AnB-API: Extending AnB with APIs and persistent storage

Jóhann Björn Björnsson

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

In this thesis we introduce a security protocol modelling language, AnB-API, which is designed as an extension to another security protocol modelling language AnB. With AnB-API we extend the capabilities of AnB to support modelling of protocols that make use of an API based communications model and persistent storage. We formally describe the semantics of AnB-API and how it is compiled into AIF, a lower level language, and how that is translated into Horn Clauses that can be verified to give proof of security. We also give examples of protocol specifications written in AnB-API and explain how they utilise the new features of AnB-API.

Preface

This thesis was prepared at DTU Compute in fulfilment of the requirements for acquiring a Master's Degree in Computer Science and Engineering.

The thesis deals with the design and implementation of a security protocol modelling language.

The thesis consists of the formal description of AnB-API, a security protocol modelling language and it's compiler along with a description of AnB and AIF to which the language is related.

Lyngby, 25-June-2016

Jóhann B.

Jóhann Björn Björnsson

Acknowledgements

I would like to thank my supervisor Sebastian Mödersheim whose immense patience and skills were invaluable to me. His willingness to assist me and positive atmosphere kept me motivated to the end and I feel privileged to have had access to such a talented and knowledgeable man.

I would also like to thank my girlfriend Edda who has shown me devotion and love during all those late nights I've spent at school.

Contents

Chapter 1

Introduction

As our modern world moves online, security protocols are becoming an ever bigger part of our daily lives. They provide a foundation for secure online applications such as online banking, e-commerce and various privacy applications. Designing security protocols can be a complex task and modelling and analysis of such protocols is important. Various tools exist for protocol analysis and in this thesis one such tool is introduced called AnB-API. It is a high-level modelling language for modelling security protocols with certain properties. The language is meant as an extension to the Alice and Bob notation (AnB) [8], a security protocol modelling language with a simple and elegant Alice and Bob-style syntax. The AnB-API compiler compiles code written in AnB-API into Abstract Intermediate Format [7], an expressive lower-level security protocol modelling language.

CHAPTER 2

Background

Our new language AnB-API borrows features and syntax from both AnB and AIF. To get a good understanding of the requirements and behaviour of AnB-API we'll look at both languages and compare them to AnB-API.

2.1 Abstract Intermediate Format

As stated earlier, AnB-API is translated into Abstract Intermediate Format (AIF) [7]. AIF (related to AVISPA's Intermediate Format [1]) is a language that supports verifying a wide variety of protocols and was chosen as the language to compile AnB-API into for its features, mainly persistent storage and its flexible communications model. AnB-API also borrows syntax from AIF and since AnB-API is compiled into AIF, some AIF features are also offered more or less unchanged in AnB-API (namely Facts and Sets). Since AnB-API offers a more high-level syntax than AIF, some of AIF's expressiveness is sacrificed in AnB-API. AIF supports the definition of functions that allow the user to define custom protocol functionality where AnB-API supports a predefined set of commonly used operations. Moreover, AIF requires users to define an attacker model that the intruder will operate within. In AnB-API, an attacker model is built in and is not editable by programmers. All specifications written in the language are thus translated to AIF with the same attacker model.

2.1.1 Synax and semantics

AIF supports persistent storage in the form of sets. Sets are defined over variables which in turn are defined over constants. To state that the key `PK` is in agent `a`'s keyring we could write `PK in keyring(a)`. In this case `keyring` is a set that has been defined over a variable in which `a` is a constant. Sets can be defined over multiple variables which is useful when more complex database structures need to be expressed. If we wanted to define a server database that stores keys for users we could write `PK in db(s,a,valid)` to state that `PK` is registered as a `valid` key in the server's (`s`) database for the user `a`. This set's definition could be `db(s,U,Status)` where the variables might be defined as `U:{a,b}` and `Status:{valid,revoked}`. Here we see that variable names start with an uppercase letter and constant names start with a lowercase letter. A definition like this actually generates four sets `db(s,a,valid)`, `db(s,a,revoked)`, `db(s,b,valid)`, `db(s,b,revoked)`. In AIF, protocols are defined as rules in a state transition system. Rules have a left-hand side and a right-hand side, separated by an arrow. The right-hand side of the rule describes a state that will be reached if the rule is executed and the left-hand side of the rule describes a state that the system needs to be in for the rule to be executable. These states are defined with expressions. If the expressions on the left-hand side all evaluate to true, that rule can be executed and the expressions on the right-hand side describe the state of the system after the rule has been executed. Values can also be freshly generated, in which case value names are put on the arrow itself. In order to demonstrate AIF rules we'll look at a simple example of a key server protocol. For now we'll skip Type, Set, Function and Fact declarations and get back to them later when we talk about AnB-API.

```
=[PK]=>iknows(PK).PK in keyring(a).PK in db(s,a,valid);
```

In the above rule, the fresh value PK is created and inserted into both `a`'s keyring and the server's database for `a` as a `valid` key. We also add PK to the intruder knowledge with `iknows(PK)`, where we assume that `iknows` has been defined to represent intruder knowledge in an attacker model similar to the Dolev-Yao attacker model [4]. This rule can be executed multiple times in any state since no conditions are put forth on the left-hand side of the arrow. In the next rule a new key `NPK` is created and put into `a`'s keyring (intuitively `a` creates the key and puts it into their keyring). `a` sends `sign(inv(PK),pair(a,NPK))` to the server letting them know that `a` now has a new key and by sending it to the server, the message gets added to the intruder knowledge. Expressions that dictate membership in sets need to be on both sides of the rule in order for the membership to persist. This means that since `PK in keyring(a)` is on the left-hand side but not on the right-hand side, `PK` gets removed from `keyring(a)`.

```
iknows(PK).PK in keyring(a)
=[NPK]=>
NPK in keyring(a).iknows(sign(inv(PK),pair(a,NPK)));
```

As we see on the left-hand side of the rule, `PK` must be in `keyring(a)` and in the intruder knowledge, effectively making the first rule a prerequisite for this one. Here, `sign`, `inv` and `pair` refer to a signed message (signed with `inv(PK)`), a private key corresponding to the given public key (`PK`) and message concatenation, respectively. In the next rule the server receives the new key from `a`, checks if it already exists and if not, revokes the old key and inserts the new key into their database as a `valid` key.

```
iknows(sign(inv(PK),pair(a,NPK))).PK in db(s,a,valid).
forall Sts. NPK notin db(s,a,Sts)
=>
PK in db(s,a,revoked).NPK in db(s,a,valid);
```

By repeating `iknows(sign(inv(PK),pair(a,NPK)))` from the rule before on the left-hand side of this rule we again make the previous rule a prerequisite for executing this rule. In this rule we check if `NPK` exists in the server's database, both as a `valid` key and a `revoked` one. This is done with a `forall` statement. In `forall` statements we specify variables to loop over and a set. In this case we loop over `Sts` which contains `valid` and `revoked`. This means that we're checking if `NPK` exists in `db(s,a,valid)` and `db(s,a,revoked)` and if it doesn't, the `forall` expression evaluates to true. These rules describe our protocol but we need another one to describe a state in which an attack happens.

```
iknows(inv(PK)).PK in db(s,a,valid)
=>attack;
```

In this rule we simply state that if an intruder learns a private key and if that private key is in the server's database for our user `a` as a `valid` key, an attack takes place.

AIF is compiled into Horn clauses (both SPASS [9] and ProVerif [2] syntax) using the fpaslan [7] tool. By default fpaslan outputs Horn clauses in SPASS syntax which can be directly fed to the SPASS tool for verification. SPASS takes as input axioms and a conjecture. The Horn clauses from AIF are the input axioms and the conjecture is `attack`. The principal output from SPASS is either "Proof found" or "Completion found". When SPASS outputs "Proof found" it is indicating that it has found a proof that the Horn clauses (our protocol description) indeed lead to an attack. If SPASS however finds proof that the protocol does not lead to an attack it outputs "Completion found". The problem of determining wether or not the protocol leads to an attack is undecidable [7] so SPASS may run forever without producing a result.

2.2 Alice and Bob

As stated earlier, AnB-API is also takes a lot from Alice and Bob (AnB) [8]. AnB is one of the input languages to the Open Source Fixedpoint Model Checker (OFMC) [6]. It can model various protocols with multiple communicating agents and supports exponentiation, various security goals, pre-existing knowledge and a flexible channel notation supporting confidential, authentic and secure channels. AnB-API builds on this language and uses much of the same syntax. AnB uses the Dolev-Yao intruder model so all communications are intercepted by the intruder and get added to the intruder knowledge. The intruder can also synthesise any message from their knowledge and send at any time to any agent. The intruder can use all cryptographic operations and knows the identity of all agents in the protocol. In order to demonstrate AnB we quickly look through the single sign-on protocol in figure 2.1. We declare the types used in the protocol in the Types section, agents with the `Agent` keyword, fresh values with the `Number` keyword and functions with the `Function` keyword. Types that start with a capital letter represent variables (meaning that they'll be instantiated with an arbitrary number of values during protocol execution) and types starting with lower-case letters represent constants. The Knowledge section specifies knowledge that agents have before the protocol run starts. The aim with the protocol is for the user to be able to consume `SP`'s service without having to register with them beforehand (note that `SP` is a variable and represents multiple service providers). Instead the user and the service provider both trust the identity provider `idp` which facilitates the needed trust between `U` and `SP`. The Actions section describes communications in the protocol. The user sends `URI` to `SP` (intuitively a request for some service from `SP`) and gets an ID back which identifies the user to the service provider (note that `U` is a variable and represents multiple different users so `SP` needs to keep track of users). The user sends `ID` to `idp` encrypted with a shared key between the user and `idp`, which effectively authenticates the user to `idp`. `idp` now sends back (among other things) a signed message with the user's identity and `SP`'s identity which

```
Protocol: SingleSignOn

Types:
Agent U,idp,SP;
Number URI,Payload,ID;
Function sk

Knowledge:
U:   U,idp,SP,sk(U,idp);
idp: U,idp,SP,sk(U,idp),pk(idp),inv(pk(idp)),pk(SP);
SP:  idp,SP,pk(SP),inv(pk(SP)),pk(idp)

Actions:
U -> SP: URI
SP -> U: ID
U -> idp: {| ID |}sk(U,idp)
idp -> U: {| pk(SP),pk(idp),{U,SP}inv(pk(idp)) |}sk(U,
    idp)
U -> SP: { Ktemp,ID,{U,SP}inv(pk(idp)) }pk(SP)
SP -> U: {| Payload |}Ktemp

Goals:
SP authenticates U on ID
U authenticates SP on Payload
Payload secret between SP,U
```

Figure 2.1: A single sign-on protocol in AnB

represents trust between U and SP. U now sends that to SP along with the ID from before and a freshly generated key to use for secure communications going forward (this is all encrypted with SP's public key). SP — now trusting U — sends the payload (the service U requested originally) to the user encrypted with the fresh key from U.

Like in AIF we need to define security goals. In the Goals section we specify that SP uses ID to authenticate the user, that the user authenticates the service provider when sending the payload and that the payload is confidential between U and SP.

Chapter 3

AnB-API

3.1 Introduction to AnB-API

AnB is well suited for modelling protocols where multiple agents communicate but lacks support for modelling communications with devices that make use of Application Programming Interfaces (APIs). Various security hardware like Hardware Security Modules (HSMs), smart cards and security tokens expose an API that is used to communicate with the device. These APIs can have multiple operations that can be executed in an arbitrary order and that is where AnB falls short. These devices also often store sensitive data (such as key material) which is also impossible to model in the current implementation of AnB. The goal with AnB-API is to add these two core functionalities to AnB. AnB-API is a modelling language for security protocols that make use of APIs and persistent storage and although AnB-API is meant as an extension to AnB, in order to accommodate some of the features in AnB-API and simplify development, some AnB features were dropped, namely exponentiation and advanced channels. The language expresses different API calls in the protocol as subprotocols which can be called in any order. Data stores are supported in AnB-API in the same way as in AIF where Sets are defined over variables which in turn are defined over constants. This means that databases can be modelled where agents can insert, look up and remove stored data. In the following sections we will look at AnB-API's attacker model, sections of an AnB-API specification and look at how it is translated into AIF.

3.2 Attacker model

We utilise rules from the Dolev-Yao attacker model [4] where an adversary will overhear all transmitted messages (transmitted messages are added to the intruder knowledge) and can synthesise and send messages from their knowledge. Underlying cryptography is assumed to be sound and all cryptographic operations are known by all agents, including the adversary. Below is a formal description of the intruder model used in AnB-API. Here $iknows(\cdot)$ represents things in the intruder knowledge.

- The intruder knows all agents that take part in the protocol
$$\forall u \in Agents : iknows(u)$$
Where *Agents* contains all agents in the protocol
- The intruder knows the public keys of all agents that take part in the protocol
$$\forall u \in Agents : iknows(pk(u))$$
Where *Agents* contains all agents in the protocol
- The intruder knows the private keys of all dishonest agents that take part in the protocol. Note that this rule means that having multiple dishonest users might not be very useful unless they are all meant to cooperate to some degree.
$$\forall d \in Dishonest : iknows(inv(pk(d)))$$
Where *Dishonest* contains all dishonest agents in the protocol
- The intruder can hash messages
$$\frac{iknows(M)}{iknows(h, M)}$$
where $h \in HashConstants$
- The intruder can build composed messages and decompose messages
$$\frac{iknows(M_1) \qquad iknows(M_2)}{iknows(pair(M_1, M_2))} \qquad \frac{iknows(pair(M_1, M_2))}{iknows(M_1) \qquad iknows(M_2)}$$

- Given the right key, the intruder can symmetrically encrypt and decrypt messages

$$\frac{iknows(M) \qquad iknows(K)}{iknows(\{|M|\}_K)} \qquad\qquad \frac{iknows(\{|M|\}_K) \qquad iknows(K)}{iknows(M)}$$

- Given the right key, the intruder can asymmetrically encrypt and decrypt messages

$$\frac{iknows(M) \qquad iknows(K)}{iknows(\{M\}_K)} \qquad\qquad \frac{iknows(\{M\}_K) \qquad iknows(inv(K))}{iknows(M)}$$

- Given the right key, the intruder can sign and open signed messages

$$\frac{iknows(M) \qquad iknows(inv(K))}{iknows(\{M\}_{inv(K)})} \qquad\qquad \frac{iknows(\{M\}_{inv(K)})}{iknows(M)}$$

3.3 Sections of the language

A protocol specification written in AnB-API has six sections containing the protocol name, types, sets and facts used in the subprotocols section, the subprotocols section itself, along with an attacks section declaring our security goals. In the following sections we go into each one in detail and as an example present a key server protocol similar to the one we looked at before (The entire AnB-API key server specification can be found in appendix A).

3.3.1 Protocol name

The first section simply contains the protocol name. This is translated into the similar "Problem" section in AIF. This section starts with the keyword "Protocol" following the desired name of the protocol like so:

```
Protocol: Keyserver
```

In order to keep the code minimal and readable, AnB-API does not use semicolons at the end of lines.

3.3.2 Types

The second section defines types to be used in the specification. In the Types section we define the constants and variables used in the specification of subprotocols later on. Types follow the same format as Types in AIF where variables range over constants. Variable names start with an upper-case letter and constants with a lower-case letter, e.g.:

```
Types:
Agents       : {a,b,s,i}
Dishonest    : {i}
H            : {a,b}
S            : {s}
U            : {a,b,i}
Sts          : {valid,revoked}
PK,NPK       : value
```

Variables are declared as either a set of constants, a fresh value with the `value` keyword, or arbitrary untyped values with the `untyped` keyword. Three sets are a part of the language and are used to formulate the attacker model during compilation. These are `Agents`, `Dishonest` and `HashConstants`. `Agents` should contain all agents that participate in the protocol, honest and dishonest. `Dishonest` should contain dishonest agents that participate in the protocol and `HashConstants` should contain all hash constants that are used in the protocol. Note that all dishonest agents know each other's private keys as mentioned in section 3.2. Declaring the `HashConstants` variable is optional and only needs to be done if the protocol uses hashes (we don't use hashes in our key server example so the variable is not defined in the above code). Hash constants were introduced to allow users to use multiple hashing functions and should always be used with hashes like so: `h(h1,M)` where `M` is a message and `h1` is a hash constant.

In the code above we define our honest agents `a` and `b`, our server `s` and the intruder `i`. We declare a variable for our honest users `H`, server `S` and all users (clients of the server) `U`. We also declare a set `Sts` which represents the statuses of the keys that the server holds. `PK` and `NPK` will be used to refer to keys during the protocol run. If multiple variables range over the same set of constants or types they can be comma separated in the declaration (like `PK` and `NPK`).

3.3.3 Sets

Sets (data stores) are declared in the third section. Each set declaration has a name and a list of variables over which the set is defined (note that declaring a set over constants is not supported).

```
Sets:
ring(U), db(S,U,Sts)
```

Here we define two sets `ring(U)` and `db(S,U,Sts)`. The set `ring` represents each agent's keyring which holds that agent's keys. The `db` represents the server's database that holds keys for all users of all statuses. As in AIF, defining sets over variables like this actually represents a set for each constant that the variable is defined over. The declaration for the set `db` above actually represents 4 sets because a set is created for every combination of the constants that it's variables range over. `S` ranges over 1 constant `s`, `U` ranges over 2 constants `a` and `i`, and `Sts` ranges over 2 constants `valid` and `revoked`. This gives $1 * 2 * 2 = 4$ sets. If no sets are defined, the sets section should still be in the protocol but the set list should be empty like so (the comment is optional but recommended for the sake of clarity):

```
Sets:
#empty
```

3.3.4 Facts

A fact in AnB-API represents some event occurring. A fact can involve messages, indicating that those messages have something to do with that event occurring. The Facts section is similar to the Facts section in AIF:

```
Facts:
cnfCh/2, request/2
```

A fact declaration has a name and an integer number denoting the arity of the fact. The arity represents how many messages are used to establish the fact. Our key server example does not make use of facts so the facts in the code above are only to demonstrate what the Facts section looks like. If no facts are defined, the facts section should still be in the protocol but the fact list should be empty like so (the comment is optional but recommended for the sake of clarity):

```
Facts:
#empty
```

3.3.5 Subprotocols

In AnB-API, API calls are represented as subprotocols where each subprotocol is a list of actions. These actions can be the sending of messages to other agents or actions performed locally by an agent of the protocol. Subprotocols make up the bulk of a specification and are separated by `---`. Let's look at the subprotocols in our key server example:

```
Subprotocols:
#Honest user creates a new public key
H: create(PK)
H: insert(PK,ring(H))
H->S: sync
S: insert(PK,db(S,H,valid))
S->H: PK
---
```

The specification has three subprotocols. The first one describes an operation in which an honest user creates a new key, inserts it into their own keyring and then sends it to the server, which inserts it into their database. The sync command represents that `PK` arrives at the server without the user sending it there. This represents an out-of-band initialisation where the user might have physically visited the key server and given them the key in person. The server inserts the key into their database for `H` as a `valid` key and then sends it back to the user over the network. This final transmission serves the purpose of inserting the key into the intruder knowledge.

```
#Dishonest user creates a new public key
i: create(PK)
i: insert(PK,ring(i))
i->S: sync
S: insert(PK,db(S,i,valid))
S->i: PK, inv(PK)  #The intruder's new private key is
                   #shared with other dishonest users
---
```

The second section is similar to the first one except it deals with a dishonest user creating a public key. The process is the same except for the last line where the intruder's new private key is shared with other dishonest users (we assume that they're working together to attack the server).

```
#A user sends a new key to be registered with the server and
#the old one is revoked
U: select PK from ring(U)      #We select PK because we need to use it
U: create(NPK)                 #for signing when we send the new key
U: insert(NPK,ring(U))
U->S: {U,NPK}inv(PK)
S: if NPK notin db(S,_,_)      #The server checks if the new key
S: insert(NPK,db(S,U,valid))   #is in its database of all users
S: select PK from db(S,U,valid)#and all states before inserting
S: delete(PK,db(S,U,valid))    #it as a valid key and revoking
S: insert(PK,db(S,U,revoked))  #the old one
```

In the third subprotocol a user renews their key. The user creates a new key NPK and inserts it into their own keyring. The user then sends their identity along with the new key to the server — signed by the user's currently valid key. The if statement (explained in detail in section 3.4.1) checks that NPK is not registered for any user with any state in the database. If it's not, the server inserts it as a valid key for U. The server then deletes PK (the old key) from their valid set and inserts it into their revoked set.

3.3.6 Attacks

In this section we define security goals for the specification. We introduce the notion of a referee to do this. The referee is a reserved keyword in the language that represents the intruder in the Attacks section. To define attacks we specify conditionals that all need to evaluate to true in order for an attack to happen. Multiple attacks can be defined and are then separated by "---".

```
Attacks:
->referee: inv(PK)              #The referee receives a private key
referee: if PK in db(S,H,valid)
```

Here we state that if the referee (the intruder) is able to receive a private key whose public key is registered in the server's database as the valid key of an honest user, we have an attack.

3.4 Actions and Messages

As we can see in the key server example, AnB-API supports multiple different actions and borrows the message syntax from AnB. In the following sections we take a closer look at each AnB-API action along with the syntax for messages.

3.4.1 Actions

To allow agents to perform local operations the syntax `A: action` is used. We saw most of these actions in the key server example but `action` can be any of the following:

- `A: create(x)` Here agent `A` locally creates value `x` which needs to have been defined in the Types section. At this point `x` only exists locally with `A` and has not been saved to persistent storage. Example:

  ```
  A: create(NPK)
  ```

- `A: insert(x,s)` This is how persistent storage is used. Here, agent `A` saves variable `x` into set `s` where `s` is a set expression. Set expressions have a similar form as set declarations where the set name is followed by parentheses enclosing a comma-separated list of variables or constants. In the following example agent `A` inserts `NPK` into `ring(A)` which could indicate that the agent is inserting a newly created public key into their own keyring:

  ```
  A: insert(PK,ring(A))
  ```

- `A: select x from s` fetches a value previously saved in set `s`. This is required to perform actions on values that have been saved to persistent storage. `s` is again a set expression. In the following example `NPK` is fetched from `ring(A)`, making it available to use in other commands such as the `delete` command.

  ```
  A: select NPK from ring(A)
  ```

- `A: delete(x,s)` This command deletes value `x` from set `s`. `s` is again a set expression. `x` needs to have been declared earlier in the current scope (either with the `create` command or the `select` command). In this example, agent `A` deletes `NPK` from their keyring, where `NPK` has earlier been declared with a `select` command.

```
A: select NPK from ring(A)
A: delete(NPK,ring(A))
```

- `A: if x notin is` This conditional statement determines wether or not to continue executing actions in the subprotocol. If the condition evaluates to true, command execution will continue as usual and the next lines of the protocol will be executed. If the conditional evaluates to false, execution of the subprotocol will stop and the next subprotocol will be executed. `x` is a variable and `is` is an in-set expression. In-set expressions are similar to set expressions but are used in conditional statements. They have a bit of extra syntactic sugar to make writing these statements more robust. Similar to set expressions, they have the set name followed by parentheses enclosing a comma-separated list, but this list can contain both variable/constant names and the underscore. The statement below means that we proceed if NPK is not found in any of the sets associated with `S` and `a`. If we look at the definition of `db` from above we see that the third parameter in the definition of `db` is `Sts`. Thus, the statement checks wether NPK is found in `db(S,a,valid)` and `db(S,a,revoked)`, and if it's not in either of them, then the conditional evaluates to true and execution continues.

```
if NPK notin db(S,a,_)
```

- `A: if x in is` This is the negated form of the conditional statement above, meaning that if the condition is met subprotocol execution continues and otherwise stops.

- `A: f` Here, f denotes a fact expression. Fact expressions indicate that some event has taken place. Fact expressions have the name of the fact followed by parentheses enclosing a comma separated list of messages. The number of messages must be the same as the arity of the fact defined in the fact declaration. An example of this is below where we assume `cnfCh` has been defined in the `Facts` section with arity of 2.

```
cnfCh(A,{NA}PK)
```

- `A: if f` This is a conditional that can be used to check if a certain fact has been stated (check if some defined event has happened during protocol run). This, along with the fact statement, can be used to model out-of-band channels, e.g.:

```
referee: if cnfCh(A,(NA,NB))
```

- `_->referee: M` The rest of the actions pertain to the Attacks section. This actions means that the referee (attacker) receives M. If the referee is able to receive `M` at any point during the protocol run, this evaluates to true and we continue executing the attack.
- `referee: if x in s` This is similar to the "if in" check in the subprotocol section and means that execution of the attack continues if `x` is in set `s`.
- `referee: if x notin s` This is the opposite of the "if in" action where attack execution continues if `x` is not found in `s`.
- `referee: if f` This is similar to the subprotocol action where the referee checks if a certain fact has been stated. Attack execution continues if the fact has been stated.

3.4.2 Messages

When agents send messages to each other they use the syntax `A->B: M` in which `A` sends message `M` to `B`. The syntax of a message is the same as in AnB [8] where a message can be:

- A variable, e.g. `A->B: NA`
- A key, e.g. `A->B: NPK`
- Comma separated list of messages, e.g. `A->B: M1,M2`
- Asymmetrically encrypted message, e.g. `A->B: {M}K` where `K` is a key
- Symmetrically encrypted message, e.g. `A->B: {|M|}K` where `K` is a key
- Messages enclosed in parentheses, e.g. `A->B: (M)`
- A hashed message, e.g. `A->B: h(h1,M)` where `h1` is a constant in the HashConstants variable.

As we see later, messages that are sent between agents in the protocol are automatically added to the intruder knowledge. The keys in the above examples can be a freshly generated value (e.g. `A->B: {M}PK`), a public key (e.g. `A->B: {M}pk(A)`), a private key (e.g. `A->B: {M}inv(PK)`) or a shared key, in which case the key is shared between two agents (e.g. `A->B: {M}sk(A,B)`). `pk`, `inv` and `sk` are reserved keywords in the language and are used to refer to these keys.

3.5 Compilation to AIF

The AnB-API compiler is written in Haskell and in order to work with input code on an abstract level we use a set of Haskell data types and type synonyms to represent the code. These entities are then checked for errors and mapped to AIF commands. This mapping may be as simple as printing the input verbatim to the output file. In other cases we might need for a single AnB-API command to check different parts of the input in order to produce meaningful AIF code. In order to understand the compilation process from AnB-API to AIF, we take a brief look at the data types and type synonyms that we use in the following section.

3.5.1 Abstract Syntax Tree

The parser is responsible for abstracting the incoming AnB-API code to the aforementioned set of entities that we can work with in Haskell. This set of entities is known as the Abstract Syntax Tree (AST). The data types used in the AST are defined in the file Ast.hs. Below is a description of the AST's components.

3.5.1.1 Basic type synonyms

To make it easier to work with data types in the AST, we use a few type synonyms to represent the most basic data structures. Haskell type synonyms allow us to define names to use for data types. An identifier (`Ident`) is simply a Haskell String and is used for all symbols (constants, variables, set names, etc.). Many aspects of the compiler require dealing with lists of identifiers so we define the `IdentList` as a Haskell list that holds identifiers and finally, to hold the entire AST we define a type synonym `Protocol` as a six-tuple that corresponds to the six sections of an AnB-API specification.

```
type Ident = String
type IdentList = [Ident]
type Protocol =
(Ident,TypeDecls,SetDecls,FactDecls,Subprotocols,AttackDecls)
```

3.5.1.2 Protocol sections

As we see in the definition of `Protocol` we use other type synonyms to refer to the various sections of the protocol. The protocol name is simply an `identifier` so that's not explained further here. TypeDecls, SetDecls and FactDecls are simply lists of TypeDecl, SetDecl and FactDecl respectively. These sections host all declarations in the code and each of these can hold 0 or more declarations of each kind. A Type declaration simply consists of an identifier (the name of the type) and a Type. Type can be a set (e.g. `Agents : {a,b,i}`) a value (e.g. `NPK : value`) or untyped (e.g. `M1 : untyped`) as explained in section 3.3.2. A Set declaration consists of an identifier (again, the name of the set) and a list of identifiers corresponding to the variables over which the set is defined (e.g. `db(S,U,Sts)`). Fact declarations consist of an identifier and an integer denoting the arity of the fact (e.g. `confCh/1`).

```
type TypeDecls = [TypeDecl]
type TypeDecl = (Ident, Type)
data Type = Set IdentList
          | Value
          | Untyped

type SetDecls = [SetDecl]
type SetDecl = (Ident, IdentList)

type FactDecls = [FactDecl]
type FactDecl = (Ident, Int)

type Subprotocols = [Subprotocol]
type Subprotocol = [Action]

type AttackDecls = [Attack]
type Attack = [Action]
```

3.5.1.3 Subprotocols and Attacks

Subprotocols and Attack declarations are the bulk of the protocol specification and are a little more complex. Both are divided into multiple segments (subprotocols and attacks) that contain actions. The action data type represents all the actions that can be modelled in the protocol (detailed in section 3.4.1)

```
data Action = Insert Ident SetExp
```

```
              | Delete Ident SetExp
              | Select Ident SetExp
              | Create Ident
              | Ifnotin Ident InSetExp
              | Ifin Ident InSetExp
              | Fact FactExp
              | Iffact FactExp
              | Transmission Channel Msg
              | Sync Channel
              | ToRefAction Msg
              | RefSelect Ident SetExp
              | RefIfnotin Ident InSetExp
              | RefIfin Ident InSetExp
              | RefIffact FactExp
```

They carry names corresponding to the keywords they use in code (e.g. `Ifnotin` for `if x notin s` where `x` is an `Ident` and `s` is an `InSetExp`). ToRefAction, RefSelect, RefIfnotin, RefIfin and RefIffact are actions that can only be used in the Attacks section. The rest of the actions can only be used in the Subprotocols section. Using actions in wrong sections results in a parsing error. As we can see in the definition of Action, various expressions are used.

```
type SetExp = (Ident,IdentList)
type InSetExp = (Ident,InSetIdentList)
data InSetIdent = Ident Ident
                | Underscore Ident
type InSetIdentList = [InSetIdent]
type FactExp = (Ident,Msg)
```

These are mostly straight forward, `SetExp`, `InSetExp` and `FactExp` all have an identifier representing the set or fact name. A `SetExp` has a list of identifiers denoting variables, `InSetExp` as well except they can also be the underscore, and `FactExp` has a message (note the message can be a concatenation of multiple messages).

3.5.1.4 Messages

In order to represent messages and keys we use the following data structures:

```
data Msg = Atom Ident
```

```
            | Cat Msg Msg
            | Key Key
            | Crypt Msg Key
            | Scrypt Msg Key
            | Hash Ident Msg

data Key = GenericKey Ident
            | PublicKey Ident
            | PrivateKey Ident
            | SharedKey IdentList
```

The message is a recursive data structure that can be a concatenation of multiple messages (`Cat Msg Msg`), a key (`Key Key`), asymmetrically encrypted (`Crypt Msg Key`), symmetrically encrypted (`Scrypt Msg Key`) and a hashed message (`Hash Ident Msg` where `Ident` is the hash constant). The Key also has their own data type and can be a `GenericKey` (freshly generated key), `PublicKey`, `PrivateKey` and a `SharedKey` as explained in section 3.4.2.

3.5.2 Translation

The syntax for the Types, Sets and Facts sections of AnB-API is — aside from semicolons at the end of lines — the same as for these sections in AIF. Therefore they are printed like they are on the input with the addition of a few types and facts, and semicolons at the end of lines. As for the additional types, in order to accommodate the intruder model we add two types to the `Types` section, `IntruderRuleM1` and `IntruderRuleM2` of type `untyped`, which represent M_1 and M_2 from section 3.2 respectively. We don't need to add the key K from section 3.2 to the intruder model declaration since it would simply be a variable of type `untyped` like `IntruderRuleM1` and `IntruderRuleM2` are. We therefore simply use `IntruderRuleM2` in it's stead as it is not used anyway in rules that use a key. The facts we add are `attack/0` and `iknows/1` since both are needed in AIF and are used in a different way in AnB-API.

AIF has a section that AnB-API does not have and that is the Functions section. This section is used in AIF to define functionality like encryption and hashing. In AnB-API creating functions is not supported in order to keep the language as easy to use as possible. Instead, AnB-API supports a set of common features and the Functions section in the output AIF specification is populated with these features. These are `sign` (for message signing), `scrypt` (for symmetric encryption), `crypt` (for asymmetric encryption), `pair` (for concatenation), `pk` (to denote an agent's public key), `inv` (to denote a public key's private key), `sk` (to denote a shared key between two agents), and `h` (to use hashes). All specifications written in AnB-API thus have the following Functions declaration:

```
Functions:
public sign/2, scrypt/2, crypt/2, pair/2, pk/1, sk/2, h/2;
private inv/1;
```

The Subprotocols and Attacks sections of the AST are a little more complicated and are compiled with two recursive functions `compileSubprotocol` and `compileAttack`. These functions take as input the list of actions in each subprotocol or attack along with any other information they need in the translation process (type declarations, set declarations, etc.) and output a string — the corresponding AIF code.

Each subprotocol is compiled into at least one whole AIF rule and each AIF rule is compiled from at most one subprotocol. This means that we can compile each subprotocol separately and the compilation of a subprotocol does not influence the compilation of another subprotocol. The compiler thus calls `compileSubprotocol` on each subprotocol in the input AnB-API code which has the following type: `compileSubprotocol :: Subprotocol -> Left -> Center -> Right -> Agent -> TypeDecls -> SetDecls -> FactDecls -> String` where:

`Subprotocol` is a list of all actions in the subprotocol. As we saw earlier, `Subprotocol` is a type synonym for a list of actions: `[Action]`.

`Left` is a list of actions that will be compiled into AIF code on the left-hand-side of the arrow in an AIF rule

`Center` is a list of actions that will be compiled into AIF code on the arrow of an AIF rule

`Right` is a list of actions that will be compiled into AIF code on the right-hand-side of the arrow in an AIF rule

`Agent` corresponds to the agent in the AnB-API code whose turn it is during that rule

`TypeDecls` are the type declarations from the Types section

`SetDecls` are the set declarations from the Sets section

`FactDecls` are the fact declarations from the Facts section

`String` is the return type (The generated AIF code).

The function is called recursively with each call removing the first action from the list `Subprotocol` and adding it to one or more of the sets `Left`, `Center` and `Right`. When an action calls for a different agent to act, i.e. `Transmission` and `Sync` actions, the function is called with the `Agent` parameter updated with the receiving agent. The compiler produces an error if an action is attempted by an agent who is not allowed to act. When a message is sent or when the subprotocol has no more actions, the rule is printed. This functionality of `compileSubprotocol` is formalised in the translation rules below where `x` represents a variable, `s` represents a set expression, `is` represents an in-set expression, `f` represents a fact expression, T represents type declarations, S represents set declarations, A represents the agent performing the action, and L, C and R represent `Left`, `Center` and `Right` respectively.

insert

$$_{(T,S,F)}[\![A\texttt{:insert(}x\texttt{,}s\texttt{)}\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L,C,R')}$$

where $R' = R \cup \{\text{Insert } x\ s\}$ and $\alpha = A$

create

$$_{(T,S,F)}[\![A\texttt{:create(}x\texttt{)}\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L,C',R)}$$

where $C' = C \cup \{\text{Create x}\}$ and $\alpha = A$.

select

$$_{(T,S,F)}[\![A\texttt{:select }x\texttt{ from }s\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L',C,R')}$$

where $L' = L \cup \{\text{Select x s}\}$, $R' = R \cup \{\text{Select x s}\}$ and $\alpha = A$.

delete

$$_{(T,S,F)}[\![A\texttt{:delete(}x\texttt{,}s\texttt{)}\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L,C,R')}$$

where $R' = R \setminus \{\text{Select x s}\}$. Select x s must be in L and $\alpha = A$.

if not in

$$_{(T,S,F)}[\![A\texttt{:if }x\texttt{ notin }s\texttt{(}t_1,...,t_i\texttt{)}\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L',C,R)}$$

where $L' = L \cup \{\text{Ifnotin x is}\}$ and $\alpha = A$.

fact

$$_{(T,S,F)}[\![A\texttt{:}f\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L,C,R')}$$

where $R' = R \cup \{\text{Fact f}\}$. `f` must have been declared in F and `f` must have the same arity as it's definition in F and $\alpha = A$.

if fact

$$_{(T,S,F)}[\![A\texttt{:if }f\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L',C,R)}$$

where $L' = L \cup \{\text{Fact f}\}$. `f` must have been declared in F and `f` must have the same arity as it's definition in F and $\alpha = A$.

if in

$$_{(T,S,F)}[\![A\texttt{:if } x \texttt{ in } is\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha}_{(L',C,R')}$$

where $L' = L \cup \{\text{Select x s}\}$, $R' = R \cup \{\text{Select x s}\}$, $\alpha = A$ and s $=$ $isetose(is, S)$ where *istose* is a function that for an in-set expression looks up the corresponding set expression in S. Note that this produces the same AIF code as the select statement. It's in the language to make it more intuitive (more information about this is in 3.4.1).

sync

$$_{(T,S,F)}[\![A\texttt{->}B\texttt{:sync}\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha'}_{(L,C,R)}$$

where $\alpha = A$ and $\alpha' = \texttt{B}$. Note that this action only switches agent's turns and does not produce AIF code. It's in the language to make the code more meaningful by representing atomic actions within a subprotocol (as we saw in the key server example).

transmission

$$_{(T,S,F)}[\![\texttt{\_->}B\texttt{:}M\rho]\!]^{\alpha}_{(L,C,R)} = {}_{(T,S,F)}[\![\rho]\!]^{\alpha'}_{(L',C,R)}$$

where $L' = L \cup \{\text{Transmission "\_" B}\}$, $\alpha = A$ and $\alpha' = \texttt{B}$.

transmission

$$_{(T,S,F)}[\![A\texttt{->}B\texttt{:}M\rho]\!]^{\alpha}_{(L,C,R)} =$$
$$\{enums(L,R,T)L\texttt{=[}C\texttt{]=>}R \cup \texttt{iknows(M)}\} \cup {}_{(T,S,F)}[\![\texttt{\_->}B\texttt{:M;}\rho]\!]^{\alpha'}_{(\epsilon,\epsilon,\epsilon)}$$

where $\rho \neq \epsilon$, $\alpha = A$ and $\alpha' = \texttt{B}$ and *enums* is a function that generates lambda expressions in front of rules if needed. Every time an AIF expression is used that has variables in it we need to tell AIF to enumerate over the variables. AIF uses lambda expressions in front of the left-hand side of the rule to do this. *enum* takes the sets of actions that can contribute to the lambda expression (L and R) and the type declarations (T), finds variable sets within them and generates a lambda expression.

sendlast

$$_{(T,S,F)}[\![A\texttt{->}B\texttt{:}M]\!]^{\alpha}_{(L,C,R)} = \{enums(L,R,T)L\texttt{=[}C\texttt{]=>}R \cup \texttt{iknows(M)}\}$$

where $\rho = \epsilon$, $\alpha = A$ and *enums* generates lambda expressions for the rule.

empty

$$_{(T,S,F)}[\![\epsilon]\!]^{\alpha}_{(L,C,R)} = \{enums(L,R,T)L\texttt{=[}C\texttt{]=>}R\}$$

Where *enums* generates lambda expressions for the rule.

The `compileAttack` function is very similar to `compileSubprotocol` and works the same way. The function is called for every attack in the Attacks section where every attack produces one AIF rule and every AIF rule in the output corresponds to one AnB-API Attack. The function has the type `compileAttack :: Attack -> Left -> TypeDecls -> SetDecls -> FactDecls -> String` where `Attack` is the list of actions. AIF rules that describe attacks don't allow fresh variables to be created and only have conditions on the left-hand side of the rule, with the right-hand side containing only the fact `attack`, so `Center` and `Right` are omitted. We also don't maintain `Agent` like before since the referee is the only agent that we model in the Attack section. Below is a description of how `compileAttack` works.

referee receive

$$_{(T,S,F)}[\![\texttt{->referee:M;}\rho]\!]_{(L)} = {}_{(T,S,F)}[\![\rho]\!]_{(L')}$$

Where $L' = L \cup \{\text{ToRefAction msg}\}$

referee if in

$$_{(T,S,F)}[\![\texttt{referee:if x in is;}\rho]\!]_{(L)} = {}_{(T,S,F)}[\![\rho]\!]_{(L')}$$

Where $L' = L \cup \{\text{Select x s}\}$. As with `Ifin` before the in-set expression `is` is converted to a set expression `s`.

referee if notin

$$_{(T,S,F)}[\![\texttt{referee:if x notin is;}\rho]\!]_{(L)} = {}_{(T,S,F)}[\![\rho]\!]_{(L')}$$

Where $L' = L \cup \{\text{RefIfNotin x is}\}$

referee if fact

$$_{(T,S,F)}[\![\texttt{referee:if f;}\rho]\!]_{(L)} = {}_{(T,S,F)}[\![\rho]\!]_{(L')}$$

Where $L' = L \cup \{\text{RefIfFact f}\}$

referee empty

$$_{(T,S,F)}[\![\epsilon]\!]_{(L)} = \{enums(L, R, T)L\texttt{=>attack;}\}$$

Where *enums* generates lambda expressions for the rule (same as in the rules for `compileSubprotocol`).

As described above, when the list of actions is empty or when a message is sent, an AIF rule is printed. To print an AIF rule, we translate each action in `L` and `R` to an AIF expression and concatenate them with a "." in between each (AIF syntax for AND). The actions from `L` make up the left-hand side of a rule and the actions from `R` make up the right-hand side of the rule. Translating AnB-API actions to AIF expressions is straight-forward for most actions and below is a description of how each action is translated:

- **Insert `x s`** translates to `x in s`, e.g. `PK in ring(a)`.
- **Delete `x s`** translates to `x in s` where the rule's right-hand side does not contain `x in s` (see translation rules above), e.g. `PK in ring(a)`.
- **Select `x s`** translates to `x in s`, e.g. `PK in ring(a)`.
- **Create `x`** translates to `x`, e.g. `=[PK]=>`.
- **Ifnotin `x is`** translates to `forall` $[t'_i | t_i =' \_']$`.x notin is(`$t_1, ..., t_n$`)` and

$$t'_i = \begin{cases} t_i & \text{if } t_i \neq' \_' \\ sparam(s,i) & \text{if } t_i =' \_' \end{cases}$$

  where $sparam(s,i)$ returns the name of the i-th parameter of set s, e.g. `forall U,Sts. NPK notin db(S,U,Sts)`.
- **Ifin `x is`** is never printed since it yields two Select actions which are printed instead.
- **Fact `f`** translates to `f`, e.g. `cnfCh(A,(NA,NB))`.
- **Iffact `f`** translates to `f`, e.g. `if cnfCh(A,(NA,NB))`.
- **Transmission ("_",receiver) msg** translates to `iknows(msg)`.
- **Transmission (sender,"_") msg** translates to `iknows(msg)`.
- **Transmission (sender,receiver) msg** translates to `iknows(msg)`.
- **Sync channel** yields nothing when printed.
- **ToRefAction msg** translates to `iknows(msg)`
- **RefSelect `x s`** is the same as `select` from above. It translates to `x in s`, e.g. `PK in ring(a)`.
- **RefIfnotin `x is`** translates to `forall` $[t'_i | t_i =' \_']$`.x notin is(`$t_1, ..., t_n$`)` and

$$t'_i = \begin{cases} t_i & \text{if } t_i \neq' \_' \\ sparam(s,i) & \text{if } t_i =' \_' \end{cases}$$

  where $sparam(s,i)$ returns the name of the i-th parameter of set s. This is the same as `Ifnotin` above, e.g. `forall U,Sts. NPK notin db(S,U,Sts)`.
- **RefIfin `x is`** translates to `if x in is`, e.g. `if PK in ring(a)`
- **RefIffact `f`** translates to `if f` and is the same as `Iffact` above, e.g. `if cnfCh(A,(NA,NB))`

Chapter 4

Working with AnB-API

4.1 Modelling device behaviour

As we saw in the keyserver example (appendix A) we can use AnB-API to model multiple agents communicating but one of the main features of the language is the ability to model protocols that have a single user communicating with a pre-programmed device such as a HSM. PKCS#11 is a standard by RSA Laboratories that defines an API for security devices that store cryptographic tokens. The API is called Cryptoki[5] and specifies various operations on those tokens like encryption, decryption, random number generation, signing and more. Various security vulnerabilities have been identified in the standard[3] and in order to demonstrate AnB-API code working against an API we will take a look at a security hole in PKCS#11 called a key separation attack. The attack exploits the fact that keys in a token can be assigned multiple roles. It describes a sequence of valid Cryptoki operations that cause a sensitive key which is not supposed to leave the token unencrypted to be revealed off-token to an attacker. In PKCS#11 keys have attributes that determine their role and which operations they can be used for. These rules include *extract* (key can be wrapped and extracted off the token), *wrap* (key can be used to wrap keys (wrapping keys means encrypting them for extraction)), *decrypt* (key can be used for decryption) and *sensitive* (key may not be revealed unencrypted off-token).

```
Protocol: key_separation

Types:
Agents  : {token, i}
Dishonest: {i}
HashConstants: {h1}
Token   : {token}  #We use the token constant in the protocol and
K1,K2   : value    #the Token variable in the set declarations
M       : untyped

Sets:
extract(Token), wrap(Token), decrypt(Token), sensitive(Token)
```

Here we define the token, two keys and an arbitrary message. In order to represent key attributes we define sets for each attribute and we state that if a key is a member of a set then they possess the corresponding attribute.

In our example we include two operations, `createExtract` and `createWrap`, where keys can be generated on token and the user gets back a handle to the generated key. These represent API calls that the intruder can execute in any order:

```
Subprotocols:
#createExtract
token: create(K1)
token: insert(K1, sensitive(token))
token: insert(K1, extract(token))
token->_: h(h1,K1)
---
#createWrap
token: create(K2)
token: insert(K2, wrap(token))
token: insert(K2, decrypt(token))
token->_: h(h1,K2)
---
```

`createExtract` creates a key that has the attributes *extract* and *sensitive* and `createWrap` creates a key that has the attributes *wrap* and *decrypt*. Despite the counter-intuitive nature of these actions, both operations reflect a legitimate sequence of Cryptoki actions. In order to represent handles we use AnB-API's hash functionality since its behaviour is essentially the same as a handle's in this case.

Note that in order to represent the token's functionality and communications with it's user we use the Underscore Notation. This is useful when modelling communications with devices. By using `_->token: M` and `token->_: M` we model a message `M` being sent to the token (from whatever user communicating with it) and from the token (back to whatever user communicating with it) respectively. Here we are simply expressing how the token behaves when talked to.

In order to perform the attack we implement two on-token operations that represent functionality specified in Cryptoki. These are *wrap* and *decrypt*:

```
#wrap
_->token: h(h1,K1),h(h1,K2)
token: if K1 in extract(token)
token: if K2 in wrap(token)
token->_: {|K1|}K2
---
#decrypt
_->token: h(h1,K2),{|M|}K2
token: if K2 in decrypt(token)
token->_: M
```

The wrap operation takes a handle to a key K_2 that can be used for wrapping, and a handle to a key K_1 that can be extracted and returns K_1 wrapped with K_2. The decrypt operation takes a handle to a key K_2 that can be used for decryption and an encrypted message and returns the message decrypted with K_2. Finally we declare that if the referee can somehow receive an unencrypted key that has the attribute *sensitive* an attack takes place.

```
Attacks:
->referee: K1
referee: if K1 in sensitive(token)
```

In order to perform the attack the intruder creates a key K_1 that is both sensitive and extractable. The intruder then creates another key K_2 that can be used to decrypt and can wrap other keys. The intruder now simply calls the `wrap` operation and extracts our sensitive (and wrappable) key K_1 encrypted with K_2. The intruder now has $\{|K_1|\}K_2$ and can simply feed that along with K_2 into the `decrypt` operation which decrypts the message and returns to the intruder our sensitive key K_1. The full AnB-API specification for the key separation attack can be found in appendix B.

The key separation specification described here is a good example of a protocol that can not be modelled in AnB. Instead of describing sequential communications between multiple users, we simply model how a token responds when different operations are called and those operations can be called in any order. Moreover, the token maintains a database of keys with different attributes that can change and persist throughout the entire run of the protocol.

4.2 Compiler

The AnB-API compiler is written in Haskell. The lexical analyser generator Alex was used to generate a lexical analyser for the language and the parser generator Happy was used to generate a parser. Both tools along with the Glasgow Haskell Compiler (GHC) are included in the Haskell platform which is freely available at `https://www.haskell.org/`. The AnB-API compiler's source code and an executable for Mac OS X can be found at `https://github.com/jbb10/AnB-API` along with the two examples in appendix A and B. The source code resides in the `/src` folder along with a Makefile for easy compilation and the examples are in the `/examples` folder. To compile the compiler simply issue the `make` command from the `/src` folder. For the compilation to succeed, `ghc`, `alex`, and `happy` need to be in the user's PATH variable. The compiler was written and compiled using version 7.8.3 of GHC.

The compiler yields the `anbapi` program which is the compiler. To use it simply type `./anbapi keyserver.anbapi` to compile a file `keyserver.anbapi`. The output is a file containing AIF code with the same file name ending in `.aif` (e.g. `keyserver.aif`). The `.aif` file can be compiled into Horn clauses with the `fpaslan` tool and the output from that can then be input into the SPASS tool for verification. A script for Mac OS X is included in the `/tools` folder called `checkAIFprotocol` to make this process more convenient. It takes a `.aif` file as input (e.g. `./checkAIFprotocol keyserver.aif`) and prints the output from the SPASS verification tool. The script assumes that the `fpaslan-mac` and `SPASS` tools are in the user's PATH variable.

Chapter 5

Conclusion

We set out to extend AnB with API support and persistent storage. To achieve this we added the notion of operations performed locally by agents along with new syntax and semantics to support it. The AnB syntax was used as a basis for the new syntax and the message syntax in AnB is retained in its original form. In order to interact with preprogrammed devices we introduced the Underscore notation which allows device behaviour to be modelled directly. Utilising AIF's set features we also added support for simple data stores to be modelled allowing databases and long-term storage to be expressed in protocol specifications. We provide a formal description of how the semantics in AnB-API are translated into AIF. In [7] we're also provided with with a formal definition of AIF and how it is translated into Horn Clauses giving us a solid and clear description of what AnB-API specifications do and what they produce.

5.1 Future work

In order to keep focus on the language and it's functionality some common compiler features were omitted. The compiler catches syntactic errors and many semantic errors as well but producing line numbers along with the errors was not implemented. Variable names used in the attacker model are also reserved keywords (`intruderRuleM1` and `intruderRuleM2`) in the implementation and as part of future work could simply be generated at runtime. Support for exponentiation is useful for modelling e.g. Diffie-Hellman key exchange and support for that was dropped in order to simplify development. This is something that should be introduced to further extend the language's usability.

Bibliography

[1] Deliverable AVISPA. D2. 3: The intermediate format. `http://www.avispa-project.org/delivs/2.3/d2-3.pdf`, 2003. [Online; accessed 17-June-2016].

[2] Bruno Blanchet, Ben Smyth, and Vincent Cheval. Proverif 1.90: Automatic cryptographic protocol verifier, user manual and tutorial. `http://prosecco.gforge.inria.fr/personal/bblanche/proverif`, 2015. [Online; accessed 14-June-2016].

[3] Jolyon Clulow. *On the Security of PKCS #11*, pages 411–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[4] Danny Dolev and Andrew C Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[5] RSA Laboratories. Pkcs11: Cryptographic token interface standard. `https://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm`, 2016. [Online; accessed 23-June-2016].

[6] Sebastian Mödersheim and Luca Viganò. *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*, chapter The Open-Source Fixed-Point Model Checker for Symbolic Analysis of Security Protocols, pages 166–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[7] Sebastian Alexander Mödersheim. Abstraction by set-membership: Verifying security protocols and web services with databases. In *Proceedings of*

the 17th ACM Conference on Computer and Communications Security, CCS '10, pages 351–360, New York, NY, USA, 2010. ACM.

[8] S. Mödersheim. Algebraic properties in alice and bob notation. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 433–440, March 2009.

[9] Christoph Weidenbach, Renate A Schmidt, Thomas Hillenbrand, Rostislav Rusev, et al. System description: Spass version 3.0. In *Automated Deduction–CADE-21*, pages 514–520. Springer, 2007.

Appendix A

Key server specification in AnB-API

The following page contains an example of a protocol written in AnB-API. It describes a simple key server protocol in which a server stores public keys for users. A user may submit a key to the key server which will store it for the user. If the user submits a new key (signed by a currently valid key), the server will check if the new key is in it's database of already existing valid or revoked keys (for any user) and if not, it will store the new key and mark the old one as revoked. Finally we define an attack state in which the intruder intercepts a private key which is stored in the server's database for an honest user and a valid key. See the comments in the code and section 3.3 for more detailed information. When this code is compiled with the AnB-API compiler and the fpaslan tool, and verified with the SPASS tool the following result is reached (SPASS output is trimmed for clarity). We see that SPASS produces "`SPASS beiseite:  Completion found.`" meaning that the protocol is secure.

```
...
SPASS V 3.5
SPASS beiseite: Completion found.
Problem: keyserver.aif.spass
...
--------------------------SPASS-STOP------------------------------
```

```
Protocol: Keyserver

Types:
Agents      : {a,b,s,i} #Mandatory; contains all agents in the protocol
Dishonest   : {i}       #Mandatory; contains dishonest agents in the protocol
H           : {a,b}     #These are our honest users
S           : {s}       #This is our server
U           : {a,b,i}   #These are our users
Sts         : {valid,revoked} #These are statuses for our keys
PK,NPK      : value     #Keys used during protocol run

Sets:
ring(U), db(S,U,Sts)    #Our sets; user keyring and server database

Facts:                  #This protocol does not make use of facts. We
#empty                  #leave an optional comment there for clarity.

Subprotocols:
#Honest user creates a new public key
H: create(PK)
H: insert(PK,ring(H))
H->S: sync
S: insert(PK,db(S,H,valid))
S->H: PK
---
#Dishonest user creates a new public key
i: create(PK)
i: insert(PK,ring(i))
i->S: sync
S: insert(PK,db(S,i,valid))
S->i: PK, inv(PK)  #intruder shares their new private key with dishonest users
---
#A user sends a new key to be registered with the server and
#the old one is revoked
U: select PK from ring(U)      #We select PK because we need to use it
U: create(NPK)                 #for signing when we send the new key
U: insert(NPK,ring(U))
U->S: {U,NPK}inv(PK)
S: if NPK notin db(S,_,_)      #The server checks if the new key is in its
S: insert(NPK,db(S,U,valid))   #database of all users and all states before
S: select PK from db(S,U,valid)#inserting it as a valid key and revoking
S: delete(PK,db(S,U,valid))    #the old one
S: insert(PK,db(S,U,revoked))

Attacks:
->referee: inv(PK)              #The referee receives a private key
referee: if PK in db(S,H,valid)
```

Appendix B

Key separation in AnB-API

The following page contains an example of a protocol written in AnB-API. It is meant to give an example of communications with a HSM (hardware security module) and demonstrates a key separation attack. In the specification we have a token which the user can interact with using API calls. The token is used to store keys, decrypt and encrypt messages. The intruder is able to extract a sensitive key from the token using a combination of the `wrap` and the `decrypt` operations on keys that have multiple roles (more information in section 4.1). When this code is compiled with the AnB-API compiler and the fpaslan tool, and verified with the SPASS tool the following result is reached (SPASS output is trimmed for clarity). We see that SPASS produces "`SPASS beiseite: Proof found.`" meaning that an attack is present in the protocol.

```
...
SPASS V 3.5
SPASS beiseite: Proof found.
Problem: keyseparation.aif.spass
...
--------------------------SPASS-STOP------------------------------
```

```
Protocol: key_separation

Types:
Agents  : {token, i}
Dishonest: {i}
HashConstants: {h1}
Token   : {token}  #We use the token constant in the protocol and
K1,K2   : value    #the Token variable in the set declarations
M       : untyped

Sets:
extract(Token), wrap(Token), decrypt(Token), sensitive(Token)

Facts:
#empty

Subprotocols:
#createExtract
token: create(K1)
token: insert(K1, sensitive(token))
token: insert(K1, extract(token))
token->_: h(h1,K1)
---
#createWrap
token: create(K2)
token: insert(K2, wrap(token))
token: insert(K2, decrypt(token))
token->_: h(h1,K2)
---
#wrap
_->token: h(h1,K1),h(h1,K2)
token: if K1 in extract(token)
token: if K2 in wrap(token)
token->_: {|K1|}K2
---
#decrypt
_->token: h(h1,K2),{|M|}K2
token: if K2 in decrypt(token)
token->_: M

Attacks:
->referee: K1
referee: if K1 in sensitive(token)
```