due to space constraints, we look only at the initiator part of the model in detail. The data structures for the initiator are as follows:

```
const
  NumInitiators: 1;
type
  InitiatorId: scalarset (NumInitiators);
  InitiatorStates: enum{I_SLEEP,I_WAIT,I_COMMIT};
  Initiator: record
    state: InitiatorStates;
    responder: AgentId;
  end;
var
  ini: array [InitiatorId] of Initiator;
```

The number of initiators is scalable and defined by the constant `NumInitiators`. The type `InitiatorId` can be thought of as a subrange `1...NumInitiators` with the difference that automatic symmetry reduction is invoked by declaring this type a `scalarset`. The state of each initiator is stored in the array `ini`. In the startstate statement of the model, the local state (stored in field `state`) of each initiator is set to `I_SLEEP`, indicating that no initiator has started the protocol yet.

The behavior of an initiator is modeled with two Murφ rules. In the first rule, the initiator starts the protocol by sending the initial message to some agent and changes its local state from `I_SLEEP` to `I_WAIT`. The second rule models the reception and checking of the reply from the agent, the commitment and the sending of the final message.

The Murφ code of the first rule looks as follows:

```
ruleset i: InitiatorId do
  ruleset j: AgentId do
    rule "initiator starts protocol"
      ini[i].state = I_SLEEP &      -- condition
      !ismember(j,InitiatorId) &
      multisetcount (l:net, true) < NetworkSize
    ==>
    var
      outM: Message;   -- outgoing message
    begin                            -- action
      undefine outM;
      outM.source  := i;
      outM.dest    := j;
      ... set remaining fields of outM
      multisetadd (outM,net);
      ini[i].state     := I_WAIT;
      ini[i].responder := j;
    end;
  end;
end;
```

Note that the rule is enclosed by two `ruleset` statements. These statements make the rule scalable: it is instantiated for each initiator of type `InitiatorId` and for each agent of type `AgentId`. Thus, when one changes the constant `NumInitiators`, the number of rules automatically adapts to this change. The identifiers of the initiator and the agent of a particular instantiation are assigned to the local variables `i` and `j`, respectively, and can be used in the rule.

The condition of the rule is that initiator `i` is in the local state `I_SLEEP`, that agent `j` is not an initiator (and hence either reponder or intruder), and that there is space in the network for an additional message. The network is modeled by the shared variable `net`. Each network cell can hold one message of the protocol. In the action of the rule, the outgoing message is constructed and added to the network. In addition, the local state is updated and the identifier of the intended conversant is stored in state variable `responder`.

Now, we consider the second rule of the initiator.

```
ruleset i: InitiatorId do
  choose j: net do
    rule "initiator reacts to nonce received"
      ini[i].state = I_WAIT &   -- condition
      net[j].dest = i
    ==>
    var
      inM, outM: Message;
    begin                       -- action
      inM := net[j];
      multisetremove (j,net);
      if inM.key=i &
         inM.mType=M_NonceNonce &
         inM.nonce1=i then
        ... set fields of outgoing message outM
        multisetadd (outM,net);
        ini[i].state := I_COMMIT;
      end;
    end;
  end;
end;
```

Since this rule is enclosed by a `ruleset` and a `choose` statement, it is instantiated for each initiator `i` of type `InitiatorId` and for each cell `j` of the network `net`.

The condition of the rule is that initiator `i` is in the local state `I_WAIT` and that the destination of the message in network cell `net[j]` is actually this initiator. In the action of the rule, first the message is deleted from the network. The initiator then checks if it can decrypt the message, if the message type is correct and if the message contains the correct nonce. If all conditions hold, an outgoing message is constructed and added to the network, and the initiator commits to the session by changing its local state to `I_COMMIT`.

In our Needham Schroeder model, only one nonce per agent is needed, since we considered neither multiple runs of the protocol nor multiple conversants at the same time for the same agent. This allowed us to simply model nonces by variables of type `AgentId`, which represents the identifiers of all agents (initiators, responders and intruders) in the system and thus offers one value for each agent. In the general case, however, we would have to change the model to allow more than one nonce per agent. This change can be readily accomplished by calling a "nonce generator" function that yields a new value of type `1...MaxNonces` each time a nonce is needed. In fact, similar functions were used in the case of the TMN and Kerberos protocols.

Now we give one of the two invariants we use to specify the correctness condition of the protocol, which is that initiator and responder must be correctly authenticated.

```
invariant "responder correctly authenticated"
  forall i: InitiatorId do
    ini[i].state = I_COMMIT &
    ismember(ini[i].responder, ResponderId)
    ->
    res[ini[i].responder].initiator = i &
    ( res[ini[i].responder].state = R_WAIT |
      res[ini[i].responder].state = R_COMMIT )
  end;
```

This invariant basically states that for each initiator `i`, if it committed to a session with a responder, this responder (whose identifier is stored in `ini[i].responder`), must have started the protocol with initiator `i`, i.e. have stored `i` in its field `initiator` and be in state `R_WAIT` or `R_COMMIT`.

The intruder in our model maintains a set of overheard messages and an array representing all the nonces it knows. The intruder is modeled with three rules: one for overhearing and intercepting messages, one for replaying messages from the set of overheard messages, and one for generating messages using the known nonces and injecting them into the network.

3.3 Verification results

Murφ was able to discover the protocol error described by Gavin Lowe [10] in 1.7 seconds. After fixing the protocol as suggested in [10], Murφ failed to find any additional bugs in the protocol.

Table 1. Number of reachable states dependent on the model parameters in the Needham-Schroeder example

number of ini.	res.	int.	size of network	states	time
1	1	1	1	1706	3.1s
1	1	1	2	40 207	82.2s
2	1	1	1	17 277	43.1s
2	2	1	1	514 550	5761.1s

Table 1 shows the number of reachable states and runtime on a 200 MHz SGI INDY when varying the parameters of the corrected model. The size of the network is given in the number of messages it can hold. Since the intruder is in complete control of the network, one message seems sufficient here. Similarly, since the intruder is modeled to be very powerful, adding a second intruder does not to increase the intruder power.

4 The TMN Protocol

The Tatebayashi, Matsuzaki, and Newman (TMN) Protocol [18] is a key distribution protocol for digital mobile communication systems, such as cellular networks. When one network node (the initiator) wishes to communicate with another (the responder), the initiator and responder attempt, with the aid of a trusted central server, to obtain a shared secret for use as a session key. The original version of the protocol can be formalized as:

$$\begin{array}{lcl} A \rightarrow S & : & B, \{N_a\}_{K_S} \\ S \rightarrow B & : & A \\ B \rightarrow S & : & A, \{N_b\}_{K_S} \\ S \rightarrow A & : & B, \{N_b\}_{N_a} \end{array}$$

At the conclusion of the protocol, it is expected that N_b can be used as a session key for communication between A and B.

We know of two major classes of attacks against this version of the TMN protocol. In the first, an intruder impersonates either A or B to the other, and one of the parties is tricked into communicating with the imposter. In the second, the intruder is able to eavesdrop on a communication between the participants.

Although we had prior knowledge of both attacks [18, 7], we were able to uncover them in a natural way using Murφ. Our analysis of the TMN protocol provides a useful benchmark, since it has been used as a basis for comparing three other systems for cryptographic protocol analysis [7]. The operators of those systems also had knowledge of the attacks against TMN, so we had no obvious advantage. In contrast to two of those tools, Murφ requires no operator intervention once the model is specified.

We use a single model to obtain several variations on these two classes of attacks. The model has various parameters that can be adjusted to indicate, for example, how many participants are available, and how much information the intruder can remember. In the simplest model, the intruder has no memory, and does not even attempt to decrypt encrypted messages. Murφ is able to find the following attack in a few seconds, after searching 7126 states:

$$\begin{array}{lcl} A \rightarrow S & : & B, \{N_a\}_{K_S} \\ S \rightarrow B & : & A \\ I(B) \rightarrow S & : & A, \{N_i\}_{K_S} \\ S \rightarrow A & : & B, \{N_i\}_{N_a} \end{array}$$

Here, the notation $I(B)$ indicates I impersonating B. In this execution of the protocol, A is fooled into thinking it is communicating with B, when in fact A now has a shared secret with I. However, it is perhaps reasonable to assume that it is impossible (for reasons specific to the hardware involved) for intruders to impersonate other parties, or that some authentication step could be added to the protocol to ensure that impersonation of other parties is impossible. Each message in our model had a "source" field. In order to model the impossibility of impersonation, we simply ensured that the intruder put no identifier other than its own in this field. When modeling this restricted situation, Murφ finds another bug, after searching 3327 states:

$$\begin{array}{lcl} I \rightarrow S & : & I, \{N_i\}_{K_S} \\ S \rightarrow I & : & I \\ A \rightarrow S & : & B, \{N_a\}_{K_S} \\ S \rightarrow B & : & A \\ B \rightarrow S & : & A, \{N_b\}_{K_S} \\ I \rightarrow S & : & I, \{N_b\}_{K_S} \\ S \rightarrow I & : & I, \{N_b\}_{N_i} \end{array}$$

Here, I uses a replay attack to force S to decrypt the session key used between A and B. If, however, we allow the server to keep track of previously used session keys, Murφ searches 1070 states, and finds no bugs.

There is, none-the-less, one final twist. The original TMN paper suggested the use of RSA for the public-key encryption algorithm. RSA encryption has the property that the encryption of n multiplied by the encryption of m is equal to the encryption of mn. In other words,

$$\{m\}_K \{n\}_K = \{mn\}_K.$$

Thus, an encrypted message m can be "blinded" by multiplying it by the encryption of some fixed factor n. After decryption, the result may be divided by the same factor n to obtain the original message.

By allowing the intruder to "blind" messages, we obtain the same attack as above (in 10,341 states), except that in the penultimate message, the intruder blinds N_b, so that the server does not realize that a replay has occurred. Our sever refuses to accept session keys that are duplicates of previously used keys, but in this case is fooled because the session key it is given is not an exact duplicate of a previous session key; instead, it is such a session key multiplied by some factor.

As far as we know, we have uncovered (again) all the attacks against the TMN protocol found previously by either humans or machines. The search times (reflected in the number of states searched) were reasonably short. Unfortunately, attaining those times required care when coding the model description for

Murφ. It was important to bound the amount of information the intruder could store, and the amount of information the server could store rather tightly. We allowed the intruder to generate arbitrary messages, but it was necessary to prevent the intruder from generating messages that would never be accepted by the other parties, since this lead to state explosion. We were unable to model the actual RSA encryption algorithm, as the range of values required was too large for our finite-state tool.

We did encounter some difficulties using Murφ to model the TMN protocol. In particular, Murφ does not at present have a way to list all the errors found in a particular model; as soon as one error is found, the program exits. This meant that we had to make changes to the model to find various errors. Specifically, when our adversary was sufficiently powerful to find one immediate error, Murφ would not reveal additional bugs that required only a weaker adversary. However, there seems no obvious reason that Murφ could not be modified to output all violations of the specified correctness conditions. Overall, Murφ severed our purposes very well and was not unduly difficult to use.

5 Kerberos

5.1 Overview of the protocol

The Kerberos protocol [9] aims at mutual authentication between a client C and a server S. Unlike the Needham-Schroeder protocol, it uses secret-key cryptography. Before the client authenticates itself to the server, it communicates with two other agents: a Key Distribution Center (KDC) and a Ticket Granting Server (TGS). For authenticating itself to the server, the client needs a ticket issued by the TGS, and to get this ticket, another ticket issued by the KDC is needed.

We study a reduced version of the protocol, which can be described in the following five steps. This version can be obtained from the Kerberos description in [9] by leaving out timestamping and nonces. In addition, the reduced protocol only authenticates the client to the server; for mutual authentication, a sixth step were required.

$$
\begin{aligned}
C \rightarrow \mathrm{KDC} &\quad : \quad C,\ \mathrm{TGS} \\
\mathrm{KDC} \rightarrow C &\quad : \quad \{K_{s1}\}_{K_c},\ \overbrace{\{C, K_{s1}\}_{K_{\mathrm{TGS}}}}^{\text{ticket for TGS}} \\
C \rightarrow \mathrm{TGS} &\quad : \quad \overbrace{\{C\}_{K_{s1}}}^{\text{authent.}},\ \overbrace{\{C, K_{s1}\}_{K_{\mathrm{TGS}}}}^{\text{ticket for TGS}},\ S \\
\mathrm{TGS} \rightarrow C &\quad : \quad \{K_{s2}\}_{K_{s1}},\ \overbrace{\{C, K_{s2}\}_{K_S}}^{\text{ticket for } S} \\
C \rightarrow S &\quad : \quad \overbrace{\{C\}_{K_{s2}}}^{\text{authent.}},\ \overbrace{\{C, K_{s2}\}_{K_S}}^{\text{ticket for } S}
\end{aligned}
$$

In the first step of the protocol, the client C requests a ticket for the TGS from the KDC. The KDC generates a new session key K_{s1}, which will be used for the communication between C and the TGS. The KDC also generates a ticket for the TGS (encrypted with the TGS's secret key K_{TGS}) and sends the ticket to C. In addition, the session key is also sent to the client (encrypted with the C's secret key K_c). The client decrypts the session key, generates a so-called authenticator and sends authenticator, the received ticket and a request for a ticket for S to the TGS. The TGS checks this message by decrypting the ticket, learning the session key and decrypting the authenticator. Since the session key and the client knowing this key were transmitted in the ticket and the authenticator was encrypted using this very session key, the client is now authenticated to the TGS. In a corresponding fashion, the client is authenticated to the server in the last two steps of the protocol.

5.2 Modeling the protocol

The basic ideas in modeling the Kerberos protocol were the same as in the Needham-Schroeder case. The Murφ model of Kerberos, however, had more than twice as many lines as the Needham-Schroeder model because of the greater complexity, especially of the intruder model. In the Kerberos case, we explicitly modeled a "session key generator" (as a Murφ function) that yields a new session key every time it is called. The correctness criterion was specified with invariants in a similar fashion to the Needham-Schroeder case.

5.3 Verification results

We started verification on a Kerberos configuration with one instance each of client, server, TGS and KDC. Murφ did not find any bug in this configuration and explored 109 282 states in 872.1 seconds.

After we increased the number of servers to two, however, Murφ found a problem in the protocol after exploring 3405 states in 18.5 seconds: in the third step, the name of the server that the client wants to get a ticket for is transmitted unencrypted. An intruder can change this server name (in our model by intercepting the message first and then replaying it with a new server name) and later re-direct the message transmitted from the client to the server in the last step of the protocol to the new server. Thus the client's communication can be re-directed to another server, without either of them realizing. In this way communication from an honest client might also be re-directed to a "compromised" server.

While this problem can occur when the protocol is implemented according to [9], the exact definition of the protocol in RFC-1510 [8] prevents this attack by calculating a checksum over the unencrypted part of the third message and including this checksum into the authenticator. Upon receipt of the message, the TGS checks the integrity of the message with this checksum and possibly rejects it.

After fixing the Murφ model of Kerberos by including the previously unencrypted server name into the authenticator, Murφ finished the two-server case without finding any errors, exploring 79 583 states in 389.1 seconds.

There are several possible extensions to our current Murφ model of Kerberos that might reveal problems in the protocol definition in RFC-1510. We expect the two most promising areas to be to allow multiple runs of the protocol in combination with adding timestamps to the model, and to also model the remote authentication functions of the Kerberos protocol. We anticipate that one prerequisite for successfully using Murφ on the extended model is an optimization of the intruder model to avoid the generation of non-accepted messages.

6 Discussion

In this section, we make some remarks on our modeling assumptions and those aspects of Murφ that seemed particularly helpful or cumbersome. In the process, we comment on two aspects of our approach that could be automated: formulation of the message format and optimization of the adversary, based on global analysis of the behavior of all participants.

6.1 Modular system description and the "network"

In general, it takes some effort to describe a simple protocol in the form needed to run Murφ. For one of us, an experienced programmer with no prior familiarity with Murφ, it took about half a day to follow the example of the Needham-Schroeder protocol and prepare a model of the TMN protocol. Modifying the description to eliminate all variants of one protocol error and find a second and third error took many more hours, but significantly less than a week.

Murφ is intended to allow modular description of a system. For example, there is no need to define a single global state variable; each component may have its own local state, which the Murφ system automatically integrates into a global state when the system description is compiled. However, since the network is used by all of the participants in a distributed protocol, it is necessary to understand all of the possible messages by all participants when modeling the network. In the Needham-Schroeder protocol, which only has three messages, the Murφ characterization of the message format must have enough fields for all of the kinds of information that occur in any of the three intended messages. These leads to a message type that contains the following fields:

```
source: AgentId;      -- source of message
dest:   AgentId;      -- intended destination
key:    AgentId;      -- encryption key
mType:  MessageType;  -- type of message
nonce1: AgentId;      -- nonce1
nonce2: AgentId;      -- nonce2
                    -- OR sender identifier
                    -- OR empty
```

with comments that hopefully make the intent of each message field clear. The message type `MessageType` is an enumeration type with three possible values:

```
M_NonceAddress -- {Na, A}Kb: nonce and address
M_NonceNonce   -- {Na,Nb}Ka: two nonces
M_Nonce        -- {Nb}Kb:    one nonce
```

If we choose to refine the model by incorporating more information about the encryption function, then the message format may have to be changed, and this may require changes to all of the components of the system. This problem suggests that it would be worthwhile to develop translations from a higher-level notation into Murφ.

6.2 Modeling the adversary

Formulating the adversary is often relatively complicated and can consume more than half of the time required to prepare a Murφ description of a protocol. There are two main challenges in formulating an adversary to a protocol:

- Formalize the "knowledge" of the adversary, as a function of some initial conditions and the set of messages an adversary has observed up to that point in the run of the protocol. By "knowledge," we mean the set of possible entries that an adversary may insert in each message field.

- Select a finite set of possible adversary actions at any point in the run of the protocol, using the knowledge the adversary has at that point.

The first activity is more conceptual; the later involves pragmatic considerations.

For the protocols we have examined, we have characterized the knowledge of the adversary as simply the union of some set of initial data, such as public keys and the names of participants, and the data obtained by overhearing any message sent from one participant to another. This is relatively straightforward, given the message format and the assumption that an encrypted message may be decrypted only if the adversary knows the associated key. In particular, the initial data is finite (and small) and the data remembered from previous communication is also finite (and small), keeping us within the domain of finite-state systems. For adversaries of this form, we believe it will be possible to generate some or all of the adversary description automatically from formulations of other parts of the system.

With more subtle cryptographic assumptions, as discussed below, it becomes more difficult to model the adversary accurately. For example, if we want to allow the adversary to transmit $\{n \cdot i\}_K$, on the basis of seeing only $\{n\}_K$, for small integers $i = 1, 2, 3$, say, then we currently have to specify this explicitly in our intruder description. Moreover, we either have to model the message content as an integer and perform multiplication, or model the message content as a pair and put both n and i into the message. In this way, the intruder model affects the message format.

The second general intruder consideration is that while we may want to run the protocol against the most capable, most nondeterministic adversary possible, this may cause the analysis to run more slowly, or consume more space, than desirable. While we have not had to weaken the adversary, we have found it useful to optimize the adversary using information about the set of messages that other participants will actually accept. For example, suppose the receiver of a message computes a checksum and discards any message without the correct checksum. Then an unoptimized nondeterministic adversary may generate messages that do not have a correct checksum. However, these messages would have no effect on the protocol, if every other participant would reject them. Therefore, we can improve the running time and space of the analysis by rewriting the adversary to avoid generating these useless messages. Depending on the size and complexity of the system, we have found this sort of optimization to alter the size of the state space by one to several orders of magnitude. Therefore, we consider it a useful direction to explore automatic optimization of the adversary.

6.3 Black-box cryptography

With the exception of the third TMN attack, we have analyzed protocols only under the strict and somewhat unrealistic assumption that an adversary may generate encrypted text only if it knows the text and the encryption key. There are two ways we have considered relaxing this assumption.

In an environment where participants have private keys, it may be possible for an adversary to know, or observe, that two participants seem to be using the same key. We can model this by giving our adversary the *interkey* capability of generating a message containing $\{x\}_{K_i}$ whenever it has overheard $\{x\}_{K_j}$, for some specific pair of participants i and j. If this allows the adversary to compromise a third client that does not share this key, then we would have uncovered a weakness in the protocol.

A second way we might weaken the assumption of perfect black-box cryptography is through modeling *malleability* [3] properties of cryptosystems. In brief, a cryptosystem is malleable if it is possible to compute the encryption of one message from the encryption of another, without decrypting. Put more mathematically, an encryption scheme $\{\,\cdot\,\}_{(\,\cdot\,)}$ is a family of encryption functions, indexed by keys, together with associated decryption functions. An encryption scheme is malleable if it is possible to compute $\{f(x)\}_K$ from $\{x\}_K$, for some function f, without knowing the decryption key K^{-1} or decrypting the message. An example of a malleable encryption scheme is RSA, which allows the encryption of a multiplicative factor of x to be computed easily from the encryption of x.

In the TMN protocol, we explored the possibility of sending $\{xi\}_K$, on the basis of seeing only $\{x\}_K$, without knowing the decryption key. However, to avoid the complexity of multiplication, we did this with some simplification. Since this kind of "blinding" disguises the contents of the message completely, the particular multiplicative factor used to blind a message makes no difference. Therefore, we used a single bit, indicating that the message is blinded. If we were delving further into other detailed properties of the particular encryption algorithms (in this case RSA), we would be unable to make this simplification, but for our purposes, it was perfectly adequate. By this method, we were able to explore the impact of blinding without suffering undue state-space explosion.

While an interkey assumption is relatively straightforward to incorporate using our approach, malleability may be harder to handle using only finite-state systems since allowing a function f to be applied repeatedly to some datum could violate any specific finite bound the set of possible messages.

7 Conclusion

This paper reports the results of a feasibility study on finite-state exploration tools for analyzing cryptographic and security-related protocols. In comparison with related studies using other automated tools, such as [7, 10, 11, 14], we believe we have achieved promising success: in all cases, we seem to have done as well or better than other tools. In order to facilitate comparison between the various tools and approaches, we have focussed on protocols that have been analyzed by others. We are also analyzing newer protocols, in hopes of reporting new discoveries not already uncovered by previous researchers.

As a tool for analyzing cryptographic protocols, Murφ has some notable strengths and some mildly irritating rough spots. We found the input language and many of the features easy to learn and adapt for our purposes. The efficiency of the tool was also warmly appreciated. We did, however, wish for better control of the search strategy. Even with a large search space, we believe it would be useful to be able to set interrupt points (much as one might do for an interactive debugger) and then direct the search by hand from that point. We also had difficulty disabling certain error traces. Instead of modifying the adversary in the TMN analysis, to avoid one form of attack, we would have liked to have supplied a separate Murφ directive to avoid extending certain traces.

One promising direction for further development of our tools and methodology would be to devise translations from a higher-level notation into Murφ. As explained in Section 6, we believe that automatic generation of the network message format and automatic or partially-automatic generation of an adversary are both practical and beneficial.

Finally, we expect to integrate work on formal logic into our state-exploration approach. This could provide a more accurate method for determining correctness of a protocol, independent of size limitations. For this purpose, we anticipate investigation of some form of protocol homomorphism properties. We also hope to use formal proofs of invariants to narrow the search space for state enumeration.

Acknowledgments: We would like to acknowledge the assistance and encouragement of David Dill, Cynthia Dwork, Stephen Freund, Li Gong, Francois Guimbretiere, John Rushby, Jens Skakkebaek and Vitaly Shmatikov.

References

[1] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.

[2] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.

[3] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography (extended abstract). In *Proc. 23rd Annual ACM Symposium on the Theory of Computing*, pages 542–552, 1991.

[4] C. N. Ip and D. L. Dill. Better verification through symmetry. In *11th International Conference on Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.

[5] C. N. Ip and D. L. Dill. State reduction using reversible rules. In *33rd Design Automation Conference*, pages 564–7, 1996.

[6] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. In *Computer Aided Verification. 8th International Conference*, pages 147–58, 1996.

[7] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *J. Cryptology*, 7(2):79–130, 1994.

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

[9] J. Kohl, B. Neuman, and T. Ts'o. *The evolution of the Kerberos authentication service*, pages 78–94. IEEE Computer Society Press, 1994.

[10] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems.* Springer-Verlag, 1996.

[11] C. Meadows. Analyzing the Needham-Schroeder public-key protocol: a comparison of two approaches. In *Proc. European Symposium On Research In Computer Security.* Springer Verlag, 1996.

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

[13] A. W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *CSFW VIII*, page 98. IEEE Computer Soc Press, 1995.

[14] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[15] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. Submitted for publication.

[16] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[17] U. Stern and D. L. Dill. A new scheme for Memory-efficient probabilistic verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, pages 333–48, 1996.

[18] M. Tatebayashi, N. Matsuzaki, and D. Newman. Key distribution protocol for digital mobile communication systems. In *Proc. CRYPTO '89*, pages 324–333, 1990.

[19] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC™-I. In *32nd Design Automation Conference*, pages 7–12, 1995.