Protocol Security Verification Tutorial

Sebastian Mödersheim,
DTU Compute, samo@dtu.dk

December 2018

1 Roadmap

This tutorial gives an introduction to modeling security protocols and the methods for automated verification that is hopefully easier accessible than research papers. For concreteness it uses OFMC, an automated protocol verification tool written by the author, and thus this document also serves as a user manual for OFMC. Several other methods and tools are briefly discussed in order to give a broader perspective.

In the first part, we will entirely focus on precisely describing security protocols, their security goals and a model of the intruder, so that "the protocol is secure" is a mathematical statement that is either true or false, and there is a chance to prove or disprove this statement. Disprove also entails finding a counter-example to security: an attack.

The second part is concerned with methods to automatically find the correct answer, i.e., to find a proof of security or an attack. This is difficult since in general there will be an infinite number of things that can happen in a protocol, so that exhaustive search is impossible. Even under reasonable restrictions that make the search space finite, this is often still practically infeasible. We will focus on two techniques that can in practice often deal well with protocol security problems. One is based on symbolic representation with constraints and it can often find attacks quickly; the other is based on abstract interpretation and it can often find proofs of security. We will also discuss why the problem is in general undecidable, i.e., there is no hope for finding any verification method that will always answer correctly for all protocols.

In the third part, two more advanced topics, namely channels and compositionality. The idea is that on the Internet we use a lot of protocols in parallel and in a stacked fashion, e.g., using TLS to establish a secure channel and run a banking application over that channel. While one could theoretically verify such a composed system, this becomes easily too complex to handle. Also, we would like protocol designers to design a protocol like TLS independent of the actually payload protocols that it will be used for later. The key of compositional reasoning is to just allow this component-wise development, i.e., that the composed system is secure if the components are.

Contents

Part I
Modeling Protocols

A danger in designing formal models in computer science lies in mixing the model with algorithmic aspects, i.e. the question *what* we want to check with the question *how* to perform the check. This can lead to models that are somewhat un-intuitive and hard to use, because they are a poor compromise between what one wants to express and what one can handle with a particular analysis method. Therefore we try to clearly separate the modeling in this part and the formal verification methods in the next part, and obtain here a model regardless of whether this can be automatically analyzed or not.

A clear, simple and declarative modeling language is actually the aim of the main input language of the OFMC tool: *AnB* [21]. AnB is based on the popular Alice-and-Bob notation that is informally used in many textbooks. However AnB is a formal language – like a programming language or a logic – because it has:

- a syntax (what is a valid protocol in the language AnB?)
- and a formal semantics (what does an AnB specification mean?)

That is, it has a programming language flavor, since there is a translator that generates executable protocol implementations in JavaScript; since this is however for an extension of AnB called *SPS* that is currently still in a prototypical stage, we refer here only to the literature [1].

OFMC also uses a lower-level input language, called *IF* (Intermediate Format) [2]. This is based on set-rewriting and considerably more tailored to the needs of the automation, in fact it is OFMC's "native language". Internally, AnB is translated into IF. IF is more expressive than AnB, but hard to use directly (i.e., without translation from a language like AnB). We do not discuss IF in this tutorial since its technical details may be too distracting, but will rather use a much simpler and nicer formalism to describe the behavior of honest agents in a protocol: *strands*. In fact, we define the semantics of AnB by translation into strands and how strands – together with the intruder – give rise to a *state transition system*.

Notation: A few central points are summarized in such a box.

2 Example: Building a Key-Establishment Protocol

Before we go into formal details, we want to first introduce AnB informally at hand of some examples. 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 [6]. 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.

2.1 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 form 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 to (and records) the transmitted messages and 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.[2]

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 by default all protocol roles should be specified as variables of type `Agent`, allowing everybody to participate in the respective role under their real name, including a *dishonest* person. This basically models that not all protocol participants are necessarily honest. It turns out that many protocols have surprising attacks when allowing dishonest participants. A

[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] This is not a limitation, since for instance to model `A->B: M1` followed by `C->D: M2` one could insert another message `B->C: dummy` where `dummy` is part of the initial knowledge of `B`. See, however, the more efficient solution of "piggy-backing" in the next variants of the protocol.

specification of a constant like s here is only to be used to model a *trusted third party*: a party that we require to be honest for the protocol to work. [3] In a key-exchange protocol, a dishonest server can often trivially break the security goals. Therefore, when we want to model such an honest participant, we specify a constant like s that cannot be instantiated by the intruder.

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 all terms in the initial knowledge contain only variables of type agent. For instance, in our specification it would be an error to declare the variable KAB as part of the knowledge.

We show in the next version of the protocol, how to model long-term keys (using functions).

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, KAB is freshly generated, and the generator is s since it sends the first message that KAB occurs in. 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 initially does not know 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 is 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 allowed 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.

Interpreting Attacks We run OFMC with the command line `ofmc KeyEx1.AnB` (the file is found in the tutorial folder of OFMC). This will start a search with a *bounded number of sessions*, i.e., it limits how many runs of each role of the protocol we have (although with arbitrary agents playing the roles). OFMC starts with 1 session (i.e., one "copy" of each role), then 2 sessions, and so on, until it finds an attack or the user interrupts. So for a correct protocol, OFMC does not terminate (in this setting).[4]

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

```
SUMMARY
  ATTACK_FOUND
GOAL
  secrets
...
```

[3] 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".

[4] To enforce termination, one may also specify a fixed number of sessions, say 3, with option `--numSess 3` .

```
ATTACK TRACE
i -> (s,1): x29,x28
(s,1) -> i: KAB(1)
i -> (i,17): KAB(1)
i -> (i,17): KAB(1)
```

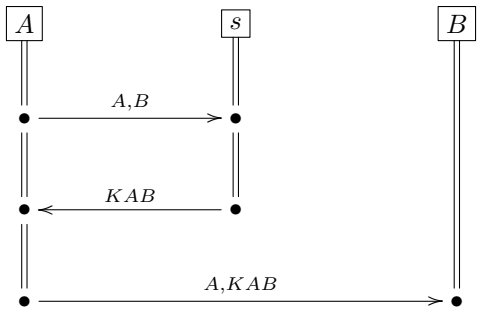
This indicates, unsurprisingly, that we have an attack against the secrecy goals. In the first step of the attack, the intruder sends a message to the server s. Here $(s, 1)$ indicates that it is the server in session 1. The point of this identifier is to tell us, when several sessions of the protocol run in parallel, which messages belong to the same run of an agent. The message that the intruder sends is $x29, x28$. This is a pair of variables. 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.[5]

Note that the intruder started the protocol with the server without the agent $x29$ having done anything. This is because the network is asynchronous, i.e., there is no guarantee that the intended recipient will actually receive the message. In fact it is often the goal of a protocol to get to a state where all parties "are on the same page". 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 a session number).

Usually attacks will consist of pairs of steps where the intruder sends a message to an honest agent and receives an answer from that agent, like the first two messages in the attack here. This reflects an efficient view of the protocol analysis problem: the intruder *is* the communication medium in the sense that all messages received by an honest agent come from the intruder and all messages sent by an honest agent are received by the intruder. Thus the intruder uses the honest agents like *oracles*.

With this answer from the server (the second message of the attack), 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 produce a secrets that he was not supposed to see.

Message Sequence Charts A popular graphical notation for protocols, closely related to Alice and Bob notation, is the message sequence chart, or MSC for short, where we have one column for each role of the protocol and denote with arrows the messages exchanged between the roles. For instance the message exchange of the above protocol as a message sequence chart looks as follows:



Similarly, an attack trace can be represented by a message sequence chart, for instance using the OFMC option `--attacktrace`, an SVG file is created containing the attack as an MSC.

[5]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 would not be a violation of secrecy. The exclusion of particular values is currently not shown by OFMC.

2.2 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.[6]

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. Recall that we are not allowed to have any variable like KAB that is not of type Agent in the initial knowledge of a role. However, we can declare new function symbols and use them to model long-term keys as a function of the agents who share them, e.g., use $sk(A, s)$ to represent the shared key of A and s. As such a term contains only variables of type `Agent`, it is allowed to include it 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

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, e.g., we did not specify that `sk` should be a function of two arguments, and if we use it with a different number of arguments, OFMC will not complain (this may be a source of errors).

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.

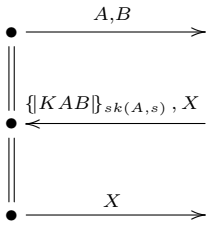
Executability In this new version of the protocol, the server does not transmit the key KAB unprotected as in the first version. Instead, it creates two encryptions, one using the shared key with A and another one using the shared key with B. These two encrypted messages are sent to A. According to her 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.

[6] 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.

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 any message X. A will then pass X 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);
```

In fact, this sequence of send and receive events is called a *strand* [14] and can be used to describe the behavior of an honest agent. We sometimes also like to use the following graphical notation for strands where outgoing arrows represent sending messages and incoming arrows represent receiving messages:



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 encrypted text so that the recipient cannot detect the 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 `{|M|}K` 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 (with 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 `{|M|}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.

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

```
SUMMARY
  ATTACK_FOUND
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,pBsKABA,KAB(1),x27,1)
% witness(s,x29,pAsKABB,KAB(1),x401)
% witness(s,x401,pBsKABA,KAB(1),x29)
% ...
```

The attack is a violation against *weak authentication* (which corresponds to Lowe's *non-injective agreement* [18]). The weak authentication is part of the standard (strong) authentication goal (which corresponds to Lowe's *injective agreement* [18]) that we have specified. We have displayed in the attack trace three facts from the comments that OFMC gives out as part of the *Reached State comment*. These facts are sometimes 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. But let us first understand it just from the message trace.

What the intruder has done here is that he chose two arbitrary agent names `x29,x401` and sent them to the server, who then created a new shared key `KAB(1)` for these two agents. The intruder sends this message now to `x401`, but claiming to be `x27`, i.e., a different person than who the server generated the key for. To `x401` this message looks like a perfectly correct step 3 of the protocol, so it will believe that `KAB(1)` is a shared key with `x27`. So the intruder has not found out any secret, but he managed to break the authentication: server `s` and recipient `x401` disagree on who is playing role A of the protocol in this session, i.e., who the key is shared with, thus it is a violation of the goal `B authenticates s on A,KAB`.

We will define the authentication goals formally later of course, but it can sometimes help understanding the point of view of an agent by looking at the witness/request facts (in comments of the attack output): the fact `witness(s,x29,pAsKABB,KAB(1),x401)` means that the server `s` intends to run the protocol with agent `x29` in role B, using `KAB(1)` as a key (for variable KAB of the protocol) and `x401` for role A of the protocol. (The identifier `pAsKABB` is just a technicality to distinguish several similar authentication goals.) In contrast, the fact `request(x401,s,pBsKABA,KAB(1),x27,1)` shows what `x401` is thinking: he thinks `x27` is playing role B. (The number `1` is for strong authentication, as explained later.)

> 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 the protocol is supposed to 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 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.

Actually, this authentication problem can be used to break secrecy – to that end the user may just comment out the two authentication goals and observe that OFMC finds then a secrecy violation.

2.3 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)
```

This time we get an attack that is not described in the book by Colin Boyd and Anish Mathuria [6] whose development we were following so far:

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


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

This is indeed a very subtle attack, and one may even argue that this should not be considered an attack. In fact, OFMC has—in the present version—a more sensitive notion of authentication. In contrast to other definitions of authentication, we do not only require that the parties agree on some data, e.g., here the agent `x401` and the server `s` on the key `KAB(1)`; rather, we also require that they agree on the which roles they play. In fact, the agent `x401` believes to play role `B` here, while the server thinks that `x401` plays role `A`. This may be considered less important, mainly because the intruder did not learn the key, and nobody got confused about the names of the partners they are talking with; however, it can in general lead to problems when there is confusion in which role the different participants are acting. (In fact, this is the first version to satisfy the secrecy goal.)

We therefore slightly divert from the development in [6] and change the message format to take this into account. In order to keep consistency with the book, we call this version "3b". Basically we need to prevent that the messages that the server sends to role A and role B could be confused. There are in fact many ways to do this, e.g., introducing new constants. Instead, we simply mention both the agent A and the agent B in the messages. This is good practice since now both encrypted messages from the server have exactly the same meaning: the first field is the key, the second field is the agent playing role A, and the third field is the agent playing role B:

```
A->s: A,B
# s creates key KAB
s->A: {| KAB,A,B |}sk(A,s), {| KAB,A,B |}sk(B,s)
A->B: {| KAB,A,B |}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
  ATTACK_FOUND
GOAL
  strong_auth
...
```

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

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

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 to accept the exact same message more than once.

In fact Lowe's definition of injective agreement [18] is more complicated: it requires basically (in our terminology) that there is an injective mapping from request facts to corresponding witness facts. If we 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 bearing a timestamp that is more than a few minutes old. This 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.

2.4 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 a 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,A,B |}sk(A,s), {| KAB,A,B |}sk(B,s)
A->B: {| KAB,A,B |}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 it 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.

This version has a similar attack as the following famous variant:

Needham-Schroeder Shared Key We have now arrived at a protocol very similar to a classic protocol, the Needham-Schroeder Shared Key (NSSK) protocol [25]. 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 [9]. For NSSK we obtain:

```
SUMMARY
  ATTACK_FOUND
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. 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.

2.5 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 [6]). However, OFMC still finds an attack! First, we get again the role confusion problem of the 3rd attempt. So the stricter goals of authentication that OFMC is using are still not satisfied. Let us fix that the same way we did before changing the protocol into:

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

Anyway we *still* get an attack! This time it is a replay 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,x701,NB(2)|}_(sk(x701,s)),{|KAB(3),x701,x701,NB(1)|}_(sk(x701,s))
i -> (x701,1): {|KAB(3),x701,x701,NB(1)|}_(sk(x701,s))
i -> (x701,2): {|KAB(3),x701,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 [24]):

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

2.6 Final Version

A simple way to fix this last attack is to simply add a constraint to the knowledge section:

```
where A!=B
```

This prevents all instantiations of the roles where A and B are played by the same agent. This should of course be noted when implementing the protocol: the implementation should always check whether the identity of the other partner claims to be the same agent identifier (that would have the same key).

3 Example: 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 [26].

```
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)|}
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.

3.1 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 acting under a pseudonym and being authenticated with respect to that pseudonym as proposed in [23]. We will come back to this when discussing channels in Section 15.

3.2 Diffie-Hellman

Diffie-Hellman [10] is a cryptographic primitive that can be regarded as the beginning of public key cryptography (as it pre-dates RSA); it has also an interactive/protocol aspect and we shall give an short introduction, since it is usually a good idea to create fresh keys using Diffie-Hellman.

To go slightly into cryptography, the idea is to pick first a large prime number p (which is also publicly known) and make computations in $\mathbb{Z}_p^\star$, that is the group of numbers $\{1, \ldots, p-1\}$ with multiplication modulo p. For instance let $p = 7$ (to have a small prime number of the example) and write $\equiv_p$ for equivalence modulo p, then $3^3 \equiv_7 3 \cdot 3 \cdot 3 \equiv_7 9 \cdot 3 \equiv_7 2 \cdot 3 \equiv_7 6$. Thus in all multiplications (and thus exponentations) we "stay" within zp. Note also that any of the intermediate results can be taken modulo p (so also intermediate results do not get larger than $p-1$). Next, we fix also a *generator* $g \in zp$: a generator is an element so that every for every $z \in \mathbb{Z}_p^\star$, there is an $x \in \mathbb{Z}_p^\star$ such that $g^x \equiv_p z$. Both g and p are fixed and publicly known. An important property is that given x, it is easy to compute $g^x(modp)$, but the opposite direction it is believed to be a hard problem, i.e., the best known algorithm that given z as input finds an x such that $g^x \equiv_p z$ is exponential (in the size of p).

Now Diffie-Hellman between two agents A and B is essentially the following: both A and B generate random numbers X and Y, respectively, and make the following exchange:

```
A->B: exp(g,X)
B->A: exp(g,Y)
```

where for simplicity we omit the fixed prime modulus p and write exp for modular exponentiation. After this exchange, the numbers X and Y are still secret to the participant who created them. Now A can exponentiate the value $exp(g, Y)$ from B with her secret X, i.e., $\exp(exp(g, Y), X)$. Similarly B can exponentiate the value $\exp(g, X)$ from A with his secret Y, i.e., $\exp(\exp(g, X), Y)$. By the laws of exponentiation (that also hold modulo p) that is the same: $exp(exp(g, Y), X) = exp(exp(g, X), Y)$. Thus they have a secret shared key that can only be constructed by the creators of X and Y.

So in a way, we have created here a secret "out of thin air", i.e., without A or B having to have any prior relationship with each other and without the help of a trusted third party. Of course, that is not entirely true, we need to be precise here: we do not have a secret between A and B necessarily, but between whoever created X and Y. Any attacker could have generated

his own secret X and send $\exp(g, X)$, claiming to be A (or similar impersonating B). There is no relationship here between A and $\exp(g, X)$ or between B and $\exp(g, Y)$. However we can make such a relationship using any approach to *authenticate* the exchange. Here is a simple Diffie-Hellman based protocol that uses digital signatures to authenticate the exchange:

```
Protocol: DH
# A simple protocol based on Diffie-Hellman

Types: Agent A,B;
       Number X,Y,g,MsgA,MsgB;
       Function pk;

Knowledge: A: A,B,pk(A),pk(B),inv(pk(A)),g;
           B: A,B,pk(A),pk(B),inv(pk(B)),g
where A!=B

Actions:

A -> B: {A,B, exp(g,X)}inv(pk(A))
B -> A: {A,B, exp(g,Y)}inv(pk(B))
A -> B: {|A,B,MsgA|}exp(exp(g,X),Y)
B -> A: {|B,A,MsgB|}exp(exp(g,X),Y)
Goals:

A authenticates B on exp(exp(g,X),Y),MsgB
B authenticates A on exp(exp(g,X),Y),MsgA
exp(exp(g,X),Y) secret between A,B
MsgA secret between A,B
MsgB secret between A,B
```

Here we use the Diffie-Hellman key to exchange some "payload" messages $MsgA$ and $MsgB$. It is actually left implicit how A and B create the key, i.e., that A rather has to construct $exp(exp(g, Y)X)$ to be able to encrypt here payload message to B and decrypt B's payload message to her. OFMC does this automatically, since the property that $exp(exp(g, X), Y) = exp(exp(g, Y), X)$ is understood by the compiler (and the protocol analysis). How this is actually computed is discussed in Section 10.

What was actually revolutionary when Diffie-Hellman was introduced becomes clear if we compare it to the key exchange protocols that can be built with symmetric cryptography only (like the first example of this section). Suppose two parties who already have a shared secret want to update it to a new secret. We could have a simple update protocol like this:

```
A->B: \scrypt{K}{update,K'}
```

where K is the current shared secret key of A and B and K' is a fresh key that A just created. One may make a more complicated protocol to also make key confirmation etc., but let us focus on this. Basically the only choice to exchange a new secret like K' is to already have a shared secret K. Diffie-Hellman requires a bit less: we do not have to have a secret, it suffices to be able to authenticate each other, while confidentiality is not important: the intruder may well see the exchanged exponents. This is why we consider this actually like the beginning of public key cryptography: one can regard X and Y as private keys and $exp(g, X)$ and $exp(g, Y)$ as public keys. It suffices to authentically distribute the public keys, then we can build any secure channels out of it.

Another advantage is built in: both parties contributed to the Diffie-Hellman key, so both have an implicit guarantee of freshness. (However, this can also be achieved with symmetric cryptography.) Finally Diffie-Hellman has perfect forward secrecy: consider the following Diffie-Hellman-based key update protocol where A and B currently shared a symmetric key K (may be

constructed earlier using Diffie-Hellman) and they update as follows:

```
A -> B: {|update,A,exp(g,X)|}K
B -> A: {|update,B,exp(g,Y)|}K
```

and suppose the agents from now on use the resulting $K' = exp(exp(g, X), Y)$ as new shared key. Suppose now, the key K becomes known to the intruder (e.g., by a brute force attack or some mistake) *after* this exchange. Then still, he cannot construct K' (except by a separate brute force attack). In contrast, in the conventional key update before, if the intruder has logged all the trafic and finds out just one key, all the following keys fall like domino stones.

In a very similar way, consider our key exchange protocol with the trusted third party from the beginning of this section. If the intruder manages to hack the trusted third party at any time and obtain the long-term keys of some agents, then he can decrypt every conversation where he has logged the key-exchange messages. That is not the case if we exchange the protocol to use Diffie-Hellman for creating the shared keys.

For all these reasons, it is usually a good idea to use Diffie-Hellman to construct shared keys – and why *exp* has "the full support" of AnB/OFMC.

4 The Syntax of AnB

After the examples we have seen so far, we want to first define precisely the syntax of Alice and Bob notation, and then define formally, what it actually means – the semantics. To that end, let us consider a full specification, the Needham-Schroeder Shared-Key protocol, which is one of the protocols we have "walked into" in our development in Section 2.4.

```
Protocol:    NSSK # Needham Schroeder Shared Key
Types:       Agent A,B,s;
             Number NA,NB;
             Symmetric_key KAB;
             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,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
Goals:
  A authenticates s on KAB,B
  B authenticates s on KAB,A
  KAB secret between A,B,s
```

Obviously the first item is to give the protocol a name and comments can be made using the # sign. The we have to declare the types of all identifiers that we use in the protocol (actually OFMC allows omitting type declarations, but this may lead to unexpected results). As possible types we have:

- `Agent`: the type of all agents (participants) in the protocol, i.e., who can send or receive messages.
- `Number`: random nonces and the like.
- `Symmetric_key`: similar to nonces (in fact either can be used as keys in encryptions).

- `Public_key`: for declaring a public key, i.e. asymmetric encryption); if P is a public key then $\mathsf{inv}(P)$ is the corresponding private key, so there is no type for specifying private keys.
- `Function`: modeling a function (of arbitrary arguments). These functions will by default *not* be available to the intruder, so one can, like in the example, use it to define a key-infrastructure.

Note that the types `Symmetric_key` and `Public_key` are only used for *freshly created keys*, i.e., that are newly created during the protocol run. For long-term keys (that exist before a particular protocol run), one must use functions like `sk` in the example.

Variables are identifiers that start with an uppercase character, while constants and functions start with a lower-case character.

We can now build *terms*, or *messages*, using the functions and constants symbols that have been declared including the following *built-in* functions:

- For any term `p` (that represents a public key) the corresponding private key is `inv(p)`.
- Public key encryption is denoted `{m}p` where `p` is the public key and `m` is the message encrypted. (In order to decrypt one needs the corresponding private key `inv(p)`.) The same symbol is also used for signatures: `{m}inv(p)` is a signature on the message `m` with private key `inv(p)`. (To verify the signature, one needs the corresponding public key `p`.)
- Symmetric encryption is denoted `{|m|}k` where `k` can be an arbitrary term representing the encryption key, and `m` is the encrypted message. In the text, where we are not limited by ASCII, we may rather write more nicely $\{\!|m|\!\}_k$.
- A concatenation of messages is simply written with commas between the messages, e.g., `KAB,B`. Note that "," is thus an infix operator, and it binds less than any other, e.g. $\{\mathtt{c}\}\mathtt{a},\mathtt{b}$ is equivalent to $(\{\mathtt{c}\}\mathtt{a}),\mathtt{b}$.[7] Also, the operator is right-associative, i.e., `A,B,C` is understood like `A,(B,C)`.
- The function $\exp(b, e)$ represents modular exponentiation (where the modulus is not explicitly written) of basis b with exponent e. This is for use in Diffie-Hellman based protocols, and we cover it more precisely below in Section 3.2.
- There is also a partial support for bitwise exclusive or (XOR), but it is actually not recommended to use XOR: such a function should only occur in the implementation of cryptographic primitives, but not in the protocols itself, and often it is indicative of poor protocol design.[8]

Note that there are no hash or MAC functions built in; one simply declares them as user-defined functions and makes them public (i.e., add to the knowledge of every protocol role).

Further, there are no decryption functions; this is because they are not used to construct any messages (but only to extract information from messages) and thus, they are not part of any term in AnB.

In fact, we later see that even in modeling the actions of honest agents and intruder, we do not need function symbols for decryption.

[7] To express a key that is a concatenation, one uses parentheses, e.g., $\{\mathtt{c}\}(\mathtt{a},\mathtt{b})$. However, usually one would not have such a raw concatenation, but some key derivation function like in the TLS example.

[8] There are some exceptions. First, distance-bounding protocols, but here so many other assumptions are needed that standard protocol models and verification tools cannot help. Second, protocols for devices with very low power consumption; but these cannot protect against any intruder with at least modest computational power. Third, one-time pads; however these are best modeled by a confidential channel.

The next part of the AnB specification is the initial knowledge. The knowledge should be specified for any *role* of the protocol, i.e., for every variable and constant of type agent (except for ones that never send or receive any messages like the server `s` in the TLS example).

> The knowledge is a list of messages that the role initially knows. All variables in the knowledge must be of type `Agent`. All variables that do not occur in the knowledge section represent messages freshly created during the protocol run by the agent who first uses them. The knowledge of a role must be sufficient to *execute* the protocol (as defined below).

Intuitively, the requirement of "executable" means that at any step of the protocol agents must have sufficient knowledge (from initial knowledge and received messages) to construct the next message they are supposed to send. In fact, this is at the very core of understanding the *meaning* of AnB.

The core of an AnB specification is the action section, where we have steps of the form $A \to B : m$ for two roles A and B and a message m. The roles can be constants or variables of type agent. Later in Section 15 we will also introduce here a channel notation to allow for communication channels with some built-in security properties. The channel $A \to B$ in contrast means that there are no guarantees on the security of the communication medium, e.g. the Internet or a wireless transmission where potentially an attacker could listen, send, or even intercept messages.

Finally in the goals section we can specify three kinds of goals (and their formal meaning will be defined in Section 9):

- `m secret between A,B,C` where `m` is a message and `A,B,C` is a list of protocol roles that are allowed to know the message.
- `B weakly authenticates A on m` where `m` is a message and `A` and `B` are roles of the protocol. The intuitive meaning is that when `B` finishes the protocol, then `A` has at least started the protocol and agrees with `B` on each others name and the message `m`.
- `B authenticates A on m`. Like the `weakly` variant, but additionally we require that there is no replay, i.e., that `B` did not complete more sessions than `A` started.

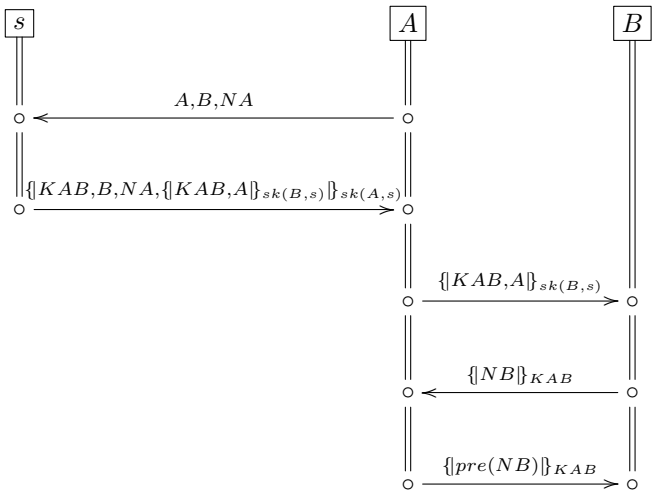
The "direction" (from A to B) that is implicit in the authentication goals may be confusing at first. Suppose m is a message generated by A in the protocol and then somehow transmitted (even indirectly) to B. Then it is clear that authentication is a goal from B's point of view, i.e., B wants to be sure that this message indeed comes from A. In contrast, A does not have have any guarantees from this goal: it does guarantee neither that the message will only reach B (that would have to be a secrecy goal) nor that the message will at all reach B (since the intruder can disrupt communication). Consider again the example of the authentication goals for key `KAB` in the NSSK example: here we declare the goal that both agents A and B authenticate the server s on the key, who is actually the one generating the key. We could however also formulate a goal that A and B should authenticate each other on the key. It is instructive to experiment with such goals and try to understand the attacks that result in some cases.

5 From Alice and Bob to Strands – Intuition

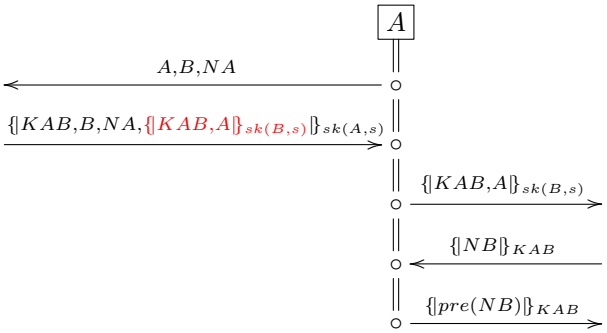
Even though Alice and Bob notation seems fairly simple and intuitive, it is surprisingly difficult to define it precisely. Essentially it describes how honest agents would behave, i.e., those that faithfully execute the protocol without trying to cheat. The intruder in contrast can do anything he likes, limited only by his knowledge and his cryptographic abilities. It turns out that, in order to define the behavior of the honest agents and the notion of executability, we also need a kind of intruder model: no agent can for instance perform encryptions or decryptions to which they do not posses the necessary keys. In other words, in order to execute a protocol, one should not need to break the cryptography.

We will now define the meaning of Alice and Bob notation by first a translation to strands – one for each role – that describes the behavior of each honest agent playing in this role. Together with a model of the intruder, this will then give rise to a *state transition system*, where each state is a simple model of the protocol world, and both the intruder and the honest agents can perform actions that lead to new states. The security goals will then be checks on states whether any goal has been violated (e.g., has the intruder found out a secret that he was not supposed to see); we call a state where a goal is violated, an *attack state*. The security of a protocol means then: no *reachable* state is an attack state. In fact there will be, in general, infinitely many states so that an exhaustive search directly will not work, but we will see some methods that can often "cope" with the infinity.

A first step in the translation is to write the action section of a protocol as a message sequence chart. For the NSSK example, we obtain the following MSC:



The main idea is now to split this into several strands, one for each role. For the NSSK example, for the role A this would be simply the messages that A is involved in (with some red highlighting explained below):



Suppose we choose concrete values for the variables A and B – some agent names – and for the variable NA – a fresh nonce, then this can be regarded as a fully specified program how this agent should behave:

1. It first sends out the message `A,B,NA`

2. It then waits for receiving any message that *matches* the *pattern* of the first incoming arrow. That is, it would have to be encrypted with the key `sk(A,s)` (the shared key of A and s), so A can decrypt this. The result of the decryption is a list of messages:

   - The first field is an arbitrary key KAB. That means, this variable is *bound* by whatever is received here – any value is accepted.
   - The second field must be the name of B sent in step 1.
   - The third field must be the random number NA sent in step 1.
   - The fourth one, highlighted in red is actually a bit strange, let us get back to that a little later.

   If any of the checks on the fields (or the decryption) fails, then this message is not accepted. It could then either be that the agent terminates the session, or keeps on waiting for another message that meets the expectations.

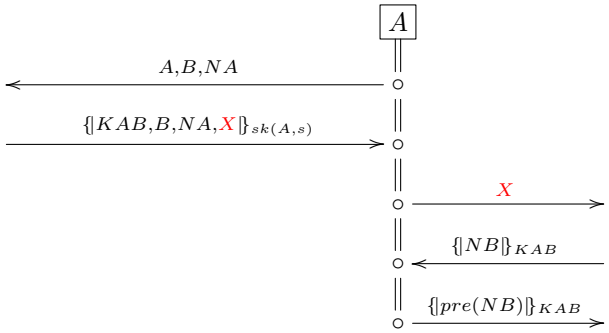
3. If everything went fine in step 2, the agent proceeds to step 3, sending out the "red" message.

4. The agent waits now to receive any message that is encrypted with the novel key KAB (that was learned in step 2). Again, the value NB is *bound* here – the agent accepts any value and remembers it as NB.

5. In the final step the agent applies the *pre* function to NB (representing $NB - 1$), encrypts it with KAB and sends it.

More generally, we are thus first instantiating with concrete terms all variables of type agent and the values that the agent in question freshly generates (note that A generates only NA, while KAB and NB are generated by others). For incoming messages, we only accept concrete terms that match the pattern we have, and this instantiates the remaining variables of that message. We will define this precisely below as a transition system.

But there is a strange thing about the highlighted red part in the example: this is a message encrypted with the shared key of B and s that agent A does not know. Therefore, there is no way for A to decrypt this message and check that it is really encrypted with that key, that it contains the same value KAB that was received in the first part of the message, and that it also contains the name A. The red part represents a couple of checks that A actually cannot perform! In fact, A should be willing to accept any message in this position. Of course, A sends out as step 3 whatever she received here. Thus, we get an accurate representation by replacing the red part with another variable X in both the step 2 and step 3:



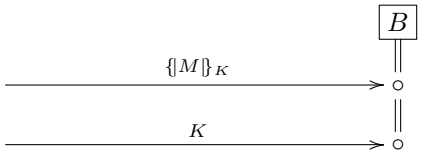
Thus, the translation of AnB to strands must take care of

- what agents can actually check on incoming messages and
- how they can generate outgoing messages.

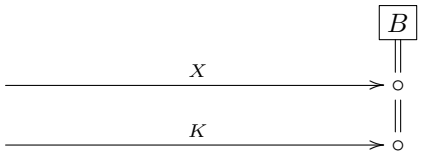
It may seem fairly simple: we just have to figure out subterms (like the red one in the above example) that an agent cannot decrypt and check, and that has therefore to be replaced by a variable. The following example shows that it is not so easy:

```
A->B: {| M |}K
# some time passes
A->B: K
```

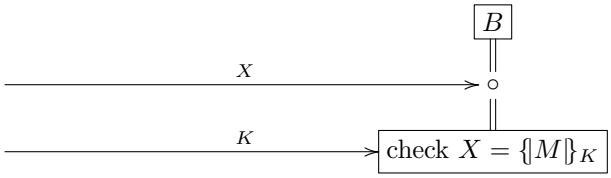
where K is a freshly generated key and M is some message that A wants to send to B but so that B cannot open it yet. At a later point she can now send K and open the message. How would the strand for B look like? The simple solution



is not correct, since B does not know K yet, and thus cannot check that he really received an encrypted message. Replacing it simply with X:



is also not correct, because we now lost any the crucial relation between the two received messages: in particular B would now accept any message X and then any message K (even if it does not decrypt the first one). For this one we would have to add a check in form of an equation:



In fact, this check would bind the variable M to whatever had been encrypted in the message X. Note that a protocol may well now contain steps that involve M, since B has learned it by decryption.

To precisely define the translation from Alice and Bob to strands, there is a sequence of increasingly more complicated papers, see for instance [15]. Few of these papers deal with another complication: that operators may have algebraic properties like

$$\exp(\exp(A, X), Y) = \exp(\exp(A, Y), X)$$

which is required for the translation of any Diffie-Hellman based protocols (without this property, these protocols would not be executable).

An interesting question was therefore: how can we define the meaning of Alice and Bob notation for an *arbitrary* given algebraic theory? This generalization of the question to arbitrary properties had a surprising effect: it lead to a much simpler definition than any of the previous specialized efforts [21, 7, 1].

This concludes for now our first overview of the Alice and Bob notation and its meaning. We now need to first look a bit at terms, algebraic properties, and intruder deduction, and then get back to Alice and Bob and strands.

6 Term Algebra and All That

We introduce now the basic notions of terms and algebras that are needed in the rest of this tutorial. For a more thorough account we suggest [3, 16].

6.1 Signatures

The basic logical term *signature* (not to be confused with for instance digital signatures) means the symbols from which we can build terms, and it is thus sometimes also called an alphabet:

Definition 1 (Signature)**.** *A* signature Σ *is a set of* function symbols *where every function symbol has an* arity *(number of arguments). We write f/n for a function symbol f of arity n. Constants are a special case of function symbols with arity* 0*.*

Example 1. $\Sigma_{\mathbb{Z}} = \{add/2, mult/2, minus/1, zero/0, one/0\}$*. While the names of these function symbols somehow suggest a meaning, there is none attached to them.*

Table 1 shows the function symbols we usually need in protocol verification, together with their intuitive meaning and whether or not they are public, i.e., whether the intruder can apply them. For instance, the function $\mathsf{inv}(\cdot)$ is not public because it shall later be used to assign a private key to a given public key; if $\mathsf{inv}(\cdot)$ *were* public, then the intruder would know all private keys that he knows the public key of. Let us denote with Σ_p in the following the subset of symbols from Σ that are declared *public*.

We also use the "mixfix" notation for encryption from AnB, i.e., we write $\{m\}_k$ instead of introducing a prefix function symbol like $crypt(k, m)$. We allow the user to declare their own constants and function symbols, so we cannot really give one fixed signature Σ – it all depends on what the user specifies in an AnB file. Similarly, we will in other parts of the document need to introduce new function symbols. That is why all the following definitions are given for an *arbitrary* Σ, i.e., the definitions are *parameterized* over Σ.

Symbol	Arity	Meaning (informal)	Public
i	0	name of the intruder	yes
$\mathsf{inv}(\cdot)$	1	private key of a given public key	no
$\{\cdot\}_\cdot$	2	asymmetric encryption	yes
$\{\!\|\cdot\|\!\}_\cdot$	2	symmetric encryption	yes
$\langle\cdot,\cdot\rangle$	2	pairing/concatenation	yes
$\exp(\cdot,\cdot)$	2	exponentiation modulo fixed prime p	yes
$a, b, c, \ldots$	0	User-defined constants	User-def.
$f(\cdot)$	$\star$	User-defined function symbol f	User-def.

Table 1: Standard function symbols for protocol verification.

Definition 2 (Terms)**.** *Let Σ be a signature and $V = \{X, Y, Z, \ldots\}$ be a set of* variable symbols *disjoint from Σ. Define $\mathcal{T}_\Sigma(V)$ to be the set of* terms *(over Σ and V) as follows:*

- *All variables of V are terms.*
- *If $t_1, \ldots, t_n$ are terms and $f/n \in \Sigma$, then also $f(t_1, \ldots, t_n)$ is a term.*

Terms without variables are called ground *terms. We write $\mathcal{T}_\Sigma$ for the set of all ground terms.*

Example 2. $\mathcal{T}_{\Sigma_{\mathbb{Z}}}(\{X, Y\}) = \{X, Y, zero, one, minus(X), minus(Y), minus(zero), minus(one), add(X, X), \ldots\}$*. Again, these terms have no meaning attached to them yet.*

A central concept is that of a substitution: it expresses that some variables should be substituted (replaced) by some terms:

Definition 3 (Substitution)**.** *Let $X_1, \ldots, X_n$ be variables and let $s_1, \ldots, s_n$ be terms. A* substitution *σ is written as $\sigma = [X_1 \mapsto s_1, \ldots, X_n \mapsto s_n]$ and means the following function from terms to terms:*

$$\sigma(t) = \begin{cases} s_i & \text{if } t = X_i \text{ for any } i \in \{1, \ldots, n\} \\ f(\sigma(t_1), \ldots, \sigma(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \\ t & \text{otherwise} \end{cases}$$

Thus, a substitution replaces every X_i with s_i, leaves all other variables untouched, and for a composed term $f(t_1, \ldots, t_n)$, the substitution is recursively applied to the subterms t_i.

Example 3. *For $\sigma = [X \mapsto f(Z), Y \mapsto Z]$ we have $\sigma(g(Z, Y, f(X))) = g(Z, Z, f(f(Z)))$.*

6.2 Algebra

While signatures and terms are pure syntax, an algebra gives symbols and terms a *meaning*. The idea is that we have to have a "universe" (or "domain") in which we want to interpret terms, and then interpret every function symbol as a function in that universe:

Definition 4 (Σ-Algebra)**.** *Given a signature Σ, a* Σ-Algebra *$\mathcal{A}$ consists of*

- *a non-empty set U, called the* Universe*, and*
- *for every $f/n \in \Sigma$ a function $f^{\mathcal{A}} : U^n \to U$.*

For a ground term $t \in \mathcal{T}_\Sigma$ and a Σ-Algebra $\mathcal{A}$, we denote with $t^{\mathcal{A}}$ the meaning of t in $\mathcal{A}$ which is:

$$f(t_1, \ldots, t_n)^{\mathcal{A}} = f^{\mathcal{A}}(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}})$$

Example 4. *One possible algebra $\mathcal{Z}$ for our example signature $\Sigma_{\mathbb{Z}} = \{add/2, mult/2, minus/1, zero/0, one/0\}$ is to interpret all terms as integer numbers:*

- *Universe:* $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$
- $add^{\mathcal{Z}}(a, b) = a + b$
- $mult^{\mathcal{Z}}(a, b) = a \cdot b$
- $minus^{\mathcal{Z}}(a) = -a$
- $zero^{\mathcal{Z}} = 0$
- $one^{\mathcal{Z}} = 1$

We can evaluate any ground term in $\mathcal{Z}$, for instance:

$$add(mult(one, minus(one)), one)^{\mathcal{Z}} = 0 \ .$$

Another choice is the algebra $\mathcal{B}$ that interprets terms as Booleans:

- *Universe:* $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$
- $add^{\mathcal{B}}(a, b) = a \vee b$ *(logical* or*)*
- $mult^{\mathcal{B}}(a, b) = a \wedge b$ *(logical* and*)*
- $minus^{\mathcal{B}}(a) = \neg a$ *(logical* not*)*
- $zero^{\mathcal{B}} = \mathsf{false}$
- $one^{\mathcal{B}} = \mathsf{true}$

It holds that

$$add(mult(one, minus(one)), one)^{\mathcal{B}} = \mathsf{true}\ .$$

Compare the result with the one for $\mathcal{Z}$; it is normal that in two different algebras we get different results for terms.

A possible algebra $\mathcal{C}$ for the signature of Table 1:

- *Universe: $\mathbb{B}^*$ (all bit-strings)*
- *$\{\!|m|\!\}_k^{\mathcal{C}}$ is AES 128 bit with MAC (returning an error message if k is not 128 bit long).*
- *...*

Observe that the algebras $\mathcal{Z}$ and $\mathcal{B}$ of this example share some properties, for instance both *add* and *mult* are associative and commutative and *mult* is distributive over *add*. More formally, let us write $s \approx_{\mathcal{A}} t$ if two (ground) terms s and t are equal in algebra $\mathcal{A}$, i.e., if $s^{\mathcal{A}} = t^{\mathcal{A}}$. Then we have for every ground terms s, t, u that

$$mult(s, add(t, u)) \approx_{\mathcal{Z}} add(mult(s, t), mult(s, u))$$

and the same holds for $\approx_{\mathcal{B}}$.

An important idea of mathematical logic is that, instead of studying a concrete algebra like $\mathcal{Z}$ and $\mathcal{B}$ and see what properties they have, we could rather take the opposite direction and start with a set of equations like distributivity above and ask: what is the class of algebras in which this property holds? This allows to *abstract* from the concrete algebra and prove statements that hold in *every* algebra that satisfies a number of equations. In that spirit, in cryptographers have studied cryptographic operators on an abstract level, i.e., considering only what (algebraic) properties the cryptographic building-blocks need to satisfy, allowing for the study of entire classes of cryptographic implementations [19].

6.3 The Free Algebra

In logic programming and theoretical computer science, one extreme form of algebra plays a central role: if we do *not assume any algebraic properties*. This is in some sense the most abstract and also somehow the most easy algebra to work with. (Hence the popularity in computer science...) It is often called the initial or free algebra because it is "freely generated" by its terms. Even though we have already used the notation $\mathcal{T}_\Sigma$ for the set of all ground terms, we will "overload" it and also use it to denote the free algebra:

Definition 5 (Free Algebra)**.** *Given a signature Σ, the* free algebra $\mathcal{T}_\Sigma$ *is defined as follows:*

- *The universe is the set of all ground terms $\mathcal{T}_\Sigma$.*
- *For every $f/n \in \Sigma$, define $f^{\mathcal{T}_\Sigma}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$.*

Lemma 1. *For all ground terms s and t, we have $t^{\mathcal{T}_\Sigma} = t$ and $s \approx_{\mathcal{T}_\Sigma} t$ iff $s = t$.* [9]

Thus, in the free algebra, the meaning of a term t in $\mathcal{T}_\Sigma$ is just the term t itself and two terms are equal in the free algebra, if they are syntactically equal (there are no syntactic equations). In other words, terms do not have a deeper meaning, they just are what they syntactically are. (No wonder this resonates so well with computer science...)

In Example 4, we have sketched how a algebra $\mathcal{C}$ for the symbols of Table 1 could look like. This algebra would be a concrete cryptographic interpretation where every message is a bit string and most function symbols are actual cryptographic algorithms. We will now go for the other extreme and interpret messages in the most abstract way – in the free algebra.

This has some interesting consequences. In $\mathcal{C}$, we may for instance interpret a function $h(\cdot)$ by a cryptographic hash function and there will be collisions, i.e., different messages n and m such

[9] "*iff*" abbreviates "if and only if"

that $h(n) \approx_{\mathcal{C}} h(m)$. In contrast, in the free algebra it holds that $h(n) \approx_{\mathcal{T}_\Sigma} h(m)$ iff $h(n) = h(m)$ iff $n = m$. So the free algebra models hash functions as *collision free* – which is actually absurd in reality, since no function $h : A \to B$ with infinite A and finite B can be injective.

To see that it makes sense to interpret terms in the free algebra anyway, consider that the example of the collision is a (necessary) imperfection of the concrete hash function used in $\mathcal{C}$, and a different concrete hash function may have collisions for different pairs of inputs. The free algebra is thus abstracting from the (undesirable) properties of the concrete implementation, and we will thus arrive at an intruder who can only perform attacks that work in *any* implementation of the cryptographic functions.

One could thus say that we model an intruder who is oblivious to the actual cryptography, and will not attempt any attacks on the crypto-level. Thus, the security results that we formally prove in such a model by default tell us only about the security against an intruder who does not perform crypto-analytic attacks. We come back to this issue when we define the intruder deduction relation below, i.e., the intruder's ability to analyze terms.

6.4 ⋆ Quotient Algebra

A section title in blue and marked with a ⋆ signals an advanced topic for the readers who would like to know more. We always give a short *executive summary paragraph* that suffices for following the rest of the tutorial.

Summary 1. *For some advanced protocols and primitives we simply need a few algebraic properties in the model. We then define a set of equations E (like $s + t = t + s$) and define an algebra where the equations of E hold but that is "as close as possible" to the free algebra, i.e., two terms are equal iff that is a consequence of E.*

Protocols that use Diffie-Hellman are only executable if the model supports at least the following property:

$$\exp(\exp(g, X), Y) \approx \exp(\exp(g, Y), X)$$

This is because A knows X and $\exp(g, Y)$ and can thus construct the term on the right, while B knows Y and $\exp(g, X)$ and can thus construct the term of the left. That they arrive at the same shared secret thus requires the equality to hold.

Definition 6. *Let E be a set of equations $s \approx t$ where s and t are terms with variables. Let $\approx_E$ be the smallest congruence relation that includes all ground instances of E:*

- *(Equations of E.) If $s \approx t$ is an equation of E, and σ is a substitution that maps every variable of s and t to a ground term, then $\sigma(s) \approx_E \sigma(t)$ holds.*
- *(Reflexivity.) $s \approx_E s$ for every term s.*
- *(Transitivity.) If $s \approx_E t$ and $t \approx_E u$ then also $s \approx_E u$.*
- *(Symmetry.) If $s \approx_E t$ then also $t \approx_E t$.*
- *(Structure.) If $s_1 \approx_E t_1, \ldots s_n \approx_E t_n$ and f has arity n, then $f(s_1, \ldots, s_n) \approx_E f(t_1, \ldots, t_n)$.*

The equivalence class $[t]_E$ *of a term t is the set of equivalent terms in $\approx_E$:*

$$[t]_E = \{s \mid s \approx_E t\} .$$

In general, a relation that is reflexive, transitive, and symmetric is called an *equivalence relation*, and if it satisfies also the (Structure.) condition, it is called a *congruence relation*. We have defined $\approx_E$ to be the *smallest* relation that satisfies the above properties, i.e., $s \not\approx_E t$ unless it follows from the properties that $s \approx_E t$. Therefore $\approx_E$ is "the closest we can get to the free algebra" while still respecting the equations of E.

Example 5. *Let E be just the equation* $\exp(\exp(B,X),Y) \approx \exp(\exp(B,Y),X)$. *Then*

$$\begin{aligned}[exp(exp(exp(a,b),c),d)]_E = \{\ & exp(exp(exp(a,b),c),d))),\\ & exp(exp(exp(a,b),d),c))),\\ & exp(exp(exp(a,c),b),d))),\\ & exp(exp(exp(a,c),d),b))),\\ & exp(exp(exp(a,d),b),c))),\\ & exp(exp(exp(a,d),c),b)))\}\end{aligned}$$

The remaining question is how to define an algebra $\mathcal{A}$ such that $\approx_{\mathcal{A}}$ is the relation $\approx_E$: the free algebra is too abstract (no equations hold), several other algebras are too concrete (too many equations hold). Recall that the free algebra is constructed by using the set of all terms as the universe, and every term is interpreted by itself. The idea is now to interpret every term t by its equivalence class $[t]_E$ – automatically if $s \approx_E t$ we then have $[s]_E = [t]_E$, thus two terms are interpreted as equal iff they are equal according to $\approx_E$. This means that the universe is now a set of sets of terms, namely for every term, the equivalence class of that term is an element of the universe:

Definition 7 (Quotient Algebra)**.** *Given a signature Σ, and a set of equations E, the* quotient algebra $\mathcal{T}_{\Sigma/\approx_E}$ *is defined as follows:*

- *The universe is the set $\{[t]_E \mid t \in \mathcal{T}_\Sigma\}$ of equivalence classes of ground terms.*
- *For every $f/n \in \Sigma$, define $f^{\mathcal{T}_{\Sigma/\approx_E}}([t_1]_E, \ldots, [t_n]_E) = [f(t_1, \ldots, t_n)]_E$.*

The second item of this definition requires some more explanation: it defines how a function symbol f can be applied to n elements of the universe. Since these elements are equivalence classes, we can write them in the form $[t_i]_E$ where the t_i are some element of the equivalence class. The definition does not tell how this t_i should be picked from the equivalence class, e.g., what happens if we choose some other terms $s_1, \ldots, s_n$ with the property that $s_i \approx_E t_i$? The answer is that this does not matter for the result since $f(t_1, \ldots, t_n) \approx_E f(s_1, \ldots, s_n)$ since $\approx_E$ is a congruence relation. (Thus, also $[f(t_1, \ldots, t_n)]_E = [f(s_1, \ldots, s_n)]_E$.) This definition tells us to arbitrarily pick any "representatives" t_i from the given equivalence classes, and this is not ill-defined, since result is independent of that choice.

As a closing remark, most algorithms that work in $\approx_E$ are more complicated than their counterparts for the free algebra. In general $\approx_E$ is not even a decidable relation, i.e., for some E there is no algorithm that can – for every pair of terms s and t as input – correctly answer whether or not $s \approx_E t$.

7 The Dolev-Yao Intruder Model

One of the most cited papers of protocol verification is one by Danny Dolev and Andrew Yao [11] because they suggested a simple but comprehensive intruder model that has become the de-facto standard for modeling an intruder if one does not consider the cryptographic level. The original paper considers only public-key encryption as a cryptographic primitive, but it is common to treat other primitives in the same spirit. Here are the key points:

- Every user has a public/private key pair.
- Every user knows the public key of every other user.
- The intruder is also a user with his own key pair.
- The intruder can decrypt only messages that are "meant" for him, i.e., that are encrypted with his public key.
- The intruder controls the network: he can read messages, block them, divert them to a different recipient, and insert new messages.

It may seem excessive to assume the intruder controls the entire network (e.g., the entire Internet and also the Intranet of a company). However, this expresses simply that we do not rely the network to offer any protection, and we should not if a message may travel over any point that could be insecure. We will later see how to integrate a notion of channels on which the intruder has no, or limited, control.

The insecure network is the classical view of secure communication: Alice wants to send to Bob some messages, they are honest people, but between them is a hostile world that tries to read, manipulate, or even forge messages between them. One may think of a spy novel where Alice is a secret agent operating in a foreign country and Bob is the home base of the secret service. Dolev and Yao make an important change to this classical view: that the participants of a protocol are not necessarily honest people (who stick to the rules, in particular the protocol). One may think for instance of an e-Banking protocol where Alice is a customer and Bob is a bank: it should not be a requirement of this protocol that all customers are honest, some clients may be trying to cheat. Maybe even a bank may be dishonest, e.g., due to dishonest employees trying to manipulate transactions. In fact, several of the most surprising attacks involve a dishonest participant (while the protocol is secure when considering only honest agents and the intruder can only control the network) e.g. [17]. Thus, we recommend to consider by default that every role of the protocol may be played by a dishonest agent. In some cases like the keyserver example from before, we see that the protocol is (trivially) broken if the keyserver is dishonest. We have thus explicitly made the keyserver a trusted party by using a constant s that cannot be instantiated by the intruder, while all other roles of the protocol (where we have variables) can be instantiated by the intruder i. [10]

7.1 Intruder Deduction

At the core of the Dolev-Yao intruder model is the question of the cryptographic abilities. While Dolev and Yao only consider public-key cryptography, the general idea is that the intruder "knows" all cryptographic algorithms – these algorithms should not be considered a secret themselves, but only the secret keys. This is sometimes called *Kerckhoff's principle*. It is particularly important if you have dishonest participants, because they obviously need to know the algorithms. Note that only functions like $\mathsf{inv}(\cdot)$ are exempt from this, because they do not represent a cryptographic operation, but a function in our model that assigns to every public key its corresponding private key.

Another key idea of Dolev-Yao is now: the intruder can use all (public) cryptographic operations – but that is it, there is no other possibility to analyze messages, like cryptanalysis. Thus we model an intruder who has access to a *Crypto API*, i.e., a library of encryption and decryption algorithms, but all he ever does cryptographically are function calls to that library. All the security results we prove in this model thus are against an intruder who is oblivious to cryptography and may no longer hold against an attacker with crypto-analytic abilities.

One could of course instead perform "cryptographic security proofs", i.e., prove that a protocol is secure in an algebra like $\mathcal{C}$ in Example 4. This gets extremely complex as one has to then define bounds on the intruder's resources, consider probabilities (since the intruder may make guesses) and use assumptions of the hardness of certain mathematical problems like prime factorization. If we think of a developer in industry whose goal is to design complex distributed systems then security against such a full-fledged cryptographic model may be infeasible: we should not require that all developers of systems that use cryptography are also cryptographers themselves.

Our suggestion is to clearly distribute "responsibility" and a reasonable distribution can be: the goal of a crypto API should be, roughly speaking, that the intruder cannot derive any message that he cannot obtain from an API call. This is not trivial to formalize and prove, but there are several results that show the soundness of a Dolev-Yao model with respect to a concrete cryptographic implementation, e.g. [5].

[10] One may wonder what happens if there is more than one intruder: in the worst case, they all collaborate, therefore we see that as a special case of one intruder and the dishonest agents are bots under the intruder's control. It is a bit of a simplification that there is only one dishonest agent called i, but as long as no inequalities on agent names are required, this is sound.

We formalize now the cryptographic abilities of the intruder according in the style of Dolev and Yao for our set of operators from Table 1. This is a relation of the form $M \vdash m$ where M is a finite set of messages, called the intruder knowledge, and m is a message that is derivable from that knowledge:

Definition 8. *We define* $\vdash$ *as the* least *relation that satisfies the following rules:*

$$\frac{}{M \vdash m} \text{ if } m \in M \ (Axiom) \qquad \frac{M \vdash m_1 \quad \ldots \quad M \vdash m_n}{M \vdash f(m_1, \ldots, m_n)} \text{ if } f/n \in \Sigma_p \ (Compose)$$

$$\frac{M \vdash \langle m_1, m_2 \rangle}{M \vdash m_i} \ (Proj_i) \qquad \frac{M \vdash \{\!|m|\!\}_k \quad M \vdash k}{M \vdash m} \ (DecSym)$$

$$\frac{M \vdash \{m\}_k \quad M \vdash \mathsf{inv}(k)}{M \vdash m} \ (DecAsym) \qquad \frac{M \vdash \{m\}_{\mathsf{inv}(k)}}{M \vdash m} \ (OpenSig)$$

$$\frac{M \vdash s}{M \vdash t} \text{ if } s \approx_E t \ (Algebra)$$

(*Axiom*) This rule just says that the intruder can derive any term m that is already in his knowledge M.

(*Compose*) This rule says that for any public function symbol f of n arguments he can apply f to any terms $m_1, \ldots, m_n$ that he can already derive.

(*Proj$_i$*) If the intruder knows the pair $\langle m_1, m_2 \rangle$, then he also knows its components m_1 and m_2.

(*DecSym*) If the intruder knows a symmetrically encrypted message $\{\!|m|\!\}_k$ and also knows the key k, then he can derive m.

(*DecAsym*) Similarly for asymmetric encryption, only here he has to know the private key $\mathsf{inv}(k)$.

(*OpenSig*) If the intruder knows a signature $\{m\}_{\mathsf{inv}(k)}$, then he also knows the signed message m.[11]

(*Algebra*) For the case that one wants to analyze a protocol under some algebraic properties E (e.g. for exponentiation), this rule closes the deduction under $\approx_E$.

The fact that the intruder cannot do anything else than these rules is captured by saying that $\vdash$ is the *least* relation satisfying the rules: if $M \vdash t$ does not follow from these rules (by any number of steps), then $M \nvdash t$.

Example 6. *Let* $M = \{\, k_1, \{\!|m_1|\!\}_{k_1}, m_2, \{\!|m_3|\!\}_{k_2} \,\}$. *Then for instance we have:*

- $M \vdash m_1$
- $M \nvdash m_3$
- $M \vdash \{\!|\langle m_1, m_2 \rangle|\!\}_{k_1}$
- $\ldots$

We can actually write derivations as a proof tree *where the leaves of the tree are (Axiom) steps and the root of the tree is the result we want to prove:*

$$\dfrac{\dfrac{}{M \vdash k_1}\ (Axiom) \qquad \dfrac{\dfrac{\dfrac{}{M \vdash \{\!|m_1|\!\}_{k_1}}\ (Axiom) \quad \dfrac{}{M \vdash k_1}\ (Axiom)}{M \vdash m_1}\ (DecSym) \qquad \dfrac{}{M \vdash m_2}\ (Axiom)}{M \vdash \langle m_1, m_2 \rangle}\ (Compose)}{M \vdash \{\!|\langle m_1, m_2 \rangle|\!\}_{k_1}}\ (Compose)$$

[11] We assume here a signature scheme where the message m being signed is actually included in clear-text and the actual signature is applied only to a hash of that message. To obtain m, the intruder does not need to know the public key, but only for verifying signatures. Verifying signatures however is not a message deduction problem (he does not learn a new message from that).

Example 7. *As an example for reasoning with algebraic properties consider again the property* $\exp(\exp(B,X),Y) \approx \exp(\exp(B,Y),X)$.

Let the intruder knowledge be $M = \{\ x, \{|b, \exp(g,y)|\}_k, k, m\ \}$. *Observe that* x, y *are constants (lower-case letters) i.e., concrete exponents that the intruder and another participant have randomly chosen.*

We show that the intruder can derive from M *for instance the message* $\{|m|\}_{\exp(\exp(g,x),y)}$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{M \vdash \{|b,\exp(g,y)|\}_k}\,(Axiom) \quad \dfrac{}{M \vdash k}\,(Axiom)}{M \vdash \langle b, \exp(g,y)\rangle}\,(DecSym)}{M \vdash \exp(g,y)}\,(Proj_2) \qquad \dfrac{}{M \vdash x}\,(Axiom)}{M \vdash \exp(\exp(g,y),x)}\,(Compose)}{M \vdash \exp(\exp(g,x),y)}\,(Algebra) \qquad \dfrac{}{M \vdash m}\,(Axiom)}{M \vdash \{|m|\}_{\exp(\exp(g,x),y)}}\,(Compose)$$

7.2 Automating Dolev-Yao

Let us quickly think about implementing an algorithm that, given M and m as input should tell us whether $M \vdash m$, and consider the free algebra (i.e., $E = \emptyset$, i.e., we can omit the ($xAlgebra$) rule). A problem is that there are infinitely many things the intruder can do, and thus there are infinitely many proofs. How can we limit the search for a proof of $M \vdash m$, so that the algorithm does not run into an infinite loop?

The idea is to solve first an easier problem: let $M \vdash_c m$ denote derivations that only use ($Axiom$) and ($Composition$), i.e., an intruder who does not analyze any terms. This can be solved by a simple backwards search: if the goal is to derive $t = f(t_1, \ldots, t_n)$, then check if t is contained in M, and if not, and if $f \in \Sigma_p$, try to recursively derive every t_i. Otherwise the answer is no. This algorithm terminates since in every recursive call, the terms get smaller.

Next we solve the problem to *analyze* the intruder knowledge: if the intruder knows a message of the form $\{|m|\}_k$, use the above algorithm for checking $M \vdash_c k$, i.e., whether the key can be obtained by composition steps only. If so, add m to the intruder knowledge M. This modification of M does not change the set of terms that the intruder can derive from M via $\vdash$, the intruder is just explicitly remembering what he can derive (in this case using ($DecSym$)). Repeat this for all of the decomposition rules ($DecSym$), ($DecAsym$), ($Proj_i$), and ($OpenSig$) until no new terms can be obtained. This algorithm terminates because it can only add sub-terms of the original intruder knowledge M (and since M is finite, there are only finitely many subterms).

Example 8. *The complete analysis of the intruder knowledge* M *in Example 7, would add the message* b *and* $\exp(g,y)$.

Now that the intruder knowledge is completely analyzed, for any term m to derive it suffices to use only composition steps:

Theorem 1 ([1])**.** *For an analyzed knowledge* M*,* $M \vdash m$ *iff* $M \vdash_c m$.

This algorithm can also be extended to handle many equational theories E (e.g. the exponentiation example), but in general $\approx_E$ is undecidable, and thus, so is $\vdash$.

8 Transition Systems

We can now put it all together and define a world of honest agents, an intruder, and an intruder-controlled network. We proceed as follows:

- We start with an AnB description of a protocol. As explained in Section 5, we can first see such a description as a message sequence chart, and then split it into several strands, one for each role of the protocol.

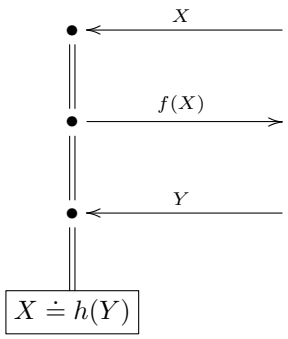
- We have pointed out also in Section 5 there are some cases where this does not yet give an accurate description of the protocol execution, but let us postpone this problem at first and come back to it in section 10.
- We define how to instantiate the protocol for concrete sessions of the protocol, yielding an infinite set of "closed" strands.
- We then define a state transition system where both the honest agents (represented by the strands) and the intruder can make transitions.

Let us start by giving a formal definition of strands:

Definition 9. *A* strand *is a sequence of steps where each step is either*

- $\mathsf{Snd}(t)$ *for sending a message* t*.*
- $\mathsf{Rcv}(t)$ *for receiving a message* t*.*
- $\mathsf{s} \doteq \mathsf{t}$ *for checking whether two terms* s *and* t *are equal. (We write* $\doteq$ *to indicate that this is an operation performed during protocol execution.)*
- $\mathsf{Evt}(t)$ *generates a special event* t *(that the intruder cannot see and that we use only for formulating the goals).*

For a strand of zero steps we write 0*. The graphical notation of strands is straightforward, e.g. the strand* $\mathsf{Rcv}(X).\mathsf{Snd}(f(X)).\mathsf{Rcv}(Y).X \doteq h(Y)$ *would be represented as*

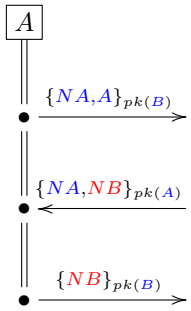


Let $S = S_1.\mathsf{Rcv}(t).S_2$ *be a strand, and let* X *be a variable that occurs in* t *but not in* S_1*. Then we say* X *is* bound *in* S *by the receive step* $\mathsf{Rcv}(t)$*. We say that all variables that are not bound by such a receive step are* free variables *of* S*. We say that a strand is* closed *if it does not have free variables.*

Example 9. *Consider the following AnB specification, called Needham-Schroeder Public-Key protocol:*

```
Protocol: NSPK
Types: Agent A,B;
       Number NA,NB;
       Function pk
Knowledge: A: A,pk,inv(pk(A)),B;
           B: B,pk,inv(pk(B))
Actions:
A->B: {NA,A}(pk(B))
B->A: {NA,NB,B}(pk(A))
A->B: {NB}(pk(B))
Goals:...
```

Extracting the strand for role A yields (when there is no danger of confusion, we sometimes write a pair $\langle s,t\rangle$ of messages simply as "s,t" without the angle brackets):



The variable NB is bound by a receive step, while the blue variables are free.

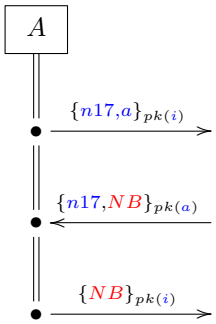
8.1 ⋆ Instantiation

Summary 2. *The roles that come out of AnB have free variables that represent either agent names (and can be instantiated with any concrete agent name, including the intruder* i*), and freshly generated constants (that are instantiated with constants that are not used anywhere else). This gives an (in general infinite) set S_0 of closed strands. Also this induces an initial intruder knowledge M_0 (derived from the knowledge specification in AnB).*

We consider the free variables of a role as its *parameters*, i.e., when we see a role as a program, the parameters could be inputs to the program. In fact, recall that in the syntax of AnB we had required that

- only variables of type *Agent* can occur in the initial knowledge of any agent – and they can be instantiated with any concrete agent name; and
- all other variables (that do not occur in the initial knowledge) are freshly created by the agent who first uses them.

Example 10. *Continuing Example 9, the variables A and B are agent names and shall be instantiated with agent names, e.g. $\sigma(A) = a, \sigma(B) = i$. The variable NA is freshly created by A since it is not in the initial knowledge and first occurs in a message sent by A. Thus we can instantiate it with a constant that is unique to this strand – to model a random number generator, e.g., $\sigma(NA) = n17$. Under substitution σ we obtain the following closed strand:*



The precise semantics of AnB includes an executability check that can only succeed if all free variables are either agent names of the initial knowledge of that role or freshly created by that role. For instance in the example, if we remove NB from the receive step, but leave it in the following send step, then this protocol is unexecutable (since NB is created by B and thus a priori not known to A.

Thus, the instantiation of agent names and fresh constants will always yield *closed* strands. We can thus create an arbitrary large supply of protocol *sessions*:

Definition 10. *Let $R_1, \ldots, R_k$ be the roles of a protocol (as strands). For each role R_i let I_{R_i} be a set of substitutions that instantiate the free variables of R_i with agent names and freshly generated constants. Further let K_i be the knowledge of role R_i and A_i be the agent name of role R_i. We define the* initial set *of closed strands as:*

$$S_0 = \bigcup_{i=1}^{k} \{\sigma(R_i) \mid \sigma \in I_{R_i} \wedge \sigma(A_i) \neq \mathsf{i}\}$$

and the initial intruder knowledge as

$$M_0 = \bigcup_{i=1}^{k} \{\sigma(K_i) \mid \sigma \in I_{R_i} \wedge \sigma(A_i) = \mathsf{i}\} \ .$$

Thus, for every instantiation σ that does not instantiate the role name with the intruder, we create a closed strand as part of S_0, and otherwise, we give the instantiated knowledge to the intruder.

Example 11. *In the previous example, we have the instance $\sigma = [A \mapsto a, B \mapsto \mathsf{i}, NA \mapsto na_{17}]$ for role A, producing the already depicted closed strand. Let us have a matching instance for role B: $\sigma' = [A \mapsto a, B \mapsto \mathsf{i}, NB \mapsto nb_{18}]$. Since this role B is now played by* i*, this instance does not produce a strand but gives the intruder knowledge $\{b, pk, inv(pk(i))\}$, i.e., the knowledge that was specified for role B under substitution σ'. Thus the intruder is supplied with all messages he needs to know in order to faithfully execute the protocol.*

Note that in general there is no bound on the agent names or on the number of sessions that can execute a protocol. Also between the same set of agents, we can have an unbounded number of sessions.

8.2 States and Transitions

We define now our abstract protocol world by first describing what a state of this world is:

Definition 11 (State)**.** *A state consists of three things:*

- *a set of closed strands representing the honest agents,*
- *a set of ground messages M that the intruder currently knows,*
- *and a set E of events that have occurred so far (for later analyzing the security goals).*

The *initial state* of this world consists of the initial set S_0 of closed strands and the initial intruder knowledge M_0 that were produced by the instantiation. The set E of events is initially empty.

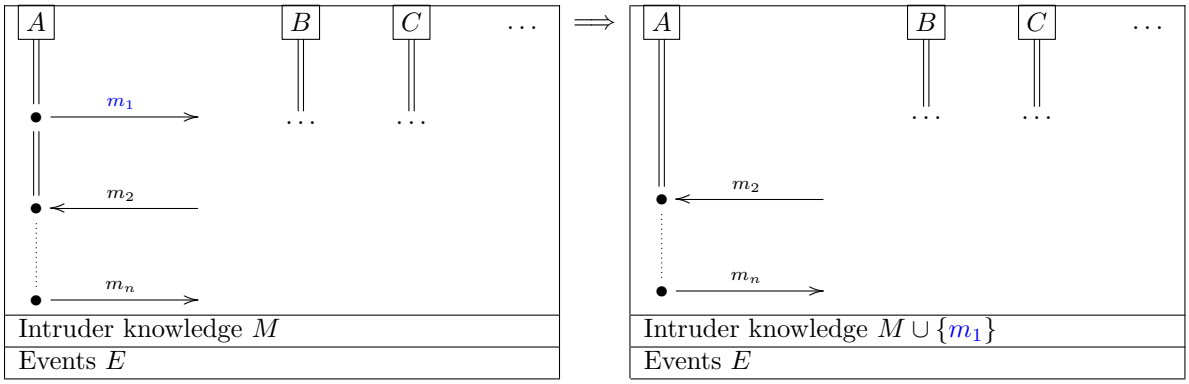
We now define how this world can evolve by giving possible *transitions* from one state to another caused by actions of honest agents and the intruder. In every state there can be several possible transitions, yielding different possible successor states of the world. We are then interested in all states that our world can reach, the *reachable* states. Later we define the security goals by declaring which reachable states we consider an *attack state* and then security is defined by: no attack state is reachable.

We now define the possible transitions, based on the current state and a possible next state:

8.2.1 Transition: Sending

If the current state contains an honest agent whose next step is to send a message m_1, then a possible next state is obtained by removing the message m_1 and adding it to the intruder

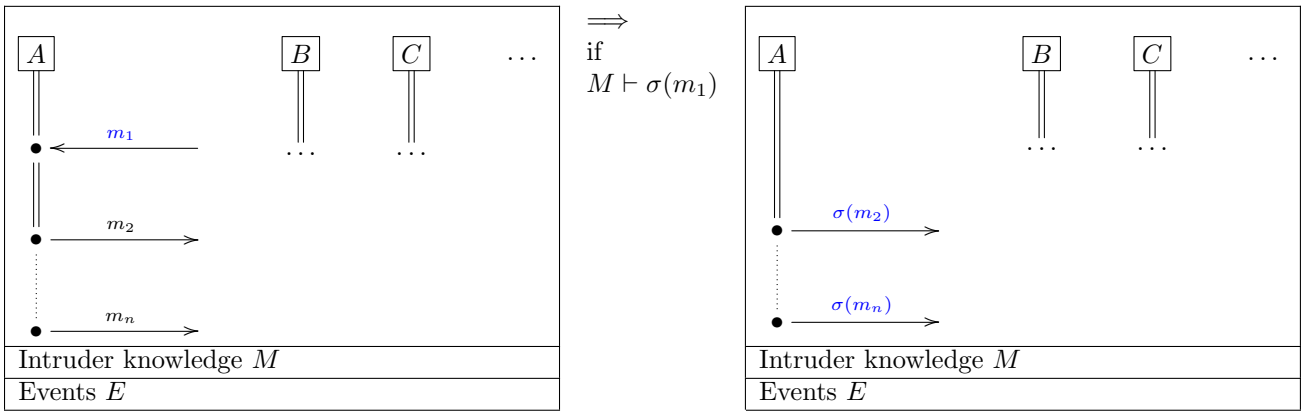
knowledge:



Since the intruder controls the entire network, in this model all messages are simply "received" by the intruder – observe that the strand notation does not even write who the intended recipient is (it does not matter anyway in the presence of this intruder).

8.2.2 Transition: Receiving

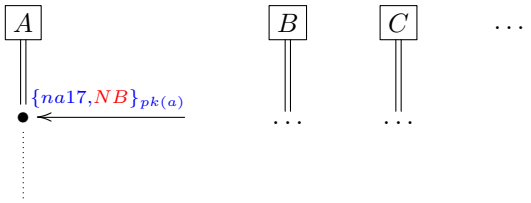
The next kind of transition is for a state that contains an honest agent whose next step is to receive a message m_1. The situation is more complicated, because m_1 may contain variables that are bound by this receive step. The question is whether the intruder can generate any instance of m_1 from his current knowledge M, i.e., whether $M \vdash \sigma(m_1)$ for some substitution σ. Then the following transition is possible:



The substitution σ is applied to the rest of the strand since all variables that have been bound by the receive step are bound to that value for the rest of the strand. Again this reflects the fact that the intruder controls the network: all messages that an honest agent receives come from the intruder. Observe that the strands do not carry any information about who the claimed sender is, since the intruder could anyway insert any name he knows.

Observe that with this strong intruder all communications are subsumed, e.g., that a message from honest a to be received by honest b: by the send event of a, the intruder learns (and intercepts) the message. If he pleases, he sends it to b in the next step, so the possibility is in the model, but he could choose to send a different (modified) message to b (that matches what b is waiting for) and so on.

Example 12. *Consider the state*



with intruder knowledge $M = \{a, b, i, pk(a), na5, na17, \{na17, nb3\}_{pk(a)}\}$.

Then there are infinitely many substitutions σ *under which the receive step can work, i.e., so that* $M \vdash \sigma(\{na17, NB\}_{pk(a)})$*:*

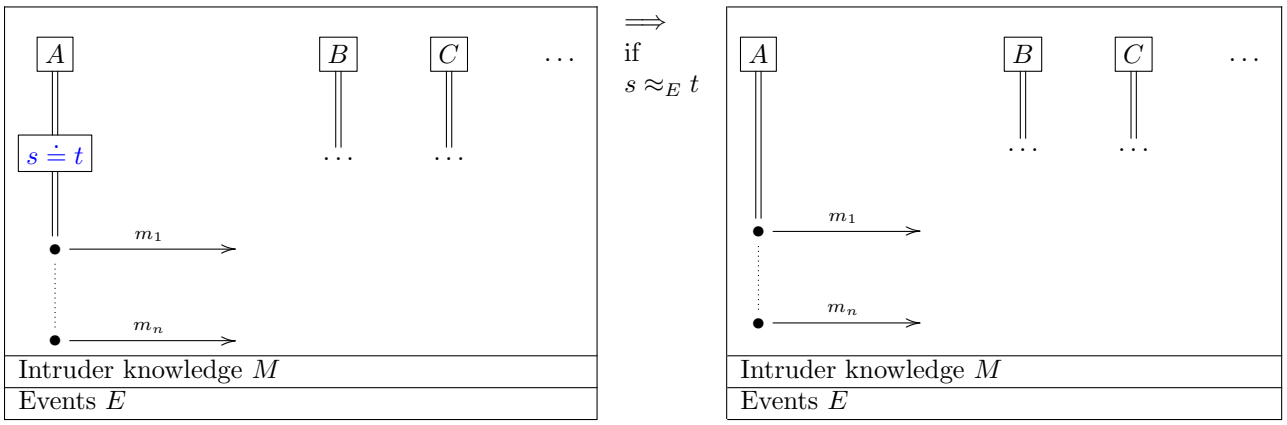
$\sigma(NB) =$

- $nb3$ *(using the encrypted message)*
- $na5$ *or* $nb17$ *(construct himself)*
- $a, b, i, pk(a), \{na17, nb3\}_{pk(a)}, \ldots$, *i.e. "ill-typed" messages: the intruder can actually use any message he can construct.*

This demonstrates that a single honest strand can induce an infinite set of reachable states, since the intruder has an infinite choice of terms to construct.

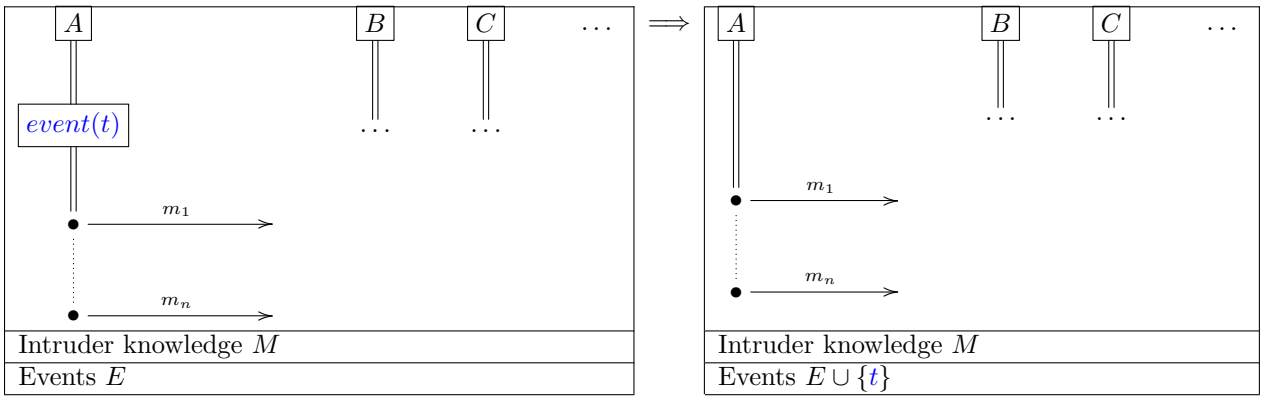
8.2.3 Transition: Checking

The next kind of transition is pretty easy: when the next step of an agent is an equality $s \doteq t$ then it can proceed only if $s \approx_E t$ (i.e., $s = t$ in the free algebra); otherwise this strand is simply stuck:

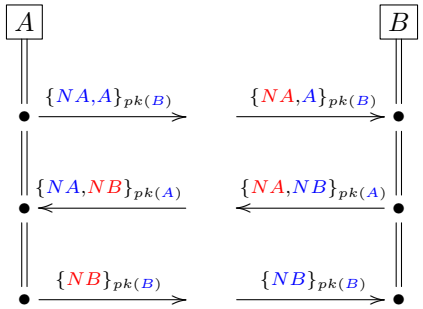


8.2.4 Transition: Events

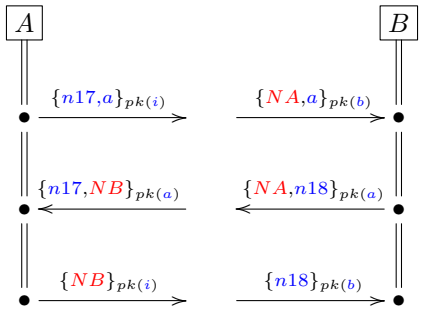
Finally, if an agent creates an event, this is simply added to the set of events:



Example 13. *Consider again the NSPK protocol from example 9 (where we have marked the free variables blue and the bound variables red):*



Let us consider for each role only one instance with an honest player: $\sigma_A = [A \mapsto a, B \mapsto i, NA \mapsto n17]$, *so agent* a *would like to talk to (the dishonest) agent* i. *Remember this is because our model does not rely on all parties to be necessarily honest. For role* B *we consider instance* $\sigma_B = [B \mapsto b, A \mapsto a, NB \mapsto n18]$*:*

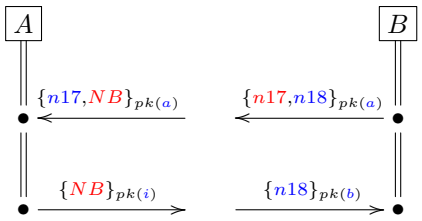


For the initial intruder knowledge we have $M_0 = \{a, b, i, pk(a), pk(b), pk(i), \mathsf{inv}(pk(i))\}$, *i.e., he knows the agents, all public keys and his own private key.*

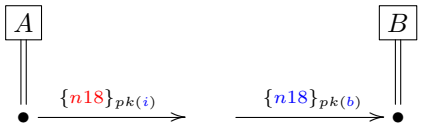
Let us look at just one trace, *i.e., sequence of transitions, that this world can take:*

- *First* a *sends out her first message* $\{n17, a\}_{pk(i)}$ *that is added to the intruder knowledge* M. *Note that* $M \vdash n17$ *since the message is encrypted with the public key of the intruder.*
- *The intruder has now many choices, he could reply to* a, *basically making a normal run with her, or he could try to talk to* b *and impersonate* a. *Also for that there are many choices,*

since he can take any term he knows to be NA and compose a message that b is waiting for. He could use the value $n17$ that he just learned, giving this state:



- *Now b can send out his reply $\{n17, n18\}_{pk(a)}$ which is again added to the intruder knowledge. Note that the intruder cannot decrypt this message, and thus he cannot learn the secret $n18$ yet. But this message matches what a is waiting for:*
- *Let the intruder simply forward the message he learned from b to a we have the state:*



- *When a sends out her next message, $\{n18\}_{pk(i)}$, the intruder learns the secret n18.*
- *The intruder is now also able to complete the run with b, because he can produce $\{n18\}_{pk(b)}$.*

In fact, with the standard definition of secrecy and authentication goals – that we more formally discuss next – it is not *a secrecy problem that the intruder learned n17, because that was a secret meant for i, but it is a violation of secrecy that he learned n18 since it was meant for a. So at the next to last step we had already reached an attack state. That a finished her protocol run does* not *violate any authentication goals, since a did indeed talk with i like she intended, but that b finished his run is a violation of authentication since b believed to be talking with a while a has never intended to talk to b.*

> This protocol is the "canonical" example why one should consider dishonest agents: it is secure when all participants are honest. This is maybe a reason that after NSPK was published in 1978 it took 18 years until this attack was discovered by Lowe [17].

9 Security Goals

We now define formally secrecy and authentication goals for protocols. There are of course many other interesting goals, such as sender invariance, anonymity, privacy, non-repudiation, and availability. Some of these we will actually discuss later, but they require additional measures and infrastructure, so for now we only focus on the basic goals.

9.1 Secrecy

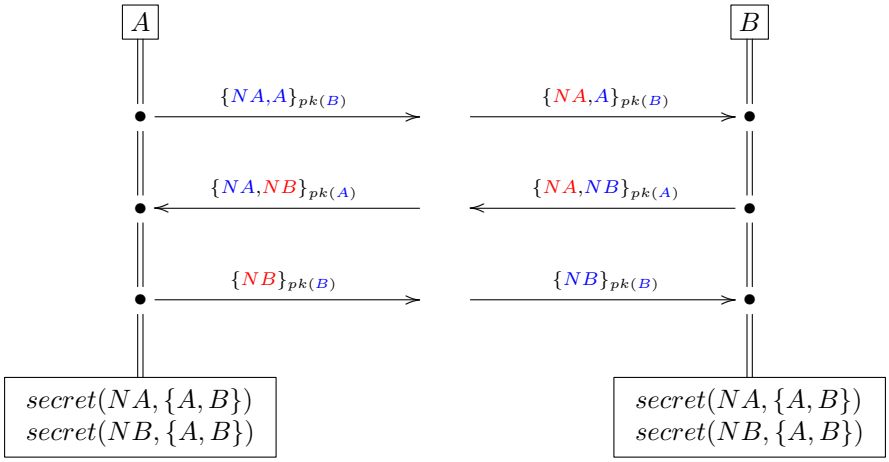
For a protocol like the NSPK example above, one could formulate the following secrecy goals in AnB:

```
NA secret between A,B
NB secret between A,B
```

It is thus important to specify *what* should be a secret, and *who* is cleared to know it. The "who" is important at least if we have dishonest participants, because it is not a violation of secrecy if a dishonest participant learns a secret that is meant for him (like $n17$ in the NSPK attack).

A convenient way to define goals is to insert special events into the protocol execution that indicate what an agent "thinks". For secrecy we use an event $secret(m, s)$ where m is the secret message and s is a set of agents that may know it. This event is inserted at the end of every role that is a participant of the secret. It is inserted at the *end* because some protocols may have a "candidate secret" that only becomes a secret when the party completed some steps. It is inserted in every role that participates to the secret, because secrecy should also be violated if only one of the parties completed the protocol run.

Example 14. *For NSPK with the given secrecy goals we have:*



Now we define an attack on secrecy as any state where the intruder finds out a secret he is not supposed to:

Definition 12 (Secrecy)**.** *A state with intruder knowledge M violates secrecy, if it contains an event $secret(m, s)$ and $M \vdash s$ and $\mathsf{i} \notin s$.*

9.2 Authentication

Authentication, also called *agreement*, mainly means that communication partners agree on who they are talking to. It however also entails a property sometimes called *integrity*: that the intruder did not manipulate the exchanged information. This is part of authentication by simply requiring that the agents also agree on the information. Finally, authentication can also entail freshness: that the intruder cannot make one agent agree to something that truly happened, but a long time ago.

For the NSPK protocol we could have the following two goals:

```
B weakly authenticates A on NA
A weakly authenticates B on NB
```

Here "weakly" denotes the weaker variant of authentication – we come to the strong variant below. Authentication goals have a direction as already explained in Section 4: one could see the protocol as authentically (and secretly, but that is another goal) transmit NA from A to B and NB from B to A. We thus need to talk about the intention of one party to send information, and talk about a party apparently receiving information. Like for secrecy goals we define special events for this.

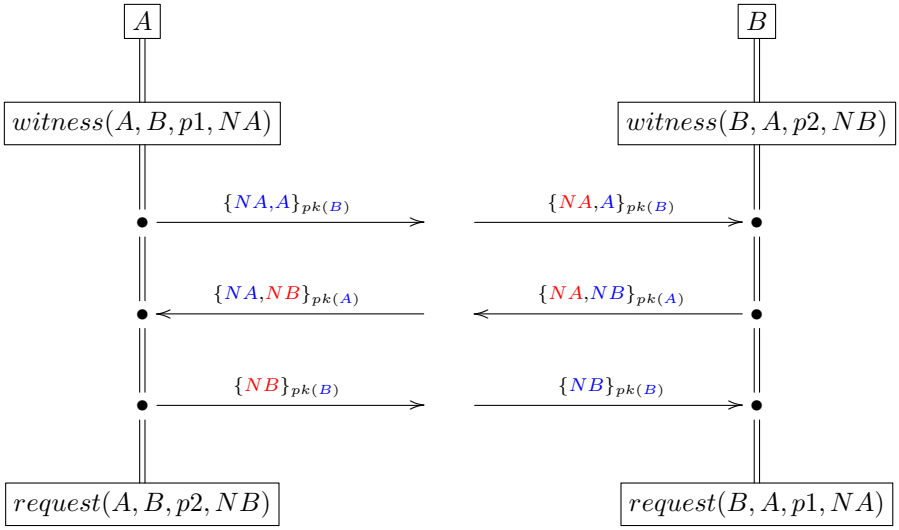
For assign to every authentication goal a unique identifier, so that events of different authentication goals are not confused. Then for any authentication goal $B(weakly)authenticatesAonM$ with identifier p:

- we insert the event $witness(A, B, p, M)$ into role A at the earliest point where A can compose M.

- we insert the event $request(B, A, p, M, ID)$ at the end of role B. Here ID is a new free variable, representing another fresh identifier to be created; this is for the strong authentication later.

Intuitively $witness(A, B, p, M)$ means that A intends to run the protocol with B and agree on message M; and $request(B, A, p, M, ID)$ is the counter-part meaning that B now believes to have been talking to A and has received message M in this session.

Example 15. *For NSPK with the given authentication goals we have:*



Now we define authentication as a state where a request without corresponding witness has occurred, i.e., one party accepts a communication that did not happen that way:

Definition 13 (Weak Authentication)**.** *An attack on weak authentication (aka* non-injective agreement *[18]) is any state where*

- *the event* $request(B, A, p, M, ID)$ *has occurred for some* $A \neq i$ *and*
- *the event* $witness(A, B, p, M)$ *has not occurred.*

We have the side condition $A \neq i$ because in a session with a dishonest participant the agreement is pointless. We could also require $B \neq i$, but that does not change anything since the intruder cannot issue any events, so an event $request(i, \ldots)$ cannot occur by construction.

We can see how different aspects are subsumed by this goal: if B believes that A has sent M it is an attack

- (Wrong Sender) if M comes from somebody else than A;
- (Wrong Receiver) if M comes from A, but was meant for C; or
- (Wrong Content) if A actually meant to talk to B, but said something different than M.

Finally, we add the aspect of freshness to achieve the (strong) authentication goal. For this we assume however that the message M that we authenticate upon contains something fresh (like NA and NB in the NSPK example are created freshly):

Definition 14 (Authentication)**.** *A state violates strong authentication (aka* injective agreement *[18]) if it violates weak authentication or:*

- *two distinct events* $request(B, A, p, M, ID)$ *and* $request(B, A, p, M, ID')$ *with* $ID \neq ID'$ *and* $A \neq i$.

Thus, we use the freshly created identifier (ID and ID') to distinguish two events that are otherwise equal in every argument. Thus B has accepted two times exactly the same message M to come from A. Since we required that M contains something fresh, no honest A would have sent this M twice, and it is thus a replay by the intruder.

10 ⋆ The Algebraic AnB Semantics

Summary 3. *In Section 5, we had remarked that in general we cannot simply split the message sequence chart into roles, because this often will not reflect how the protocol is really executed, i.e., what incoming messages an agent can accept and how outgoing messages are constructed. Here we give the precise definition for this. A key idea is that a normal protocol execution is based on using only the standard cryptographic operations just like the Dolev-Yao intruder. From this follows almost immediately what an agent can check about a received message and how an agent can compose an outgoing message. However, one must distinguish between the knowledge that an honest agent has about an actual message and how a message supposedly looks like according to the AnB specification. Finally, we can define the semantics of AnB by a translation from AnB to a set of strands, one for each role. We can also generate actual protocol implementations from this.*

This section contains several excerpts from [1] where more details and examples can be found.

10.1 Message model

We now define the AnB semantics precisely for any given set Σ of operators, whereof a subset $\Sigma_p \subseteq \Sigma$ is public, and where $\approx$ is an arbitrary congruence relation on terms.

In the previous definition of the Dolev-Yao reduction relation $\vdash$, we have used some analysis rules like (DecSym) that allow the intruder to analyze a term $\{\!|m|\!\}_k$ if he knows k and obtain m. An alternative way to model this is to use an *explicit decryption operator* $\{\!|\cdot|\!\}^{-1}_{\cdot}/2 \in \Sigma_p$ with the algebraic property:

$$\{\!|\{\!|m|\!\}_k|\!\}^{-1}_k \approx m$$

With this property the (DecSym) rule is redundant since for every terms k and m we have:

$$\dfrac{\dfrac{M \vdash k \quad M \vdash \{\!|m|\!\}_k}{M \vdash \{\!|\{\!|m|\!\}_k|\!\}^{-1}_k}\ (Compose)}{M \vdash m}\ (Algebra)$$

In a similar way, we can replace all analysis rules with explicit decryption functions. One can show that (under some reasonable restrictions) this model with explicit decryption functions is equivalent to our free algebra model with analysis rules. The advantage of the free algebra model is that it is algorithmically much easier (no algebraic reasoning) while the algebraic model has the advantage that we can handle decryption in a uniform way and more easily talk about steps that honest agents have performed.

We now distinguish two kinds of messages:

- (1) the *protocol messages* that appear in an AnB specification and
- (2) *recipes* that are the messages in the strands the semantics translates to.

It is necessary to make this distinction as the AnB specification reflects the ideal protocol run, while the semantics reflects the actual actions and checks that an honest agent performs in the run of the protocol. For the same reason, we will also distinguish between two kinds of variables:

- *protocol variables* that appear in the AnB specification like A, B, NA and the like and
- *label variables* that we introduce now: $\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \ldots$.

Intuitively, the label variables represent "memory locations" of an honest agent: given the knowledge K_R of role R in an AnB specification, we give every message in K_R one unique label:

Definition 15. *A* labeled knowledge M *is a substitution of the form* $M = [\mathcal{X}_1 \mapsto t_1, \ldots, \mathcal{X}_n \mapsto t_n]$ *where the* $\mathcal{X}_i$ *are label variables and the* t_i *are protocol messages. We then say* $|M| = n$ *is the* size

of M. The set $\mathcal{T}_{\Sigma_p}(\{X_1, \ldots, \mathcal{X}_n\})$ of terms built from the label variables of M and public operators is called the set of recipes *over M. (This is intuitively the set of all terms that an agent with knowledge M could build.) For a recipe t over M, we say the* corresponding protocol message *is $M(t)$, i.e., applying the substitution M to the recipe, replacing its label variables with protocol terms.*

Example 16. *During the execution of a Diffie-Hellman based protocol (see sec 3.2), an agent may reach the following knowledge (where X, Y, K, N are variables from the protocol description in AnB):*

$$M = [\mathcal{X}_1 \mapsto X, \mathcal{X}_2 \mapsto \{\!|\exp(g, Y)|\!\}_K, \mathcal{X}_3 \mapsto K, \mathcal{X}_4 \mapsto N] \ .$$

The agent is supposed to generate as a next step the message

$$\{\!|N|\!\}_{\exp(\exp(g,X),Y)} \ .$$

A recipe for that term is $t = \{\!|\mathcal{X}_4|\!\}_{\exp(\{|\mathcal{X}_2|\!\}^{-1}_{\mathcal{X}_3}, \mathcal{X}_1)}$ since:

$$M(t) = \{\!|N|\!\}_{\exp(\{|\{\!|\exp(g,Y)|\!\}_K|\!\}^{-1}_K, X)} \approx \{\!|N|\!\}_{\exp(\exp(g,Y),X)} \approx \{\!|N|\!\}_{\exp(\exp(g,X),Y)} \ .$$

This already takes care of the question how agents can generate messages. It remains to define what they can check about messages they receive. The idea is simply: does the reached knowledge M allow for any pair of recipes s, t such that $M(s) \approx M(t)$. That means there are two ways to derive a term that should give the same result if the messages are correct, i.e., follow the AnB specification:

Definition 16. *For a knowledge M, we say a formula ϕ is a* complete set of checks (for M) *if it implies every equation $s \doteq t$ where s and t are recipes over M such that $M(s) \approx M(t)$.*

Note that there are trivially always infinitely many equations, since if $s \doteq t$ is one and $f/1 \in \Sigma_p$, then also $f(s) \doteq f(t)$ is one. We are therefore happy with any finite collection ϕ that implies all the others.

Example 17. *Suppose according to the protocol an agent shall first receive $h(N)$, but does not know N, so cannot check the received message at first. In a later step N is revealed. The knowledge is then: $M = [\mathcal{X}_1 \mapsto h(N), \mathcal{X}_2 \mapsto N]$. Then $\phi \equiv \mathcal{X}_1 \doteq h(\mathcal{X}_2)$ is a complete set of checks.*

The entire translation/semantics from AnB to roles is now as follows:

- First split the message sequence chart from AnB into a set of strands, one for each role. We call it the plain strands.
- Label for each strand the initial knowledge M_0 of that role, which is the messages from the knowledge section labeled with $\mathcal{X}_1, \ldots, \mathcal{X}_n$.
- Go step by step through the roles:
  - For sending a protocol message m, find a recipe t over the current knowledge M such that $M(t) \approx m$. If no such recipe exists the protocol is refuted as *unexecutable*. If there is more than one such label, one can choose any: they are all equivalent due to the checks we perform on received messages. The step $\mathsf{Snd}(m)$ is replaced by $\mathsf{Snd}(t)$ for the chosen label.
  - For receiving a protocol message m, extend the current knowledge M by $\mathcal{X}_k \mapsto m$ for a new label variable $\mathcal{X}_k$ (that does not occur in M). Find a complete set ϕ of checks for the augmented M. Replace the receive step $\mathsf{Rcv}(m)$ by $\mathsf{Rcv}(\mathcal{X}_k)$ followed by the equations ϕ.

Note that this now contains explicit decryption functions that our standard model from the previous sections does not support. It is however easy to get rid of them: For instance consider the strand $\mathsf{Rcv}(\mathcal{X}_1).\{\!|s|\!\}^{-1}_{(}, \mathcal{X}_1) \doteq t.S$ for some terms s, t and a rest strand S. Here we can extract the substitution $\sigma = [\mathcal{X}_1 \mapsto \{\!|s|\!\}_{(}, t)]$ (otherwise the decryption would not work), and apply σ to the strand, yielding $\mathsf{Rcv}(\{\!|s|\!\}_{(}, t)).\sigma(S)$. In this way all decryption functions can be turned into pattern matching.

Part II
Automated Verification

11 Introduction

It would be great to have a general verification procedure for computer programs. Such a procedure would receive as input a program P (choose your favorite programming language[12]) and a *specification* what the program should compute. A specification should in some way describe a function from inputs to desired outputs of the program. For instance a program for sorting integers should as input receive a list of integers and as output return a *permutation* of the input list that is in ascending order. In general, a specification give the programmer even more freedom and specify a set S of functions and it is fine if the program computes one of the functions of S. We do not discuss here how a language or logic for describing S could look like, because it will be irrelevant for our point. The question that we want to solve is the following:

Definition 17. *Given a set S of computable functions, then an S-verifier is an algorithm that gets a program P as input and returns* yes *if P computes a function in S, and* no *otherwise.*

The following theorem tells us that we cannot even conceive such an algorithm for any S – with two trivial exceptions.

Theorem 2 (Rice)**.** *Let S be any set of computable functions, except for the emptyset and the set of all computable functions. Then there is no S-verifier.*

It is important to keep this principle limitation in mind, because it appears in similar shape again and again, and one can save a lot of time if one recognizes earlier that the problem one tries to solve is actually undecidable. Many questions of logic and mathemtics fall into this: it would be nice to have procedures to tell us whether a claimed statement is actually true (i.e., prove that it is a theorem) or not (and give a counter-example).

The principle limitation does not mean, however, that one cannot solve practically relevant parts of the problem, in particular

- identifying restrictions under which the problem becomes decidable, i.e., deciding a *fragment* of the problem;
- procedures that on some inputs give the answer *Inconclusive* (instead of yes or no) or do not terminate. In particular
  - Procedures may be focused on finding counter-examples (attacks, in our case) and be inconclusive or non-terminating on correct inputs.
  - Procedures may be focused on proving correctness and use some over-approximation. They may then fail to find a correctness proof or even present a counter-example that could be a *false positive*, i.e., a counter-example that arises from the over-approximation.

11.1 The Sources of Infinity

The reasons for undecidability in verification is that some aspect of the described system is infinite. Infinity does not necessarily lead to undecidability, for instance the set of even integers in infinite but obviously decidable. We therefore would like to first distinguish several aspects of infinity that arise from the security protocol model we have introduced in the first part and see how it relates to decidability. In fact that restriction of some aspects yields decision procedures that are the most successful ones in practice.

We will in general see our security protocol models as *tree* of states:

[12] We just have to require that the language is Turing complete, which holds for all standard programming languages like C/C++, Java, Haskell, ...

- we have an initial state that is the root node, and
- we can make state transitions from one state to another that gives a *child* relation between nodes, i.e., if one can reach from state S in one transition a set S', then S' is a child of S.
- The security goals are described as a check on states, i.e., we should later check if any node of the tree violates this check.

In general this tree has several aspects of infinity:

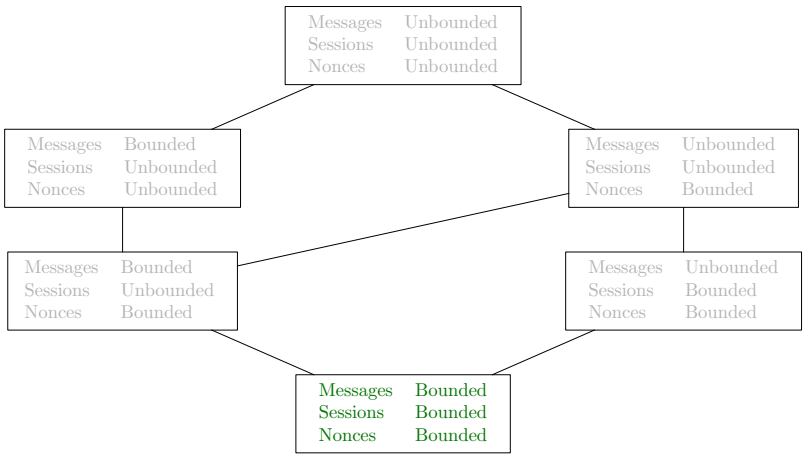
Unbounded Messages Recall the transitions for an honest agent receiving a message m (where m may contain variables). If the current intruder knowledge is M, then for every substitution σ such that $M \vdash \sigma(m)$, we can have a transition. In general this is infinite, i.e., we have a tree that can be *infinitely branching*.

Unbounded Sessions By default, the set of strands for honest agents that we start with is infinite. That is because there we do not want to limit how many people can use the protocol in parallel, and how many sessions between the same people can be open at the same time. An infinite set of sessions (no matter how represented) means both infinite branching of the tree and infinite depth (i.e., infinitely long paths).

Unbounded Nonces Agents can in any session generate fresh nonces, so in an unbounded number of sessions, also the number of nonces may be unbounded. It may seem that this automatically follows from infinite sessions, but we later want to ask what happens if we have unbounded sessions but without fresh nonces.

Algebraic Properties Algebraic properties alone can lead to undecidable problems, we want to limit this here, and just mention that the algorithms we present here can be at least be extended to work with some standard algebraic properties like Diffie-Hellman.

Leaving out algebraic properties and having unbounded nonces only with unbounded sessions, we get a lattice of six possible variants how we could restrict the "infinities" of the problem, from restricting everything to restricting nothing:



Here we have noted the most restricted model already in green, because restricting everything gives a finite tree and then a decision procedure obviously exists.

11.2 ⋆ An Undecidability Result

We show that if protocol security *were* decidable, then we could construct a decision procedure for a classic problem: Post's Correspondence Problem (PCP). Since PCP is already known to be undecidable, so must be protocol security. See also: the updated decidability lattice at the end of this section.

To show the undecidability of a problem, i.e., that it is *impossible* to construct an algorithm that decides that problem, it is often helpful to relate it to other undecidable problems for which undecidability has already been proved. Some people like to use the classical Halting problem of Turing machines, but that can often lead to cumbersome encodings. A much "cleaner" undecidability problem to work with is the correspondence problem proposed by Emil Post:

Definition 18 (Post's Correspondence Problem (PCP))**.** *Consider the following function:*

Input *A finite sequence of pairs of strings* $(s_1, t_1), \ldots, (s_n, t_n)$.

Output Yes *if there is a finite sequence of indices* $i_1, \ldots, i_k \in \{1, \ldots, n\}$ *such that* $s_{i_1} \ldots s_{i_k} = t_{i_1} \ldots t_{i_k}$*;*
and No *otherwise.* □

Example 18. *The correspondence problem*

$$\begin{array}{lll} s_1 = 1 & s_2 = 10 & s_3 = 011 \\ t_1 = 101 & t_2 = 00 & t_3 = 11 \end{array}$$

has a solution: $s_1 s_3 s_2 s_3 = 101110011 = t_1 t_3 t_2 t_3$.

The infinite aspect that makes this problem undecidable is that there is no bound on the length of the sequence of indices: if we have checked that there is no correspondence for any sequence up to length, say, 100, there is no guarantee that there is no sequence at all that has a correspondence.

We now give a reduction: we "encode" PCP into protocols by giving a computable translation f from PCP to protocols, so that the a PCP problem p has a correspondence (i.e. the answer should be *Yes*) iff the protocol $f(p)$ has an attack. It follows, if there *were* any decision procedure for protocol security, then we could build one for PCP using this translation f. Since PCP is undecidable, it follows that protocol security is undecidable.

Definition 19 (Reduction from PCP to Security Protocols inspired by [13])**.** *For the given PCP problem* $((s_1, t_1), \ldots, (s_n, t_n))$ *define the following strands:*

- *An initialization strand:* $\mathsf{Rcv}(\langle\langle X, Xs\rangle, \langle \$, \$\rangle\rangle).\mathsf{Snd}(\{\!|\langle\langle X, Xs\rangle, \langle \$, \$\rangle\rangle|\!\}_k)$. *Here* \$ *is a public constant. This takes (from the intruder) a sequence of indices that cannot be empty (there is at least one element* X*, but any number), and encrypts it with a secret key* k*. Assume also all characters of the strings* s_i *and* t_i *and the indices* $1, \ldots, n$ *are public constants.*

- *For each of the* (s_j, t_j) *(where* $j \in \{1, \ldots, n\}$*), we have an* infinite *number of strands of the form:*[13]

  $$\mathsf{Rcv}(\{\!|\langle\langle j, Xs\rangle, \langle Y_l, Y_r\rangle\rangle|\!\}_k).\mathsf{Snd}(\{\!|\langle Xs, \langle s_j \cdot Y_l, t_j \cdot Y_r\rangle\rangle|\!\}_k)$$

  The notation $s \cdot X$ *is* not *a new operator but a notation for the following: let* $s = c_1 c_2 \ldots c_n$*, then* $s \cdot Y$ *shall denote the term* $\langle c_1, \langle c_2, \langle \ldots, \langle c_n, Y\rangle\rangle\rangle\rangle$*. Intuitively these strands step by step construct the strings that are induces by the indices that the intruder had chosen originally.*

- *Finally the strand:* $\mathsf{Rcv}(\{\!|\langle \$, \langle X, X\rangle\rangle|\!\}_k).\mathsf{Snd}(secret)$*. This means that all indices by the intruder have been transfered into a string (i.e., we have reached the end marker* \$*) and the obtained strings are identical (both* X*) then the intruder has indeed found a solution to the PCP. In this case he gets the secret constant* $secret$*.*

The goal is that the intruder never obtains $secret$*.*

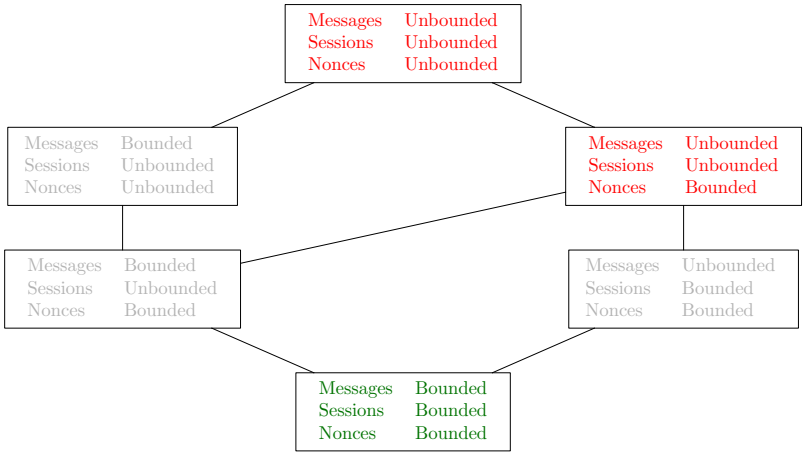
Theorem 3. *The intruder finds out* $secret$ *iff the given PCP has a correspondence. Thus protocol security is undecidable.*

Observe that this reduction relies on two infinities:

[13] We want infinitely many copies of each strands, so that this input-output behavior can be performed any number of times. Since we have a *set* of strands, we achieve this by a renaming of the strands bound variables.

- The intruder must be allowed to compose arbitrary large messages (i.e., sequences of indices).
- There must be an unbounded number of sessions so that the honest agents can check the solution that the intruder has submitted, no matter how large it is.
- There are however no fresh nonces used in the protocol. So even without fresh nonces we have undecidability.

We update the decidability lattice by two marking the proved undecidable settings in :



In the upcoming sections, we will further fill this lattice.

12 Symbolic Transition Systems

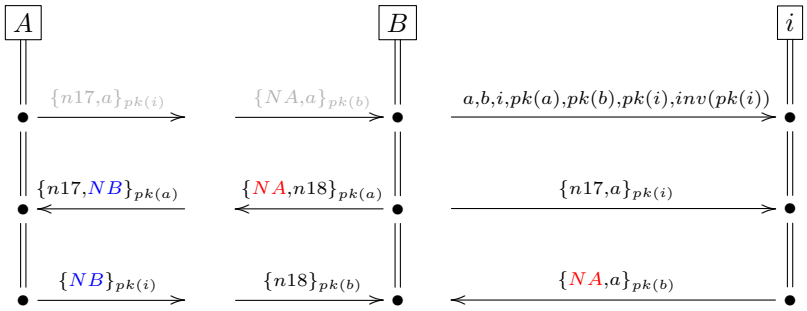
Let us now first bound the number of sessions (and thus of the nonces). Recall that even for a finite number of sessions we can get an infinite-state transition system if we do not bound the messages:

Example 19. *In Example 13, there is an infinite choice of messages that the intruder can send to b: b expects a message of the form* $\{NA, a\}_{pk(b)}$ *and the intruder knows both* $pk(b)$ *and* a*. So he can choose an arbitrary term* t *from his knowledge and construct* $\{t, a\}_{pk(b)}$*.*

This gives at this state infinitely many successor (so, as a tree, an infinite branching degree). Actually, even under some reasonable bounds on the intruder messages this is the point where search for an attack becomes really infeasible. The key idea is now that it is not really productive to list all kinds of terms t that the intruder could use as NA. Let us be *lazy* and *procrastinate* this choice of NA for now.

This means that we get a *symbolic state* that has a free variable NA and we must remember that, whatever NA is, it is something that the intruder can generate from his current knowledge. To store this information what the intruder has generated and at what state of this knowledge, we introduce the concept of an *intruder strand* – the messages the intruder has sent and received so far. We also will call this an (intruder) *constraint*, because we are interested for which values of the variables it can be fulfilled. Let us first look at the NSPK example how that looks like:

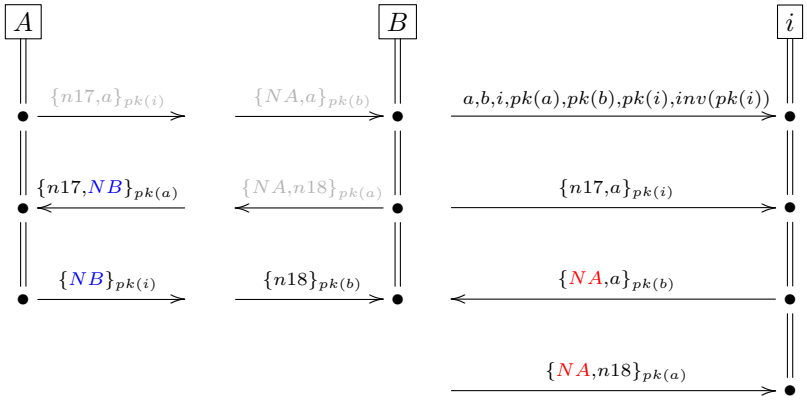
Example 20.



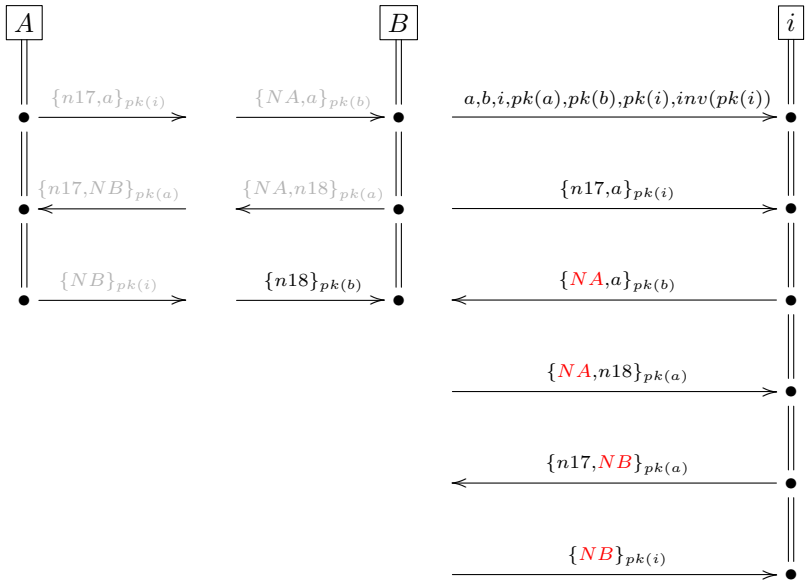
We here have an intruder strand with the following (intuitive) meaning:

- *Incoming messages are messages that the intruder has learned. (In the first line we have the initial intruder knowledge, in the second line we have the message that a has sent to i first.)*
- *Outgoing messages are messages that the intruder has produced. (In the third line we have the message that the intruder has sent to b.)*
- *This should represent all solutions σ of the free variables such that the intruder can generate every outgoing message (under σ) when knowing all previous incoming messages (under σ). We are just lazily procrastinating that choice of σ!*

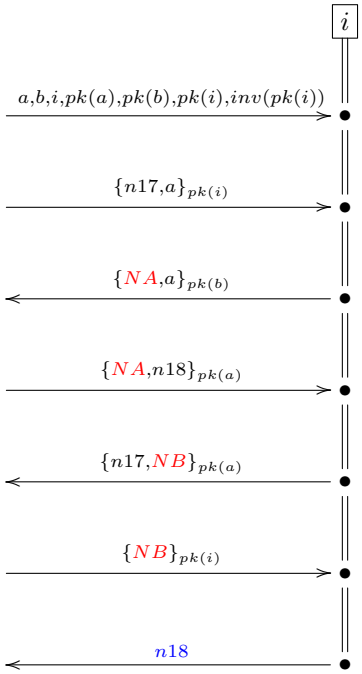
The next step could be that b answers the message; note that b's answer depends on the free variable NA – we have not instantiated in the previous step as of our laziness, so now also honest agents are sending around messages with variables:



Let us say that the intruder next talks to a and gets an answer:



Let us finally consider the secrecy goal and check whether the intruder can produce the secret $n18$ *right now. To that end, we can just add put it as a message he has to derive on the intruder strand. Then secrecy is violated, if we can find a solution for this constraint:*



In section 13 we give a decision procedure for such constraints. In fact there is a solution for this example (and only one): $\sigma(NA) = n17$ *and* $\sigma(NB) = n18$*; that's the secrecy attack from Example 13.* □

Before we have a closer look at the constraints, let us define a symbolic transition and how that induces constraints:

Definition 20 (Symbolic Transition System)**.** *A* symbolic state *consists of*

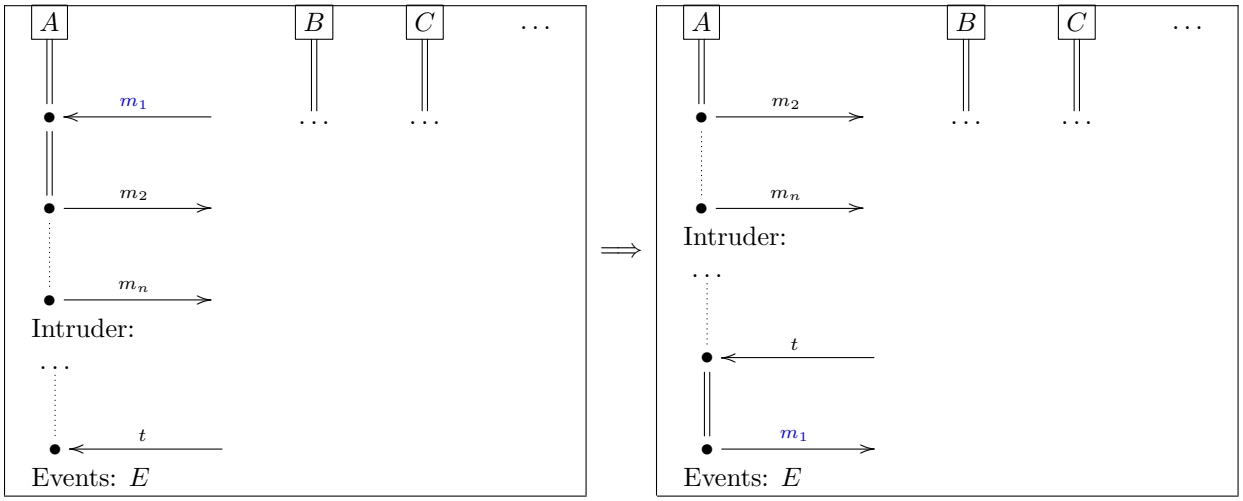
- *a set of honest strands*
- *an intruder strand*
- *a set of events that have occured.*

The initial state *has a set of closed strands for the honest agents, a receive step for the intruder strand containing the initial intruder knowledge, and an empty set of events.*

We have transition rules for honest agents sending and receiving messages, checking equations and emitting events; these are similar to the ones in the original (concrete) transition system, and we define them in the following subsections.

12.1 Transition: Receiving

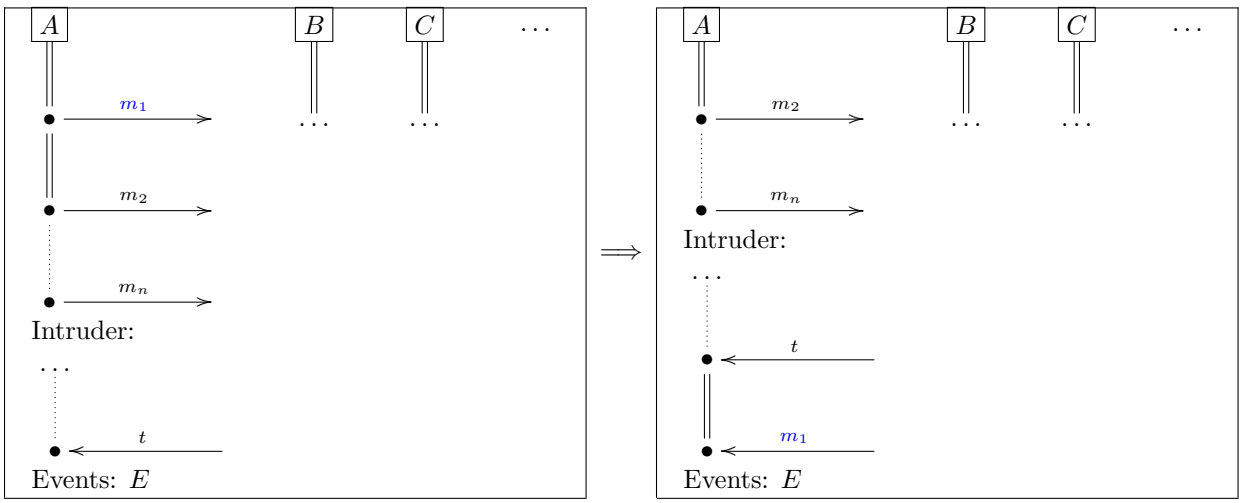
Messages that honest agents are receiving are simply moved (with inverse direction) to the intruder knowledge:



Note that this rule is actually much simpler than the corresponding ground rule, because it does not care for expressing the solutions of the intruder constraint.

12.2 Transition: Sending

Messages that honest agents are sending are also simply moved (with inverse direction) to the intruder knowledge:



12.3 Events and Equations

Events can just be carried over as before. Equations we leave to the reader as an exercise.

13 The Lazy Intruder

We first define the meaning of lazy intruder constraints and then show how to solve them.

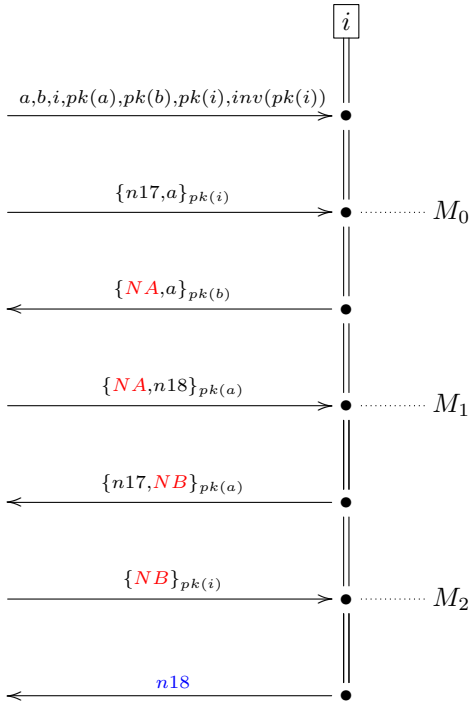
Definition 21. *An intruder constraint is a strand where all variables are free, i.e., all variables first occur in an outgoing message.*

At any point $\bullet$ in the constraint, the intruder knowledge at that point is the set of all messages received so far.

Given an intruder constraint C, and σ a substitution of all its free variables with ground terms. Then σ is called a solution of C iff:

- *for every outgoing message m of C, it holds that $\sigma(M) \vdash \sigma(m)$ where M is the intruder knowledge at that point.*

Example 21. *Consider again the constraint in from the NSPK example. Let us label the intruder knowledge at different points $M_0, \ldots, M_2$:*



The meaning of this constraint is any substitution σ such that the following Dolev-Yao deductions hold:

$$\begin{array}{rcl} M_0 & := & \{a, b, i, pk(a), pk(b), pk(i), \\ & & inv(pk(i)), \{n17, a\}_{pk(i)}\} \\ \sigma(M_0) & \vdash & \sigma(\{NA, a\}_{pk(b)}) \\ M_1 & := & M_0 \cup \{\{NA, n18\}_{pk(a)}\} \\ \sigma(M_1) & \vdash & \sigma(\{n17, NB\}_{pk(a)}) \\ M_2 & := & M_1 \cup \{\{NB\}_{pk(i)}\} \\ \sigma(M_2) & \vdash & \sigma(n18) \end{array}$$

□

13.1 Solving Constraints

We now give a procedure for solving constraints. This will be done with rules of the form:

$$\boxed{\text{S}} \rightsquigarrow \boxed{\text{S'}}$$

The meaning of such a rule is that to solve S, one way is to try to solve S'. Put another way, if we can solve S', then also we can also solve S.

For any given constraint S_0, the relation $\rightsquigarrow$ induces a search tree for solutions of the constraints as follows:

- The root node is the constraint S_0
- Every node S has as children the strands that are reachable in one step with $\rightsquigarrow$:
  - i.e., if $S \rightsquigarrow S'$ then S' is a child of S.

Important feature we prove about this constraint tree are:

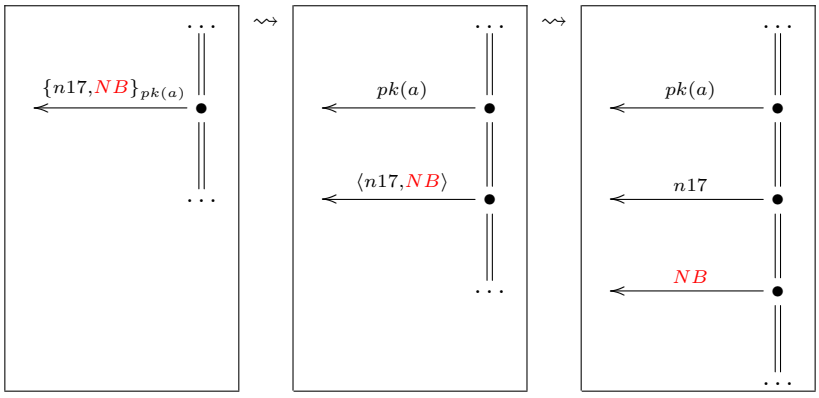
- Termination: the tree has finitely many nodes (we do not run into an infinite search of solutions)
- Soundness: we only find correct solutions.
- Completeness: we do not miss solutions.

This means in particular that every leaf of the tree is either *simple* or unsolvable (Explained below, easy to check). The root has a solution iff any leaf has a solution.

13.2 Composition

Idea: to construct an outgoing message of the form $f(t_1, \ldots, t_n)$ it is sufficient that f is a public symbol and the intruder can construct the submessages t_i:

Example 22.



More general we define the intruder composition rule as follows:

Definition 22 (Lazy Intruder Composition Rule)**.**

$$S_1.\mathsf{Snd}(f(t_1, \ldots, t_n)).S_2 \rightsquigarrow S_1.\mathsf{Snd}(t_1).\ldots.\mathsf{Snd}(t_n).S_2$$

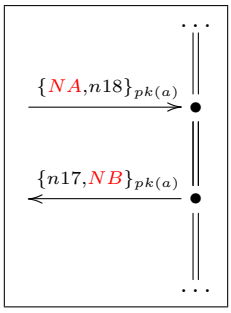
if $f/n \in \Sigma_p$, i.e., f is a public function symbol.

The lazy intruder composition rule corresponds to the (Compose) rule of the Dolev-Yao model: the intruder can compose terms he knows by applying a public function symbol. One may consider the lazy intruder as a backward search: rather than blindly composing terms (that may be useless) in a forward exploration of terms we can generate, we rather start at the terms we want to obtain, and if the toplevel symbol is public, we backwards-apply composition, i.e., try if we can compose the subterms.

13.3 Unification

Idea: another way to construct an outgoing message is to use a previously received message that has the right "shape":

Example 23.



These messages can be unified *under* unifier $\sigma = [NA \mapsto n17, NB \mapsto n18]$. *This unifier has to be applied to the entire intruder constraint.*

13.3.1 Computing the Most General Unifier – mgu

At this point we need a classic algorithm: computing the most general unifier, or mgu for short.

Definition 23 (Unification)**.** *A unification problem is a set* $\{(s_1 \doteq t_1), \ldots, (s_n \doteq t_n)\}$ *of equations of terms. (We use the symbol* $\doteq$ *again to distinguish from the normal equality symbol.)*

A unifier σ *for this unification problem is a substitution such that*

$$\sigma(s_1) = \sigma(t_1) \text{ and } \ldots \text{ and } \sigma(s_n) = \sigma(t_n) \; .$$

In general, there are infinitely many unifiers, for instance the problem $\{x \doteq f(y)\}$ has infinitely many unifiers like $\tau = [x \mapsto f(c), y \mapsto c]$ (if you replace c by any other term, then you get another unifier). However, there is in some sense a *most general* unifier for this problem: $\sigma = [x \mapsto f(y)]$ since all other unifiers are a special case of σ. This notion of generality can be defined as follows:

Definition 24. *We say that a substitution* σ *is* at least as general as *substitution* τ*, and write* $\sigma \preceq \tau$*,* [14] *if there is a substitution* θ *such that* $\theta \circ \sigma = \tau$*.*

Intuitively, this means that we can obtain the more special substitution τ from the more general σ by composing σ with another substitution θ. In the above example $\{x \doteq f(y)\}$, we have with $\theta = [y \mapsto c]$ that $\theta\sigma = \tau$, and thus $\sigma \preceq \tau$. In the free algebra we have the nice property that a unification problem has either no unifier or there is a most general unifier (that is at least as general as any other unifier). There is a fairly simple recursive algorithm that either computes the most general unifier if it exists, or returns *failure* otherwise:

Definition 25 (Algorithm $mgu(U, \sigma)$)**.**
Input: *a unification problem* U*, a substitution* σ *(initially the identity* $[\,]$*).*
Output: *A substitution or answer failure.*
If $U = \emptyset$ *then return* σ*. Otherwise pick any equation* $(s \doteq t)$ *in* U *and*

- *if* $s = t$*, continue to* $mgu(U \setminus \{s \doteq t\}, \sigma)$*.*
- *if* s *is a variable:*
  - *if* $s \in vars(t)$*: return failure*

[14] The direction $\preceq$ may seem counter-intuitive since it calls the more general substitution "smaller": this convention becomes intuitive if you see the sizes of terms as the size of the substitution, and then the smallest (and thus most general) element in this partial order is the identity $[\,]$ that does not substitute anything.

– *otherwise: update* σ *with* $[s \mapsto t]$
and continue to $mgu(\sigma(U \setminus \{s \doteq t\}), \sigma)$

- *if* t *is a variable: symmetric to previous case*
- *otherwise, i.e.,* $s = f(s_1, \ldots, s_n)$ *and* $t = g(t_1, \ldots, t_m)$*:*
  – *if* $f \neq g$*: return failure.*
  – *if* $f = g$ *(and thus* $n = m$*): continue with* $mgu((U \setminus \{s \doteq t\}) \cup \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}, \sigma)$.

We sometimes just write $mgu(s \doteq t)$ *for* $mgu(\{s \doteq t\}, [])$ □

Example 24.

$$\begin{aligned}
& mgu(\{\ \{NA, n18\}_{pk(a)} \doteq \{n17, NB\}_{pk(a)}\ \}, []) \\
= \ & mgu(\{\ pk(a) \doteq pk(a)\ ,\ \langle NA, n18\rangle \doteq \langle n17, NB\rangle\ \}, []) \\
= \ & mgu(\{\ \langle NA, n18\rangle \doteq \langle n17, NB\rangle\ \}, []) \\
= \ & mgu(\{\ NA \doteq n17\ ,\ n18 \doteq NB\ \}, []) \\
= \ & mgu(\{\ n18 \doteq NB\ \}, [NA \mapsto n17]) \\
= \ & mgu(\emptyset, [NA \mapsto n17, NB \mapsto n18]) \\
= \ & [NA \mapsto n17, NB \mapsto n18]
\end{aligned}$$

Theorem 4. *Consider any unification problem* U*. If* $mgu(U) =$ *failure, then* U *has no unifier, otherwise, if* $mgu(U) = \sigma$ *then* σ *is the most general unifier of* U*, i.e., for every unifier* τ *of* U*, we have* $\sigma \preceq \tau$.

13.3.2 The Lazy Intruder Unification Rule

Using the *mgu* function, we can now finally define the lazy intruder unification rule. If the intruder received a term s and needs to send a term t, and s and t are unifiable, and σ is the most general unifier of s and t, then we can apply σ to the entire constraint and consider the sending of t as "done":

Definition 26 (Lazy Intruder Unification Rule)**.**

$$S_1.\mathsf{Rcv}(s).S_2.\mathsf{Snd}(t).S_3 \rightsquigarrow \sigma(S_1.\mathsf{Rcv}(s).S_2.S_3)$$

if s *and* t *are not variables, and* $\sigma = mgu(s \doteq t)$.

> The lazy intruder unification rule corresponds to the (Axiom) rule of the Dolev-Yao intruder. A crucial condition of the unification rule is that neither s nor t can be a variable. This is exactly what makes the intruder lazy: when the term to generate is a variable, we do not bother to select a value for it (at this point).

One may wonder why we do not allow at least the received term s to be a variable. This is because this variable then must have originated in an earlier Send-step (since all variables in an intruder constraint must first occur in a Send-step); if that Send-step is a composed term, we should solve that first; otherwise, i.e., if we have a constraint of the form

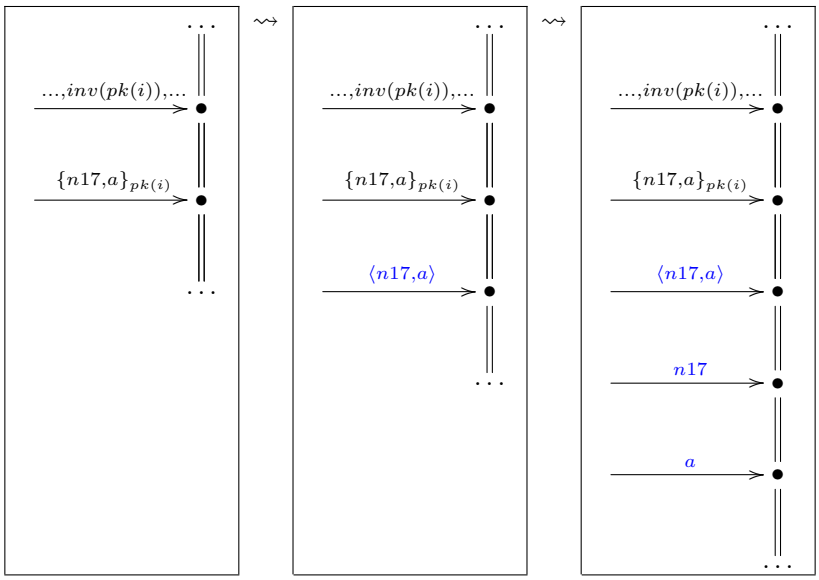
$$\ldots Snd(x) \ldots Rcv(x) \ldots Snd(t)\ .$$

Thus the intruder has sent a message x that he then received back. Thus unifying t with x would not be wrong, but redundant, and go against the philosophy of laziness here. For this reason we also forbid unification steps where the received message is a variable.

13.4 Simple Analysis

Idea: if the intruder received an encrypted message and has the decryption key in his knowledge, then we can also add the decrypted message. (Similar for pairs, he can obtain the components immediately.)

Example 25.



Definition 27 (Lazy Intruder Simple Analysis Rules)**.**

$$
\begin{aligned}
S_1.\mathsf{Rcv}(\mathsf{inv}(k)).S_2.\mathsf{Rcv}(\{m\}_k).S_3 &\rightsquigarrow S_1.\mathsf{Rcv}(\mathsf{inv}(k)).S_2.\mathsf{Rcv}(\{m\}_k).\mathsf{Rcv}(m).S_3\\
S_1.\mathsf{Rcv}(k).S_2.\mathsf{Rcv}(\{|m|\}_k).S_3 &\rightsquigarrow S_1.\mathsf{Rcv}(k).S_2.\mathsf{Rcv}(\{|m|\}_k).\mathsf{Rcv}(m).S_3\\
S_1.\mathsf{Rcv}(\langle m_1, m_2\rangle).S_2 &\rightsquigarrow S_1.\mathsf{Rcv}(\langle m_1, m_2\rangle).\mathsf{Rcv}(m_1).\mathsf{Rcv}(m_2).S_2\\
S_1.\mathsf{Rcv}(\{m\}_{inv(k)}).S_2 &\rightsquigarrow S_1.\mathsf{Rcv}(\{m\}_{invk}).\mathsf{Rcv}(m).S_2
\end{aligned}
$$

Note: this now can lead to non-termination, if one repeatedly applies a rule to the same term. But since this is redundant (not adding new knowledge), we can exclude repeated application to the same term.

13.5 ⋆ Full Analysis

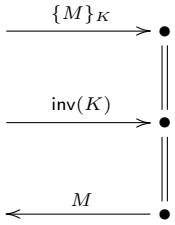
Analysis in the symbolic model is tricky and we can avoid it, if we "out-source" analysis into special strands, representing honest agents that perform analysis steps for the intruder.

The analysis rules given so far are for the simple case that the intruder receives a composed message and already has the decryption key for it in his knowledge. In general, analysis is more difficult for the following reasons:

- The key-term may contain variables, like $\{m\}_{pk(A)}$. Suppose the only private key that the intruder knows is $\mathsf{inv}(pk(a))$. Then we actually get a case split: if $A = i$ then he can decrypt it, otherwise he cannot.
- The key-term may be composed, like $\{|m|\}_{h(n1,n2)}$. If the intruder knows n_1 and n_2 and h is a public function, he can compose the decryption key, but the simple analysis rule would miss this.

- The intruder may receive a decrypted message that he cannot decrypt at first, but learn the decryption key in a later step.

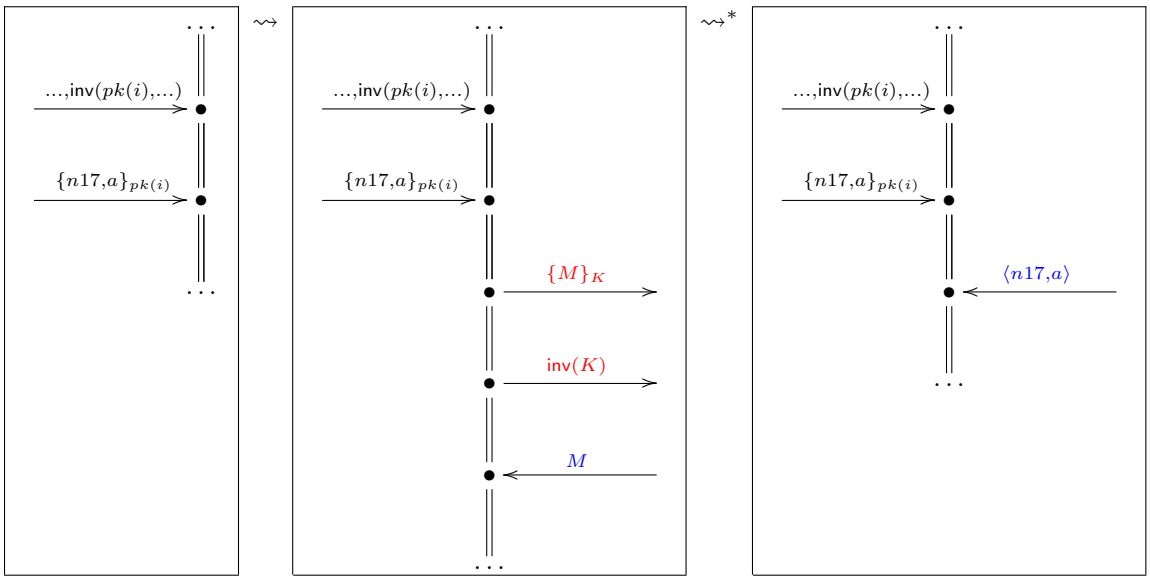
An idea is to "out-source" analysis in the following sense: suppose we add special strands to the protocol like this one:



This means an agent "offering as a service" the same functionality that the analysis rule for asymmetric encryption provides: give me a public-key encrypted message and the private key, then you get the result. Since all agents but the intruder are honest, we do not have to worry that the intruder transmits his private key to this service – nobody in our model will attack the intruder. Also this would not insert any new attacks since the intruder only obtains something that the normal Dolev-Yao model gives him anyway. If we do this for all decryption rules, we can consider an intruder who does not decrypt himself. In fact, we would then obtain a transition system where the intruder for any decomposition would ask one of the special agents.

This corresponds to the intruder to be able to insert an analysis step at any point into the constraints. For instance, we would then have for NSPK:

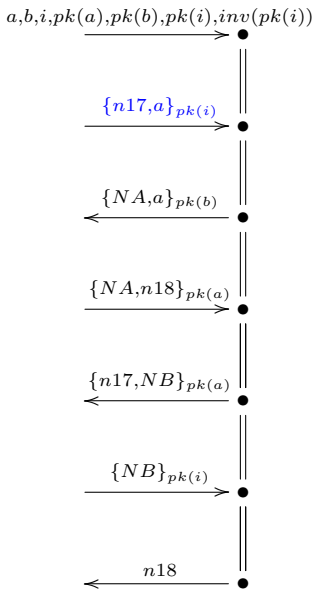
Example 26.



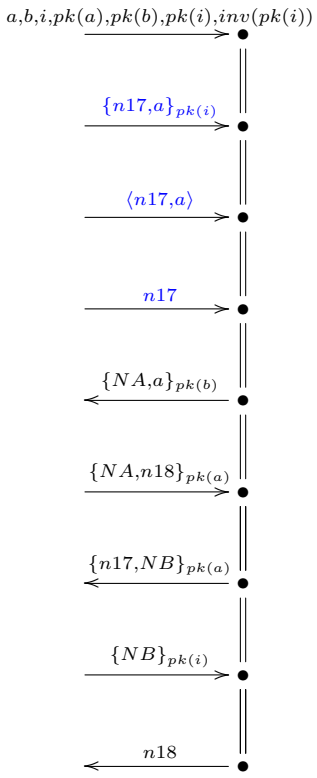
Without care, this can easily lead to non-termination. However, if we are focusing on the problem of a bounded number of sessions (i.e., infinitely many strands without the special analysis strands) then we can also limit the number of analysis strands: for every constructor in the honest strands that permits analysis, we shall have one corresponding analysis strand.

13.6 Constraint Solving Complete Example

We now give the complete example of solving the intruder constraint from Example 21. The initial constraint to solve is the following:

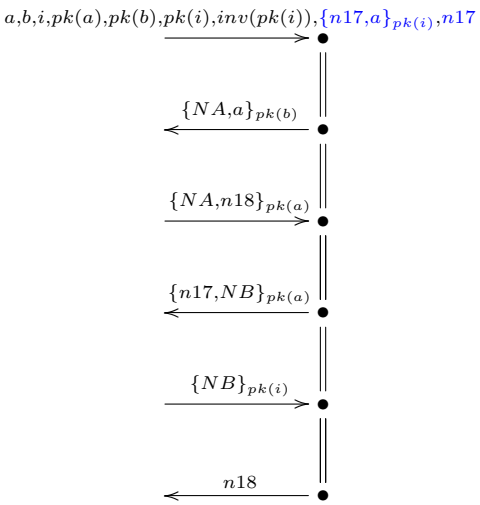


Here we can decrypt $\{n17, a\}_{pk(i)}$ since we know the private key $inv(pk(i))$. We can also can also decompose the resulting pair $\langle n17, a\rangle$. Since a is already known, we only really learn $n17$ from this. With this we obtain the constraint:

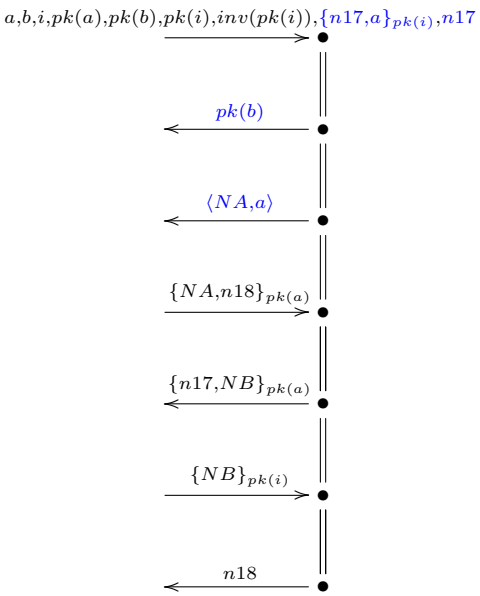


Let us simplify notation to put these messages into the initial intruder knowledge. Also note

that it is not a restriction to remove now the pair $\langle n17, a\rangle$ since we have the components and the intruder can always compose the pair again. (In fact this later reduces a few redundant cases.)

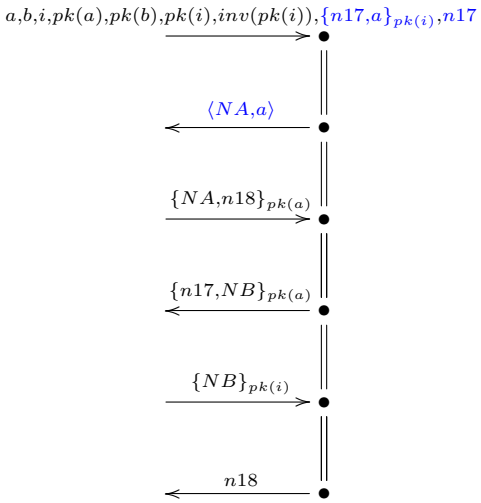


Consider the first outgroing message $\{NA, a\}_{pk(b)}$. For each outgoing message, there are always basically two possibilities: composition or unification. Unification does not work here since the intruder has nothing fitting in his knowledge. So let us go for composition:

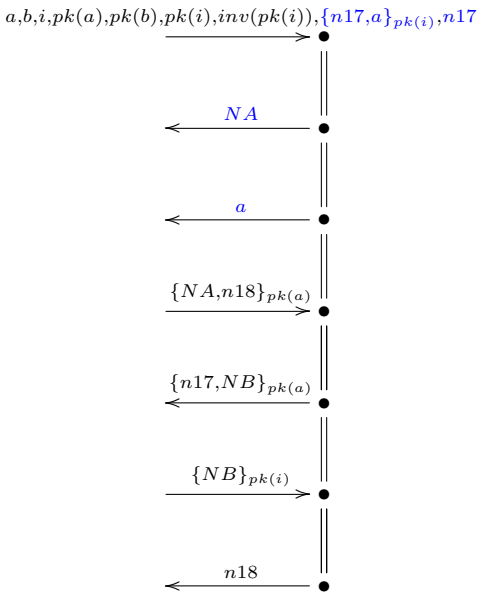


The intruder is now required to produce the key $\mathsf{pk}(b)$ which is directly in the knowledge, so

we can remove it using unification:

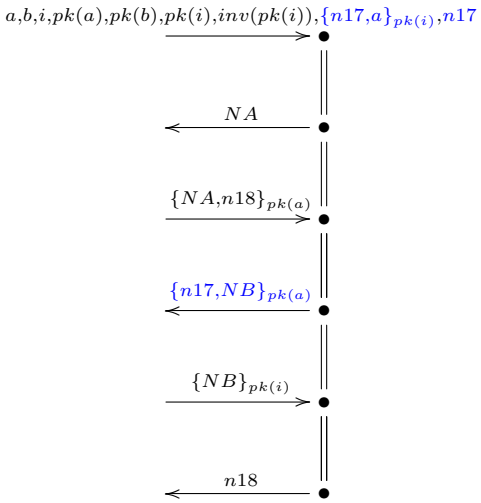


For $\langle NA, a\rangle$ we can only use composition (in fact, had we not removed the incoming pair before, we would have to deal with it here as an extra case):

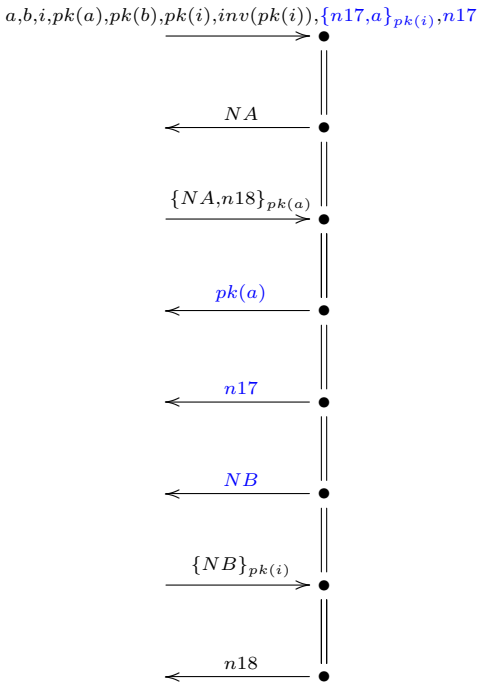


Now an important point is that we have to generate NA which is a variable. The composition rule cannot be applied to it, because it is not of the form $f(\ldots)$ for a public function f. The unification rule can only be applied between two terms s and t that are not variables. Thus, no rule can be applied – we leave NA for now. This is why the intruder is lazy – any value for NA will do, so why bother!

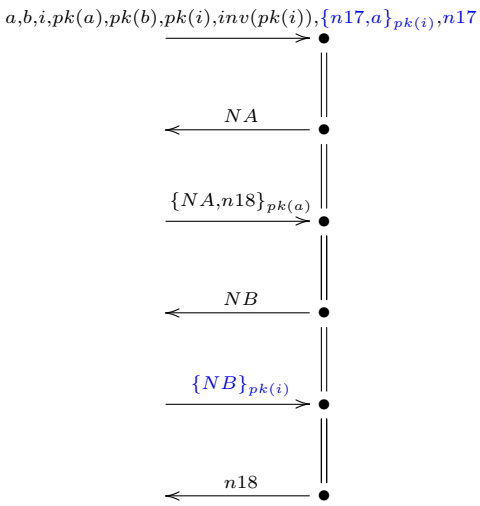
The next item to generate is a: it is already known and can be removed with unification:



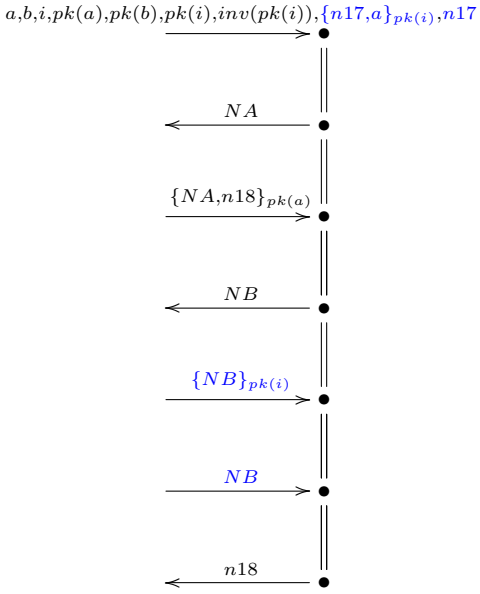
Consider now the first outgoing message that is not a variable: $\{n17, NB\}_{pk(a)}$. Again two general possibilities: unification or composition. Both cases are possible here, and to build the $\rightsquigarrow$ tree, we have to follow both. Let us follow composition first (both for the encryption and the pair):



$pk(a)$ and $n17$ are known and can be done with unification. NB is a variable and we are lazy again here, just leave it for now:



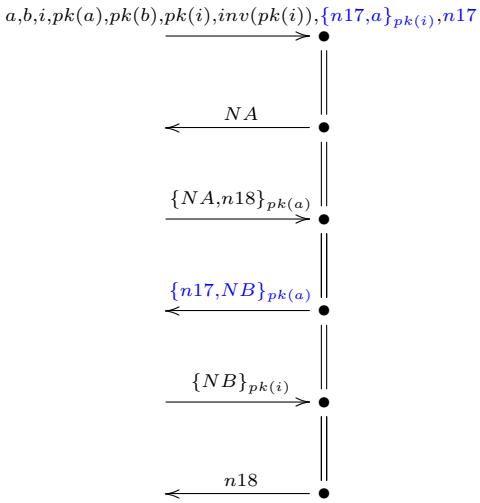
Looking at the next (incoming message), we can apply decryption here, giving us NB:



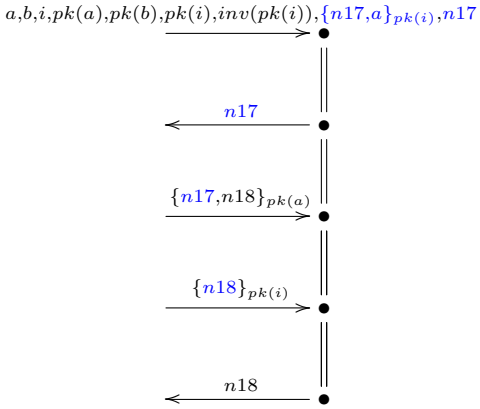
This is pretty useless: whatever NB is, we have constructed that ourselves earlier! Actually – this is the normal protocol execution where the intruder has generated some value NB and now received it back from Alice.

It remains to construct $n18$. That's impossible, because we cannot apply composition (since $n18$ is not a public function), and unification is impossible: we do not have $n18$ in our knowledge.

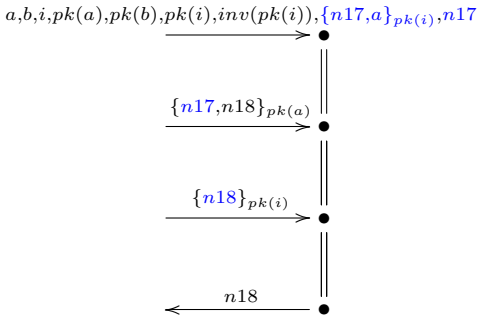
So this branch of the $\rightsquigarrow$ tree fails and we have to backtrack to the branching point from before:



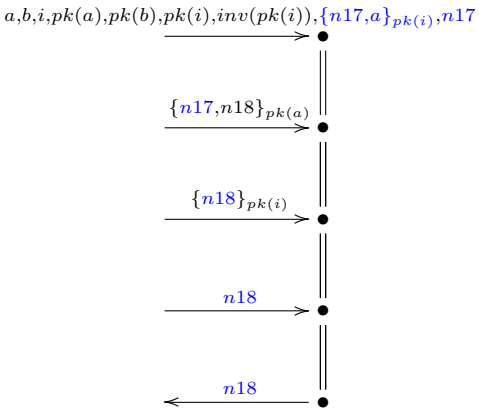
We can apply unification between the incoming message $\{n17, NB\}_{pk(a)}$ and the outgoing message $\{NA, n18\}_{pk(a)}$. The unifier (according to the mgu algorithm) is $\sigma = [NA \mapsto n17, NB \mapsto n18]$. We apply it to entire constraint and remove the outgoing message we have just "done" by unification:



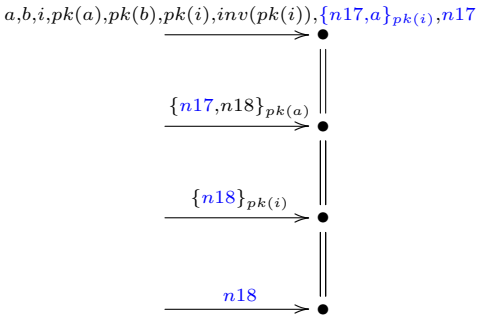
This unification is actually the central point in finding the solution. We have "retro-actively" decided that the intruder used $n17$ as nonce NA. This of course requires that the intruder knows $n17$. He does – so one unification step:



The message $\{n18\}_{pk(i)}$ can be decrypted, so we get $n18$:



The remaining outgoing message $n18$ is one simple unification step:



Solved!

13.7 Summary

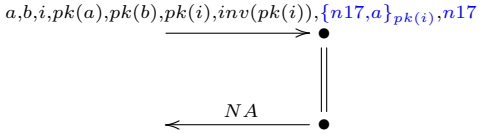
The procedure for solving constraints (with simple analysis) can be summarized as follows:

- Always start with the first step that has not been considered yet.
- If incoming: can simple decryption be applied?
- If outgoing:
  - If it is a variable, leave it for now and continue with the next step.
  - If it is not a variable, consider independently (with backtracking!) the following cases:
    - Composition (if it is a public operator)
    - Unification – with any incoming message that is not a variable.
- Whenever a unification is done where variables are substituted, apply to the entire constraint and go back to the first message that was affected!
- When all remaining outgoing messages are variables, the constraint is solved. We call that a simple constraint.

13.8 Simple Constraints

Definition 28. *An intruder constraint is called* simple *if all outgoing messages are variables.*

For instance, the constraint we obtained at the end of the NSPK example is simple since it has no more outgoing messages. Moreover, in the intermediate steps, the strands have a simple *prefix* like this one:



This is simple since the only outgoing message NA is a variable. Note that we cannot apply composition or unification steps to such a message – and it would be pointless, because it requires the intruder to only send *some* message. Since the intruder can always construct *some* message we have:

Lemma 2. *Every simple constraint has a solution.*

Thus, when we arrive at a simple constraint, we have found a solution. If we arrive at a constraint that is not simple and no further intruder rules can be applied, then the constraint is not satisfiable – as the following lazy intruder correctness theorems show.

13.9 ⋆ Correctness of the Lazy Intruder

Important properties of the lazy intruder:

- Termination: every unification and composition step makes the constraint simpler, this cannot go on forever. The analysis steps can only produce subterms of terms we already have.
- Soundness: the lazy intruder procedure finds only correct solutions (covered by the Dolev-Yao model)
- Completeness: if a constraint has a solution, the lazy intruder will find it:
  - Consider any solution of a constraint.
  - Then the constraint is either already simple or one of the lazy intruder steps gets us to a new constraint that still supports that solution.
  - By termination, we eventually arrive at a simple constraint that supports the considered solution.

Finally, one can show that the problem of protocol security with bounded sessions (and nonces) but unbounded messages is co-NP-complete.

Theorem 5 (Termination)**.** *The lazy intruder terminates, i.e., for a given constraint S, there are only finitely many S' such that $S \rightsquigarrow^* S'$.*

Proof. For a constraint S let us say its *weight* is a triple (l_1, l_2, l_3) of positive integers where

- l_1 is the number of variables in S.
- l_2 is the number of terms in S that have not been analyzed (i.e., where the intruder does not have the subterms).
- l_3 is the size of all terms to be sent (number of characters) together.

We order the components lexicographically, i.e., the weight l_1 is the most signaficant, l_2 second, and l_3 least. For instance $(3, 2, 10) < (3, 3, 3)$. Then every rule of the lazy intruder reduces the weight, and it is positive in all three components, so there cannot be an infinite chain of lazy intruder steps. Also, for every constraint S there are only finitely many constraints S' reachable in one step. □

Theorem 6 (Soundness)**.** *The lazy intruder is sound: if $S \rightsquigarrow^* S'$ and S' has a solution, then also S has a solution.*

Proof. Every reduction rule can be justified as sound by the Dolev-Yao rules. □

Theorem 7 (Completeness)**.** *The lazy intruder is complete: if S has a solution, then there is a simple S' with $S \rightsquigarrow^* S'$.*

Proof. It is sufficient to show that for every non-simple S that has a solution, we have $S \to S'$ for some S' that has a solution. From this plus termination then follows completeness.

Let S be a non-simple constraint so that σ is a solution of S, i.e., every outgoing message $Snd(t)$ in S with M being the set of messages received before $Snd(t)$, it holds that $\sigma(M) \vdash \sigma(t)$ (where $\vdash$ is the Dolev-Yao intruder deduction). Consider the first such step where t is not a variable (if there is none, the constraint is already simple). We have the following cases:

- If the last step in the deduction $\sigma(M) \vdash \sigma(t)$ is (Axiom): then there is a term $s \in M$ such that $\sigma(s) = \sigma(t)$. If s is not a variable, then this covered by the unify rule. Otherwise, if s is a variable, we have a constraint of the form

  $$\ldots \mathsf{Snd}(s) \ldots \mathsf{Rcv}(s) \ldots$$

  and thus there must be solution that does not rely on $\mathsf{Rcv}(s)$, because the intruder sent that earlier himself. We can thus proceed with that "simpler" solution.

- If the last step in the deduction $\sigma(M) \vdash \sigma(t)$ is (Compose): then we can similarly apply compose.

- If the last step in the deduction $\sigma(M) \vdash \sigma(t)$ is an analysis step. Let s_0 be the term being analyzed here. We distinguish two cases.

  - Let us first suppose that s_0 is obtained by an axiom rule. If there is a non-variable term $s \in M$ such that $s_0 = \sigma(s)$, then an out-sourced analysis steps is applicable right before $\mathsf{Snd}(t)$ and lead to the desired analysis result. Otherwise, if there is a variable $x \in M$ such that $s_0 = \sigma(x)$ then the constraint has the form:

    $$\ldots \mathsf{Snd}(x) \ldots \mathsf{Rcv}(x) \ldots$$

    then again there must be a simpler solution that does not rely on $\mathsf{Rcv}(x)$ and that we can proceed with.
  - If s_0 is obtained from an analysis step, then proceed with that analysis step first.
  - If s_0 is obtained from a composition step, then the intruder has first composed a message that he then analyzed. We can simplify this solution to eliminate this composition-analysis-pair, and proceed with that solution.

Thus in all cases, we get closer to a solved constraint. □

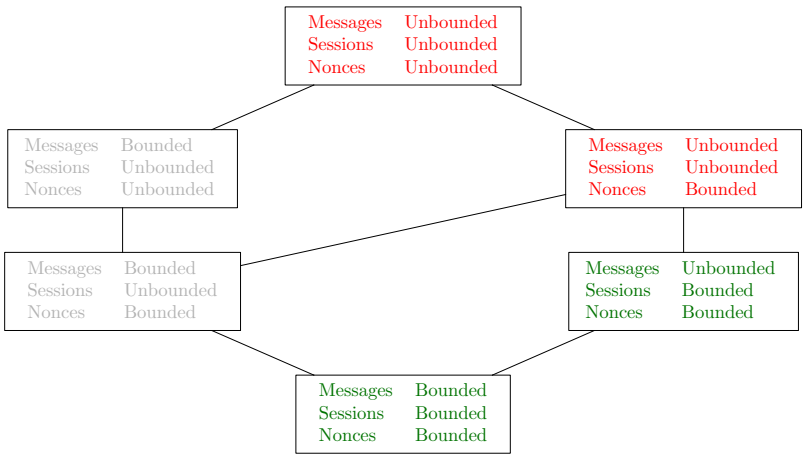
Theorem 8.

Theorem 9 (NP-Completeness)**.** *The problem of protocol security with bounded sessions (and nonces) but unbounded messages is co-NP-complete.*

Proof. First, one can show that for every Boolean formula, we can generate a protocol of a size that is polynomial in the size of the formula (and this generation takes polyniomal time) and so that the formula is satisfiable iff the protocol has an attack. Thus insecurity is NP-hard.

Second, we can show that there is a non-deterministic algorithm in polyniomial time (in the size of the initial state of the protocol) to decide protocol insecurity (without bounding messages). To that end, consider that in the symbolic transition system a trace cannot be longer thant the total number of steps in all strands; for each constraint the number of lazy intruder reductions is also polynonially bounded by the weight of the constraint.[15] □

This gives us one more entry in the decidability lattice: we can handle unbounded messages if sessions (and thus nonces) are bounded:



The remaining two items will be covered in the next chapter.

14 Abstract Interpretation

While the technique of the lazy intruder is actually very good for quickly finding attacks, it can only verify for a bounded number of sessions: when we have verified a protocol for 3 sessions, it does not tell us if there is maybe an attack for 4 sessions. Since of protocols like TLS millions of sessions exist worldwide at the same time, and the verification is growing exponentially with the number of sessions, this is not really feasible. One often refers to this as the *state-explosion problem* of model-checking. Moreover this way we never ontain a statement about *any* number of sessions.

There is however another approach that can side-step the state-explosion problem and verify for infinitely many sessions: abstract interpretation. This approach is used in static program analysis. A simple example is a compiler that shall check that every variable of a program is initialized before it is used. In general we cannot tell that for sure because of the halting problem: if an uninitialized variable is used first after a while loop, then the answer depends on whether this while loop will ever terminate (which is undecidable in general). The simple solution is: let us assume the while loop could terminate, then the compiler should flag this as a use of an uninitialized variable. We are thus considering an *over-approximation* (of what can happen), potentially ruling out a few good programs, but we get an analysis that is much simpler, that will never accept a bad program, and that always terminates.

In protocol verification this idea has been also very successful using logical formulas to give an over-approximation of what can ever happen in a protocol, in particular what the intruder can ever know. Since this is literally all a bit *abstract*, let us consider a concrete example step by step.

[15] A naïve implementation of substitutions can actually lead to an exponential runtime here, but there is an implementation that avoids this blow-up.

14.1 An Example

The basic formalism we use here are logical implications like `p.q => r.t` meaning "when p and q hold, then also r and t hold". This is often called *(definite) Horn clauses.*[16] We use here a notion from AIF$_\omega$ [22], a novel add-on for OFMC for denoting the implications, while AIF$_\omega$ rules actually mean state transitions. For the formulas we are using in this section, however, it does not make a difference. The full language of AIF$_\omega$ will be explained in a special chapter later.

Let us begin with the intruder, and let $iknows(m)$ stand for "the intruder knows m". We could specify first his initial knowledge like this:

```
=> iknows(a); => iknows(pk(a));
=> iknows(b); => iknows(pk(b));
=> iknows(i); => iknows(pk(i)); => iknows(inv(pk(i)));
```

to mean that the intruder knows the agents a, b, i their public keys and his own private key. Here we use implications without a left-hand side, i.e., the right-hand side facts holds unconditionally.

If we want to consider more agents, then we would have to make long enumerations. Therefore AIF$_\omega$ allows to first define some data types:

```
Honest = {a,b};
Dishonest = {i};
User = Honest ++ Dishonest;
```

We can then use these types in rules and later change the number of agents without changing any rules and without making long enumerations. To that end, every rule has a rule head of the form

$$rulename(Variable : Type, Variable : Type, ...)$$

The type can be any of the user-defined types (like Honest here) or *Untyped*. Untyped variables, however, have an important restriction: every untyped variable must occur in the left-hand side (and may occur in the right-hand side also). In other words, it is not allow to have untyped variables that only occur in the right-hand side.

The listing above is then written simply as follows:

```
users(A: User) => iknows(A);

publickeys(A: User) => iknows(pk(A));

privatkeys(D: Dishonest) => iknows(inv(pk(D)));
```

The first rule has the name "users" and says that for any A of type $User$, the intruder knows A. The other rules are similar.

Let now

- $crypt(k, m)$ stand for $\{m\}_k$, i.e., the asymetric encryption with key k of message m,
- and $pair(m1, m2)$ stand for the pair of $m1$ and $m2$.

Then the Dolev-Yao model for asymmetric encryption and pair is described by the following formulas:

```
asymenc(M1: untyped, M2: untyped)
iknows(crypt(M1,M2)).iknows(inv(M1)) => iknows(M2);

asymdec(M1: untyped, M2: untyped)
iknows(M1).iknows(M2) => iknows(crypt(M1,M2));
```

[16] Actually, by definition, Horn clauses can only have one fact on the right of the arrow, but `p.q => r.t` can be regarded as an abbreviation of the two Horn clauses `p.q => r` and `p.q => t`.

```
pair(M1: untyped, M2: untyped)
iknows(M1).iknows(M2) => iknows(pair(M1,M2));

proj(M1: untyped, M2: untyped) iknows(pair(M1,M2))
=> iknows(M1).iknows(M2);
```

The first formula says: if the intruder knows any messages $M1$ and $M2$, then he also knows $crypt(M1, M2)$, and similar the second, that he can decrypt $crypt(M1, M2)$ if he knows (the private key) $inv(M1)$. The rules for pair follow the same principle. The dot (.) is in this syntax logical "and" and the arrow (=>) is implication. In the initial intruder knowledge before, the (=>) was simply used without anything on the left-hand side, i.e., the right-hand side holds unconditionally.

From all the formulas so far we can now for instance derive the fact $iknows(crypt(pk(a), pair(a, b)))$, but not for instance $iknows(inv(pk(a)))$. Note that there are already infinitely many facts, e.g. $iknows(pair(a, pair(a, pair(a, \ldots))))$ is derivable. Most of the time this is however not an issue for the technques we consider.

Let us now model the honest agents of the NSPK protocol. The first step is that A sends out the message $crypt(pk(B), pair(NA, A))$, but here is a problem: NA is supposed to be freshly created. The logical formulas we have been writing so far, however, do not have a notion of time, as we are building an over-approximation of what can ever happen. Thus we cannot directly work with fresh nonces here.

Suppose now that A would actually not create a fresh random number in every session and instead use always the same random number. Then the intruder would learn this number in any session where he is B, and thus destroying secrecy. So this would be a too *coarse* over-approximation where all nonces are the same, so the intruder learns them. Let us refine a bit: suppose every has one single nonce for every other agent B; thus A is using in every session with the same B also the same nonces. We can thus write $na(A, B)$ for the nonce that A uses as na to B, and thereby make all nonces a function of the agent names. This gives us the following formula:

```
nspk1(A: User, B: User)
=> iknows(crypt(pk(B),pair(na(A,B),A)));
```

Thus the intruder knows the message that every agent in role A can produce as a first step for any agent in role B. He can decrypt this message, however, only if he is B, i.e., he learns $na(A, i)$ for every agent A. These are exactly the nonces where he is the intended recipient.

Note that the abstract $na(A, B)$ for the fresh nonce NA is still very coarse; we will not be able to check that the protocol is safe against replay attacks for instance, since all freshness is lost. However for secrecy this is fine enough. In fact we can now specify a violation of secrecy by the following rule:

```
secrecy1(A: Honest, B: Honest)
iknows(na(A,B)) => attack;
```

This means that it is an attack, if the intruder ever finds out the nonce $na(A, B)$ of two *honest* agents A and B. This indeed holds so far.

Now we can model how B can receive a message of the first step, i.e., of the form $\{p\}_(k(A), pair(NA, A))$ and send the second step, i.e., $crypt(pk(A), pair(NA, NB))$ as an answer. Here, B does not have any "control" over the value NA, he will accept any term here, so let us model NA as an untyped variable. NB however, is again freshly created. We use the exact trick as before and make this a function $nb(B, A)$, so we get:

```
nspk2(A: User, B: User, NA: untyped)
iknows(crypt(pk(B),pair(NA,A))) =>
iknows(crypt(pk(A),pair(NA,pair(nb(B,A),B))));
```

The intruder can decrypt the answer message if $A = i$, thus the intruder learns $nb(B, i)$ for every agent B – as before this is alright, since these are the nonce created by B for i. If $A = i$, he also

can analyze NA, however, no honest agent will claim the name of the intruder in the incoming message, so the intruder must have constructed that himself, and thus, he already had NA before.

Again we can make a secrecy goal for the nonce of B:

```
secrecy2(A: Honest, B: Honest)
iknows(nb(B,A)) => attack;
```

Next, A is waiting for a message of the form $\{NA, NB\}_{pk(A)}$ where NB can be anything and NA is the nonce that A created earlier for B. Then she will respond with the third step of the protocol $\{NB\}_{pk(B)}$. To model that NA must be the nonce that A has sent earlier, we can use again the abstraction: it must be of the form $na(A, B)$. Thus we have the following rule:

```
nspk3(A: User, B: User, NB: untyped)
iknows(crypt(pk(A),pair(na(A,B),NB))) =>
iknows(crypt(pk(B),NB));
```

Now the fact *attack* is derivable:

- From nspk1 the intruder can derive $crypt(pk(i), pair(na(a, i), a))$ and obtain $na(a, i)$.
- He can then construct $crypt(pk(b), pair(na(a, i), a))$.
- Using this message with nspk2, the intruder can then derive $crypt(pk(a), pair(na(a, i), nb(b, a)))$. Note that nothing in here says not explicitly who b is, but you can see it in the abstractions: a constructed $na(a, i)$ for i, and b constructed $nb(b, a)$ for a.
- Using this message with nspk3, the intruder gets $crypt(pk(i), nb(b, a))$.
- From that he gets $nb(b, a)$ which is the nonce of two honest agents, thus it is an attack (secrecy2).

This is the classical attack of Lowe we have discussed previously. Let us now change the protocol according to Lowe's suggestion, i.e., the second message additionally contains the name of B, i.e. $crypt(pk(B), pair(NA, pair(NB, B)))$.

The full specification is now:

```
Problem: nspk;

Types:
Honest    = {...};
Dishonest = {...};
User      = Honest ++ Dishonest;

Sets:
dummy(User, User); % not using this here, just for syntax

Functions:
public crypt/2, pair/2, pk/1;
private inv/1, na/2, nb/2;

Facts:
iknows/1, attack/0, secret/3;

Rules:

users(A: User)
=> iknows(A);
```

```
publickeys(A: User)
=> iknows(pk(A));

privatkeys(D: Dishonest)
=> iknows(inv(pk(D)));

asymenc(M1: untyped, M2: untyped)
iknows(crypt(M1,M2)).iknows(inv(M1)) => iknows(M2);

asymdec(M1: untyped, M2: untyped)
iknows(M1).iknows(M2) => iknows(crypt(M1,M2));

proj(M1: untyped, M2: untyped) iknows(pair(M1,M2))
=> iknows(M1).iknows(M2);

pair(M1: untyped, M2: untyped)
iknows(M1).iknows(M2) => iknows(pair(M1,M2));

nspk1(A: User, B: User)
=> iknows(crypt(pk(B),pair(na(A,B),A)));

nspk2(A: User, B: User, NA: untyped)
iknows(crypt(pk(B),pair(NA,A))) =>
iknows(crypt(pk(A),pair(NA,pair(nb(B,A),B))));

nspk3(A: User, B: User, NB: untyped)
iknows(crypt(pk(A),pair(na(A,B),pair(NB,B)))) =>
iknows(crypt(pk(B),NB));

secrecy1(A: Honest, B: Honest)
iknows(na(A,B))
=> attack;

secrecy2(A: Honest, B: Honest)
iknows(nb(B,A))
=> attack;
```

In the declaration of Honest and Dishonest, we are using here a special feature of AIF$_\omega$, the ... that allows us to define an infinite set of new constants, instead of a finite enumeration, so we have now an infinite set of honest and dishonest participants.

We feed this specification into AIF$_\omega$ which translates it for the tool ProVerif (or other solvers for this kind of Horn clauses) and within seconds, we get the result: "goal unreachable: attack:", meaning the fact *attack* is no longer derivable from our specification.

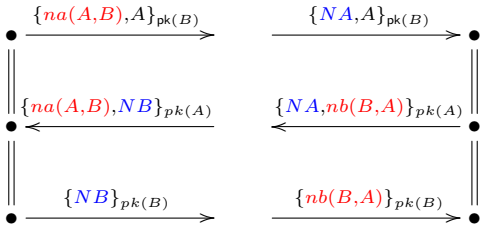
14.2 Two Abstractions

We can actually distinguish two "abstractions" we have made in the above example:

- We have abstracted the fresh nonces into functions like $na(A, B)$ and $nb(B, A)$. This gives us essentially a finite number of nonces (whenever the number of honest agents is finite).
- We have completely disgarded the transition system: there is no notion of state anymore, but we rather talk only about what messages the intruder may obtain in any number of runs.

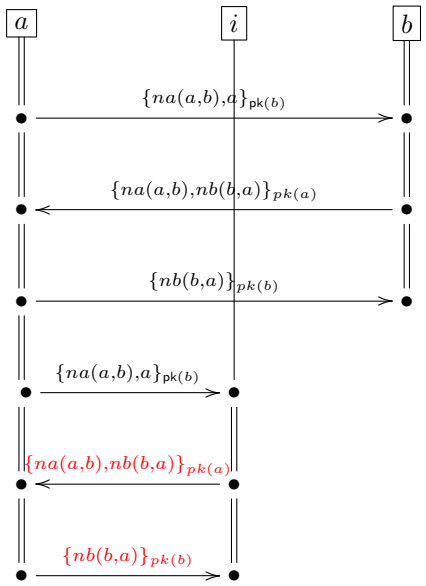
Formally, these two abstractions can be seen as a so-called *Galois-connection* [4].

Let us first look at the abstraction of the fresh data in isolation. We can see this actually as a transformation of the original protocol. For the NSPK example this means replacing in the original strands for role A and role B the freshly created variables with functions, obtaining the following two strands:



It is relatively easy to see that whatever protocol execution was possible with the original ("concrete") protocol, is still possible with the new ("abstract") protocol: take any execution of the original one, and replace all fresh nonces with the abstract ones, you get an execution of the new protocol. In this sense the abstraction is *sound*: we do not loose any execution of the original protocol, so if we can prove the new one to be correct, so is the old one.

The inverse however is not true: the new protocol has executions that the original one does not have:



Here the intruder, having observed one session between A and B, can play B in any future session. The protocol in this abstract model is therefore trivially vulnerable to replay, while that is not an issue in the original protocol.

This also means that not all goals can easily be applied to the new protocol. For instance authentication uses negation: it is an attack if there is request-event and *no* corresponding witness-event. Imagine, that in the original protocol a state is reachable where only the following witness and request events have occured:

$$witness(a, b, n_1).request(b, a, n_2)$$

i.e., a violation of authentication, and suppose both nonces n_1 and n_2 are created by a for communication with b, so they get abstracted into the same constant $na(a, b)$. Then, the new protocol would not have here a violation of authentication.

In fact, it is hard to deal with authentication (and freshness) in this abstract model, but it is possible with some further refinements. Let us however only deal with secrecy in the following, since a violation of secrecy is only described by the positive condition that the intruder can produce a secret, and thus works fine with the over-approximation.

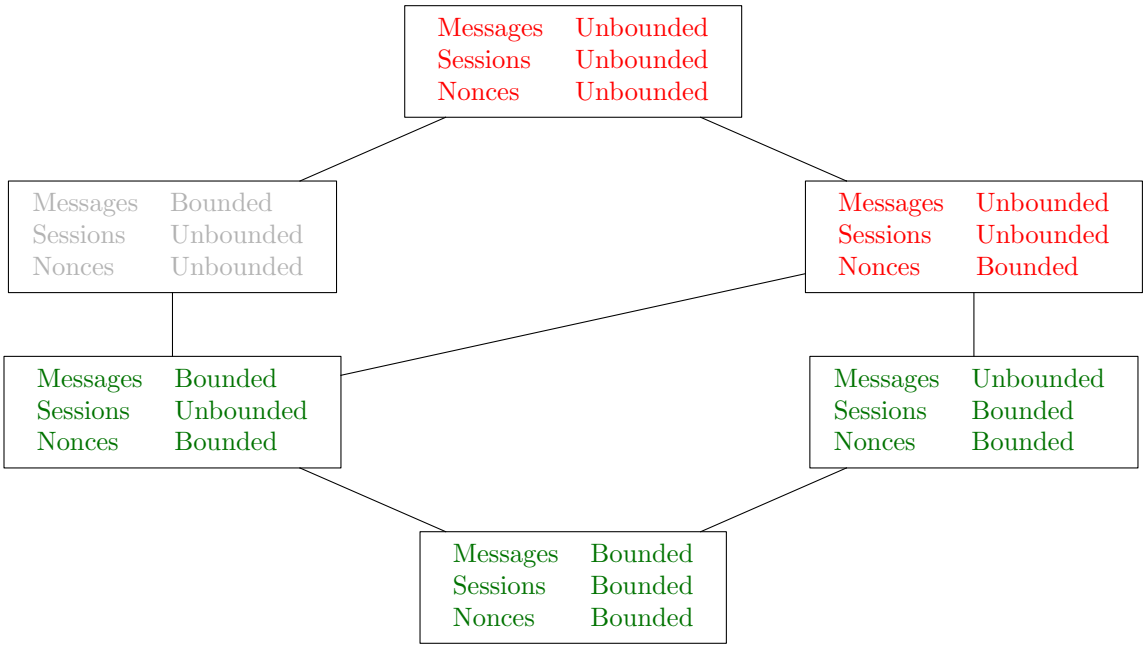
It has been shown that for secrecy goals, it is sufficient to consider only a fixed number of agents, e.g. $\{a, b, i\}$ [8]. Sufficient here means that for every attack with more agents, there is an attack with only these. Then we get also a fixed number of nonces, namely $na(A, B)$ and $nb(B, A)$ for every $A, B \in \{a, b, i\}$, and have removed the infinity of the nonces, even though we can consider infinitely many sessions.

Still, if the intruder is unbounded, this created an infinite set of reachable states, since agents still have variables for the nonces from the other parties, and thus the intruder can create an arbitrary message and use it as a nonce. Let us therefore briefly consider another restriction:

Definition 29 (Typed Model)**.** *In a* typed model *all variables have a type (e.g. nonce) and we allow only instantiation of variables with terms of the correct type.*

Then for instance, the variables NA and NB can only be instantiated with the nonces $na(A, B)$ or $nb(B, A)$ for some $A, B \in \{a, b, i\}$. This forbids the intruder to send *ill-typed* messages. Together with the finite number of nonces, there are now only finitely many terms that can be constructed, sent, and received.

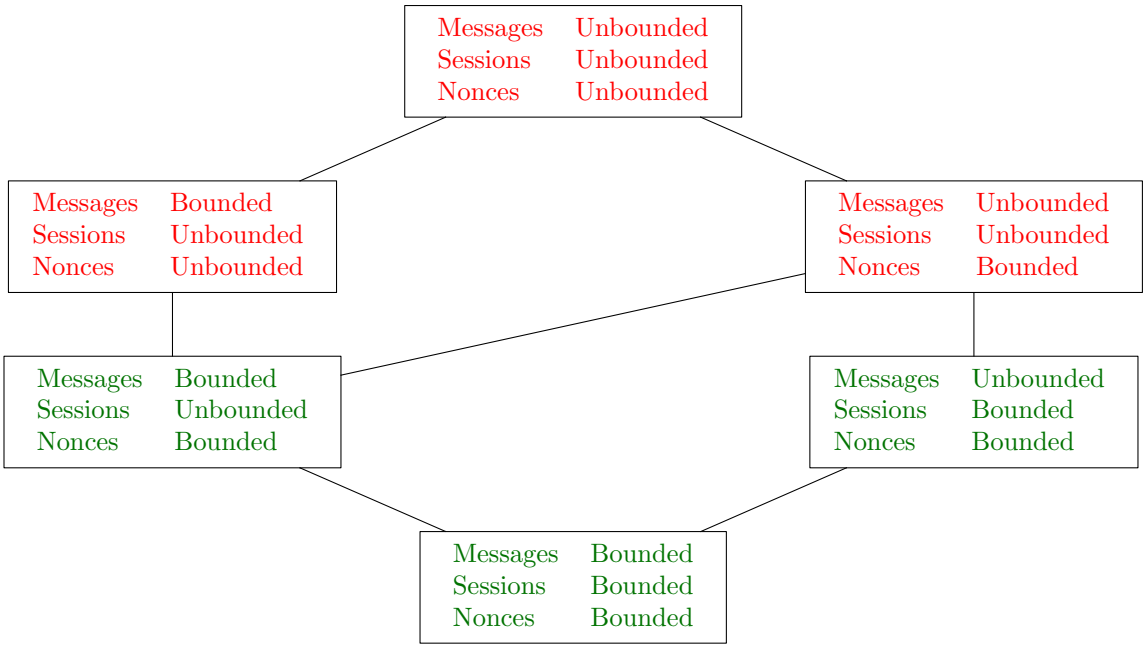
With these restrictions – bounded nonces and bounded messages – it is actually possible to decide secrecy goals for protocols and we thus have:



To complete our picture, we mention the following result that is similar to the first undecidability proof from the previous section:

Theorem 10 ([12])**.** *For an unbounded number of sessions and an unbounded number of nonces, protocol security is undecidable, even when bounding messages.*

This gives finally:



The conclusion is thus: for decidability, we may have either unbounded messages or unbounded sessions (with bounded nonces), but not both.

Part III
Advanced Topics

15 Channels and Composition

We now introduce a notion of *channels* that allow us to describe complex systems as a composition or layering of several simpler systems. A standard example can be the log-in at a bank. The user (resp. the user's browser) establishes a TLS connection with the bank. There could be now a password based authentication, or a login via a single-sign on solution like NemID. The latter is a protocol between the Bank, the User, and the Identity provider (e.g. NemID) that is also running over TLS channels. Finally, after authentication the actual banking application is communicating with the user via the inital TLS channel. It is considerably easier to consider not the resulting composed protocol as a monolithical system, but rather consider the individual components, where TLS provides a secure channel and the other protocol assume a secure channel. We shall in particular clarify what it means to have a secure channel.

15.1 Bullet Notation

We start with a very notation for channels called the bullet-calculus [20] which was started as an abstract way to model cryptography as the properties of a communication channel. While the original application is a calculus about what channels can be achieved given a number of existing channels, we will use it in AnB as a notation for channels over which messages are sent.

There are basically three kinds of channels; let us first give an intuition before we come to the formal definition.

- $A \bullet\!\!\rightarrow B$: A sends a message on an *authentic* channel to B, so B gets the guarantee that the message comes from A (but there is no guarantee that the message is confidential).

- $A \rightarrow\!\bullet\, B$: A sends a message on a *confidential* channel to B, so only B can read it (but no guarantee that it comes from A).[17]
- $A \bullet\!\rightarrow\!\bullet\, B$: A sends a message on a *secure* channel that is both authentic and confidential: A can be sure that only B can read it, and B can be sure it comes from A.

Note that for each channel there are lots of ways to achieve this in practice, e.g. asymmetric encryption and signatures.

Example 27 (NSL)**.** *We can write the Needham Schroeder Public Key protocol with Lowe's fix (NSL) completely without cryptography, when we replace the public key encryption with confidential channels:*

$$\begin{array}{llll} A & \rightarrow & B: & \{NA, A\}_{\mathsf{pk}(B)} \\ B & \rightarrow & A: & \{NA, NB, B\}_{\mathsf{pk}(A)} \\ A & \rightarrow & B: & \{NB\}_{\mathsf{pk}(B)} \end{array} \qquad \begin{array}{llll} A & \rightarrow\!\bullet & B: & NA, A \\ B & \rightarrow\!\bullet & A: & NA, NB, B \\ A & \rightarrow\!\bullet & B: & NB \end{array}$$

One of the best illustrations of channels is actually Diffie-Hellman (compare Sec. 3.2). As we have seen, what Diffie-Hellman requires is an authentic change of the "public keys" $exp(g, X)$ and $exp(g, Y)$. Thus:

$$\begin{array}{llll} A & \bullet\!\rightarrow & B: & \exp(g, X) \\ B & \bullet\!\rightarrow & A: & \exp(g, Y) \end{array}$$

This expresses the essence of Diffie-Hellman: the authenticated exchange of Diffie-Hellman public exponents, while we leave completely open how the authentication may happen:

- Cryptographic measures like digital signatures, symmetric encryption/MACs, asymmetric encryption and there are correspondingly many different protocols using these measures like the IPSEC protocols IKE, IKEv2, JFK, (Diffie-Hellman mode of) TLS, and many others.
- Relying on a trust third party.
- Relying on face to face meeting between friends, e.g. device-pairing is a protocol between two mobile devices where an un-authenticated Diffie-Hellman exchange is performed between the two devices and the owners of the devices are presented a fingerprint of the two public exponents, so they can compare and approve the exchange.

Channels can also be thought of as a goal: the goal of Diffie-Hellman for instance is to establish a secure channel between two parties:

$$\begin{array}{llll} A & \bullet\!\rightarrow & B & : \exp(g, X) \\ B & \bullet\!\rightarrow & A & : \exp(g, Y) \\ A & \rightarrow & B & : \{\!|A, B, MsgA|\!\}_{\exp(\exp(g,X),Y)} \\ A & \rightarrow & B & : \{\!|B, A, MsgB|\!\}_{\exp(\exp(g,X),Y)} \\ Goals: & & & \\ A & \bullet\!\rightarrow\!\bullet & B & : MsgA \\ B & \bullet\!\rightarrow\!\bullet & A & : MsgB \end{array}$$

We have added that A and B exchange two payload messages $MsgA$ and $MsgB$. As a goal we have specified that these payload messages are like transmissions on a secure channel, i.e., the goal $A \bullet\!\rightarrow\!\bullet\, B : MsgA$ means that both

- the message is authentically transmitted: `B weakly authenticates A on MsgA`, and
- the message is confidentially transmitted: `MsgA secret between A,B`.

[17] We use the words confidential and secret as synonymous; the term *confidential channel* had already been established in the literature and thus we did not change it here.

Thus, we can summarize Diffie-Hellman very generally by saying: *Diffie-Hellman creates secure channels from authentic channels*. A very general definition of public key cryptography is: *any scheme that allows us create secure channels from authentic ones*. Observe that this definition completely avoids even talking about any technical aspect like public and private keys.

15.2 A Cryptographic Implementation

A question is what *authentication*, *confidential*, and *secure* channel should precisely mean. In fact there are several quite different answers. We define them here using asymmetric cryptography. There are many other ways to define them, e.g. describing by their behavior.

Assume every agent A has two key pairs:

- $\langle \mathsf{ck}(A), \mathsf{inv}(\mathsf{ck}(A))\rangle$ for asymmetric encryption,
- $\langle \mathsf{ak}(A), \mathsf{inv}(\mathsf{ak}(A))\rangle$ for digital signatures,

and assume that every agent knows the public keys $\mathsf{ck}(A)$ and $\mathsf{ak}(A)$ of every other agent A. We require that neither $\mathsf{ck}(\cdot)$ nor $\mathsf{ak}(\cdot)$ occur in the AnB specification.

Then we can implement authentic channels by signing, confidential channels by encryption, and secure channels by both signing and encrypting:

Definition 30. *Channel implementation using asymmetric cryptography:*

$$\begin{array}{llllllll} A & \bullet\!\!\rightarrow & B: & M & for & A & \rightarrow & B: \ \{B,M\}_{\mathsf{inv}(\mathsf{ak}(A))} \\ A & \rightarrow\!\!\bullet & B: & M & for & A & \rightarrow & B: \ \{M\}_{\mathsf{ck}(B)} \\ A & \bullet\!\!\rightarrow\!\!\bullet & B: & M & for & A & \rightarrow & B: \ \{\{B,M\}_{\mathsf{inv}(\mathsf{ak}(A))}\}_{\mathsf{ck}(B)} \end{array}$$

This ensures the basic properties of channels:

- Only A can produce messages on the channel $A \bullet\!\!\rightarrow B$, since nobody else knows $\mathsf{inv}(akA)$.
- Only B can read messages on the channel $A\rightarrow\!\!\bullet B$, since nobody else knows $\mathsf{inv}(ckB)$.
- Both restrictions hold on a secure channel.

Note that the intruder can still intercept and replay messages – the channels we have defined do neither guarantee resilience against network disruption nor do they guarantee freshness.

A question is why we would need to include the name of the intended recipient in authentic and secure channels implementations. For the secure channels consider the protocol

$$\begin{array}{lllll} A & \rightarrow & B & : & \{\{MsgA\}_{\mathsf{inv}(()ak(A))}\}_{ck(B)} \\ Goals: & & & & \\ A & \bullet\!\!\rightarrow\!\!\bullet & B & : & MsgA \end{array}$$

Recall that the goal stands for `B authenticates A on MsgA` and `MsgA secret between A,B`. As an exercise, the reader (manually or with OFMC) should try to find the attack against the authentication goal of this protocol. In fact this is a "classical" mistake in protocol design that appears again and again over the years.

Authentic channels must guarantee that authenticate the information who is the intended recipient of the message. This is different from ensuring that only the intended recipient can *read* the message.

15.3 Channels as Assumptions – Channels as Goals

Many protocols we consider can be regarded as protocols for establishing a channel. For instance TLS is essentially a key-exchange (handshake) that establishes symmetric keys for communication between a client and a server, including a transmission protocol that uses these keys to encrypt arbitrary *Payload* messages with them (e.g. from an email application). We can thus also allow the definition of protocol goals using channels as we did it in previous examples, where authentic channel means a weak authentication goal, confidential channel means a secrecy goal, and secure channel means both goals.

Note that we have used only authentication here: the reason is that we want a correspondence between channels as *assumptions* (when the protocol transmits messages of authentic, confidential, or secure channels) and channels as *goals*. When we look at authentic channels again (and similar secure channels) in our cryptographic implementation, there is no protection against replay built into the channels: an intruder can record any message and replay it as is later. Thus, if replay prevention is needed, it remains the responsibility of the protocol that uses the channel to prevent replay. One may indeed make a variant of authentic channels (and similarly of secure channels) that also include replay protection (e.g. using timestamps, sequence numbers, or challenge response). In fact, one may define a variety of different channel properties.

15.4 Compositionality

The relation between channels as assumption and channels as goals gives rise to an interesting question:

- Given a protocol P_1 has as goal to establish a certain channel type C, e.g. TLS with the goal to establish a secure channel.
- Given another protocol P_2 assumes such a channel C, e.g. a web-application for an online email box.
- Suppose also that both P_1 and P_2 have been verified individually.
- Suppose finally, that we "plug" P_1 into P_2 together, i.e., running the traffic of P_2 over the channel established by P_1. Let us denote this composition as $P_2[P_1]$, pronounced *running P_2 over P_1*.
- Is the resulting protocol $P_2[P_1]$ correct?

Example 28. *Let P_1 be the following protocol:*

$$\begin{array}{rrl} & A \to s: & A, B, Payload, mac(sk(A,s), A, B, Payload) \\ & s \to B: & A, B, Payload, mac(sk(B,s), A, B, Payload) \\ \hline Goal: & A \bullet\!\!\to B: & Payload \end{array}$$

This protocol establishes only an authentic channel. It assumes that A and a trusted server s share a secret symmetric key $sk(A,s)$, and similarly B has key $sk(B,s)$ with s. Since only authentication is the goal, the Payload is actually not even encrypted, we just build a mac *(Message Authentication Code) that we model like a hash-function that receives as one argument a secret key. This* mac *can thus be checked by anybody who knows the respective key. In this way, A can authentically send the Payload message via s to B, since s is honest.*

Let P_2 be the following protocol:

$$\begin{array}{rrl} & A \bullet\!\!\to B: & \exp(g,X) \\ & B \bullet\!\!\to A: & \exp(g,Y) \\ & A \to B: & \{\!|ApplicationPayload|\!\}_{\exp(\exp(g,X),Y)} \\ \hline Goal: & A \bullet\!\!\to\!\!\bullet B: & ApplicationPayload \end{array}$$

Here we have an application protocol uses Diffie-Hellman to first establish a secure shared key between A and B to transmit a more high-level payload (called ApplicationPayload) securely. It assumes authentic channels from A to B and vice-versa, but it is independent of how this application is established.

The composition $P_2[P_1]$ is the following protocol:

$$\begin{array}{rl}
A \to s: & A, B, \exp(g,X), mac(sk(A,s), A, B, \exp(g,X)) \\
s \to B: & A, B, \exp(g,X), mac(sk(B,s), A, B, \exp(g,X)) \\
B \to s: & B, A, \exp(g,Y), mac(sk(B,s), B, A, \exp(g,Y)) \\
s \to A: & B, A, \exp(g,Y), mac(sk(A,s), B, A, \exp(g,Y)) \\
A \to B: & \{|ApplicationPayload|\}_{\exp(\exp(g,X),Y)} \\
\hline
Goal: A \bullet\!\!\to\!\!\bullet B: & ApplicationPayload
\end{array}$$

Note that this protocol no longer assumes any channels, i.e., it is completely implemented now. It is more complex that the original protocols, in the sense it is harder to read and understand, and also harder to analyze automatically.

15.5 Pseudonymous Channels

Consider again the TLS handshake protocol from Section 3:

```
Protocol: TLS
Types: Agent A,B,s;
       Number NA,NB,Sid,PA,PB,PMS;
       Function pk,hash,clientK,serverK,prf
Knowledge: A: A,pk(A),pk(s),inv(pk(A)),{A,pk(A)}inv(pk(s)),B,
              hash,clientK,serverK,prf;
           B: B,pk(B),pk(s),inv(pk(B)),{B,pk(B)}inv(pk(s)),
              hash,clientK,serverK,prf
Actions:
A->B: A,NA,Sid,PA
B->A: NB,Sid,PB,
      {B,pk(B)}inv(pk(s))
A->B: {A,pk(A)}inv(pk(s)),
        {PMS}pk(B),
        {hash(NB,B,PMS)}inv(pk(A)),
        {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
          clientK(NA,NB,prf(PMS,NA,NB))
B->A:   {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
          serverK(NA,NB,prf(PMS,NA,NB))
Goals:
  B authenticates A on prf(PMS,NA,NB)
  A authenticates B on prf(PMS,NA,NB)
  prf(PMS,NA,NB) secret between A,B
```

This is in fact not the most common way to use TLS, because it requires the client A to own a certificate of its public key, what is simply modeled here as the message $\{A, pk(A)\}_{inv(pk(s))}$, i.e., a trusted server s (the certificate authority) has signed the statement that A has public key $pk(A)$. Normally, only the server B will have such a certificate, but not the ordinary Internet user who runs TLS in role A. Essentially, the protocol is simply executed simply without A's certificate (while B's certificate must always be present):

```
Protocol: TLS
Types: Agent A,B,s;
       Number NA,NB,Sid,PA,PB,PMS;
       Function pk,hash,clientK,serverK,prf
```

```
Knowledge: A: A,pk(A),pk(s),B,
              hash,clientK,serverK,prf;
           B: B,pk(B),pk(s),inv(pk(B)),{B,pk(B)}inv(pk(s)),
              hash,clientK,serverK,prf
Actions:
A->B: A,NA,Sid,PA
B->A: NB,Sid,PB,
      {B,pk(B)}inv(pk(s))
A->B: A,PK,
        {PMS}pk(B),
        {hash(NB,B,PMS)}inv(PK),
        {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
          clientK(NA,NB,prf(PMS,NA,NB))
B->A:   {|hash(prf(PMS,NA,NB),A,B,NA,NB,Sid,PA,PB,PMS)|}
          serverK(NA,NB,prf(PMS,NA,NB))
Goals:
  #B authenticates A on prf(PMS,NA,NB)
  A authenticates B on prf(PMS,NA,NB)
  #prf(PMS,NA,NB) secret between A,B
```

We have here chosen to model that A has no key with a certifcate as A creating a fresh public key PK and sending it to B.

Obviously, without the client certificate, this protocol does not allow B to authenticate A, and thus both secrecy and part of authentication do no longer hold (thus commented in the listing). In other words, the user who claims to be A (and the creator/owner of PK) could in fact be anybody. So what does it even help to have this exchange without client authentication in the first place?

The answer is, we get slightly more than the remaining authentication goal (that A can be sure about the server). In fact, we still get something like a secure channel: the intruder cannot read or impersonate any messages in a session between an honest client A and an honest server B. This is since only the creator of PK (who knows $inv(PK)$) and B know the resulting keys of this exchange. So the intruder can start any number of sessions with fresh public keys for PK and under any agent name, but only read in these sessions (and in the sessions where he is B), but he cannot interfere with sessions between an honest client who created PK and an honest server (since he does not have $inv(PK)$ then).

We can actually regard the key PK as a *pseudonym* that A has chosen, nobody else can claim the ownership of that pseudonym, since that requires to know $inv(PK)$, and TLS thus establishes a secure channel between the owner of PK (whoever it is) and B.

We write for this kind of channel $[A] \bullet\!\!\rightarrow\!\!\bullet\, B$ (respectively, $B \bullet\!\!\rightarrow\!\!\bullet\, [A]$) to indicate that A is not authenticated with respect to her real name, but only with respect to a some pseudonym. We could express this for TLS without client authentication as follows:

	TLS handshake	
		$\ldots$
	$A \to B:$	$\{\lvert Payload_A \rvert\}_{clientK(NA,NB,prf(PMS,NA,NB))}$
	$B \to A:$	$\{\lvert Payload_B \rvert\}_{serverK(NA,NB,prf(PMS,NA,NB))}$
Goal :	$[A] \bullet\!\!\rightarrow\!\!\bullet\, B:$	$Payload_A$
	$B \bullet\!\!\rightarrow\!\!\bullet\, [A]:$	$Payload_B$

This is sometimes also called sender/receiver invariance: B cannot be sure about A's real identity, but that it is the same entity in several transmissions (namely the owner of a certain key pair).

This kind of channel is good enough for many applications such as transmitting credit card data: [A] •→• B : Order & Credit Card Data The intruder can also make orders, or, as a

dishonest merchant B receive credit card data, but cannot see the credit card data from an honest A sent to an honest B.

The secure pseudonymous channel is also good enough for a login protocol:

$$\frac{\begin{array}{rl}[A] \bullet\!\!\rightarrow\!\!\bullet\, B: & A, password(A,B) \\ B \bullet\!\!\rightarrow\!\!\bullet\, [A]: & Payload\end{array}}{\begin{array}{rl}Goal:\ B \bullet\!\!\rightarrow\!\!\bullet\, A: & Payload\end{array}}$$

where $password(A, B)$ is A's password at server B. We establish a "classical" secure channel in two steps:

1. We establish a secure pseudonymous channel $[A] \bullet\!\!\rightarrow\!\!\bullet\, B$ using TLS without client authentication.
2. We use this channel to authenticate the client by a shared secret (which possibly has low entropy).

Further replies (e.g. the data of client A stored on server B) are now bound to this authentication.

References

[1] O. Almousa, S. Mödersheim, and L. Viganò. Alice and bob: Reconciling formal models and implementation. In *Festschrift in honor of Pierpaolo Degano*, 2015.

[2] 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`.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[4] B. Blanchet. Security protocols: from linear to classical logic by abstract interpretation. *Inf. Process. Lett.*, 95(5):473–479, 2005.

[5] F. Böhl, V. Cortier, and B. Warinschi. Deduction soundness: prove one, get five for free. In *CCS 2013*, 2013.

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

[7] Y. Chevalier and M. Rusinowitch. Compiling and securing cryptographic protocols, 2010.

[8] H. Comon-Lundh and V. Cortier. Security properties: two agents are sufficient. *Sci. Comput. Program.*, 50(1-3):51–71, 2004.

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

[10] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[11] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, Mar. 1983.

[12] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In *Proc. of the Workshop on Formal Methods and Security Protocols (FMSP'99)*, July 1999.

[13] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *Symposium on Foundations of Computer Science*. IEEE Computer Society, 1983.

[14] F. J. T. Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, (2/3):191–230, 1999.

[15] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. In M. Parigot and A. Voronkov, editors, *LPAR*, volume 1955 of *Lecture Notes in Computer Science*, pages 131–160. Springer, 2000.

[16] C. Kirchner and H. Kirchner. *Rewriting, Solving, Proving*. 1999-2006. Available at `https://wiki.bordeaux.inria.fr/Helene-Kirchner/lib/exe/fetch.php?media=wiki:rsp.pdf`.

[17] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In T. Margaria and B. Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

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

[19] U. Maurer and R. Renner. Abstract cryptography. In *ICS*, 2011.

[20] U. Maurer and P. Schmid. A calculus for security bootstrapping in distributed systems. *Journal of Computer Security*, 4(1):55–80, 1996.

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

[22] S. Mödersheim and A. Bruni. Aif-omega: Set-based protocol abstraction with countable families. In *Principle of Security and Trust (POST)*, 2016. Tool webpage: `http://imm.dtu.dk/~samo/aifom.html`.

[23] 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.

[24] 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.

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

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