HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **14**; September 1995

ISSN 0783–540X
ISBN 951–22–2756–8

MODELLING THE NEEDHAM-SCHRÖDER AUTHENTICATION PROTOCOL WITH HIGH LEVEL PETRI NETS

Tuomas Aura
Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **B:** Technical Reports
No. **14**; September 1995

ISSN 0783–540X
ISBN 951–22–2756–8

Modelling the Needham-Schröder authentication protocol with high level Petri nets

Tuomas Aura

Abstract: In this paper, security of the Needham-Schröder key distribution protocol is modelled and analyzed with predicate/transition nets, and along that, a methodology for modelling cryptographic protocols with high level Petri nets is developed. The main goal is clarity of the model and its feasibility for automated analysis. The intruder and the communication channels are modelled as one entity that has complete control over all messages in the system. The intruder model is based on the concepts of memory and learning. Special care is taken that the model captures all possible actions of the intruder. Guidelines are given for finding the minimal number of model parts that represents a system with an arbitrary number of entities and concurrent protocol runs. Techniques of coping with state space explosion in reachability analysis, the stubborn set method and prioritizing of transitions, are discussed. Introduction of set type places or negative arcs in the net formalism is proposed as a solution for reducing the storage space requirements of the reachability graph. In experiments with the reachability analysis tool PROD, the model proved efficient in detecting protocol failures. Further optimization of the formalism and tools is necessary for the presented methods to be useful in full verification of real size protocols.

Keywords: Cryptographic protocols, Needham-Schröder, formal model, Petri nets, predicate/transition nets, PROD

Printing: TKK Monistamo; Otaniemi 1995

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358–0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–465 077
E-mail: lab@saturn.hut.fi

Contents

1 Introduction

With the growth of computer networks and network applications, security of communication has become an increasingly important issue. Cryptographic protocols are rules specifying how secure communication is achieved over insecure communication channels using cryptographic techniques. Although the protocols are based on public key and symmetric cryptographic algorithms, they form an independent area of study from that of the algorithms. In cryptographic protocol design, encryption and authentication algorithms are taken as unbreakable in the time span of a single communication process.

Our goal is to devise a formal analysis method for the security of protocols. The approach taken is to model the protocol with high level Petri nets and to locate errors or verify security with reachability analysis. For concreteness, we give a case study of a well known key distribution protocol. The observations made in the example case are generalized to all protocols, where possible. Special care is taken to keep the intruder model as simple as possible in order to convince the reader that we have captured in the model all possible attacks by the infinitely resourceful enemy. Security is our only goal; no attempt is made to analyze other properties of the protocol.

This paper is based on ideas evolved in professor Leo Ojala's seminar course on Applications of formal methods in computer science in spring 1995 in Digital Systems Laboratory of Helsinki University of Technology. The main source for the idea of modelling and analyzing safety of cryptographic protocols with high level Petri nets was Benjamin B. Nieh's thesis [7, 6]. The model in that paper, however, provoked some criticism. A less complicated model of the intruder is needed for automated analysis. Also, with a simpler model, it is easier to see what assumptions have been made in its design. In this paper, we present a conceptually simple model of the intruder and communication channels. We have completely restructured the model of [7] to make it easily understandable and feasible for automated analysis.

In Sec. 2 we describe our view of a communication system with an intruder. The Needham-Schröder (N-S) authentication protocol and a predicate/transition net formalism for protocol modelling are introduced. After that, we can build a detailed model of the N-S protocol entities in Sec. 3. The intruder model in Sec. 4 is the key part of the model. We make the intruder as simple and general as possible. In Sec. 5, we give the criteria that the protocol has to fill in order to be secure. The quantities of the model parts are discussed separately in Sec. 6. The model has to represent a system with an arbitrary number of parts, but can only be of limited size for analysis. Finally, we complete the discussion with methods of reachability analysis in Sec. 7.

2 Communication system architecture

In this paper, we consider communication systems that are composed of protocol entities and unreliable channels for transferring messages between them. All messages sent to a channel are prone to attacks by an intruder. The intruder can read, remove, replace any messages in the channels. It can also insert new messages to the channels and move messages from one channel to another. Thus, the intruder has a complete control over the communication channels. We view all channels and intruders of the system as a single intruder who takes all the messages sent by protocol entities and gives them ones to receive (Fig. 1).

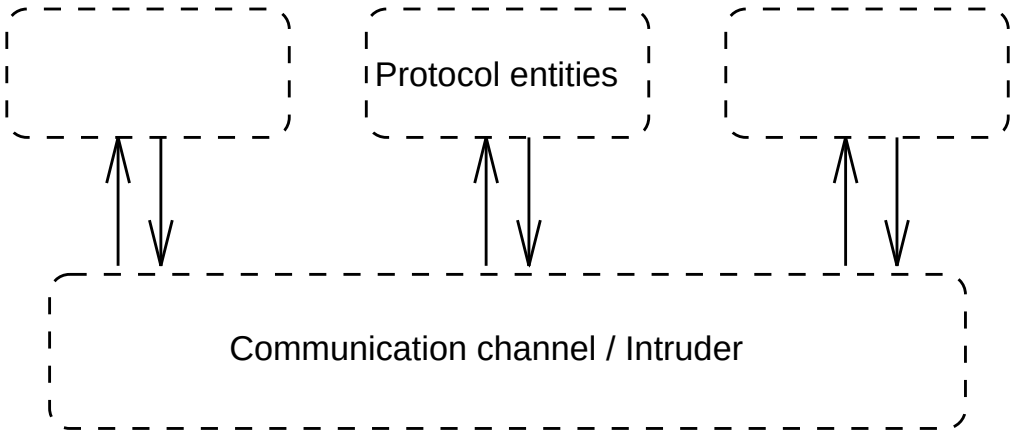

Figure 1: Communication system

In a cryptographic protocol, there are usually several different kinds of entities. There are communicating users that take the cryptographic system as a kind of service and there are trusted key and ticket servers that only exist because they are needed as a part of the protocol. A single entity may be able to adapt to different roles in the process, depending on the messages it receives and the inherent needs it has. In a system of homogeneous users, for example, a user may behave entirely differently when initiating communication itself and when responding to other's messages. The number of entities in the system is often unlimited or large.

In a typical system, the protocol is used unlimited number of times between varying entities. We have to capture in our model a snapshot of the system and analyze behavior of the system starting from that moment. We cannot follow the system running forever but have to stop at some time. Therefore we have to identify a protocol run, a sequence of events after which the state of the system is similar to the one it started in. Then we can observe the behavior of the system during a single protocol run and conclude something about a system where an arbitrary number of successive and interleaved protocol runs take place.

2.1 Types of data

Messages are composed of atomic data items such as text, keys, identifiers, serial numbers and time stamps. The text of a message can be confidential, non-confidential, incorrect (substituted by an intruder) and even randomly generated. There are public, private and symmetric keys for encrypting and decrypting messages. Serial numbers, time stamps and random numbers are used to identify messages.

Messages are either plain or encrypted sequences of encrypted submessages and atomic data items. In practical protocols, the encrypted submessages usually consist only of atomic data items, although, in theory, the recursive structure of submessages could be deeper. Even though some messages are encrypted, we assume that differently structured messages can always be recognized. The protocol entities only send and receive messages that have a correct structure. An intruder does not create malformed messages either since no entity would read them from the channel.

All entities have some inherent data, such as keys and time, before the communication process begins. They use this data to compose messages and decrypt and validate received ones.

2.2 The Needham-Schröder authentication protocol

As an example of a protocol we have chosen one with a well-known weakness, which we will try to rediscover. The Needham-Schröder (N-S) authentication protocol uses a trusted key distribution center (KDC) and symmetric private key cryptography to establish secure connections between users. We give a concise description of how the protocol is used. For a better introduction to the N-S protocol and private key cryptography, see any standard textbook on the topic [9, 1].

When a user wants to communicate with another, it is called user A and the other is user B. User A first sends a message to the KDC requesting a new session key. (message 1 in Fig. 2) The request contains identifiers of both A and B (AId, BId) and a random number ($Rn1$) generated by A.

The KDC answers to user A with a message (message 2) encrypted with A's private key (KA). The message contains the random number ($Rn1$), B's identifier (BId), the new session key (SK) and a submessage ($E[SK, AId; KB]$) encrypted with B's private key (KB). A uses the random number to identify the specific request as it may want to establish several simultaneous connections. If the random number and B's identifier match the previous request, A accepts the new session key and forwards the encrypted submessage to B.

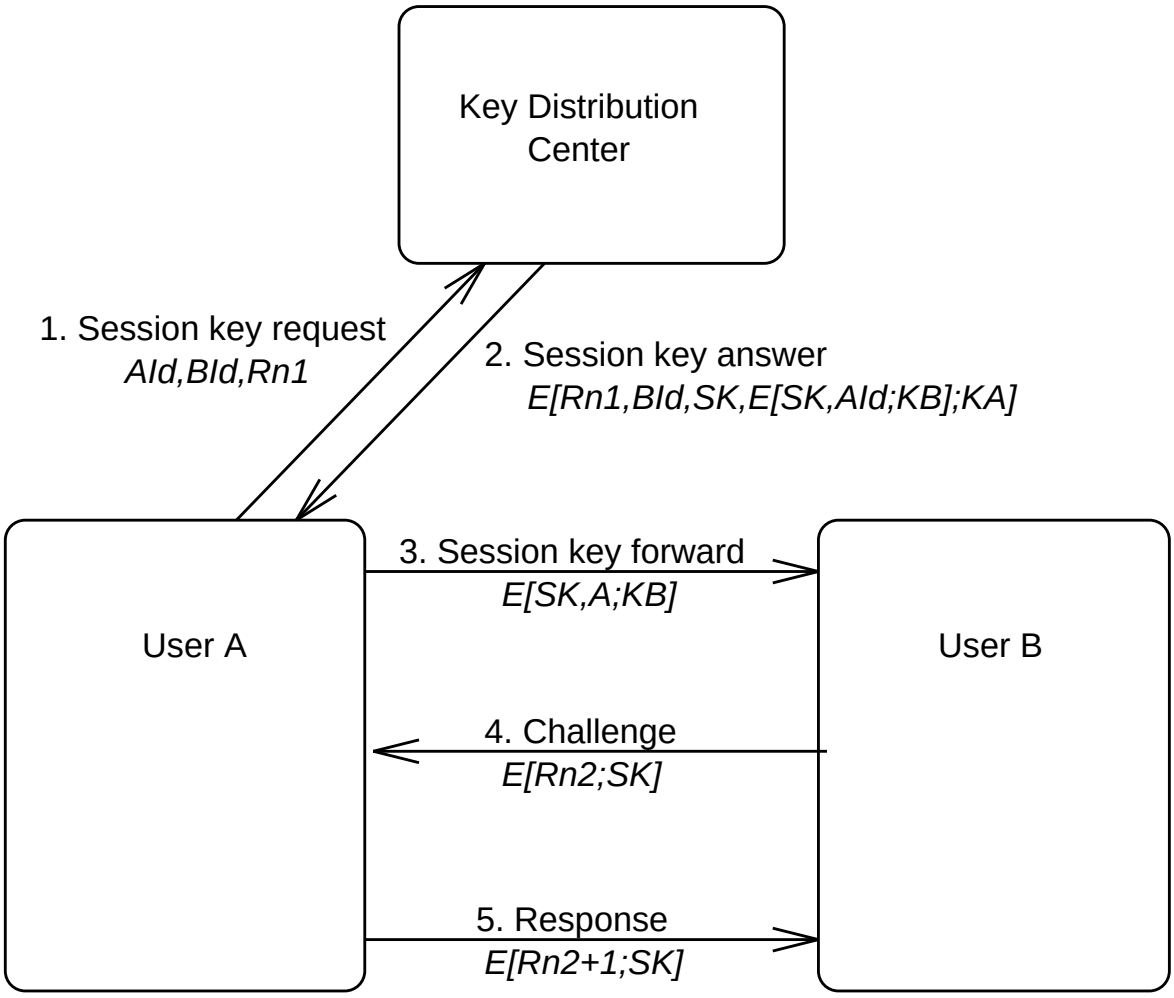

Figure 2: The Needham-Schröder protocol

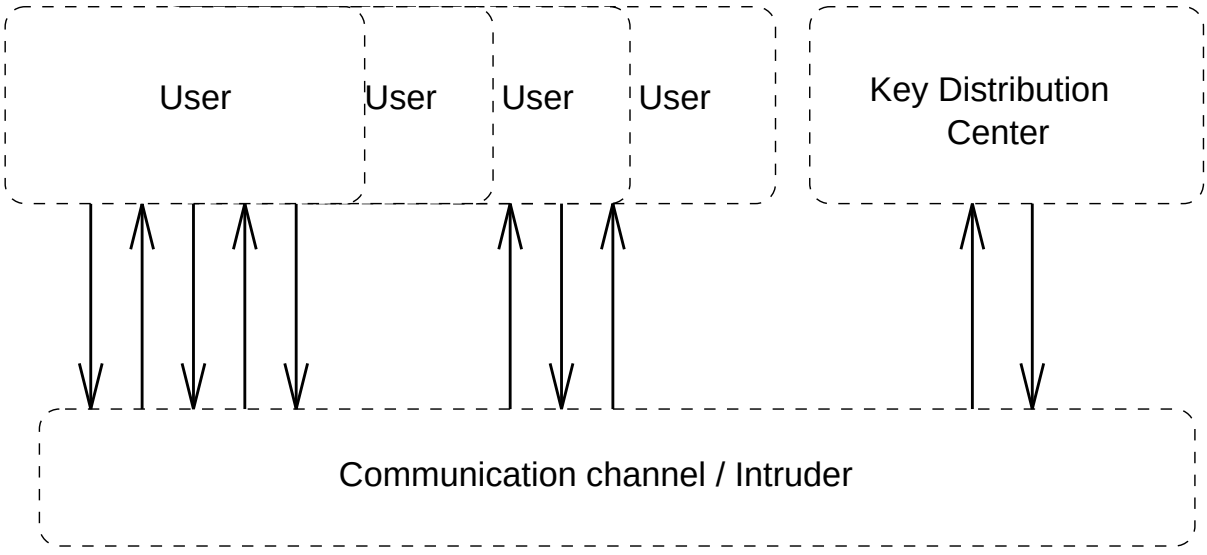

Figure 3: Top level model of the Needham-Schröder protocol

When user B receives the forward (message 3), it finds the session key (SK) and A's identifier (AId) in the message. It challenges A's knowledge of the session key by generating a random number ($Rn2$), encrypting it with the session key and sending it to A (message 4). User A can now decrypt the random number with the session key. It adds one to the number, encrypts it with the session key and sends it back to B (message 5). If B receives the random number properly increased ($Rn2 + 1$) and encrypted, it accepts the session key as authentic.

Thus, A and B think a secure connection with the new session key has been established and they have correctly authenticated the other entity.

2.3 Predicate/transition net formalism

In this paper, we have chosen to model the protocol with predicate/transition nets (Pr/T nets) [2] . In order to keep the model readable, we will use concepts of the problem domain where appropriate. These can be thought of as mere short hand notations that have a direct interpretation in Pr/T nets. (see Fig. 4)

- We will write E[Text;Key] for encrypted messages instead of <Text,Key> and Item for unary tuples instead of <Item>.
- A process often requires an item but does not consume it. For example, the intruder needs to know a key in order to decrypt a message but it does not forget the key after the encrypting. In Pr/T nets this means a transition consumes a token and produces a similar one in its place. In such cases, we will use the usual notation of a two-headed arrow.
- A place of a Pr/T net can represent a set if its tuples have an extra field, with value 1 or 0, indicating the current status of an element's membership in the set. Inserting an element that already belongs to the set should not change its contents. We will draw a dotted arrow for such insertion.
- In partial pictures of nets, we will sometimes draw only a single arrow with multiple labels instead of several ones leading from a transition to some places outside the picture. This is done when the items on the arrows are parts of a single clear-text message.

Predicate/transition nets were chosen because of the availability of effective tools for their analysis. It is, however, straightforward to use other high level Petri nets in their place. In particular, colored Petri nets [5] would suit the task

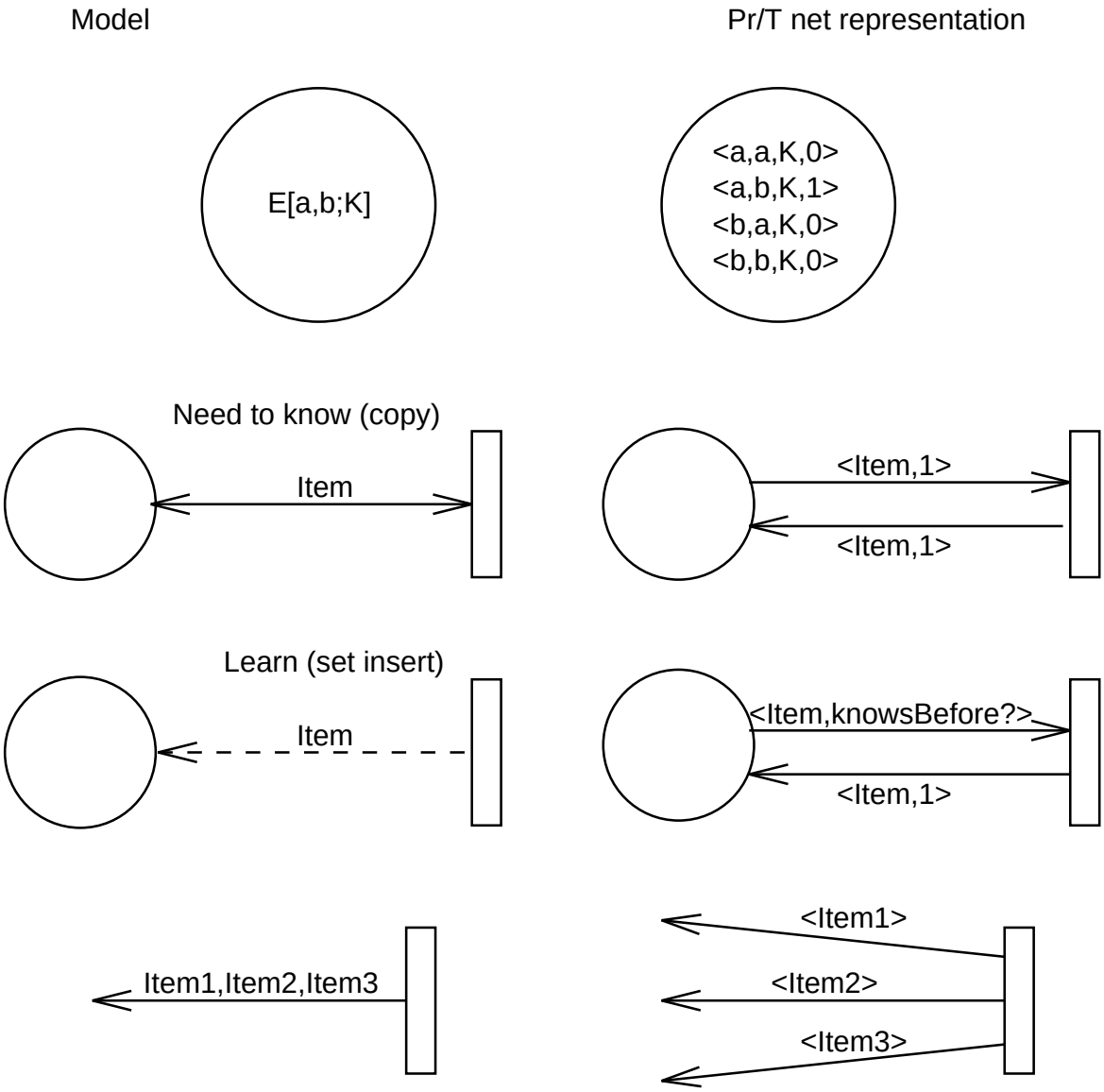

Figure 4: Model notations and Pr/T net representation

very well. Different net classes have their own specialties that can be utilized. We do not necessarily need the firing conditions of predicate/transition nets in our example, but there is no reason why they should not be used. In Section 7.6 we will discuss the possibility of defining a new net class or formalism that would be tailored for effective modelling and analysis of cryptographic protocols.

3 Model of the protocol entities

The model of the communicating entities is based on their functional description. From our point of view, the protocol completely determines the behavior of the entities. Any kind of behavior is possible as long as it conforms to the protocol specification. The implementation of the entities has to conform to the protocol specifications.

The communicating entities are sources and sinks of certain types of messages. Most entities both send and receive data. They can have their own inherent data sources and storages. Most entities can also react to received messages by releasing other messages. When receiving a message, the entity checks the integrity of the message against its inherent data and that saved from previously received messages.

The key and ticket servers are an integral part of the cryptographic system. The model must capture their possibly complex behavior accurately. The servers can usually be specified as reactive systems that receive certain types requests, check them against some inherent information of theirs and answer with messages containing the requested data.

3.1 Entities of the N-S protocol

In the Needham-Schröder protocol, the types of entities are *user* and *KDC*. A user can behave as user A initiating the communication or as user B. The user can simultaneously be A and B in several protocol runs.

Our model of the key distribution center is very simple. (Fig. 5) The KDC answers requests always in the same way. The only catch is that it has a supply session keys that are never reused.

In the N-S protocol, the two roles of a user do not interact significantly. Therefore, separate models could be built for users A and B. In general, this is not the case. In other protocols, a user in one role might reveal information that later becomes useful for the intruder when the same user is playing another

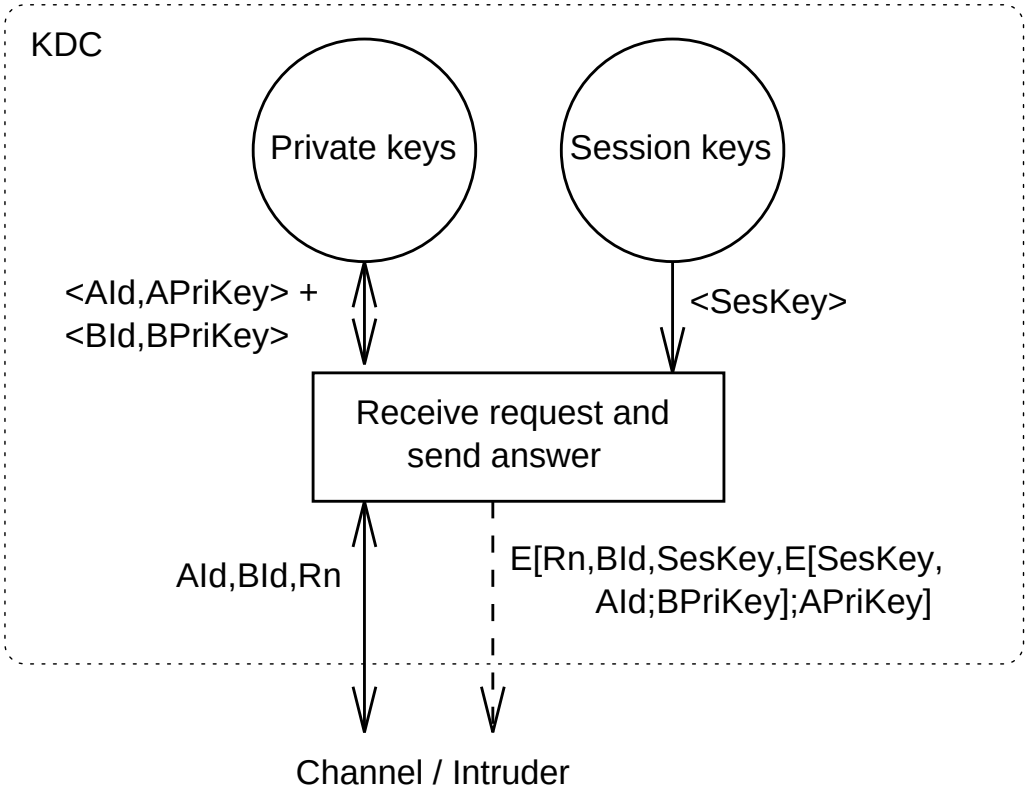

Figure 5: The key distribution center model

role. We have, therefore, decided to incorporate both A and B in the same user model.

The user model is an uncomplicated representation of the protocol specification. Fig. 6 is somewhat simplified. In practice, we want to have several users and a field for the user identifier must be added to the tuples in appropriate places. We also have to limit the number of users and who can initiate communication with whom (see Sec. 6).

4 Model of the channel and the intruder

In modelling the intruder, we have to be careful not to make any assumptions about how it manipulates the messages. The reliability of the security analysis depends crucially on the completeness of the intruder model. The model must be conceptually so simple that one can be reasonably assured it covers all possible behaviors of the intruder.

To avoid any restrictive assumptions, the communication channel is regarded as completely unreliable. The intruder has control over all messages in the channel. It can remove, alter, substitute and add messages. It is simplest to identify the intruder with the channel. We model the channel and the intruder

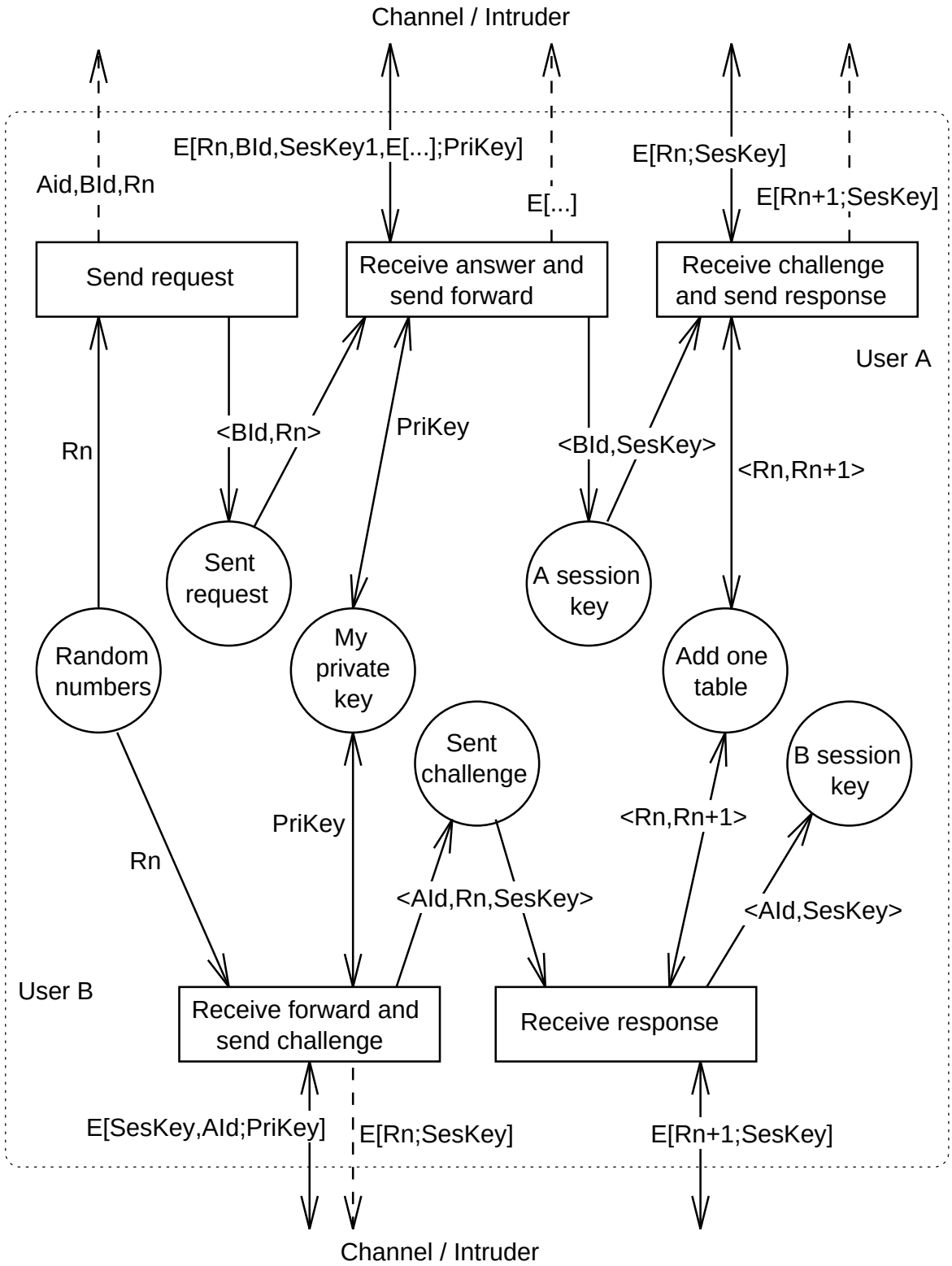

Figure 6: The user model

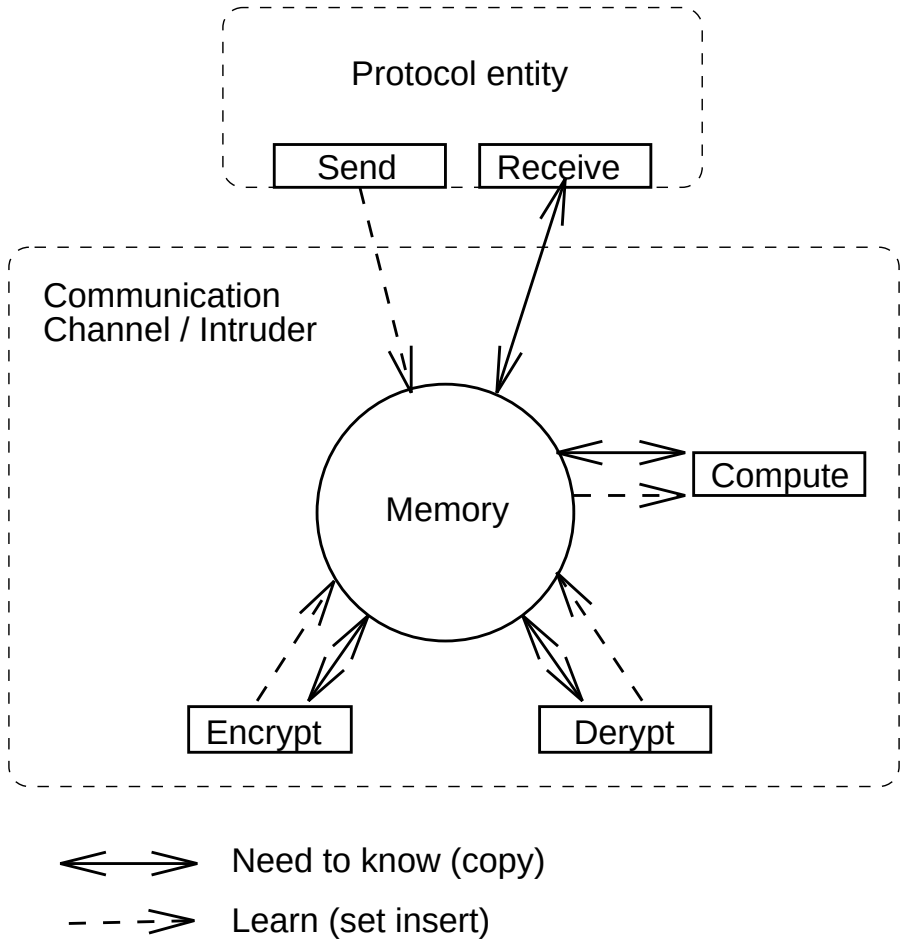

Figure 7: The intruder model concepts

as a single entity that delivers messages between protocol entities. It takes all the messages sent by them and it gives them messages to receive.

The intruder can be an outsider, a legitimate protocol entity or a group of legitimate entities. We always think of the last alternative since it is the worst. Being a conspiracy of several entities, the intruder has all their inherent knowledge: keys, passwords etc. Only trusted entities like key brokers cannot be involved in the conspiracy. The intruder can manipulate messages in negligible time. It has knowledge of the structure of all messages in the system and can encrypt and decrypt messages if only provided with the right key.

4.1 Learning

Fig. 7 shows a simplified picture of the intruder model concepts. We base our model of the intruder on the idea of learning. The intruder has a memory where it keeps all different data items. It learns new information when a protocol entity sends a message to the channel. It breaks plain sequential

messages in parts and puts every data item in the memory. (Fig. 8) It can learn the contents of an encrypted message, if it already has the message and a correct key in its memory. Reversely, the intruder can learn an encrypted message if it has the contents and the encryption key already in its memory. There can be further ways of computing new data items from the old.

In learning, a new data item is inserted in the memory only, if it is not there yet. In this sense, the memory is a set of data items. Once the intruder has learned something, it cannot forget it. Removing items from the memory could not help the intruder in its malicious tasks.

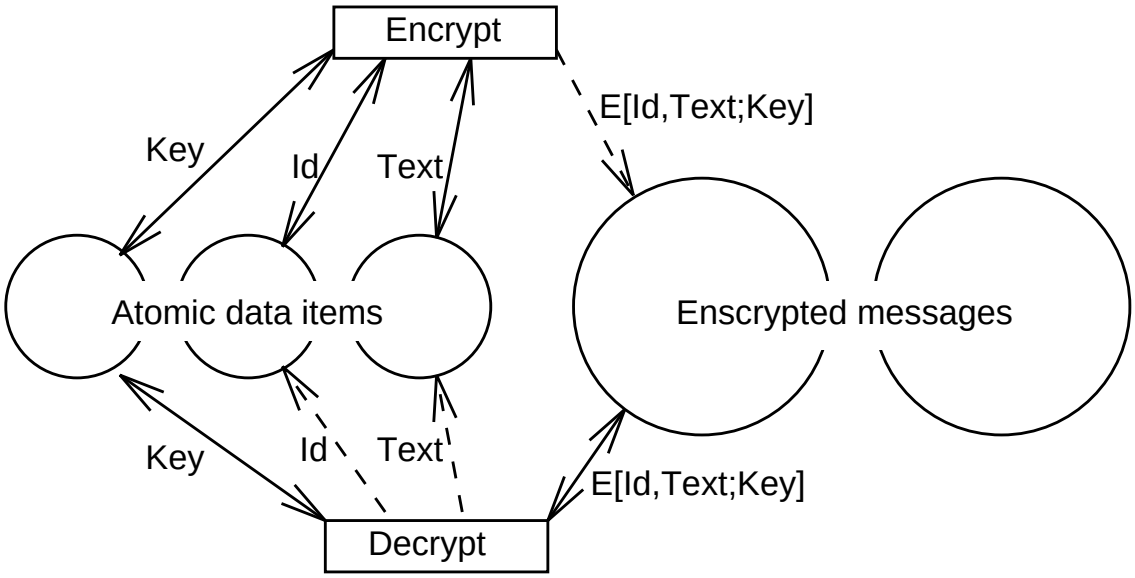

Figure 8: Encryption and decryption

4.2 Delivering messages to protocol entities

The intruder uses the items in the memory to compose sequential messages and gives them to protocol entities as if they came from a legitimate entity. Thus, the channel never carries a message directly to its destiny. The message is decomposed into the intruder memory and, when the same message is delivered further, it is re-composed from its parts. Since we will use full reachability analysis to guarantee that all possible combinations and orders of sending and receiving messages are considered, we never really have to decide which messages the intruder delivers and in which order. All possible combinations are considered automatically.

4.3 Intruder's inherent data

Like the communicating entities, the intruder has some inherent information before the communication process begins. This information includes user identifiers, public keys, time, false text, random keys and other commonly known or random data.

The intruder has recorded old messages and complete protocol runs. It may have been able acquire old session keys by means of cryptanalysis or otherwise and decrypt old messages. Therefore, we have to make the assumption that it has incorrect and previously used versions for all parts of the messages in the system.

Keeping in mind that the intruder can be a group of legitimate parties of the communication, we have to provide it with its own version of the inherent information of every type of entity in the system, such as private keys with a corresponding records in the key broker's files.

The inherent data is mostly system specific, so we can give only vague instructions here. It is easiest to first consider all different types of atomic data in the system and then, for each type, list what the intruder might have. It never hurts to give the the intruder a little more knowledge than it in reality has. That cannot lead to false acceptance of erroneous protocols in the security analysis. Moreover, it may reduce the risk of modelling errors resulting in such false acceptance.

4.4 Intruder's inherent data in the N-S protocol

In our sample protocol, the intruder knows initially all the user identifiers, and old versions of session keys and random numbers. The identifiers are publicly known and the old data items may have been obtained from previous protocol runs by recording, cryptanalysis or other methods.

The intruder knows or can generate some random numbers and random session keys, but the N-S protocol does not require logging of used keys or other critical data. In that case, there is no real difference between old and randomly generated data items. The model can be simplified by leaving out randomly generated data items from the intruder memory, for it can use the old ones in they place. The entities won't recognize any difference between old and random data items.

The intruder also has old versions of session key encrypted messages for which it has been able obtain keys. Of course, we need not provide the intruder with these messages as it can compose them itself, if we only give it the

corresponding basic data items, submessages and keys. We will, nevertheless, give it the messages to avoid re-composing them in the reachability analysis (see Sec. 7.1) Again, since the protocol entities do not remember old messages, we need not consider old session key encrypted messages that the intruder has not been able to decrypt. If the intruder has some use for them, it can use the ones with a known key instead.

The intruder does not know user's private keys and, thus, cannot compose private key encrypted messages from parts. We therefore have to give it old versions of private key encrypted messages from the KDC to every user. It is easiest to give all possible combinations of private key encrypted messages without thinking which of them the intruder could have in reality. As discussed above, it does not harm the security analysis to give the intruder too much information. If false reports of security holes result, the model can be revised.

As the intruder can be a conspiracy of several legitimate users of the system, we have to provide it with all the kinds data that different entity types have. The intruder must be given private keys of conspiring users with a corresponding entry in the KDC's files. Along with the private keys it knows all the messages that can be encrypted with them.

5 Security criteria

In order to verify the security of the protocol, we have to specify exactly what kind of behavior is unwanted in the system. We have to have a model with enough friendly entities for a full protocol run without any interference from the intruder. The security goals are stated in terms of this correct protocol run: how would the intruder want to affect it? The intruder can have two kinds of goals: it may want to learn confidential pieces of information and feed false information to an unsuspecting entity.

The intruder reaches the first goal when it learns a piece of information that is labeled or otherwise defined confidential. This might be "confidential text". Often some information becomes confidential when a protocol entity decides to use it for a certain purpose. In key distribution protocols session key is confidential only if it is accepted by someone as a secure key. In a correctly functioning protocol, there should not be any states where the intruder knows confidential data items. The second goal means that a protocol entity accepts in its internal storage a data item that should not be there. It can, for example, be "false text" or "old key".

Both of the above goals can be reduced to a situation where the same data item is in two places: inside a protocol entity and in the intruder's memory.

In the first case, confidential information from a protocol entity's data place is also known by the intruder. In the latter case, the intruder is able to lead its false data items to an entity's data places.

Our only aim is to find security failures in protocols and verify security properties. With the present model, it is not possible to verify protocols' general properties such as liveness and boundedness. For such properties to hold, we would have to limit the intruder's capability of controlling the communication channel. While security analysis of a system has a lot to do with credibility, it would not be wise to offer the intelligibility of the model or generality of the security analysis in order to model such other properties. Neither can we consider intruders that only want to hinder communication. In our model, the intruder can stop all communication by refusing to deliver any messages. We can verify that the data sent by one entity to another does not leak to the intruder and that, if the receiver accepts the data, it is authentic. We cannot guarantee that the data ever gets to the receiver.

Limiting the present discussion to security aspects is not in any way harmful, because other properties of the protocol can be analyzed separately. A more serious problem with our model is that it does not describe how the protocol recovers from an intruder attack. Security is guaranteed only if the entities halt after detecting an attack. It seems that our models could be extended to cover the recovery process. That would require an accurate specification of the behavior of communicating agents in error situations. Unfortunately, most cryptographic protocols do not specify ways of error recovery. In practical protocols the issue of recovery has to be addressed.

5.1 Security in the N-S protocol

In our example, the security criteria is fairly simple. The purpose of the protocol is to distribute session keys to two users without the intruder knowing them. Unacceptable states of the system are those where either user has a session key in a place of accepted keys and the intruder has the same key in its memory.

6 How many is enough?

So far, we have not discussed quantitative aspects of the model. Practical protocols rarely limit the number entities that may be involved in communication simultaneously. If there is a limit, it is often much too high for full reachability analysis to be feasible. We have to decide how many entities of

each type need to be included in the model for it to represent a system with an arbitrary number of entities. Does it make a difference if the key broker has two clients instead of one? Should the intruder have knowledge of more than one legitimate entity's inherent data? And so on.

This question is not easy to answer in general. The number of entities required for completeness of the security analysis depends on the protocol and the cryptographic techniques used in it. Consider, for instance, a threshold scheme where any five agents can decrypt a message but no four can. If the intruder has inherent knowledge of three agent's keys, it may be able to decrypt the message in a model with two other agents but not in a single agent model.

While the answer to the question "How many is enough?" depends heavily on the protocol, we can restrict the number of parameters that need to be given a value. First, we restate some observations from earlier sections (Sec. 3.1 and 4.4).

Observation 1 *If two roles of an entity type do not interact, the entities can be limited to playing only one of the roles. Also, different entities could be used for each of the roles.*

On the other hand, if an entity has two roles that share secret data or state information, it is not safe to limit its behavior to only one role. The entity might in one role accept false information or reveal confidential information that the intruder can later use against it when it is paying the other role. We noted (in Sec. 3.1) that in the N-S protocol the two roles of the user don't interact. In reachability analysis, we will limit users to act as either user A or user B.

Observation 2 *If the intruder has recorded old messages of a certain type and has been able to obtain a key for them, it does not need similar messages without the proper key, as it can always use the ones with a key in their place.*

The protocol entities cannot know if the intruder has been able to cryptanalyze the recorded message or not. They treat all old messages in the same way.

Observation 3 *If the protocol does not require entities of the system to remember old messages or used basic data items (session keys, random numbers etc.) and the intruder has been able to record old messages and obtain keys for them, it does not need false versions of the data, as it can use old ones instead.*

If the protocol entities do not remember past messages, they cannot see any difference between old messages and ones composed by the intruder. Usually this is the case, because it would require too much memory to remember old

messages or data items and too much time to check newly received messages against the memory.

Next, we consider the number of entities necessary for detecting security failures. As noted in Sec. 5, at least one honest, uncompromised entity of each entity type of a correct protocol run is needed in the model. Otherwise, there would be a conspiring entity involved in every communication process and secure communication would be impossible. This is the minimal number of honest entities that are needed for detecting security failures.

If security failures are detected, they are detected during some protocol run. The security goals can be stated in terms of entities of a single protocol run and the intruder memory. (Remember that we defined protocol run so that the state of the system is essentially the same before and after a protocol run.) Furthermore, if the intruder knows the inherent data of an entity, it can act exactly like the entity. It can do this by responding to all messages exactly like that entity would and generating all the same messages. Therefore, if several protocol runs between varying entities are needed to result in a security failure, the intruder can play the parts of all other entities, except the ones in the run with the detected security failure. In conclusion, the minimal number of uncompromised entities needed in a correct protocol run is sufficient for detecting any security failures in a similar run. This leads to the following observation.

Observation 4 *If we give the intruder initial knowledge of the inherent data of sufficiently many entities, it is not necessary to include in the model more than the maximum number of uncompromised entities that can take part in a full protocol run. (How much is sufficiently many, has to be determined for the protocol in question.)*

This means that the intruder can be a conspiracy of arbitrary number of legitimate agents. We have to initially insert in the intruder's memory all the inherent data of several entities of every type in the system. On the other hand, we need to have in the model only a minimal number of entities for the communication process to complete.

Still, we must decide how many conspiring agents data we initialize the intruder's memory with. This question remains for the analyst to answer separately for every protocol. It should be thoroughly answered every time. Otherwise, the security of the system cannot be proved. In many cases, upper bounds for the numbers may be easily found but these upper bounds are too high for practical analysis.

6.1 Number on entities in the N-S protocol model

In the Needham-Schröder protocol, it is fairly easy to decide the number of inherent data items in the intruder's memory. If the intruder uses two different basic data items of the same kind somewhere, there is no reason why it could not use one of them for both purposes. Nowhere in the model is there a transition that requires two data items to be different. This leads to one more observation.

Observation 5 *If the protocol does not require comparison of two data items anywhere, it suffices to initialize the intruders memory with only one data item of a kind as it can use the same one for all purposes.*

We have to be careful about what we mean with the word "kind" here. Basic data items are of the same kind, if they are of the same type (key, number etc.) and they have similar corresponding data items in protocol data storages inside protocol entities. For example, we mean that two keys are of different kind, if one is known by the intruder only and the other is the session key used by a protocol entity. In encrypted messages all the parts must be of the same kind. It may be easier to provide the intruder with all thinkable combinations of encrypted messages instead of trying to reason which really are different.

7 Reachability analysis

We have used the reachability analysis tool PROD [3, 4] to find errors in the Needham–Schröder protocol. The well known repetition attack, where the intruder sends an old forward message to user B, was rediscovered. The complete model is in Appendix A and the analysis results in Appendix B.

Experiments showed that full reachability analysis of complicated cryptographic algorithms is not a straightforward matter. Problems arise with the size of the reachability graph. Interleaving of the actions of several entities and independent learning events of the intruder result in a large state space, even though the protocol is relatively simple. In the following we will discuss some methods that were used to fight state space explosion. We are encouraged by the fact that cryptographic protocols usually allow only very restricted behavior of the entities. Therefore, there may be only a small number of conceptually different processes and states to consider. Firstly, there are some things that must be taken in consideration when building the model.

7.1 Optimizing the model

All messages that the intruder can compose from its inherent data should be given to it initially. This may not affect very much the overall size of the state space, but it saves some time at the beginning of the reachability graph generation. Giving the messages ready-made saves time when debugging the model and when the protocol under development still has errors that are quickly found.

In the previous sections we have emphasized that it is better to give the intruder too much inherent knowledge that too little. From the reachability analysis point of view, it is better to use as few different basic data items as possible. It is probably the most demanding task of the modeller to find a set of data items that sufficiently represents an arbitrary system and that still is small enough to make reachability analysis feasible.

7.2 On-the-fly verification

On-the-fly verification looks for erroneous states of the system during reachability graph generation and stops when the first such state is encountered. If the protocol has security flaws, on-the-fly method often finds them reasonably fast.

7.3 Optimizing reachability graph generation

There are also ways of reducing the generated state space during the reachability analysis. How well these methods work depends entirely on the structure of the model. Luckily, some are especially well suited for protocols with several concurrently acting agents. The main points of improvements lie in the intruder learning process and in the concurrency of the protocol.

7.4 Prioritizing learning transitions

As anyone examining a reachability graph for our model will soon notice, the intruder learning process is not very effectively described by Petri nets. The problem is that the reachability analysis considers separately all different orders in which the intruder can encrypt and decrypt messages. For example, if the intruder learns a new session key, it may be able to compose tens or hundreds of new messages from the data it already has. If we go through all different orders of these messages, the number of intermediate states will be immense. The end result of all different learning processes is, on the other

hand, the same set of messages. We can choose an single learning process to represent all of them. Also, all internal learning (encrypting, decrypting, etc.) can be done immediately after the intruder has obtained new data items.

What we did to choose only one of the possible learning processes, was prioritizing the transitions. Giving the intruder's internal learning transitions a higher priority than any other transitions in the system forces all learning to take place immediately. The different transitions inside the intruder also have to have different priorities. In a high level net even this is not enough. Different instances of transitions in the intruder model should be given different priorities so that several enabled instances of the same transition can occur only in one order. It is worth noting that transitions in conflict with higher priority ones are ignored. This does not cause problems because only the final result, after all possible learning is done, matters.

In experiments, prioritizing worked well for very small problems. When the number of different data items in the system increases, a new data item in the intruder memory may induce a tens or hundreds of states long learning process. Since we save both the set of messages known by the intruder (tuples ending in 1) and the ones not known (tuples ending with 0), the space needed for saving a single state became fairly large. (8-130kB in our experiments with the N-S protocol). Generating and saving the intermediate states of the learning processes becomes the largest part of the work. Typically they comprise at least 90% of the state space. This means high memory or disk space consumption and slow reachability graph generation. The obvious improvement would be to leave out all intermediate states of learning and only store the final result set. That in not easily done with Pr/T nets and the tools at hand.

7.5 The stubborn set method

The stubborn set method [10] proved efficient in reducing the state space. Interleaving of the concurrent actions of different entities greatly contributes to the state space explosion. Also, most of the different learning processes (see Sec. 7.4) can be arbitrarily interleaved. Changing the order in which the intruder exchanges messages with the protocol entities or learns new data often does not change the resulting state. The stubborn set method effectively utilizes this kind of concurrency in the system. The method clearly is not as effective in reducing the number of intermediate states in learning processes as the priority method proposed in the previous section.

The on-the-fly version of CFFD equivalence preserving stubborn set method [11, 12] implemented in PROD takes good advantage of concurrency of the model while preserving behavioral equivalence with respect to test transitions,

whose enabledness indicates an insecure state of the system. This was the most effective analysis method tested. It found the security flaw fast, but the path to the erroneous state was quite long. The memory requirements of saving a single state are still a problem.

In experiments, the stubborn set method found the flaw of the Needham-Schröder protocol in the basic model after 79 states and in a model with more users after 119 states. Without the stubborn set method, the former took 623 states and the latter did not complete. Without stubborn sets, breadth first generation of large state spaces is not feasible and results of depth first on-the-fly generation depend heavily on details of the model and the order of transitions in PROD input files. It is expected that priority method combined with stubborn set would reduce the size of the state space significantly, but for small problems it can slow down detecting security flaws by ignoring the shortest routes to erroneous states. Unfortunately, PROD does not allow combining priorities with on-the-fly verification or stubborn sets; no experiments with combined methods were made.

7.6 Developing a new net class for cryptographic protocols

If extensive analysis of cryptographic protocols is needed, it might be a good idea to adapt the formalism and analysis tools for them. This would make both modeling and analysis more efficient.

The short hand notations used in this paper for Pr/T net constructs make modelling easier. A new net class could be defined with set type places (instead of the usual multi-sets) like the ones we have used. In the analysis tool, the sets could be represented with a data structure especially fitted for sets. The present representation of the set and its complement is unnecessarily heavy. Set type places would partially solve the problem of states requiring too large storage space. Negative arcs, or inhibitors, that disable transition if the corresponding token exists in a place, could be used for the same purpose as the set places.

Furthermore, in a net with negative arcs it would not be necessary to store in the intruder memory any messages which the intruder can directly compose from parts. In a normal predicate/transition nets it is impossible to inhibit storing of messages when they can be composed from parts. With negative arcs, the intruder could store received messages only when it is unable to decrypt them and no similar message is already in the memory. There would be three ways to receive a message: decrypt it immediately and only store the parts, store the entire message when unable to decrypt, or not store the message when it already is in the memory. Similarly, there would be two ways

of sending a messages: compose it from parts or send an entire message that has been stored in the memory.

Set type places could be used for a partially similar effect: only store every received message once and not store any messages composed by the intruder itself. In normal Pr/T net, we have to store the complement sets, i.e. the items which are not in the memory, in any case. Therefore, it would not help any to rearrange receiving and sending of messages in our present model.

It is worth noting that nets with prioritized transitions, as suggested in Sec. 7.4, actually form a net class with far greater expressive power than the usual Pr/T nets. We only used priorities so that they did not enhance the expressiveness of the formalism but merely sped up the analysis. It is possible to use the priorities, and also negative arcs, in such a way that the net class has the power of a universal computer. [8] There is no obvious reason to avoid such use, but it is likely that any trickery in the model makes efficient analysis more difficult.

In Sec. 7.4, we came up with the idea of leaving out all intermediate places of the intruder learning processes. In a new formalism, the intruder's memory could be represented by a knowledge module that automatically does all encrypting, decrypting and other internal learning immediately when new data items are inserted. More effective algorithms and data structures than nets could be used to implement the memory's contents and the learning process. In reachability analysis, every state would still include the state of the memory. It is not clear if such a new formalism would be a Petri net any more. One possibility would be to encode the contents of the memory to a fixed number of tokens in a single place, but this task seems difficult and computationally slow for large sets of memory items.

As long as only single problems are analyzed, it is better to use existing formalisms and tools with well-implemented efficient algorithms. If the same problem needs to be solved more often or is important enough, it may be worth the work to adapt tools to the problem. The most promising improvement of the formalism and tools would be the introduction of negative arcs to the Pr/T net. To be effective in reducing the state space, any such change of formalism should still accommodate the use of the stubborn set method.

8 Conclusion

In this paper, we modelled and analyzed security of the Needham-Schröder key distribution protocol with predicate/transition nets, and along that, developed a methodology for modelling cryptographic protocols with high level Petri

nets. Our main goal has been clarity of the model and its feasibility for automated analysis.

The model of the protocol entities was constructed directly from the specification of the protocol. The intruder and the communication channels were modelled as one unreliabe entity that has complete control over all messages in the system. The intruder model was based on the concepts of memory and learning. Special care was taken that the model captures all possible actions of the intruder.

The quantities of entities and data items in the model were found to be crucial for the completeness of the security analysis. Several rules of thumbs were given for finding the minimal number of model parts that represents a system with an arbitrary number of entities and concurrent protocol runs. The issue of quantities has to be thoroughly discussed for every every protocol model separately, as was done here for the N-S protocol.

Several techniques of fighting state space explosion in reachability analysis were discussed. The stubborn set method was found to be effective in reducing the number of generated states. The large disk space required for storing complement sets was found to be a central problem in the analysis. Introduction of set type places or, preferably, negative arcs in the net formalism was proposed as a solution. In experiments with the predicate/transition net reachability analysis tool PROD, the model proved efficient in detecting protocol failures. The PROD tool has efficient implementation of the on-the-fly and stubborn set methods for reachability analysis. The tool is, however limited to the class of Pr/T nets, and we were not able to test the proposed enhancements to the formalism. Further development the formalism and tools is clearly necessary for the presented methods to be useful in full verification of the security of real size protocols.

References

[1] Dorothy Denning. *Cryptography and data security.* Addison-Wesley, Reading MA, 1982.

[2] Hartman J. Genrich. Predicate/transition nets. In *Advances in Petri Nets 1986*, pages 207–247. LNCS-254, Spring-Verlag, 1986.

[3] Peter Grönberg, Mikko Tiusanen, and Kimmo Varpaaniemi. Prod—a Pr/T-net reachability analysis tool. Technical report, Digital Systems Laboratory, Helsinki University of Technology, June 1993. <ftp: saturn.hut.fi>.

[4] Jaakko Halme, Kari Hiekkanen, Tino Pyssysalo, and Kimmo Varpaaniemi. *PROD Reference Manual.* Digital Systems Laboratory, Helsinki University of Technology, October 1994. <ftp: saturn.hut.fi>.

[5] K. Jensen. Coloured petri nets. In *Advances in Petri Nets 1986*, pages 248–299. LNCS-254, Spring-Verlag, 1986.

[6] Nieh and Tavares. Modelling and analyzing cryptographic protocols using petri nets. In *Advances in Cryptology–AUSCRYPT '92, International Conference on Cryptology*, pages 275–295. LNCS, Spring-Verlag, 1992.

[7] Benjamin B. Nieh. Modelling and analysis of cryptographic protocols using petri nets. Master's thesis, Queen's university, Kingston, Ontario, Canada, 1992.

[8] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.

[9] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., 1994.

[10] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, pages 490–515. LNCS-483, Spring-Verlag, 1991.

[11] Antti Valmari. Alleviating state space explosion during verification of behavioral equivalence. Technical report, University of Helsinki, Department of Computer Science, 1992.

[12] Antti Valmari. On-the-fly verification with stubborn sets. In *Computer Aided Verification*, pages 397–408. LNCS-697, Spring-Verlag, 1993.

[13] Kimmo Varpaaniemi. On computing symmetries and stubborn sets. Technical report, Digital Systems Laboratory, Helsinki University of Technology, April 1994. <ftp: saturn.hut.fi>.

A Needham-Schröder protocol model for PROD

```
/*
  Needham---Schröder authentication protocol
*/

/*
  Data types

  Basic data types:
      User identifier (Id)
      Random number   (Rn)
      Key             (Key)
  Encrypted message types:
      E[Rn,Id,Key,E[Key,Id;Key];Key] = <Rn,Id,Key,Key,Id,Key,Key>  (EX)
      E[Key,Id;Key]                  = <Key,Id,Key>                (EY)
      E[Rn;Key]                      = <Rn,Key>                    (EZ)
      E[RnPlus1;Key]                 = <RnPlus1,Key>               (EW)
*/

#enum Jules,Jim,Kathe
#define IdMax         Jules
#define IdMin         Kathe
#define AllIds        IdMin..IdMax
#define IdCount       IdMax-IdMin
/* Only Jules can be A, he can only choose Jim for B and
   he can start the process only once. */
#define PossibleABInit <.Jules,Jim.>
/* Only Jim can act as B. */
#define PossibleBInit <.Jim.>


#enum arn1, \
      brn1, \
      irn1
#define ARnMax        arn1
#define ARnMin        arn1
#define AllARns       ARnMin..ARnMax
#define BRnMax        brn1
#define BRnMin        brn1
#define AllBRns       BRnMin..BRnMax
#define RnMax         arn1
#define RnMin         irn1
#define AllRns        RnMin..RnMax
#define IRns          irn1..irn1
#define NonIRns       brn1..arn1
#define RnCount       RnMax-RnMin+1

#enum arn1plus1, \
      brn1plus1, \
      irn1plus1
#define RnPlus1Max    arn1plus1
#define RnPlus1Min    irn1plus1
#define IRnPlus1s     irn1plus1..irn1plus1
#define NonIRnPlus1s  brn1plus1..arn1plus1
#define AllRnPlus1s   RnPlus1Min..RnPlus1Max

#define AddOneTableInit <.arn1,arn1plus1.>+<.brn1,brn1plus1.>+ \
                        <.irn1,irn1plus1.>
```

```
#enum prikey1,prikey2, \
      prikey3
#define IPriKeyMax    prikey3
#define IPriKeyMin    prikey3
#define IPriKeys      IPriKeyMin..IPriKeyMax
#define IPriKeyCount  IPriKeyMax-IPriKeyMin+1
#define PriKeyMax     prikey1
#define PriKeyMin     prikey3
#define AllPriKeys    PriKeyMin..PriKeyMax
#define PriKeyCount   PriKeyMax-PriKeyMin+1

#enum seskey1, \
      iseskey1
#define SesKeyMax     seskey1
#define SesKeyMin     iseskey1
#define NonISesKeys   seskey1..seskey1
#define ISesKeys      iseskey1..iseskey1
#define AllSesKeys    SesKeyMin..SesKeyMax
#define SesKeyCount   SesKeyMax-SesKeyMin+1

/* The following initializer can conveniently be user for both
   the user and KDC models. */

#define PrivateKeysInit <.Jules,prikey1.>+<.Jim,prikey2.>+<.Kathe,prikey3.>

/* Everything without new random numbers or session keys,
   total 3*3*2*2*3*3*3=972,
   presently gives the intruder a bit too much knowledge */
#define memoryEXInit \
  <.NonIRns,AllIds,AllSesKeys,AllSesKeys,AllIds,AllPriKeys,AllPriKeys,0.> + \
  <.IRns,AllIds,NonISesKeys,AllSesKeys,AllIds,AllPriKeys,AllPriKeys,0.> + \
  <.IRns,AllIds,ISesKeys,NonISesKeys,AllIds,AllPriKeys,AllPriKeys,0.> + \
  <.IRns,AllIds,ISesKeys,ISesKeys,AllIds,AllPriKeys,AllPriKeys,1.>

/* Everything without new session keys, total 2*3*4=18,
   gives the intruder a bit too much knowledge */
#define memoryEYInit \
  <.NonISesKeys,AllIds,AllPriKeys,0.> + \
  <.ISesKeys,AllIds,AllPriKeys,1.>

/* Everything the intruder could compose itself */
#define memoryEZInit <.IRns,ISesKeys,1.> + \
                     <.NonIRns,AllSesKeys,0.> + \
                     <.IRns,NonISesKeys,0.>

#define memoryEWInit <.IRnPlus1s,ISesKeys,1.> + \
                     <.NonIRnPlus1s,AllSesKeys,0.> + \
                     <.IRnPlus1s,NonISesKeys,0.>


/*
  General computation tables
*/

/* Unique new random numbers,
   a separate set for each origin to reduce state space,
   intruder's share is initially in its memory */
#place ARandomNumbers  lo(<.ARnMin.>) hi(<.ARnMax.>) mk(<.AllARns.>)
#place BRandomNumbers  lo(<.BRnMin.>) hi(<.BRnMax.>) mk(<.AllBRns.>)
/* Adding one to a random number */
#place AddOneTable lo(<.RnMin,RnPlus1Min.>) hi(<.RnMax,RnPlus1Max.>) \
```

```
                mk(AddOneTableInit)


/*
  Intruder memory places
*/

/* Basic data types */
/* Intruder knows all user identifiers, cannot learn more */
#place MemoryId      lo(<.IdMin,1.>) hi(<.IdMax,1.>) mk(<.IdMin..IdMax,1.>)
#place MemoryRn      lo(<.RnMin,0.>) hi(<.RnMax,1.>) \
                     mk(<.IRns,1.>+<.NonIRns,0.>)
#place MemoryRnPlus1 lo(<.RnPlus1Min,0.>) hi(<.RnPlus1Max,1.>) \
                     mk(<.IRnPlus1s,1.>+<.NonIRnPlus1s,0.>)
/* Intruder cannot get hold of any more pivate keys in the process. */
#place MemoryPriKey  lo(<.IPriKeyMin,1.>) hi(<.IPriKeyMax,1.>) \
                     mk(<.IPriKeys,1.>)
#place MemorySesKey  lo(<.SesKeyMin,0.>) hi(<.SesKeyMax,1.>) \
                     mk(<.NonISesKeys,0.>+<.ISesKeys,1.>)
/* Encrypted messages */
#place MemoryEX lo(<.RnMin,IdMin,SesKeyMin,SesKeyMin,IdMin,PriKeyMin, \
                     PriKeyMin,0.>) \
                hi(<.RnMax,IdMax,SesKeyMax,SesKeyMax,IdMax,PriKeyMax, \
                     PriKeyMax,1.>) \
                mk(memoryEXInit)
#place MemoryEY      lo(<.SesKeyMin,IdMin,PriKeyMin,0.>) \
                     hi(<.SesKeyMax,IdMax,PriKeyMax,1.>) mk(memoryEYInit)
#place MemoryEZ      lo(<.RnMin,SesKeyMin,0.>) \
                     hi(<.RnMax,SesKeyMax,1.>) mk(memoryEZInit)
#place MemoryEW      lo(<.RnPlus1Min,SesKeyMin,0.>) \
                     hi(<.RnPlus1Max,SesKeyMax,1.>) mk(memoryEWInit)

/*
  User data places
*/

#place PossibleAB        lo(<.IdMin,IdMin.>) hi(<.IdMax,IdMax.>) \
                         mk(PossibleABInit)
#place PossibleB         lo(<.IdMin.>) hi(<.IdMax.>) mk(PossibleBInit)
#place SentRequestRn     lo(<.IdMin,IdMin,RnMin.>) hi(<.IdMax,IdMax,RnMax.>)
#place MyPrivateKey      lo(<.IdMin,PriKeyMin.>) hi(<.IdMax,PriKeyMax.>) \
                         mk(PrivateKeysInit)
#place ASessionKey       lo(<.IdMin,IdMin,SesKeyMin.>) \
                         hi(<.IdMax,IdMax,SesKeyMax.>)
#place SentChallengeRn   lo(<.IdMin,IdMin,RnMin,SesKeyMin.>) \
                         hi(<.IdMax,IdMax,RnMax,SesKeyMax.>)
#place BSessionKey       lo(<.IdMin,IdMin,SesKeyMin.>) \
                         hi(<.IdMax,IdMax,SesKeyMax.>)


/*
  KDC data places
*/

#place PrivateKeys   lo(<.IdMin,PriKeyMin.>) hi(<.IdMax,PriKeyMax.>) \
                     mk(PrivateKeysInit)
/* Unique new session keys */
#place SessionKeys   lo(<.SesKeyMin.>) hi(<.SesKeyMax.>) mk(<.NonISesKeys.>)


/*
```

```
  Tester
*/

#place Tester lo(<.0.>) hi(<.1.>) mk(<.0.>)
#tester Tester reject(<.1.>)


/*
  Intruder learning processes
*/

/* Decrypting */

#trans decryptEX
  in  { MemoryPriKey: <.PriKey2,1.>;  /* Decryption key */
        MemoryEX: <.Rn,Id1,SesKey1,SesKey2,Id2,PriKey1,PriKey2,1.>;
        MemoryRn: <.Rn,knowsBeforeRn.>;
        MemorySesKey: <.SesKey1,knowsBeforeKey.>;
        MemoryEY: <.SesKey2,Id2,PriKey1,knowsBeforeEX.>; }
  out { MemoryEX: <.Rn,Id1,SesKey1,SesKey2,Id2,PriKey1,PriKey2,1.>;
        MemoryPriKey: <.PriKey2,1.>;
        MemoryRn: <.Rn,1.>;
        MemorySesKey: <.SesKey1,1.>;
        MemoryEY: <.SesKey2,Id2,PriKey1,1.>; }
  comp () { if (knowsBeforeRn == 0 || knowsBeforeKey == 0 ||
                knowsBeforeEX == 0) Accept(); }
#endtr

#trans decryptEY
  in  { MemoryPriKey: <.PriKey,1.>;  /* Decryption key */
        MemoryEY: <.SesKey,Id,PriKey,1.>;
        MemorySesKey: <.SesKey,0.>; }
  out { MemoryPriKey: <.PriKey,1.>;
        MemoryEY: <.SesKey,Id,PriKey,1.>;
        MemorySesKey: <.SesKey,1.>; }
#endtr

#trans decryptEZ
  in  { MemorySesKey: <.SesKey,1.>;  /* Decryption key */
        MemoryEZ: <.Rn,SesKey,1.>;
        MemoryRn: <.Rn,0.>; }
  out { MemoryEZ: <.Rn,SesKey,1.>;
        MemorySesKey: <.SesKey,1.>;
        MemoryRn: <.Rn,1.>; }
#endtr

#trans decryptEW
  in  { MemorySesKey: <.SesKey,1.>;  /* Decryption key */
        MemoryEW: <.RnPlus1,SesKey,1.>;
        MemoryRnPlus1: <.RnPlus1,0.>; }
  out { MemoryEW: <.RnPlus1,SesKey,1.>;
        MemorySesKey: <.SesKey,1.>;
        MemoryRnPlus1: <.RnPlus1,1.>; }
#endtr

/* Encrypting */

#trans encryptEX
  in  { MemoryPriKey: <.PriKey2,1.>;  /* Encryption key */
        MemoryEX: <.Rn,Id1,SesKey1,SesKey2,Id2,PriKey1,PriKey2,0.>;
        MemoryRn: <.Rn,1.>;
```

```
        MemorySesKey: <.SesKey1,1.>;
        MemoryEY: <.SesKey2,Id2,PriKey1,1.>; }
  out { MemoryEX: <.Rn,Id1,SesKey1,SesKey2,Id2,PriKey1,PriKey2,1.>;
        MemoryPriKey: <.PriKey2,1.>;
        MemoryRn: <.Rn,1.>;
        MemorySesKey: <.SesKey1,1.>;
        MemoryEY: <.SesKey2,Id2,PriKey1,1.>; }
#endtr

#trans encryptEY
  in  { MemoryPriKey: <.PriKey,1.>;  /* Encryption key */
        MemoryEY: <.SesKey,Id,PriKey,0.>;
        MemorySesKey: <.SesKey,1.>; }
  out { MemoryEY: <.SesKey,Id,PriKey,1.>;
        MemoryPriKey: <.PriKey,1.>;
        MemorySesKey: <.SesKey,1.>; }
#endtr

#trans encryptEZ
  in  { MemoryEZ: <.Rn,SesKey,0.>;
        MemorySesKey: <.SesKey,1.>;  /* Encryption key */
        MemoryRn: <.Rn,1.>; }
  out { MemoryEZ: <.Rn,SesKey,1.>;
        MemorySesKey: <.SesKey,1.>;
        MemoryRn: <.Rn,1.>; }
#endtr

#trans encryptEW
  in  { MemoryEW: <.RnPlus1,SesKey,0.>;
        MemorySesKey: <.SesKey,1.>;  /* Encryption key */
        MemoryRnPlus1: <.RnPlus1,1.>; }
  out { MemoryEW: <.RnPlus1,SesKey,1.>;
        MemorySesKey: <.SesKey,1.>;
        MemoryRnPlus1: <.RnPlus1,1.>; }
#endtr

/* Adding one to a random number and subtracting one
   (the latter probably unnecessary) */

#trans addOne
  in  { MemoryRn: <.Rn,1.>;
        MemoryRnPlus1: <.RnPlus1,0.>;
        AddOneTable: <.Rn,RnPlus1.>; }
  out { MemoryRn: <.Rn,1.>;
        MemoryRnPlus1: <.RnPlus1,1.>;
        AddOneTable: <.Rn,RnPlus1.>; }
#endtr

#trans subtractOne
  in  { MemoryRnPlus1: <.RnPlus1,1.>;
        MemoryRn: <.Rn,0.>;
        AddOneTable: <.Rn,RnPlus1.>; }
  out { MemoryRnPlus1: <.RnPlus1,1.>;
        MemoryRn: <.Rn,1.>;
        AddOneTable: <.Rn,RnPlus1.>; }
#endtr


/*
  User message exchange
*/
```

```
#trans sendRequest                          /* User A */
  in  { PossibleAB: <.MyId,BId.>;     /* Don't put back for reuse! */
        ARandomNumbers: <.Rn.>;  /* Don't put back for reuse! */
        MemoryRn: <.Rn,knowsBeforeRn.>; }
  out { SentRequestRn: <.MyId,BId,Rn.>;
       /* Request <MyId,AId,BId,Rn> from user A to KDC. */
MemoryRn: <.Rn,1.>; }
#endtr

#trans recAnswerAndSendForward              /* User A */
  in  { SentRequestRn: <.MyId,BId,Rn.>;
        MyPrivateKey: <.MyId,PriKey.>;
       /* Answer EX from KDC to user A. */
        MemoryEX: <.Rn,BId,SesKey,Key1,Id1,Key2,PriKey,1.>;
        MemoryEY: <.Key1,Id1,Key2,knowsBeforeEY.>; }
  out { MyPrivateKey: <.MyId,PriKey.>;
        ASessionKey: <.MyId,BId,SesKey.>;
        MemoryEX: <.Rn,BId,SesKey,Key1,Id1,Key2,PriKey,1.>;
       /* Forward EY from user A to user B. */
        MemoryEY: <.Key1,Id1,Key2,1.>; }
#endtr

#trans recForwardAndSendChallenge           /* User B */
  in  { MyPrivateKey: <.MyId,PriKey.>;
        PossibleB: <.MyId.>;
        BRandomNumbers: <.Rn.>;  /* Don't put back for reuse! */
       /* Forward EY from user A to user B. */
        MemoryEY: <.SesKey,AId,PriKey,1.>;
        MemoryEZ: <.Rn,SesKey,knowsBeforeEZ.>; }
  out { SentChallengeRn: <.MyId,AId,Rn,SesKey.>;
        PossibleB: <.MyId.>;
        MemoryEY: <.SesKey,AId,PriKey,1.>;
       /* Challenge EZ from user B to user A */
        MemoryEZ: <.Rn,SesKey,1.>; }
#endtr

#trans recChallengeAndSendResponse          /* User A */
  in  { ASessionKey: <.MyId,BId,SesKey.>;
        AddOneTable: <.Rn,RnPlus1.>;
       /* Challenge EZ from user B to user A */
        MemoryEZ: <.Rn,SesKey,1.>;
        MemoryEW: <.RnPlus1,SesKey,knowsBeforeEZ.>; }
  out { ASessionKey: <.MyId,BId,SesKey.>;
        AddOneTable: <.Rn,RnPlus1.>;
        MemoryEZ: <.Rn,SesKey,1.>;
       /* Response EW from user A to user B. */
        MemoryEW: <.RnPlus1,SesKey,1.>; }
#endtr

#trans recResponse                          /* User B */
  in  { SentChallengeRn: <.MyId,AId,Rn,SesKey.>;
        AddOneTable: <.Rn,RnPlus1.>;
       /* Response EZ from user A to user B. */
        MemoryEW: <.RnPlus1,SesKey,1.>; }
  out { AddOneTable: <.Rn,RnPlus1.>;
        BSessionKey: <.MyId,AId,SesKey.>;
        MemoryEW: <.RnPlus1,SesKey,1.>; }
#endtr
```

```
/*
  KDC message exchange
*/

#trans recRequestAndSendAnswer
  in  { PrivateKeys: <.AId,APriKey.> + <.BId,BPriKey.>;
        SessionKeys: <.SesKey.>;  /* Don't put back for reuse! */
        MemoryRn: <.Rn,1.>;
        MemoryEX: <.Rn,BId,SesKey,SesKey,AId,BPriKey,APriKey,
                   knowsBeforeEX.>; }
       /* Request <AId,BId,Rn> from user A to KDC. */
  out { PrivateKeys: <.AId,APriKey.> + <.BId,BPriKey.>;
        MemoryRn: <.Rn,1.>;
       /* Answer EX from KDC to user A. */
        MemoryEX: <.Rn,BId,SesKey,SesKey,AId,BPriKey,APriKey,1.>; }
#endtr


/*
  Tester
*/

#trans foundProblemA
  in  { MemorySesKey: <.SesKey,1.>;
        ASessionKey: <.AId,BId,SesKey.>;
Tester: <.0.>; }
  out { MemorySesKey: <.SesKey,1.>;
        ASessionKey: <.AId,BId,SesKey.>;
Tester: <.1.>; }
#endtr

#trans foundProblemB
  in  { MemorySesKey: <.SesKey,1.>;
        BSessionKey: <.BId,AId,SesKey.>;
Tester: <.0.>; }
  out { MemorySesKey: <.SesKey,1.>;
        BSessionKey: <.BId,AId,SesKey.>;
Tester: <.1.>; }
#endtr
```

B Using Probe to track a failure of the protocol

```
0#query bspan (true) %2
   0  [9>   10  [3>   14  [1>   16  [0>   17  [0>   18  [0>   19  [0>   20  [0>
  21  [0>   22  [0>   23  [0>   24  [0>   25  [0>   26  [0>   27  [0>   28  [0>
  29  [0>   30  [0>   31  [0>   32  [0>   33  [0>   34  [0>   35  [0>   36  [0>
  37  [0>   38  [0>   39  [0>   40  [0>   41  [0>   42  [0>   43  [0>   44  [1>
  46  [0>   47  [0>   48  [0>   49  [0>   50  [0>   51  [0>   52  [0>   53  [0>
  54  [0>   55  [0>   56  [0>   57  [0>   58  [0>   59  [0>   60  [0>   61  [0>
  62  [0>   63  [0>   64  [0>   65  [0>   66  [0>   67  [0>   68  [0>   69  [0>
  70  [0>   71  [0>   72  [0>   73  [0>   74  [0>   75  [0>   76  [0>   77  [0>
  78  [0>   79
1 paths
Built set %4
-----------------------------------------------
The intruder sends user B an old message recorded from
previous protocol runs. B does not notice that the
```

```
message and session key are old and sends a challenge.
-----------------------------------------------
10#succ arrow 3
Arrow 3: transition recForwardAndSendChallenge, precedence class 0
  MyId = Jim
  PriKey = prikey2
  Rn = brn1
  SesKey = iseskey1
  AId = Jules
  knowsBeforeEZ = 0
to node 14
-----------------------------------------------
[...irrelevant transitions removed...]
Since the intruder knows the old session key, it can
decrypt the challenge and send a response.
-----------------------------------------------
44#succ
Arrow 0: transition decryptEZ, precedence class 0
  SesKey = iseskey1
  Rn = brn1
to node 45
-----------------------------------------------
[...irrelevant transitions removed...]
-----------------------------------------------
75#succ
Arrow 0: transition addOne, precedence class 0
  Rn = brn1
  RnPlus1 = brn1plus1
to node 76
-----------------------------------------------
76#succ
Arrow 0: transition encryptEW, precedence class 0
  RnPlus1 = brn1plus1
  SesKey = iseskey1
to node 77
-----------------------------------------------
77#succ
Arrow 0: transition recResponse, precedence class 0
  MyId = Jim
  AId = Jules
  Rn = brn1
  SesKey = iseskey1
  RnPlus1 = brn1plus1
to node 78
-----------------------------------------------
A tester transition becomes enabled.
-----------------------------------------------
78#succ
Arrow 0: transition foundProblemB, precedence class 0
  SesKey = iseskey1
  BId = Jim
  AId = Jules
to node 79
-----------------------------------------------
```