An AnB Tutorial

Sebastian Mödersheim,
DTU Informatics, samo@imm.dtu.dk

September 20, 2011

AnB is a formal specification language based on the popular Alice-and-Bob-style notation of security protocols. It is connected to the OFMC tool by means of a translator to OFMC's native language IF, the AVISPA Intermediate Format [1]. The expressiveness of IF is larger than that of AnB, but AnB is much easier to use and nicer to read.

A formal definition of AnB can be found in [5]. This tutorial serves as a gentle introduction to the concepts and how to use the language and also, to some extent, OFMC. To keep matters simple, we use a lot of trivial examples of protocols that can be found in the Clark-Jacob library.

Highlighting Since this tutorial does not assume that readers are familiar with formal verification, it might be boring to read for people who are. Therefore we note within the text with paragraph-headings where a particular concept is introduced or discussed in detail so it can be looked up quickly.

Moreover, a few central points are summarized in such a box.

1 Building a Key-Establishment Protocol

We follow here an example of protocol development found at the beginning of the book *Protocols for Authentication and Key-Establishment* by Colin Boyd and Anish Mathuria [2]. The point in that book is to show the stepwise development of a protocol and motivate each step as an answer to a security problem. Here, we use it to illustrate AnB and the output of OFMC instead.

1.1 First Attempt

We want to write a simple *key-exchange protocol* that establishes a shared symmetric key between two parties Alice and Bob that do not have a security relationship so far. This newly established key shall then allow them to communicate securely by encrypting messages with that key. Since we cannot establish such a secure connection out of thin air, we need some form of existing relationship to begin with, and here it will be a *trusted third party s* (for "server"). If we entirely omit all the cryptography for now, a very simple (and trivially insecure) protocol is the following:

```
Protocol: KeyEx # First Attempt

Types:  Agent A,B,s;
        Symmetric_key KAB

Knowledge:
  A: A,B,s;
  B: A,B,s;
  s: A,B,s
```

```
Actions:

A->s: A,B
# s creates key KAB
s->A: KAB
A->B: A,KAB

Goals:

A authenticates s on KAB,B
B authenticates s on KAB,A
KAB secret between A,B,s
```

Actions and the Communication Medium Let us begin with the `Actions` section. Here we see the exchange of three messages: first, A tells the server s that she[1] would like to talk to B. The server creates a fresh symmetric key KAB and sends it back to her in the second step. In the third step she forwards the key to B and they can start communicating.

In fact, the communication here is *asynchronous*. The notation `A->B: M` indicates two things:

- that A sends a message M on an *insecure* communication medium;
- that B waits for receiving a message of the given format and then (and only then) continues with his next step.

The medium could be the Internet or a wireless connection: it is possible that an unauthorized third party listens (and records) to the transmitted messages, inserts messages under a fake sender ID. Moreover there is no guarantee that a sent message will arrive at the intended destination.

AnB also requires that in a sequence of messages, the receiver of one message is the sender of the next message.

Variables and Constants A, B and KAB—and all identifiers that begin with upper-case letters—are *variables*. That means that they are placeholders for a concrete value (the real name of an agent or a concrete symmetric key in this case) that will be filled in when the protocol is actually executed. Identifiers that start with a lower-case like s are *constants*.

Roles Variables and constants that are declared to be of type `Agent` are called *roles*. The use of variables and constants is crucial here: we will allow that variables of type agent can be instantiated arbitrarily with agent names. This includes the special agent `i` — the *intruder*. We will discuss below in more detail what the intruder can and cannot do, but for now it is worth pointing out that most roles should be specified like this. It represents that anybody can participate in the respective role of the protocol under their real name, including a *dishonest* person. It turns out that many protocols have surprising attacks when allowing dishonest participants. In contrast, we sometimes want a *trusted third party*, like s in this case, that needs to be honest for the protocol to work. [2] In a key-exchange protocol, a dishonest server can often trivially break the security goals. When we want such an honest participant we specify a constant like s that cannot be instantiated by the intruder.

[1] Throughout this tutorial, we assume that A (Alice) is female, B (Bob) and i (the intruder) are male, and all others (servers etc.) are neutrum.

[2] The word "trust" has often lead to confusions, since the statement "A trusts B" has nothing to do with the question whether B is actually trustworthy. We actually do not work with trust-statements, but rather only with honest/dishonest. Terms like "trusted third party" are so common however, that we use them here as well in the sense of "honest party".

Knowledge For each role of the protocol, one needs to specify an initial knowledge. This knowledge is essential to the meaning of the AnB specification as we will discuss at several points throughout this tutorial. In particular, we will check for every role and every message that they have to send, whether this message can be constructed by that role from the initial knowledge plus all messages it has received before. If this is not the case, then the specification has an error: it is *unexecutable* and will be rejected by the translator to IF.

Variables in Knowledge MUST be of Type Agent The initial knowledge will usually include the knowledge of all roles of the protocol.

> It is crucial that no term in the initial knowledge contains a variable any other than of type agent. For instance in our specification it would be an error to declare the variable KAB as part of the knowledge.

In fact, as examples below demonstrate, the modeling of long-term keys must be done using functions instead.[3]

Fresh Values All variables that are not part of the initial knowledge are freshly generated by the agent who first uses them, in our example, this is s. Fresh means in reality: an unpredictable random number; in the abstract formal world it means: when executing this step, the variable is instantiated with a new constant (and the intruder and all other agents initially do not know it this constant).

Secrecy Goals The most simple goal is secrecy: we denote a term and say between whom it shall be secret. In this case, the secret KAB and it is shared between all participants of the protocol. The specification of a group of people that share the secret is necessary: we allow the intruder to play role A or role B and in this case, he is of course clear to know the shared key of that particular protocol run. It is however an attack, if the intruder finds out a shared key of a protocol run between two honest agents playing in roles A and B. (And due to lack of encryption, this secrecy goal is trivially violated in the given protocol.)

We postpone the discussion of the more involved authentication goals to a later example.

Classic Mode It is suggested run every example first in OFMC's classic mode, this may look like this: `ofmc KeyEx1.AnB -classic` . The classic mode means that OFMC performs a classic model-checking analysis for a bounded number of *sessions* (protocol runs). We begin with 1 session, if no attacks are found, we continue with 2 sessions and so forth, until either an attack is found or the user stops OFMC. This classic mode is both good for quickly finding attacks and for gaining confidence in a protocol (if no attacks are found within hours of search), but we can prove the security only for a fixed number of sessions in this way.

Interpreting Attacks For our first protocol we get the following output in the AVISPA Output Format:

```
SUMMARY
  UNSAFE
GOAL
  secrets
...
ATTACK TRACE
i -> (s,1): x29.x28
(s,1) -> i: KAB(1)
i -> (i,17): KAB(1)
```

[3] At the state of this writing, the AnB2IF translator accepts variables in the knowledge that are not of type `Agent`, but they do behave like (public) values of type `Agent`.

```
i -> (i,17): KAB(1)
```

This indicates, unsurprisingly, that we have an attack against the secrecy goals. In the first step, the intruder sends a message to the server s. Here $(s, 1)$ indicates that it is the server in session 1—if we run several sessions of the protocol in parallel, which messages belong to the same instance of some agent. The message that the intruder sends is $x29.x28$. This is a pair of variables. Unfortunately, the Output format dictates here that a dot is used for pairing (concatenating) messages while most other formats (including AnB) use a comma for this. The variables $x29$ and $x28$ indicate that it is completely irrelevant for the attack what the intruder chooses here. In this case, the server expects to receive two agent names, but just any will do for this attack. To be entirely precise, there are two choices of agent names that would not work: $x29 = i$ or $x28 = i$; in these cases the trace it would not be a violation of secrecy. The exclusion of particular values is currently not shown by OFMC.

Note that the intruder started the protocol with the server without the agent $x29$ having done anything. This is because the network is asynchronous and no party can be sure that the other parties are "on the same page" (that is ultimately the goal to achieve by a protocol).

The server now responds to the request by creating a new key and sending it. Fresh values in an attack will always be the name in AnB followed by a unique number (which is in fact the session number). Usually attacks will have pairs of steps where the intruder sends a message to an honest agent and receives an answer from that agent. This reflects an efficient view of the protocol analysis problem: the intruder *is* the communication medium and all communication that takes place is the intruder using the honest agents as kind of oracles.

With this answer of the server, the attack is already completed: the intruder now knows the shared key of two agents $x29$ and $x28$ that he can freely choose—violating secrecy. The last two lines of the attack are just a technicality of OFMC: all steps of the form $(i, 17)$ are just result of an internal check that the intruder could generate a secret that he was supposed not to.

1.2 Second Attempt

We clearly need to protect the transmission of the secret shared key KAB and for that, we would like to assume that every agent (including the intruder) *initially* has a shared key with the server. We may for instance imagine that s provides wireless access, but everyone who wants to use it has to first register. Let us say this registration happens offline (possibly checking a photo ID) and involves installing a unique username and password. The username would be in our abstract model the variable of type agent, and the password is a shared key with s.[4]

Modeling Long-Term Keys The important thing about the shared key is that it is not freshly generated in a session but it is perpetual information. For simplicity, let us we ignore the possibility of user revocation or updating of shared keys in regular intervals, and thus consider a model where the shared key between a user and the server never changes. Then, we could model such a shared key as a function of an agent, e.g. $sk(A, s)$ would be the shared key of A and s. As such a term contains only variables of type `Agent`, we are allowed to include them in the initial knowledge of a role.

The second attempt to our protocol is now as follows, where AnB uses the notation `{|M|}K` for the symmetric encryption of message M with key K:

```
Protocol: KeyEx # second attempt

Types: Agent A,B,s;
       Symmetric_key KAB;
       Function sk
```

[4]In reality, one would not directly use a textual password, but use a cryptographic hash function to generate a key from the password for instance.

```
Knowledge:
  A: A,B,s,sk(A,s);
  B: A,B,s,sk(B,s);
  s: A,B,s,sk(A,s),sk(B,s)

Actions:

A->s: A,B
s->A: {| KAB |}sk(A,s), {| KAB |}sk(B,s)
A->B: A,{| KAB |}sk(B,s)
```

and the goals are the same as before.

Use of Functions Note that we have declared `sk` as a "constant" of type `Function`. Here, OFMC currently does not do any type checking: one may apply such a function to any terms. Of course it is strongly recommended to keep the usage of such a function symbol consistent, e.g. in this case using it always with two arguments of type agent and using the result as a key for symmetric encryption.

Obviously every agent initially knows his or her shared key with the server. For the server we specify only the knowledge of the shared keys with the other two roles (as obviously shared keys with other principals is irrelevant for a session).

Executability In this new version of the protocol, the server does not transmit the key KAB unprotected as in the first version. Instead, it creates two encryptions, one using the shared key with A and another one using the shared key with B. These two encrypted messages are sent to A. According to her knowledge, A can only decrypt the first of the two messages it receives, while the second one cannot be analyzed by A. A is supposed to forward this second package to B who has the necessary shared key to decrypt that message. So at least in a run where the intruder does not interfere, all agents have enough knowledge to produce all messages they have to and end up with a copy of the shared key KAB.

A central observation here is that A cannot check the second part of the message from the server. Especially, if we think of an intruder producing such a message (possibly recycling older messages he has seen on the communication medium), only the first part needs to have correct format, while the second part can be anything. A will then send that anything on to B. If we look at A in isolation, we may describe it as a program of the form

```
send(A,B); receive({|KAB|}sk(A,s),X); send(X);
```

The Model of Symmetric Encryption Many cryptographers may associate with the term "symmetric encryption" only the pure encryption, without any means of protecting integrity such as a message authentication code (MAC). Such a pure encryption would be vulnerable to the intruder manipulating bits of the ciphertext and thereby changing the plaintext obtained by decryption without and the recipient cannot detect this manipulation. We believe that there are only very few cases in protocol verification when we actually need the pure symmetric encryption, but almost always we also need the integrity. We therefore model in AnB with `{|.|}.` a primitive that includes the integrity. For our concrete example that means, when A receives the two encrypted messages from the server, she will decrypt the first one to which she has the key; the integrity mechanism of the primitive allows her to check (which overwhelming probability) that the received message is indeed correctly encrypted with the right symmetric key $sk(A,s)$ and not some message manipulated by the intruder. Put another way, if the intruder sends any other message that is not of the form `{|.|}sk(A,s)`, then A will detect that and refuse it. We actually do not even model that the intruder tries sending ill-formed messages to honest agents that they will refuse.

The general semantics of AnB and the algebraic theories of OFMC allow to model primitives such as pure encryption without integrity check if they are really needed. One easy way to do it is to use the algebraic theory specification of OFMC and declare for symmetric encryption the cancellation property

```
scrypt(K,scrypt(K,M))=M
```

Here, `scrypt` is the name of the symmetric encryption primitive in the IF language.

We later look at how to model a MAC alone.

An Attack Against Weak Authentication Goals Running this second example with OFMC in classic mode, we get the following attack:

```
SUMMARY
  UNSAFE
GOAL
  weak_auth
...
ATTACK TRACE
i -> (s,1): x29.x401
(s,1) -> i: {|KAB(1)|}_(sk(x29.s)).{|KAB(1)|}_(sk(x401.s))
i -> (x401,1): x27.{|KAB(1)|}_(sk(x401.s))


% Reached State:
%
% request(x401,s,purpose,KAB(1).x27,1)
% witness(s,x29,purpose,KAB(1).x401)
% witness(s,x401,purpose,KAB(1).x29)
% ...
```

The attack is a violation against *weak authentication* (which corresponds to Lowe's *non-injective agreement* [4]). The weak authentication is part of the standard (strong) authentication goal (which corresponds to Lowe's *injective agreement* [4]) that we have specified. We have also noted three facts from the comments that OFMC gives out as part of the *Reached State comment*. These facts are helpful in understanding an authentication attack as they reflect what the honest agents think has happened from their point of view and that we review in detail now.

The attack again begins with the intruder sending two agent names to the server s and again the concrete values do not matter for the attack so it is just two variables, here $x29$ and $x401$. The server thus generates a fresh key $KAB(1)$ and encrypts it with the shared keys $sk(x29, s)$ and $sk(x401, s)$, respectively. At this point the two witness facts are created that we see in the reached state. They correspond to the two authentication goals we have specified:

```
A authenticates s on KAB,B
B authenticates s on KAB,A
```

First goal expresses that when A finishes the protocol run, she can be sure that she agrees with the server on the values of the variables KAB and B and (implicitly) also on her name A. The witness fact therefore reflects what the server s really thinks these values are: namely it thinks that its own name is s, it is communicating with $x29$ (that is whatever the intruder chooses in the concrete attack to be A). For the pair KAB, B, the concrete value that the server works with is $KAB(1), x401$, i.e. the fresh nonce it has just created and whatever agent name $x401$ the intruder chooses to be B. The entry `purpose` is a dummy entry because this field is not used by AnB currently. The second witness fact analogously reflects the servers belief with respect to the second authentication goal.

In the last step of the attack, the intruder forms a message that has the correct format of the last step of the protocol and sends it to agent $x401$, who is thus playing in role B. In this message

has chosen yet another agent name $x27$ (again the concrete value does not matter for the attack) to be the agent apparently in role A. The second part of that message is a valid encryption for $x401$ and so $x401$ believes that he has just received a shared key with the communication with $x27$, authorized by the server. This is reflected by the request fact. (The last argument of the request fact, 1, is only relevant for strong authentication and will be explained below.)

The situation of witness and request facts now shows that the intruder has caused a misunderstanding. s has meant the key for communication between $x29$ and $x401$, while $x401$ thinks the key is for communication between $x27$ and $x401$. So it is in this case the second goal being violated.

> More generally, the violation of weak authentication is given if there is a request fact without a matching witness fact. For a goal of the form `B authenticates A on M`, the witness fact reflects the point of view of A while the request fact reflects the point of view of B. This goal should thus be used if we the protocol is supposed ensure the authentic communication of a message M from A to B. It then counts as an attack, if B finishes the protocol believing that A has sent message M for him; this includes the case that A has meant the message for somebody else (as in the example attack) or it is somebody else than A sending this message, even somebody honest. Additionally, in contrast to weak authentication, the standard strong authentication includes also a freshness aspect that we discuss later.

The problem of the second-attempt protocol could be described as follows. Whenever the server produces an encryption `{|KAB|}sk(A,s)` then this indicates to A that the key has been produced by the server s for use between A and some other agent that is not mentioned in that message. Similarly, the message `{|KAB|}sk(B,s)` only indicates to B that KAB is a shared key for B and somebody else. Since everything outside the encryption can freely be manipulated by the intruder, he can easily confuse the agents and break authentication goals. One may wonder why such a confusion is such a big deal since the benefit intruder apparently does not benefit much from it. For that, consider that the established key may later be used for the transmission of sensitive information like banking transactions or medical data; it is very undesirable that such information are directed to a wrong party because of an authentication problem in the key-exchange.

1.3 Third Attempt

From the previous example we have learned that the encrypted messages by the server should explicitly mention the other agent that the key is meant for, i.e. in the encryption for A the name of B should be mentioned and vice-versa. The exchange then looks as follows:

```
A->s: A,B
s->A: {| KAB,B |}sk(A,s), {| KAB,A |}sk(B,s)
A->B: {| KAB,A |}sk(B,s)
```

Strong Authentication/Replay In this case, we get a violation of the strong authentication aspect of the goal:

```
Verified for 1 sessions

SUMMARY
  UNSAFE
GOAL
  strong_auth
...
ATTACK TRACE
i -> (s,1): x34.x501
(s,1) -> i: {|KAB(1).x501|}_(sk(x34.s)).{|KAB(1).x34|}_(sk(x501.s))
```

```
i -> (x501,1): {|KAB(1).x34|}_(sk(x501.s))
i -> (x501,2): {|KAB(1).x34|}_(sk(x501.s))


% Reached State:
%
% request(x501,s,purpose,KAB(1).x34,1)
% request(x501,s,purpose,KAB(1).x34,2)
% witness(s,x34,purpose,KAB(1).x501)
% witness(s,x501,purpose,KAB(1).x34)
...
```

The attack trace starts like the previous ones with the intruder sending a message to the server choosing two agent names, now called $x34$ and $x501$. The server answers with the corresponding message mentioning everywhere the agent names. (This produces again the corresponding witness facts.) Now the intruder sends this message to agent $x501$ which is actually as the protocol intends it. $x501$ generates a request term and this request term actually matches the second of the two witness terms; so authentication is fine here (for every request there is a matching witness). Now in the final step the intruder just sends the same message a second time—a *replay*. Note that the receiver is now $(x501, 2)$ while in the previous it was $(x501, 1)$. This means that in both cases it is the same agent $x501$, but it is playing in two different sessions of the protocol. Imagine that the last step of the attack happens much later, say a week, than the first three. That would mean $x501$ accepts a quite old key for communication again. This can be bad for several reasons. First, think of a banking transaction: if one can make the bank perform a transaction several times that was actually issued only once this is clearly a problem. Also it is in many contexts important that a message is recent and not a replay of an old message, e.g. think of electronic stock-market applications. Finally, in many scenarios such as wireless communication, shared keys may be of very limited length, allowing an intruder to find them in a brute-force attack that takes a few hours or days. Establishing a new key frequently can still provide security against such an intruder—but only if the key exchange protocol is protected against replay of course, so the intruder cannot re-introduce an old key that he has broken.

> A replay attack (and thus a violation of strong authentication without violating weak authentication) is characterized by two identical request terms with different session numbers, i.e. an agent is made accept the exact same message more than once.

In fact Lowe's definition of injective agreement [4] is more complicated: it requires basically (in our terminology) there is an injective mapping from request facts to corresponding witness facts. If assume, however, that the message being authenticated upon contains at least one part that is supposedly fresh (like the key KAB in this case), then we will never have two times exactly the same witness fact and two times exactly the same request fact occurs iff there is a replay attack.

Timestamps A very simple and natural way to ensure freshness is the use of timestamps in messages. Assuming we manage to have computers' clocks synchronized up to a few seconds, we can safely require that agents never accept messages that have a timestamp that is more than a few minutes old. This takes already ensures that only recent messages are accepted. Additionally, we can prevent any replay even within the validity of the timestamp, if all messages are stored as long as their timestamp is valid and newly incoming messages are checked against this store.

AnB (and OFMC) have no precise model of timestamps; the reason is that talking about concrete timing would require assigning also concrete times to all the normal operations and we would need to formalize also the speed at which the intruder can send messages and similar things.

However, the above sketched methods with timestamps effectively prevent old messages or replays. So if the protocol has these mechanisms in place, one may simply drop the check for replay in our model. In our example that would mean to write the authentication goals as:

```
A weakly authenticates s on KAB,B
```

```
B weakly authenticates s on KAB,A
```

With this, the protocol can actually be verified. In fact, looking closely at the attack trace against `strong_auth`, we see that the first line says (with slight grammatical problems):

```
Verified for 1 sessions
```

This means that looking at only one single session, OFMC found no attack. This is not surprising as a replay attack requires at least two sessions of some agent. Using now the weak authentication we see after some time also that it is verified for 2 sessions and so on.

1.4 Fourth Attempt

The described buffering of messages for a limited amount of time can still be an impractical solution in many scenarios, especially when dealing with large amounts of data or a distributed system. (Nonetheless the use of timestamps in electronic transaction is generally a good idea.)

Nonces An alternative way to ensure recentness is the use of *challenge-response* protocols. The challenge is random number chosen by one party; this number is often called a *nonce*. It abbreviates *number once*, indicating it should be used only one time. The point is that if another party has to include the nonce in a response, then the creator of the nonce can be sure that that response is no older than the nonce it contains. The value of these guarantees of course depends on the cryptographic operations in which the nonce is used.

Since in our case, we want to protect B against a replay of the key, we add two steps to the protocol, namely one where B generates a nonce NB and sends it encrypted with the new shared key KAB to A, and then A has to respond with $NB - 1$ encrypted with KAB. (The subtraction of 1 is so that the response is actually a different message than the challenge.) The protocol the looks as follows:

```
...
  Number NB;
  Function sk,pre
Knowledge:
  A: A,B,s,sk(A,s),pre;
  B: A,B,s,sk(B,s),pre;
  s: A,B,s,sk(A,s),sk(B,s),pre
Actions:
A->s: A,B
s->A: {| KAB,B |}sk(A,s), {| KAB,A |}sk(B,s)
A->B: {| KAB,A |}sk(B,s)
B->A: {| NB |}KAB
A->B: {| pre(NB) |}KAB
```

Public Functions To model the function -1 in our abstract model as simple as possible, we have declared a new function symbol *pre* and given that to the knowledge of every agent. As a consequence, every agent is able to produce $pre(M)$ for a message M that it knows. We do not model more aspects of arithmetic, because that is not really necessary for this model.

Needham-Schroeder Shared Key We have now arrived at a protocol very similar to a classic protocol, the Needham-Schroeder Shared Key protocol [8]. That protocol also had a nonce NA from A that is included in the server's message for A and here the two encryptions are nested, i.e. the server sends the message for B as part of the encrypted message for A:

```
A->s: A,B,NA
s->A: {| KAB,B,NA, {| KAB,A |}sk(B,s)  |}sk(A,s)
```

```
A->B: {| KAB,A |}sk(B,s)
B->A: {| NB |}KAB
A->B: {| pre(NB) |}KAB
```

Denning-Sacco attack on NSSK Both our fourth protocol and the NSSK are vulnerable for very similar attacks, first reported by Denning and Sacco [3]. For NSSK we obtain:

```
SUMMARY
  UNSAFE
GOAL
  strong_auth
...
ATTACK TRACE
i -> (s,1): i.x701.x206
(s,1) -> i: {|KAB(1).x701.x206.{|KAB(1).i|}_(sk(x701.s))|}_(sk(i.s))
i -> (x701,1): {|KAB(1).i|}_(sk(x701.s))
(x701,1) -> i: {|NB(2)|}_KAB(1)
i -> (x701,1): {|pre(NB(2))|}_KAB(1)
i -> (x701,2): {|KAB(1).i|}_(sk(x701.s))
(x701,2) -> i: {|NB(4)|}_KAB(1)
i -> (x701,2): {|pre(NB(4))|}_KAB(1)
```

The Intruder Acting Under His Real Name This is again a replay attack. As in the previous attacks, we begin with the intruder sending a message to the server s. Here for the first time, we see that the intruder chose a concrete name as a sender: his own name. The reason is that this particular attack only works if the intruder can decrypt the outermost encryption of the reply by the server, which is with the key $sk(A, s)$. The intruder does not know any shared key of an honest agent with the server, but he knows his own shared key with the server: $sk(i, s)$. So for the concrete choice $A = i$, he is actually able to decrypt the answer from the server.

The reader may wonder where it is specified that the intruder knows $sk(i, s)$. It is actually specified both by the knowledge of role A and role B, since both roles can be played by the intruder:

> In general, for the initial knowledge specification $A : m_1, \ldots, m_n$ (where A is a variable), then the intruder obtains for his initial knowledge all messages $m_1, \ldots, m_n$ where all occurrences of A are substituted by i.

The first 5 steps of the attack trace are in fact a perfectly normal protocol run: the intruder acts just like an honest agent would behave in role A. The variables $x701$ and $x206$ are again choices of the intruder, namely of the agent playing role B and the value of the nonce NA, that do not matter for the attack. The actual attack now happens in the last three steps. Here the intruder talks to a second session of the agent $x701$ (in role B) using the old message from the server and then responding to the challenge from $x701$. Note that $x701$ actually generates a fresh nonce $NB(4)$ for this second session.

Meaning of the Attack With this attack, the intruder makes an honest B accept an old session key a second time, violating the strong authentication goal between B and the server. In this form, the attack is actually not that interesting because the intruder needs to play under his real name to achieve it, so it is a session key for secure communication between i and B which is not very attractive to attack. The attack becomes more interesting if we think of KAB as a short session key (that can be broken with brute force within some hours) and $sk(A, s)$ and $sk(B, s)$ as long-term keys that have more length and cannot be broken by brute force. In this case, the attack would also work for an honest A because the intruder just needs to replay an old message of step 3 of the protocol for which he has cracked the contained session key.

AnB currently does not have a method to specify the loss of short-term secrets, although this can be done on the IF level. A (quite technical) workaround to specify it in AnB would be to add an old session key and messages in the initial knowledge, e.g.

```
A: ...,oldkab,{|oldkab,A|}sk(B,s)
```

However, the fact that we get a very similar attack by the normal specification (although it is less interesting) is often indicative that there may be other, related, problems.

1.5 Fifth Attempt

Denning and Sacco suggest to rearrange the protocol a bit and to let B start with sending a nonce NB to A, so that the server can include the nonces of both agents in its messages, and thus provide freshness guarantees to both agents. This protocol now looks as follows:

```
B->A: A,B,NB
A->s: A,B,NA,NB
s->A: {| KAB,B,NA |}sk(A,s), {| KAB,A,NB |}sk(B,s)
A->B: {| KAB,A,NB |}sk(B,s)
```

This protocol is considered secure by many (including [2]). However, OFMC still finds an attack!

```
GOAL
  strong_auth
...
ATTACK TRACE
(x701,1) -> i: x701.x701.NB(1)
(x701,2) -> i: x701.x701.NB(2)
i -> (s,2): x701.x701.NB(2).NB(1)
(s,2) -> i: {|KAB(3).x701.NB(2)|}_(sk(x701.s)).{|KAB(3).x701.NB(1)|}_(sk(x701.s))
i -> (x701,1): {|KAB(3).x701.NB(1)|}_(sk(x701.s))
i -> (x701,2): {|KAB(3).x701.NB(2)|}_(sk(x701.s))
```

In fact, the attack is more easy to understand if we reorder the messages in there (the slightly confusing ordering is due to partial-order reduction techniques used in OFMC [7]):

```
ATTACK TRACE
(x701,2) -> i: x701.x701.NB(2)
i -> (s,2): x701.x701.NB(2).NB(1)
(s,2) -> i: {|KAB(3).x701.NB(2)|}_(sk(x701.s)).{|KAB(3).x701.NB(1)|}_(sk(x701.s))
i -> (x701,2): {|KAB(3).x701.NB(2)|}_(sk(x701.s))
(x701,1) -> i: x701.x701.NB(1)
i -> (x701,1): {|KAB(3).x701.NB(1)|}_(sk(x701.s))
```

Talking to Oneself In the attack trace, we see a strange thing: the agent $x701$ who starts (playing role B) intends to talk to — $x701$. We see here that if the roles A and B can be instantiated by two agents, this does not exclude $A = B$. Some people have argued that such scenarios should be considered since a user may work on different physical machines and on all machines, the user may have the same long-term keys. Then, when a user (like $x701$ in this example) tries to establish a secure connection between the two machines (using Denning-Sacco in this case) he would instantiate both roles A and B and thus both shared keys with the server are the same, namely $sk(x701, s)$. If such a scenario is possible, i.e. if the protocol does not explicitly require that the logical name of the two endpoints are different, then the above attack is possible. Note here with *logical name* we mean the identity to which the keys are bound. This is usually *not* the concrete IP-address of the machine and could thus be completely independent from addressing mechanisms. We therefore recommend to make protocols even safe for agents

"talking to themselves" and interpret attacks as being related to different machines the agent is working on.

1.6 Final Version

The idea to fix the problem is to break the "symmetry" of the protocol in the sense that the messages for A and B respectively are actually different and cannot be confused. For simplicity we just enter the name of the server into one:

```
B->A: A,B,NB
A->s: A,B,NA,NB
s->A: {| KAB,B,NA |}sk(A,s), {| KAB,A,NB,s |}sk(B,s)
A->B: {| KAB,A,NB,s |}sk(B,s)
```

In this version OFMC finally accepts the protocol and begins searching for attacks in more and more sessions.

Using the Fixedpoint Mode Now we finally have a protocol that is accepted by OFMC for a couple of sessions. We can then proceed and try the newer and more experimental fixedpoint module that performs an abstraction similar to ProVerif . The advantage is that this verifies the protocol for an *unbounded* number of sessions. It may however happen that the abstraction leads to false attacks (that would not work in the concrete model of classic OFMC). Also the interpretation is far more restricted: it is blind for replay and reflection attacks.[5] Using the line

```
ofmc KeyEx6.AnB -fp
```

we obtain after a few seconds the result that no attacks are found and that the protocol is thus safe in this restricted model for an unbounded number of sessions. Note that it is always recommended to use both the classic and fixedpoint module to extensively check the protocol as being methods with complementary strengths that may find different kinds of attacks.

2 TLS

We now look at a more interesting example both since it is more complex and since it is one of the most widely used protocols in the Internet. Our model is inspired by the one of Paulson [9].

```
1  Protocol: TLS
2  Types: Agent A,B,s;
3         Number NA,NB,Sid,PA,PB,PMS;
4         Function pk,hash,clientK,serverK,prf
5  Knowledge: A: A,pk(A),pk(s),inv(pk(A)),{A,pk(A)}inv(pk(s)),B,
6                 hash,clientK,serverK,prf;
7             B: B,pk(B),pk(s),inv(pk(B)),{B,pk(B)}inv(pk(s)),
8                 hash,clientK,serverK,prf
9  Actions:
10 A->B: A,NA,Sid,PA
11 B->A: NB,Sid,PB,
12       {B,pk(B)}inv(pk(s))
13 A->B: {A,pk(A)}inv(pk(s)),
14         {PMS}pk(B),
15         {hash(NB,B,PMS)}inv(pk(A)),
16         {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
```

[5] The latter means that A tries to start a session with B and the intruder just reflects every message back to A. If the protocol is completely symmetric, this may appear as a second session started by B, and A actually finishes the protocol talking to herself.

```
17              clientK(NA,NB,prf(PMS,NA,NB))
18  B->A:    {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
19              serverK(NA,NB,prf(PMS,NA,NB))
20  Goals:
21    B authenticates A on prf(PMS,NA,NB)
22    A authenticates B on prf(PMS,NA,NB)
23    prf(PMS,NA,NB) secret between A,B
```

Walkthrough We discuss the messages step by step:

Client hello (line 10) A client A first contacts the server B that she wants to connect to. This includes a fresh nonce NA and session identifier Sid, as well as the security preferences PA. The security preferences cannot really be modeled here, and we replace them with a nonce (to not change the message format).

Server hello (line 11) The server replies with his own nonce NB and his own preferences PB (again represented as a nonce).

Server certificate (line 12) The server sends a certificate of his public key. This is essentially a digital signature by some trusted certificate authority s, signing for B's public key. Of course, the real certificates may contain more fields, in particular expiry dates, but we do not model that.

In our model, every agent A has a long-term public key $pk(A)$ and a corresponding private key $inv(pk(A))$. Note that pk is a function symbol that we declare similar to sk in previous examples to represent a given (static) key infrastructure. In contrast, inv is a built-in symbol that maps public keys to corresponding private keys. The general rule is that public-key encrypted messages can only be decrypted with the corresponding private key and vice-versa. Encryption with a private key thus means *signing* a message (because only the owner of the private key can have done that). We distinguish asymmetric (public/private-key) encryption from symmetric encryption by using the notation `{M}K` for encryption of message M with key K.

The initial knowledge of role A contains her public and private key as well as a certificate for her public key by the server and the server's public key. The knowledge of B is analogous. They both do not in advance know each other's public keys, modeling that they only learn them through the exchange of the certificates. In order to verify the certificates, they need the public key of the server. As a consequence, in the translation from AnB to IF, the first exchange looks like this on the side of A:

```
send(A,NA,Sid,PA).receive(NB,Sid,PB,{B,PKB}inv(pk(s)))
```

Here, the public key of B is learned by A as PKB from the certificate. A has no means to check $PKB = pk(B)$ as it is supposed to be, although this will always be the case since in this model nobody has the key $inv(pk(s))$, so nobody can forge certificates.

Client certificate (line 13) Similar to the server's certificate. Note this is optional in TLS: if omitted (which is usually the case if the client is a normal web-browser) then the client is not authenticated. The authentication and secrecy goals we state do not hold then. We discuss this interesting case of a unilaterally authenticated TLS channel below.

Client Key exchange (line 14) The client generates the *pre-master secret* PMS, which is just another fresh random number. This number is encrypted with the public key of the server.

Certificate verify (line 15) This signature is present iff the client certificate (line 13) is present. It then authenticates the PMS and links it with the nonce NB and the name of B.

What is signed is actually a *cryptographic hash* of NB, B, PMS. Recall that a cryptographic hash provides a cryptographic check function in the sense that for two random messages M and M', it is very unlikely that $h(M) = h(M')$ (low chance of collisions); it is difficult to obtain M from knowing only $h(M)$ (hard to invert); and for given M (or $h(M)$) it is hard to find M' such that $h(M) = h(M')$ (collision-resistant). We simply model this in AnB again as a new function symbol `hash` and give this function to initial knowledge of all roles, so everybody can compute $h(M)$ for given M; the hardness of finding collisions and inverses is modeled by the absence of intruder rules in the algebraic theory of OFMC.

Client finished (line 16-17) The next message for the first time contains the basis of the shared keys that A and B will obtain. This basis is $K = prf(PMS, NA, NB)$ where prf stands for *perfect random function* and is just another cryptographic hash (like in line 15). This basis K is used to create a message authentication code, i.e. a hash-function with a symmetric key. The original TLS specification tells us to MAC all messages that have been exchanged so far with K. To simplify this a bit, we use just the variables that occur so far. This hash is called the *"Finished"-Message*. It is transmitted encrypted with the client's shared key $clientK(NA, NB, K)$ where $clientK$ is yet another hash-function.

Server finished (line 18-19) The server B answers with the same finished message encrypted with his shared key $serverK(NA, NB, K)$ where $serverK$ is the last of the hash-functions we introduce. Note that both A and B can compute both client and server keys. The distinction is made so that messages from A to B can not be mistaken as messages from B to A.

2.1 Unilateral TLS

The most commonly used form of TLS is without the optional client authentication (i.e. lines 13 and 15 in the above AnB specification), because the user does not have a certificate. These connections have strictly weaker security guarantees: the server cannot be sure about the identity of the client he is talking with (while the client can, thanks to the server's certificate). Still, this client and server have a secure connection in the sense that confidentiality and integrity are preserved. We may think of a client having acting under a pseudonym and but being authenticated with respect to that pseudonym as proposed in [6].

OFMC supports special goals that reflect this weakened security goals, namely

```
[A] *->* B : prf(PMS,NA,NB)
B *->* [A] : prf(PMS,NA,NB)
```

The first goal indicates that A has securely (authentic and secret) communicated the key $prf(...)$ to B—where A's identity has not been certified hence the notation $[A]$. Similar we have secrecy and agreement in the other direction. For a formal definition of this channel notation see [6].

Exercise A very common use of the unilateral TLS is the authentication of a user with username and password. How can such a login via the unilateral TLS channel be modeled in AnB? What would be appropriate goals?

3 Construction Area

There are many more things to be said and explained and this tutorial is work in progress. If you have problems, questions, ideas, requests, please let me know.

References

[1] AVISPA. The Intermediate Format. Deliverable D2.3, Automated Validation of Internet Security Protocols and Applications (AVISPA), 2003. `http://www.avispa-project.org/delivs/2.3/d2-3.pdf`.

[2] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment*. Springer, 2003.

[3] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, 1981.

[4] G. Lowe. A hierarchy of authentication specifications. pages 31–43. IEEE Computer Society Press, 1997.

[5] S. Mödersheim. Algebraic Properties in Alice and Bob Notation. In *Proceedings of Ares'09*, 2009.

[6] S. Mödersheim and L. Viganò. Secure pseudonymous channels. In M. Backes and P. Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 337–354. Springer, 2009.

[7] S. Mödersheim, L. Viganò, and D. A. Basin. Constraint differentiation: Search-space reduction for the constraint-based analysis of security protocols. *Journal of Computer Security*, 18(4):575–618, 2010.

[8] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[9] L. C. Paulson. Inductive analysis of the internet protocol TLS. *ACM Transactions on Information and System Security*, 2(3):332–351, 1999.