# An Automatic Protocol Composition Checker

Ivana Kojovic


Kongens Lyngby 2012
IMM-M.Sc.-2012-52

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2012-52

# Summary

Formal analysis is widely used to prove security properties of protocols. There are tools to check protocols in isolation, but in fact we use many protocols in parallel or even vertically stacked, e.g. running an application protocol (like login) over a secure channel (like TLS) and in general it is unclear if that is safe. There are several works that give sufficient conditions for parallel and vertical composition, but there exists no program to check whether these conditions are actually met by a given suite of protocols.

The aim of the master thesis project is to implement a protocol composition checker and present it as a service for registering protocols and checking compatibility of the protocols among each other. In order to establish the checker, it is necessary to collect and integrate different conditions defined throughout the literature. Also, we will define a framework based on Alice and Bob notation, so the checker can examine protocols in an unambiguous manner.

Further we will develop a library of widely-used protocols like TLS that are provenly compatible with each other and define a set of negative example protocols to test the checker.

We implement the checker as an extension of the existing Open-Source Fixed-Point Model-Checker OFMC to easily integrate our composition checker with an existing verification procedure that supports Alice and Bob notation.

# Preface

This thesis is a part of a double degree Master program in Security and Mobile Computing (NordSecMob). I spent first semester on Norwegian University of Science and Technology (NTNU) and then I continued with the program on Technical University of Denmark (DTU). The thesis was prepared on the Department of Informatics and Mathematical Modelling on DTU, under the main supervision of Sebastian Alexander Mödersheim and co supervision Danilo Gligoroski from NTNU.

Lyngby, 29-June-2012

Ivana Kojovic

# Acknowledgements

Special appreciation goes to my supervisor Sebastian. His sound advices, good ideas and friendly atmosphere, provided enough encouragement and helped me to pass the thesis process less stressing. It was a pleasure working with such a dedicated researcher and a great person.

Also, I would like to thank to my co supervisor Danilo Gligoroski for contribution. Special thanks to the officers in Finland and Norway for the full support during whole master program.

Sincere thanks to all my friends and my boyfriend for unforgettable days spent in Northern Europe.

I dedicate this thesis to my wonderful family whose unconditional love and advices helped me throughout my master studies.

# Contents

Chapter 1

# Introduction

A rapid development of applications on the Internet brought the need for secure communication. That is why security protocols are implemented: they strictly define set of rules in the communication between a numbers of parties in order to assure secrecy, authenticity and integrity of the data. Assuring security goals using security protocols is a demanding task and in the previous 30 years this has been dynamic research area of a computer science.

Protocols can be described in purely formal manner. We prefer formal description because that can automate protocol verification of security features. Usually, formal descriptions consider message representation using term algebra. Dolev and Yao provided a paper [1] which main principles are still used today. In this model, we assume that cryptography primitives are perfect: message-forming operation is treated as symbolic application of function symbol to atomic abstractions of nonce, names, keys, etc. Dolev-Yao model empowers its attacker aka. intruder by setting deduction system : fixed set of his capabilities to manipulate messages. The intruder is superior in the protocol:

- he can intercept any message on the communication medium
- he can initiate protocol runs at any point of time with some party
- has has unlimited computational storage for data on which he applies inference rules

The intruder knowledge can be presented as a constraint system. Checking the confidentiality property is equal to finding a solution for a set of constraint and that is NP-complete problem, considering one specific scenario. When the number of sessions is unbounded, secrecy checking becomes even an undecidable problem.

Even when we prove security of protocols in isolation, it is not clear whether their composition is secure. This can cause problems when we run many protocols in the same environment in parallel or vertical (stack) composition. Keys used in one single protocol can be a result of key establishment of other protocol and there is no guarantee that such composition is secure. Further more, we are never sure how protocols are interacting and whether messages can be confused and misunderstood by some of the participants.

This is the central concern of the thesis. We collect some of the conditions through literature and implement checker for compositionality. First chapter defines algebraic preliminaries necessary for data structures used in checker. Later on, we present some of the techniques and algorithms that we used in application. The checker is tested against Clark-Jacob library where we get both gives negative and positive results.

## 1.1 Protocol composition

Composing different types of security protocols and using established keys is very common on the Internet. The OSI model provides security protocols on the layers where applications can use underlying mechanisms. Such examples are: TLS [2] and its predecessor SSL inside transport layer, IPSec on network layer where it is widely used to corporate security to VPNs, etc. One can arbitrarily compose protocols: an application can establish another secure channel over the established VPN or another TLS connection over existing one (self-composition). One example is given in figure 1.1 Here, applications (smtp, mail, svn) are running over the established keys over the channels protocols painted in green. This represents *vertical composition* and it is indicated with a red color. *Parallel* protocol composition corresponds to the light blue. For a composition like 1.1 is, it is unclear whether they are secure in composition and claim that it is secure although protocols in isolations, like TLS and SSH, are.

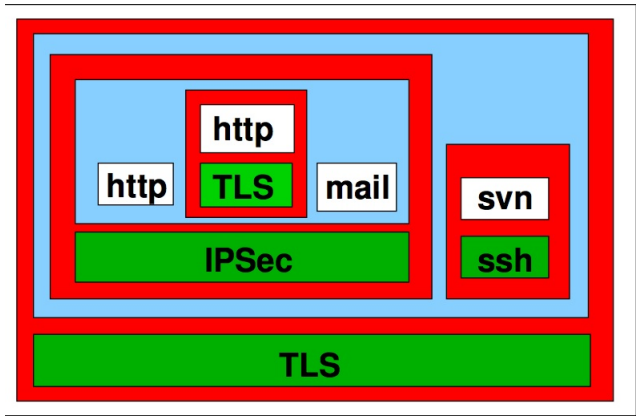


**Figure 1.1:** Protocol composition example

### 1.1.1 Vertical protocol composition

As we could see from the example above, vertical composition consists of channel protocol and application running over it. *Channel protocol* is a key exchange protocol between two parties which provides 2 symmetric keys, for each direction of the communication between parties. This kind of channel protocols has to follow design principles such as usage of fresh nonces per session. *Application protocols* are any kind of protocols that can run over the channel. We distinguish 2 types of application protocols: *abstract* and *concrete*. Abstract ones are relying on the security mechanisms of the channel, while later ones are having built-in features that preserve secrecy or authenticity of data and they are secure in isolation.

### 1.1.2 Parallel protocol composition

Every time we run protocols over the same medium we say that we are dealing with parallel composition. Protocols that are composed in parallel may also share the same key infrastructure such as: public keys and long term shared keys.

## 1.2 Potential problems with protocol composition

Most of the techniques for security protocol verification are focused on the single protocol in isolation without considering protocols that are run in parallel or on top of each other. Compositions are often too complicated system to analyse and verification methods cannot scale to that level. Protocols can be composed in infinite different ways where automated verification is not possible.

The easiest approach for handling the security of the composition is *to establish preconditions for single protocols which will give guarantee that their composition will remain secure.* Setting up the conditions is not straight forward but we can get an idea from couple of examples.

[3] considers parallel composition conditions. Intuitively, we guess that protocols running over the same medium need to have different formats so intruder cannot manipulate with the message formats. Lets look at the example with $P_1$ and $P_2$ protocol:

$$P_1 : A \to B : asycrypt(pk_B, s) \qquad P_2 : A \to B : asycrypt(pk_B, Na)$$
$$B \to A : Na$$

First protocol $P_1$ just sends a secret asymmetrically encrypted with B's public key $pk_B$. In second protocol $P_2$, agent A sends nonce asymmetrically encrypted also with B's public key. Agent B then acknowledges the message by returning the encrypted nonce. By running these 2 protocol in parallel, secrecy of protocol $P_1$ is violated: intruder intercepts the message from $P_1$ and forwards it to $P_2$ protocol which is used to decrypt the messages of this format. [3] explains and proves the solution for the attack. Main idea is tagging the similar sub-message formats with different constants so that intruder cannot perform unification between messages. Fixed protocol looks like this:

$$P_1 : A \to B : asycrypt(pk_B, [1, s]) \qquad P_2 : A \to B : asycrypt(pk_B, [2, Na])$$
$$B \to A : Na$$

where square brackets annotate concatenation of messages. Similarly, [4] gives an example where vertical composition fails. This protocol is defined with hand-

shake:

$$A \rightarrow B : asycrypt(pk_B, [A, B, sk_{(A,B)}])$$
$$B \rightarrow A : symscrypt(sk_{(A,B)}, crypt(pk_A, [B, A, N]))$$

where afterwards every message that has message M as payload and N as session identifier, looks like:

$$A \rightarrow B : [N, symcrypt(sk_{(A,B)}, M)]$$

This protocols has an attack under a self-composition. The self-composition is defined in [4] and presents the situation where established channel establishes another channel over existing one. In this example , attack trace looks like:

$$\text{a} \rightarrow \text{b} : asycrypt(pk_b, [a, b, sk_{(a,b)}])$$
Step 1 of the protocol

$$\text{b} \rightarrow \text{a} : symcrypt(sk_{(b,a)}, asycrypt(pk_a, [b, a, n]))$$
Step 2 of the protocol

$$\text{a} \rightarrow \text{b(i)} : [n, symcrypt(sk_{(a,b)}, asycrypt(pk_b, [a, b, sk_{(a,b)}]))]$$
Step 1 of the protocol, over established channel but intercepted by intruder

$$\text{b} \rightarrow \text{a(i)} : asycrypt(pk_a, b, a, sk_{(b,a)})$$
Step 1 initiated by b but intercepted by intruder

$$\text{a(i)} \rightarrow \text{b} : symcrypt(sk_{(a,b)}, asycrypt(pk_b, [a, b, sk(a, b)]))$$
Replay of the intercepted message, with $sk_{(A,B)}$ as session identifier

$$\text{b} \rightarrow \text{a(i)} : [sk_{(a,b)}, symcrypt(sk_{(a,b)}, ...)]$$
B sends a message where intruders gets shared key!

Like we stated in the definition of channel protocol, this protocol needs freshly generated keys for the session and both parties must contribute to those keys. Correspondingly to parallel composition in [3], here we also demand different message formats and additionally one more condition: disjointness from message encryption. Disjointness from message encryption ensures that we cannot confuse one message with its own encryption. In the attack above, we see that message in the protocol can be unified with message on the established channel

(encrypted with channel keys). In order to prevent this, we tag the body of the symmetric encryption in the second message. Then, fixed protocol looks like:

$$A \rightarrow B : asycrypt(pk_B, [A, B, sk_{(A,B)}])$$
$$B \rightarrow A : symcrypt(sk_{(A,B)}, [\mathbf{tag}, asycrypt(pk_A, [B, A, N])])$$

## 1.3 Motivation for the checker

Following the results from a few research articles [4], [5], [3], the idea is to translate the composability conditions from the literature and implement an Automatic Protocol Composition Checker (APCC). The current state of art has lots of issues: there are different models with too complicated notation and composability conditions are not trivial to understand and check. For this purpose, we need to establish set of conditions that are sufficient to deicide composition security. We want a simplification of theoretical conditions which will enable system administrators and protocol designers to easily and automatically check protocol composability. Therefore, APCC is given a set of protocol descriptions as an input in the widely used Alice and Bob notation and it returns the answer whether it is possible to compose the protocols. Next chapter will describe message model, data structures and certain details of the implementation process.

Chapter 2

# Preliminaries

In this chapter we describe in detail our model of protocol messages. This model is at the core of composability preconditions which are translated into implementation of checker. [4] also defines intruder model which is crucial for formal a prove of the composability result. Further more, we will review standard unification procedure which is one of the fundamental operations over the messages for a conditions checking.

## 2.1 Algebraic protocol model

Messages exchanged between agents are represented in term algebra $\mathcal{T}$ where:

- $\Sigma$ represents countable set of signature: symbols and constants all written with lower-case letters
- $\mathcal{V}$ is a countable set of variables, written with upper-case letters
- $\Sigma \cap \mathcal{V} = \emptyset$

The set of signatures $\Sigma$ is a partition set and consists of $\Sigma_0$ - the set of all constants (agents, keys and nonces), $\Sigma_p$ - the set of public operations that every

agent can perform and $\Sigma_m$ - the set of private mappings.

### 2.1.1 Public operations

Operations $\Sigma_p$ over the message terms can be performed by any agent. In this model we use:

- $asycrypt(pk, m)$ asymmetric encryption of message $m$ with public key $pk$, while $crypt(inv_{pk}, m)$ represents signing the message $m$ using private key $inv_{pk}$
- $symcrypt(k, m)$ symmetric encryption of message $m$ with symmetric key $k$ where we assume that this function also includes integrity protection such as MAC
- $hash(m)$ cryptographic hash function
- $[m_1, ..., m_n]_n, n \geq 2$ concatenation of $n$ messages. Different arity of messages eliminates basic type-flaw attacks. For instance in [6], one of the attacks in Otway-Rees protocol is possible because of the non-defined length of concatenation of messages

### 2.1.2 Private mappings

Set $\Sigma_k$ contains functions such as $inv_k$ (the private key of corresponding public key), which is obviously not a public operation. They are treated as special mappings in the model, but not as regular functions over symbols in the $\Sigma_0$. We call them *basic terms* and as consequence, mappings like $pk_A$, $sk_A$ ,$inv(pk_A)$ are treated as atomic terms together with all constants and variables.

## 2.2 Unification algorithm

The unification problem for a set of pairs of terms is concerned about the possibility of replacing term variables with some terms in order to obtain syntactically [1] equal terms. If two terms are unifiable, we can always find a solution i.e. substitution that makes two terms identical. We call this substitution *unifier*.

[1] We are only interested in syntactical unification of first-order terms

Sometimes, there can be more than one solution to unification problem but we are only interested in *most general unifier (**mgu**)*, where all the other unifiers are formed with instantiation from the general one.

Unification is a central function that helps identifying similar message formats (recall the chapter 2 and examples). We will mention the basic definitions and unification algorithm from [7] used in the implemented checker.

### 2.2.1 Definitions

We operate over the defined term algebra $\mathcal{T}(\Sigma, \mathcal{V})$ where $\mathcal{V}$ defines set of variables and $\Sigma$ all function symbols in term definition. Constants are always functions with arity 0 (which is exactly $\Sigma_0$ from above defininion). A *Substitution* is a mapping from variables to terms. We define application of substitution $\sigma$ to a term $t$ as $t\sigma$ where:

$$t\sigma := \begin{cases} x\sigma & \text{if } t = x \\ f(t_1\sigma, ..., t_n\sigma) & \text{if } t = f(t_1, ..., t_n) \end{cases}$$

Second case of the definitions also allows $n = 0$. Then, $f$ is a constant symbol and $f\sigma = f$ A substitution is usually presented with a set of bindings between variables in domain and terms: $\{x_1 \mapsto s_1, x_2 \mapsto s_2, ..., x_n \mapsto s_n\}$ and it can be applied to set of terms or set of equations in same manner. For example, for a term $t = f(a, b)$ and substitution $\sigma = \{a \mapsto x, b \mapsto y\}$, application of substitution $\sigma$ to a term $t$ is defined as $t\sigma = f(x, y)$

**Definition 3.1** A *substitution* $\sigma$ is more general than substitution $\sigma'$ if there is a substitution $\delta$ where $\sigma' = \delta\sigma$. Then, $\sigma'$ is an instance of $\sigma$ and we write $\sigma \lesssim \sigma'$

**Definition 3.2** A unification problem is a finite set of equations $S = \{s_1 =^? t_1, ..., s_n =^? t_n\}$ A *unifier* or *solution* of $S$ is a substitution $\sigma$ such that $s_i\sigma = t_i\sigma$ for $i = 1, ..., n$. $\mathcal{U}(S)$ denotes set of all unifiers of $S$. $S$ is unifiable if $\mathcal{U}(S) = \emptyset$. A substitution $\sigma$ is a *most general unifier (mgu)* of $S$ if $\sigma$ is a least element of $\mathcal{U}(S)$:

- $\sigma \in \mathcal{U}(S)$
- $\forall \sigma' \in: \sigma \lesssim \sigma'$

[7] gives necessary conditions for proving what is *mgu*. First, it proves a quasi-order relation over all substitutions. Then, it sets up lemma for idempotent

substitutions (the one where $\sigma = \sigma\sigma$). Finally, it states with theorem that *unification problem S has a solution when it has idempotent mgu but only up to renaming.* This means that even idempotent mgus are not unique, as an example $\{a \mapsto b\}$ and $\{b \mapsto a\}$ are both idempotent mgus of $a =^? b$

### 2.2.2 Unification algorithm - rule based approach

Given a set of pairs of terms, we want to determine algorithmically whether this unification problem has solution. Also, when a solution exists we want to compute *mgu*. [7] presents a rule based approach where the algorithm is an inference system.

Every idempotent substitution $\{x_1 \mapsto t_1, ..., x_n \mapsto t_n\}$ can be represented in its *solved form*: if all the $x_i$ are pairwise distinct variables and none of which occurs in any of the $t_i$.

The algorithm starts with set of unification problems that need to be solved, namely $P$. At every point the algorithm state is either $\perp$ (no solution), or it is determined with pair $P$ of unification problems and set $S$ of corresponding equations in solved form. We say that there is unifier (solution) to the system $P; S$ if it unifies every equation in both $P$ and $S$. The system state $\perp$ represents that there is no unifier.

In order to find the **mgu** of the system, we apply transformation rules:

- **Delete rule** : $\{t =^? t\} \cup P'; S \Rightarrow P'; S$ expresses that the trivial equation can be deleted from the set
- **Decompose rule** :
  If $f = g$ then $f(t_1, ..., t_n) =^? g(p_1, ..., p_n) \cup P'; S \Rightarrow \{s_1 =^? p_1, ..., s_2 =^? p_n\} \cup P'; S$ . The rule includes subterms of composed terms into the list of $P$
- **Orient rule** : If $t$ is not a variable then $\{t =^? x\} \cup P'; S \Rightarrow \{x =^? t\} \cup P'; S$. The rule moves variables to the left-hand side of equation.
- **Eliminate rule**: If variable $x$ **does not** occur in term $t$ then $\{x =^? t\} \cup P'; S \Rightarrow P'\{x \mapsto t\}; S\{x \mapsto t\} \cup \{x = t\}$. Notation for the solved form. The rule enforces substitution of variable $x$ with term $t$ in both sets and includes this equation as part of the solution while taking it out of the set of problems.

Initially, we set up system $\{x =^? t\}; \emptyset$ and we apply the transformation rules. Rules are picked arbitrarily and the order of applying them does not affect the

result at the end. It is proven that this algorithm always terminates leaving the system in two possible states: either in $\bot$ where we cannot apply some of the rules above (some preconditions are maybe violated) or in $\emptyset; S$ where $P$ set is exhausted and $S$ is in the solved form.

## 2.3 Preconditions for composition analysis

As summarized in previous chapter, composability result usually requires a set of conditions on protocols, that are proved sufficient for their secure composition. In the following pages, we will emphasise these condition from theoretical point of view. Ideas for the checker implementation relies on these preconditions.

### 2.3.1 Security in isolation

Unsurprisingly, we cannot allow insecure protocols in the composition. We have seen that security of protocols in isolation alone does not necessary imply secure composition.

Security of each of the protocols can be checked with a variety of automatic tools. Some of them are using static analysis approach, like LySa tool while some are using model checking where protocols are executed bounded number of session. One of the tools from model checking group is *OFMC* (Open-source Fixed-point Model Checker) [8].

The experimental phase will check the secuity in isolation using OFMC and even more, protocol composition checker will become one of the options in OFMC software package.

### 2.3.2 Format-type safe (FTS) protocols

Looking back to the examples in chapter 1, we see that different message formats in protocols are one of the conditions for composability result. Therefore, we pay attention on the type-flaw attacks. Type-flaw attacks are possible when intruder convince agent that message format have different type than intended. In [5] it is explained how these attacks can be prevented by typing and tagging messages. [4] goes further and demands distinct message formats even on the level of all non-variable subterms. We call this format-type safe (FTS) definition

of the protocol. We require that a type is define for every message: basic types for atomic terms:

$$\{agent, nonce, symkey, pubkey, privkey, tag\}$$

We annotate that term $t$ has type $\tau$ as $t : \tau$. Functions are defined as *composed types*: if $(t_1 : \tau_1), ..., (t_n : \tau_n)$ then $f(t_1, ..., t_n) : f(\tau_1, ..., \tau_n)$ where $f \in \{conc, crypt, scrypt, hasht\}$. As an example, message $asycrypt(p_k, [NA, A])$ has a type $crypt(pubkey, conc(nonce, agent))$.

**Definition:** Let $MP(P)$ be all the message patterns - all sent and received non-variable messages within protocol. Every element of $MP(P)$ is $\alpha$-renamed so that different messages have different variables. We call a protocol *format-type safe* when:

1. $MP(P) \cap \mathcal{V} = \emptyset$, messages are either functions or constants
2. for every 2 non-variable subterms $m_1, m_2 \in MP(P)$ with different types, there exists no unifier.

For instance a protocol with

$$\begin{aligned} MP(P) =\{&([asycrypt(pk_A, [NA, B]), A] : conc(crypt(pubkey, conc(nonce, agent)), agent),\\ &[NB, B] : conc(nonce, agent))\} \end{aligned}$$

is not FTS because non-variable subterms $asycrypt(pk_A, [NA, B]) : conc(nonce, agent))$ and $NB : nonce$ have different intended types and they are unifiable while:

$$\begin{aligned} MP(P) =\{&([hash[NA, B]), A] : conc(hasht(nonce, agent), agent),\\ &[NB, B, NA] : conc(nonce, agent, nonce))\} \end{aligned}$$

is FTS.

Unless protocol is FTS, it cannot be considered to participate in composition. This is the first implemented condition in the APCC.

### 2.3.3 Disjointness and DISE condition

While the previous ensures that we don't have type flaw attacks on a single protocol, we must make sure that messages are different from its own encryptions. There can be situations where we run some protocol over the channel

where every message is additionally encrypted with a channel key. In that case, we have to check that encrypted message is not getting confused with already existing messages defined in a protocol. This is a precondition for the vertical composition presented in [4] and based on that, we adapt the following definition:

**Definition:** Let $P$ be a protocol and $K_1, K_2, ..., K_n$ be variable symbols that do not occur in $MP(P)$. Then the *message patterns with encryptions of* $P$ is defined as:

$$EMP(P) = \alpha(\bigcup EMP_n(P))$$
$$EMP_0(P) = MP(P)$$
$$EMP_{n+1} = \alpha(\{symcrypt(K_{n+1}|m \in EMP_n(P))\})$$

where $\alpha(M)$ indicates variable renaming of the elements of $M$ (taking into account the $K_1, ..., K_n$ names) so that different elements have different variable names. Then, $P$ is called *disjoint from its own encryption* (DISE) iff there is no unifier between the $EMP_i$ and $EMP_j$ where $i \neq j$.

In [4] there is no bound on the number $n$ but for purpose of the checker we are bounding $n$ to total *depth of encryption* in the protocol.

$$depth = \max_{M \in MP(P)} \{n| \text{ number of nested symmetric encryptions in message } M\}$$

The depth of encryption presents expected upper bound for the number of $EMP_i$-s e.g. in the checker there is no need to encrypt messages more than *depth* times to find out whether we can unify some existing message with it. Let's look at the example where message pattern contains these 2 messages:

$$\begin{aligned} MP(P) = &\{symcrypt(k1, symcrypt(s_k(A,B), [NA, A])) \\ &: scrypt(symkey, scrypt(symkey, conc(nonce, agent))), \\ &[NB, B] : conc(nonce, agent)\} \end{aligned}$$

We must identify whether these messages are allowed to be in the same message pattern set. First of all, we easily notice that depth of encryption of $MP(P)$ is equal to 2, so we will make $EMP_1$ and $EMP_2$ as encrypted patterns and discover that we can unify messages $symcrypt(k1, symcrypt(s_k(A,B), [NA, A])) \in MP(P) = EMP_0$ and $symcrypt(KEY1, symcrypt(KEY2, [NB1, B1])) \in EMP_2$. This means that we cannot allow these 2 messages in the $MP(P)$ and we can either modify second message by adding tag, $[tag, NB, B] : conc(tag, nonce, agent)$ or change the order in concatenation, $[B, NB] : conc(agent, nonce)$.

The previous example shows why we bound number $n$ to the depth of encryption and shows that encrypting $MP(P)$ more than 'depth of encryption' times will be irrelevant.

**Disjointness of protocols:** Let $EST(P)$ present all non-atomic subterms of $EMP(P)$, again $\alpha$ renamed. We say that set of protocol $R$ is pairwise disjoint if for every two protocols $P_i$ and $P_j$ there is no unifier between the sets $EST(P_i)$ and $EST(P_j)$, $i \neq j$

We compose $EST(P)$ as the union of all $EMP_i$s and $NVST(MP(P))$ non-variable subterms of $MP(P)$:

$$EST(P) = MP(P) \cup EMP_1(P) ... \cup EMP_n(P) \cup NVST(MP(P))$$

This is how we achieve disjointness. Namely, every $EST$ set characterize each protocol and every element of one $EST$ must not be unifiable with elements in the other $EST$.

**Parallel composition:** The last definition will be also related to conditions in the parallel composition. Then, we will be sure that no similar message formats will be confused on the same medium. Additionally, inside parallel composition we have to take care of long-term secrets in each of the protocols that are not secret in some of them e.g. we do not want any private keys of one protocol to appear as public constants in another protocol.

Chapter 3

# Implementation of APCC

In the previous chapter we introduced theoretical foundations for *An Automatic Protocol Composition Checker* (APCC). Here, we will presents the overall idea but also the most important implementation details. First, we will define the data structures on which ACPP operates, then we will implement composibility conditions so the program can decide whether some protocols are composable. The code will be explained through snippets while some code parts are listed in appendix A

We chose Haskell [9] as language of implementation because it is a suitable for the problems where we deal with structural data types and recursions, although it has some challenges which could not be found in script or object-oriented languages. OFMC's source code is also written in Haskell, therefore it will be easy to integrate APCC into it.

## 3.1 General idea

The basic idea behind APCC is to present it as a service where users can test composability of their protocol specifications. The specification can be given in some high-level language like Alice and Bob notation [10] (AnB) that OFMC

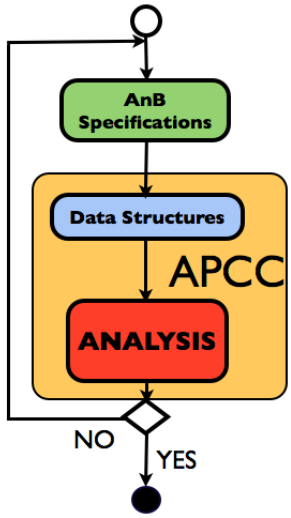


**Figure 3.1:** Interaction with APCC tool

uses. Once we feed APCC box like in 3.1 with these specifications, AnB notation is translated into data structures that are suitable for analysis. Finally, we get either positive or negative answer whether conditions hold or now and where problems occurred. The workflow is presented on 3.1 and it shows how user can interact with result of APPC. Every time when a user gets a negative answer, it should be precised what went wrong so user can fix the protocol specifications and repeat the procedure again until he gets a positive answer. This feedback can help to gain some good practices of protocol composition design.

## 3.2 Data structures

Based on the term representation of messages and theoretical analysis from previous chapter, we define the following message data structure:

```
data Atomic = Ident Id
            | Mapping Id [Atomic]
            deriving (Eq,Ord)

data Message = Atom Atomic
            | Concat [Message]
            | Asycrypt Message Message
            | Symcrypt Message Message
```

```
          | Hash Id Message
          deriving (Eq,Ord)
```

**Listing 3.1:** Message data types

*Message* is either *Atomic* or composed with recursive definition. Here, *Atomic*[1] part corresponds to either constants, variables or mappings defined over agents: public keys like $pk(A)$ or shared keys like $sk(A, B)$. We mentioned in previous chapter that mappings are not considered as normal operations, hence they are basic terms and we classify them as atomic. By convention, variables always begin with upper-case, while constants begin with lower-case letter. Composed messages are defined as:

- *Concat* is concatenation of messages, Haskell's data type *list*
- *Asycrypt* is asymmetric encryption where the arguments are key and message to be encrypted
- *Symcrypt* is symmetric encryption where the arguments are key and message to be encrypted
- *Hash* is a hash 'like' function with *Id* name, with a message as an argument

In close correspondence, to Message data type we are defining the *MessageType* data type. We mentioned that part of compositionality analysis 2.3.2 requires format-type safeness (FTS) check. This includes that all messages are supposed to be typed. Therefore, we define:

```
data BasicType = Agent
                | Nonce
        | Symkey
        | Pubkey
        | Privkey
        | Tag
       deriving (Eq,Ord)
data MessageType = Basic BasicType
        | Conc [MessageType]
        | Crypt MessageType MessageType
        | Scrypt MessageType MessageType
        | HashT MessageType
        deriving (Eq,Ord)
```

**Listing 3.2:** MessageType data types

[1]Ident is user defined type identical to String

Atomic messages are having one of the types from the set {*Agent, Nonce, Symkey, Pubkey, Privkey, Tag*[2]} while composed types correspond to composed messages. For instance, message $asycrypt(pk(B), NA)$ looks like:

```
Asycrypt(Atom((Mapping"pk") [Ident "B"]))(Atom(Ident"NA"))
```

with message type:

```
Crypt (Basic Pubkey) (Basic Nonce)
```

An APPC analysis requires that for every message/submessage protocol a type is specified, like a tuple (Message, MessageType). AnB specification (at least the extended one) contains information what are the types of the atomic part of the message: variables, constants and function names, while the rest we get just by composing.

## 3.3 Unification over *Message* data type

The theoretical overview of unification algorithm was given in the 2.2.2 and now we are implementing that idea. The algorithm is generalized to find unifier for a list of messages but we are only interested whether unifier exist for pair. *Unify* function is defined as:

```
unify :: [(Message, Message)] → Maybe [(Message, Message)]
```

The unification result is *Maybe* data type in Haskell. It is either *Nothing* (empty list) or list of substitutions expressed as a tuples. In order to implement the transformation rules, we need to consider the more complicated definition of message operations e.g. specific data structure where even atomic parts (private and public mappings) are functions in the unification algorithm.

```
--| Initial state. xx is the messages that we are trying to
    unify
unify xx = unify' xx []

unify' :: [(Message, Message)] → [(Message, Message)] →
    Maybe [(Message, Message)]

-- | When there is nothing to process, acc is the solution
unify' [] acc = Just acc
```

[2] *Tag* is a type for a unique constant within one protocol that makes message format disjoint

```
-- | Delete rule
unify' ((a, b):xx) acc | a==b = unify' xx acc

-- | Eliminate rule
unify' ((Atom(Ident x),t):xx) acc
| not (occursCheck (x,t)) && isVariable x =
 unify' (map (substitutionp (Atom (Ident x)) t) xx) ((Atom(
     Ident x),t) : map (substitutionp (Atom (Ident x)) t) acc
     )
| isVarTerm t && not (isVariable x)  =
 unify' (map (substitutionp t (Atom (Ident x))) xx) ((t,(
     Atom (Ident x))) : map (substitutionp t  (Atom (Ident x)
     )) acc)
| otherwise = Nothing

-- | Orient rule
unify' ((t,Atom(Ident x)):xx) acc = unify' ((Atom(Ident x),t
    ):xx) acc

-- | Decompose rules for all the non atomic message terms,
    including mappings!
unify' ((Asycrypt key1 mess1, Asycrypt key2 mess2):xx) acc
      = unify' ([(key1,key2)]++([(mess1,mess2)] ++ xx)) acc
```

**Listing 3.3:** Unification code

The code snippet 3.3 shows core of the unification code where decompose rule is only given for asymmetric encryption and for all other composed terms it looks similar. We start unification algorithm with pair of messages as arguments. Then, algorithm tries to perform one of the specified rules. The function *occursCheck* examines one of the conditions for the *eliminate* rule whether variable on left side occurs in the function term on the right side. *Substitutionp* performed inside one term, substitutes every occurrence of one variable with a term.

## 3.4 Implementing preconditions

After we established *Message* data type and its corresponding *MessageType* type, we can check the preconditions, starting with FTS defined in 2.3.2. An approach to this part of implementation was straightforward: we form $\alpha$ renamed set of all possible protocol message subterms and try to unify each pair of them. Implemented conditions are based on unifiability of terms that have to have disjoint variable symbols, thus we need to perform appropriate $\alpha$ renaming.

The $\alpha$ renaming task changed direction of implementation process. Idea behind the renaming is the following: every time we add new term to a growing set of subterms, we append the cardinal number of the set to the term's variable names. Hence, current number of the elements in the set has to be saved through all process of computation.

This is why we employ monads in Haskell [11, 12]. Haskell is a pure functional programming languages and programs are made of functions that cannot change global variables or state. While in other programming languages we can have a global state can be preserved using variables but, in Haskell every variable has to be part of function parameters. When a function is called twice with the same arguments the result will always be the same. Haskell offers *State monads* to bind a state transition to computations. This syntactic sugar allows imperative programming style in a pure functional programming language.

Computation over *Control.Monad.State* [13] depends on some internal state which can be modified during computations. The *State* monad used in computation for forming subterms set is defined as *State StateType StateType*, where *StateType* represents both internal state and its value:

```
type StateType = (Map.Map (Message,String) MessageType, Int)
```

StateType holds the map data structure of all subterms as a key-value pair $(Message, MessageType)$ [3]. Function *subTerms* that results with this kind of monad will always add new state by modifying existing one.

```
subTerms :: ((Message,String), MessageType) → State
    StateType StateType

subTerms ((Symcrypt k x,protocolName), Scrypt key msg) = do
      (state, counter) ← get
      put ( Map.insert ((alphaRenaming (Symcrypt k x)
          counter),protocolName) (Scrypt key msg) $ state,
          counter+1)
      subTerms ((k,protocolName),key)
      subTerms ((x,protocolName),msg)
```

**Listing 3.4:** Function subTerms

The snippet of the function 3.4 gives pattern matching with symmetric encryption. It takes the current states,using monad function *get* and changes it by appending renamed message,using monad function *put*, while incrementing counter and recursively calling functions on the key and encrypted message.

[3] String is just a name of the protocol that every Message holds for the easy error reporting

Based on the definition of FTS, 3.5 shows checking of FTS in 2 parts: whether we can unify non-variable subterms of different types and finding out whether some element of message pattern is a variable. *protocolNVST* wraps *subTerms* by calling it with array of protocol messages hence, it takes $MP(P)$ and initial state $(Map.empty, 0)$ as arguments. If the *checkUnify* function returns a positive answer for unification, the program will terminate and an error report will be given as printed trace which terms were unified.

```
propertyNum1 :: [((Message,String), MessageType)] → Bool
propertyNum1 array = let nvst = protocolNVST array (Map.
    empty,0)
 in
  if (cond1propertyNum1 array) && (checkUnify (fst nvst) (
     Map.keys (fst nvst)))
   then True
   else error ()
```

**Listing 3.5:** Format-type safe check

### 3.4.1 Important implementation notes

During formal definition in the previous chapter we emphasised that if a protocol is not FTS than it cannot be taken into consideration because it violates the first crucial condition. This is exactly what we are using in APCC. The APCC program will always terminate if it discovers violation of FTS otherwise it will proceed to check DISE condition. DISE and FTS are single protocol properties and they should be satisfied before the checker continues to examine protocols' disjointness.

### 3.4.2 DISE condition

Following the definition 2.3.3, the list of all subterms that we have to examine is getting bigger and complicated as we form list of the lists. We need to keep track of all $EMP_i$-s separately but still performing $\alpha$ renaming of all the terms. Single $EMP_i$ is also computed using state monad. New $EMP_i$ with renamed variable elements gets into a list and increases the counter value that we piggyback. For the purpose of encryption we use variable $KEY$ concatenated with counter value, so we can keep it also unique.

```
makeSingleEMP :: [((Message,String),MessageType)] → State
    State' State'
makeSingleEMP [] = do
```

```
   (state,counter) ← get
   return (state,counter)

makeSingleEMP (x:xs) = do
     (state, counter) ← get
     let aux = encryptMsg x ("KEY"++(show counter))
      in  put(Map.insert ((alphaRenaming (fst (fst aux))
          counter), (snd(fst aux))) (snd aux) $ state ,
          counter+1)
     makeSingleEMP xs
```

**Listing 3.6:** Function that makes $EMP_i$

In order to compute $EMP$ set, we need to include more complicated information in the state which will contain list of all $EMP_i$s and that will always propagate the counter. We need the counter afterwards to pass it to the analysis of the next protocol. The length of the $EMP$ strictly depends on depth of protocol encryption. This means that we will call function 3.7 over state exactly $depth+1$ times

```
-- state definition
type StateEMP=([Map.Map (Message,String) MessageType], Int)
-- EMP function
computeEMP :: Int → State StateEMP StateEMP
computeEMP 0 = do
            (state, counter) ← get
            return (state, counter)
computeEMP depth = do
                (state, counter) ← get
                let newState = evalState (makeSingleEMP (Map
                    .toList(head(state)))) (Map.empty,
                    counter)
                 in put((fst newState):state, (snd newState)
                     +counter)
                computeEMP (depth-1)
```

**Listing 3.7:** Forming EMP

Once when we get a collection of lists, it is straightforward to check whether every two elements from distinct lists have a unifier. When they do not, we proceed with forming the $EST$ list. Informally speaking, this list characterizes every protocol in ACCP. Inside $EST$, we will have all the terms, to the level of non-variable subterms, that need to be different and non-unifiable with $EST$s elements of other protocols. This is a central function in ACCP. ACCP accepts lists of protocols, computes the $EST$s of them and tries to unify any two elements

from different *EST*. If there is a unifiable pair, it is a violation of composability condition. Idea is implemented in 3.8.

```
protocolsEST :: [[((Message,String), MessageType)]] → State
    StateEMP StateValue --  [[(Message, MessageType)]]
protocolsEST [] = do
        (state,counter) ← get
        return state
protocolsEST (x:xs) = do
        (state,counter) ← get
        let newElem = makeEST (x,counter)
         in put ((fst newElem) : state, snd newElem)
        protocolsEST(xs)
disjointCheck :: [([(Message, MessageType)], String)] →
   Bool
disjointCheck protocols = let est = (evalState (protocolsEST
    (map putNameInsideMessages protocols)) ([], 0))
                          in unifyEMP(est)
```

**Listing 3.8:** Central functions

### 3.4.3 Protocol security in isolation

APCC assumes that the protocol is already secure in isolation, which can be established e.g. by calling OFMC. APCC in the first version, does not call OFMC for protocol verification, but we will rely on protocol specifications that OFMC already labelled as verified. OFMC package already contains adapted protocols from Clark-Jacob library [14]. The main goal is to integrate APCC as one of the options inside OFMC which will make all process more user friendly.

## 3.5 AnB notation

OFMC uses AnB format based on the popular Alice and Bob notation which is translated to the Intermediate Format(IF) [15], a tool-independent language more suitable for analysis. OFMC is a part of AVISPA and AVANTSSAR project [16] and IF format can be used for different back-ends, not just OFMC. APCC uses slightly modified AnB notation extended with additional information for APCC.

The first modification in the grammar of AnB language is separate definition for the mappings. In current version mappings are part of *Functions* but we

want to distinguish them as different models explained in 2.1.2

The second addition to a grammar is the definition of the keys. This only refers to a channel protocols like TLS. In that case, we specify a pair of keys one for each direction of message flow.

Finally, we define the *Public* and *Private* terms of the protocols. This is crucial for parallel composition precondition where we have to check whether some protocol considers some constants as private that are public in some other protocol.

```
Protocol: TLS

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

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

Public: pk(A),pk(B)      Private: inv(pk(A)), inv(pk(B))

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,tag2)}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

ChannelKeys:

K(A,B): clientK(NA,NB,prf(PMS,NA,NB))
K(B,A): serverK(NA,NB,prf(PMS,NA,NB))
```

**Listing 3.9:** TLS protocol

In 3.9 gives TLS deescription in AnB files. Here, public constants are public keys of the agents and they can be used to in other protocols that these agents participate in. Also, private constants are agents' certificates and they cannot

be used anywhere else except in this protocol. *ChannelKeys* section defines a pair of keys for both direction of communication. The key is shown together with the contributed nonces that agents produced during the key agreement.

Implementation of the AnB front end of is achieved using the lexer generator Alex [17] and parser generator Happy [18] for Haskell. Happy is similar to the famous Yacc parser generator and it is also using LALR parsing algorithm but it has an option for GLR parsing [19]. Firstly, we generate new tokens using Alex generator which are new reserved words in AnB file: *Functions, Public, Private, ChannelKeys* and then we extend the parser file with productions and corresponding actions which will map the parser result to the Abstract Syntax Tree (AST) file.

### 3.5.1 Parallel composition

Sections *Private* and *Public* in AnB file are presented so we can initiate checking of parallel composition condition. We will not allow any long term secrets in one protocol to be a public constant in another one. Checking will be performed using unification function between every pair of *Public* and *Private* list defined in AnB file.

### 3.5.2 Translation of AnB to APCC message type

The data that OFMC uses is quite different from the one used in ACPP. As an example, OFMC concatenation is defined using binary operation pair (e.g $[a, b, c]$ is same as $pair(a, pair(b, c))$), while in ACPP we use a list of messages. Once we parsed the file, we can easily extract all needed information from corresponding AST. All agent, nonces, functions, mappings need to be defined in the file so we can check types of all those identifiers that appear in the protocol specification.

```
processProtocol :: Protocol → ([(M.Message, M.MessageType)
    ],String)
processProtocol protocol =
 let actions = extractActions protocol
         name = protocolName protocol
         msgs = map msgToMessage actions
         typeMsgs = map (λx → assignType x protocol) msgs
    in (zip msgs typeMsgs,name)
```

**Listing 3.10:** Translation

Function 3.10 shows process of AnB protocol specification to the APCC message representation. Here, we are extract the most important details needed for APCC analysis:

- All actions from AST which are transformed as array of Messages defined in 3.2
- Name of the protocol which we will piggyback to all messages, for easier error reporting

From message list and AST, it is straightforward to discover message types. Once, we do that, all result is wrapped in the type:

```
([(M.Message, M.MessageType)],String)
```

and the protocol is ready for analysis.

## 3.6 Up and running

APCC is a command line tool in its first version and it requires protocol specification as an arguments. Listing 3.11 represents the core of the checker where lexer, parser and the main analysis functions are called. There are 2 possible outcomes of APCC: either program will terminate without analysis errors, then conditions for the composition of the protocols are fulfilled, or the program will terminate will report a violation of the conditions

```
xs ← getArgs
if null xs then do
    putStrLn "You entered no arguments!"
 else do
    putStrLn ("You entered " ++ show xs)
    file ← mapM readFile xs
    let protocols = map anbparser (map alexScanTokens
        file)
         allProtocols = map processProtocol protocols

    putStrLn ("Are conditions for vertical composition
        being violated? :  "++ show (disjointCheck
        allProtocols))
    putStrLn ("Parallel condition:    " ++ show (
        parallelChecker protocols))
```

**Listing 3.11:** Program

Chapter 4

# Experimental results

This chapter describes experimental results of APCC with a library of protocols consisting of most of the Clark-Jacob library [14] of protocols and TLS. The protocols are all given in AnB notation and we consider only those verified with OFMC. It is required to use extended AnB notation for the purpose of APCC analysis and we described it in the previous chapter . We demonstrate APCC on a small subset of the library, but at the end we will redesign specifications so all protocols from the library are composable.

The library [14] contains a list of small and medium scale authentication protocols. They are classified according to:

- cryptographic approach: using public or shared key encryptions
- usage of trust party to achieve some level of authentication
- number of messages in protocol
- whether it is a one-way (unliteral) or two-way mutual authentication

The majority of them belongs to a group of key transport protocols like Kerberos [20], Andrew Protocol RPC [21] while some of them provide some security services like: key authentication, key freshness and key confirmation.

For the composability result, we are interested in the channel protocols and we extend the library with TLS, currently omitting Diffie-Hellman because APCC does not support algebraic properties.

## 4.1 Demonstration of the tool

We first take a look at a small subset of protocols, namely: TLS, Needham–Schroeder public key protocol (NSPK) [22] and ISO Symmetric Key One-Pass Unilateral Authentication Protocol. The analysis of these 3 protocols does not consider how they can be composed, one can say that all of them are stacked (vertical composition) of they 2 of them run as parallel composition on established TLS channel. If APCC analysis comes out with positive answer, composability result guaranties that they can be composed in no matter what combination.

NSPK is one of the first authentication protocols and it is often mentioned in the research papers. There is an attack known on the protocol and we can discover it in OFMC and that is the reason why we use Lowe's fixed version of NSPK [23]. AnB specification is given in 4.1 where we defined *Public* and *Private* constants while *ChannelKey* section is empty because this is not channel protocol.

```
Protocol: NSPK # with Lowe fix

Types: Agent A,B;          Knowledge: A: A,pk,inv(pk(A)),B;
       Number NA,NB;                  B: B,pk,inv(pk(B))
       Mapping pk

Public: pk(A),pk(B)        Private: inv(pk(A)), inv(pk(B))

Actions:
A->B: {NA,A}(pk(B))
B->A: {NA,NB,B}(pk(A))
A->B: {NB}(pk(B))

Goals:

B authenticates A on NA
A authenticates B on NB
NA secret between A,B
NB secret between A,B

ChannelKeys:
```

**Listing 4.1:** Needham–Schroeder (public key) protocol (NSPK)

Verified and adapted TLS and ISO Symmetric Key One-Pass Unilateral Authentication Protocol are given in 4.2 and 4.3

```
Protocol: TLS

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

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

Public: pk(A),pk(B)      Private: inv(pk(A)), inv(pk(B))

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

ChannelKeys:

K(A,B): clientK(NA,NB,prf(PMS,NA,NB))
K(B,A): serverK(NA,NB,prf(PMS,NA,NB))
```

**Listing 4.2:** TLS protocol

```
Protocol: ISO_onepass_symm

Types: Agent A,B;
       Number NA,Text1;
       Function sk

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

Public: sk(A,B)

Private:
```

```
Actions:
A->B: {|NA,B,Text1|}sk(A,B)

Goals:
B weakly authenticates A on Text1

ChannelKeys:
```

**Listing 4.3:** ISO Symmetric Key One-Pass Unilateral Authentication Protocol

## 4.2 APCC disapproval vs. good protocol design

Although the demo subset of the library holds only 3 protocols, the functionality of APCC can be efficiently demonstrated. Through numerous iterations, user improves protocol specifications based on a feedback from APCC. The first run discovers the problem:

```
* Protocol "NSPK" is not format-type safe!
* Unification is possible between terms:{[NA2,NB2,B2]}pk([A2])
    and {NB4}pk([B4])
```

This message error report gives enough information how to proceed with fixing the protocol. Namely, APCC unified 2 terms with different types $crypt(pubkey, conc(nonce, nonce, agent)$ and $crypt(pubkey, nonce)$. This gives a hint to a user to make sure that messages, on non-variable subterm level, inside one protocol should not be confused and that he should make them as distinct as possible. One way to achieve it is to tag the different encrypted messages e.g. so that each tag represents the message number of the protocol. Thus, we define constants *tagNSPK1, tagNSPK2* and change the problematic actions to $\{tagNSPK1, NA, NB, B2\}(pk(A)), \{tagNSPK2, NB\}(pk(B))$. We chose naming convention for tags as concatenation of 3 string components: *tag*, *protocolname* and *number* to distinguish tag constants among the protocols. We run APCC again and get the following error:

```
* Protocol "NSPK" is not format-type safe!
* Unification is possible between terms: [NA1,A1] and [
    tagNSPK2,NB6]
```

This confirms the assumption that even the first protocol message needs a tag e.g *tagNSPK3* which will make make unification between these 2 terms impossible. So, third message becomes $\{tagNSPK3, NA, A\}(pk(B))$

Having fixed this error, APCC finally finishes checking the FTS condition for NSPK protocol. Then, DSE is checked and it is trivially satisfied because NSPK doesn't have any symmetric encryption involved inside its actions. ISO Symmetric Key One-Pass Unilateral Authentication Protocol has only one message where FTS and DSE conditions are also trivially satisfied.

Once APCC comes to the TLS, there will be many reported problems. This happens due to the complexity of the TLS handshake where lots of different message subterms looks similar which makes them unifiable. We start with:

```
* Protocol "TLS" is not format-type safe!
* Unification is possible between terms: [NA16,NB16,prf[PMS16,
    NA16,NB16]] and [PMS31,NA31,NB31]
```

These 2 terms are subterms of shared key $clientK(NA, NB, prf(PMS, NA, NB))$ and adding a tag, $clientK(NA, NB, prf(PMS, NA, NB), tagTLS1)$, in the outer concatenation can make them non-unifiable. This problem is also addressed to a $serverK(NA, NB, prf(PMS, NA, NB))$ where we use the same fix. On next run, we get:

```
* Protocol "TLS" is not format-type safe!
* Unification is possible between terms: [NB13,B13,PMS13] and
    [PMS33,NA33,NB33]
```

The triples are situated in the hash functions: $hash$ and $prf$ and we resolve this unification by tagging both triples. The next run brings new error message:

```
* Protocol "TLS" is not format-type safe!
* Unification is possible between terms: [A0,NA0,Sid0,PA0] and
    [{[A5,pk([A5])]}inv([pk([s5])]),{PMS5}pk([B5]),{hash[NB5,
    B5,PMS5]}inv([pk([A5])]),|hash[prf[PMS5,NA5,NB5,tagTLS3],
    A5,B5,NA5,NB5,Sid5,PA5,PB5,PMS5]|clientK[NA5,NB5,prf[PMS5,
    NA5,NB5],tagTLS1]]
```

Adding new tags into the client's shared key, a new problem emerges and unification between the first message of the handshake is being confused with the final client's message. Adding a tag to a message $[A0, NA0, Sid0, PA0, tagTLS4]$ will make error disappear.

After fixing this part, APCC finally reports positive result of analysis e.g. 3 protocols are composable. DISE and protocols disjointness conditions are satisfied without fixing the specifications. Added tags actually prevented APCC to complain about protocols disjointness. Parallel composition conditions is also satisfied, because tool could not unify any private with public constant.

One can add more protocols to this library. Then, we need to gain some routine and experience and learn how to avoid similar problems. APCC gives good directions and hints what is supposed to be done. The main techniques is certainly tagging but also permutation of concatenation parts so disjointness is satisfied. The checker is typically not the critical point for runtime. This, we can check the composition of all protocols basically in the time that is required to check the individual protocols.

We tested protocols from Clark-Jacob library (that are verified with OFMC )and all of them needed to be fixed. Surprisingly, APCC reported many unifications and all of them needed to be resolved. As a result, we have 18 protocols[1] ready to be composed: .

- TLS
- Needham–Schroeder protocol (NSPK) with Lowe's fix
- ISO Symmetric Key One-Pass Unilateral Authentication Protocol
- Needham-Schroeder Protocol with Conventional Keys
- Bilateral Key Exchange with Public Key
- ISO One-Pass Unilateral Authentication with CCFs[2]
- ISO Two-Pass Unilateral Authentication with CCFs
- ISO Two-Pass Mutual Authentication with CCFs
- ISO Three-Pass Mutual Authentication with CCFs
- ISO Symmetric Key One-Pass Unilateral AuthenticationProtocol
- ISO Symmetric Key Two-Pass Unilateral AuthenticationProtocol
- ISO Symmetric Key Two-Pass Mutual Authentication
- ISO Symmetric Key Three-Pass Mutual Authentication
- Non-ReversibleFunction protocol
- Andrew Secure RPC Protocol
- Denning-Sacco protocol
- Denning-Sacco protocol using public key cryptography
- Basic Kerberos protocol

[1] Specifications for all of them are found in a source code of APCC
[2] cryptographic check functions

## 4.3 APCC limitations

We did not consider protocols that involves Diffie-Hellman key exchange like H.530 [24]. Although Diffie-Hellman key exchange is really popular because of its nice properties like forward secrecy, most of the verification tools did not implement it. Involving modular exponentiation in the tools, affects unification algorithm as well as the intruder deduction. In the scope of the free algebra we cannot model Diffie-Hellman thus, we skipped protocols that are using Diffie-Hellman key exchange. There are couple of suggestions [25] how to resolve this problem and that could be one of the future improvements of APCC.

Chapter 5

# Conclusion and future work

The thesis work turns the theoretical conditions of several works on composability into practical usable tool and therefore adds a helpful feature to automatic verification tools: checking protocol composition automatically. The composability is all about whether message formats are confused and the checker is alerting these problems. Although the tagging is not so much used in the real situation, the messages must somehow be distinguished and that is good engineering practice. However, in our abstract term world the tags are often just a way to express that two messages are distinct.

We plan the future versions of the checker to bring better compatibility with OFMC tool as well as better user friendly environment. One can extend the current functionality of APCC by defining it as a webservice where we can register our protocols and give an answer about its composability with some other protocols in already registered set. The AVISPA project had already implemented a web service for verification of the protocols in isolation.

One of the mentioned missing features is Diffie-Hellman key exchange. This is the high priority extension and researchers are trying to efficiently adopt the algebraic properties into the current algorithms of unification and intruders deduction.

Still, formal analysis is bounded to a small and medium scale protocols with

lots of limitation comparing to the real protocol implementations. Nevertheless, impact of the research to the protocol design is enormous and modelling protocols in the term algebra is one of the most used approaches. Discovering the security flaws in algebraic models is cheaper and it can help designing better and more secured protocols.

Appendix A

# Source code

```
module Unification (unify) where

import Message

-- | Substitution of the messages on single term
substitution :: Message → Message → Message → Message
substitution (Atom (Ident x)) t (Atom (Ident s))
                        | s == x    = t
                        | otherwise = Atom (Ident s)

-- | Substitution of the messages on complex (function) term
substitution x t (Atom (Mapping id a)) = Atom (Mapping id (
    map toAtomic (map (substitution x t) (wrapIdent a))))
substitution x t (Concat xx)       = Concat (map (
    substitution x t) xx)
substitution x t (Asycrypt k xx)   = Asycrypt  (substitution
     x t k) (substitution x t xx)
substitution x t (Symcrypt k xx)   = Symcrypt  (substitution
     x t k) (substitution x t xx)
substitution x t (Hash _ xx)         = substitution x t xx

substitutionp x t (p1,p2)    = (substitution x t p1,
    substitution x t p2)
```

```
-- | if variable is "inside" the non atomic term on the
    right side -}

occursCheck:: (String, Message) → Bool

occursCheck (x, Atom(Ident t))
                | x==t = True
                | otherwise = False

occursCheck (x, Atom (Mapping k t)) = let t2 = wrapIdent t
    in foldl (λacc a → if occursCheck (x,a) then True else
        acc) False t2
occursCheck (x, Concat t) = foldl (λacc a → if occursCheck
    (x,a) then True else acc) False t

occursCheck (x, Asycrypt k t)  = occursCheck (x,t) ||
    occursCheck (x,k)
occursCheck (x, Symcrypt k t)  = occursCheck (x,t) ||
    occursCheck (x,k)
occursCheck (x, Hash _ t)      = occursCheck (x,t)
---------------------------------   Wrap message  ---------
    technical thing because of message data type

wrapIdent :: [Atomic] → [Message]
wrapIdent [] = []
wrapIdent (x:xs) = (Atom x): wrapIdent xs

 ---------------------------------- Making atomic message
     ----

toAtomic :: Message → Atomic
toAtomic (Atom (Ident t)) = Ident t
toAtomic (Atom (Mapping id t)) = Mapping id t


-- | Unification algorithm also works for set of the
    messages

unify :: [(Message, Message)] → Maybe [(Message, Message)]

unify xx = unify' xx []

unify' :: [(Message, Message)] → [(Message, Message)] →
    Maybe [(Message, Message)]

-- | When there is nothing to process, acc is the solution
unify' [] acc = Just acc
```

```
-- | Delete rule
unify' ((a, b):xx) acc | a==b = unify' xx acc


-- | Substitute rule : takes care of occurrence of the
    variable of on the right side; constants on the left side
     ⇒ do the substitution with switched arguments

unify' ((Atom(Ident x),t):xx) acc | not (occursCheck (x,t))
    && isVariable x = -- SUBSTITUTE - watching out on
    constants!
          unify' (map (substitutionp (Atom (Ident x)) t) xx)
              ((Atom(Ident x),t) : map (substitutionp (Atom (
              Ident x)) t) acc)
          | isVarTerm t && not (isVariable x)  =
               unify' (map (substitutionp t (Atom (Ident x)))
                    xx) ((t,(Atom (Ident x))) : map (
                   substitutionp t  (Atom (Ident x))) acc)

          | otherwise = Nothing


-- | Orient rule
unify' ((t,Atom(Ident x)):xx) acc = unify' ((Atom(Ident x),t
    ):xx) acc


-- | Decompose rules for all the non atomic message terms,
    including mappings!
unify' ((Asycrypt key1 mess1, Asycrypt key2 mess2):xx) acc
      = unify' ([(key1,key2)]++([(mess1,mess2)] ++ xx)) acc

unify' ((Symcrypt key1 mess1, Symcrypt key2 mess2):xx) acc
      = unify' ([(key1,key2)]++([(mess1,mess2)] ++ xx)) acc


unify' ((Hash id1 mess1, Hash id2 mess2):xx) acc
     | id1 == id2 = unify' ([(mess1,mess2)] ++ xx) acc
     | otherwise = Nothing

unify' ((Concat f1s, Concat f2s):xx) acc
     | length f1s == length f2s = unify' (zip f1s f2s ++ xx)
        acc
     | otherwise = Nothing
```

```
unify' ((Atom (Mapping id1 t1), Atom (Mapping id2 t2)):xx)
    acc
    | (id1 == id2 && length t1 == length t2) = unify' (zip (
        wrapIdent t1) (wrapIdent t2) ++ xx) acc
    | otherwise = Nothing


-- | For everything else, return Nothing
unify' (_) acc = Nothing
```

**Listing A.1:** Unification module

```
subTerms :: ((Message,String), MessageType) → State
    StateType StateType

subTerms  ((Atom(Ident x),protocolName), msgtype) = do
      (state, counter) ← get
      if isConstant x
              then put ( Map.insert  ((alphaRenaming (Atom (
                  Ident x)) counter),protocolName) msgtype $
                  state ,counter+1)
              else put (state,counter)
      get

subTerms ((Atom(Mapping id t),protocolName), keyType)  = do
      (state, counter) ← get
      put ( Map.insert ((alphaRenaming (Atom(Mapping id t))
          counter),protocolName) keyType $ state , counter+1)
      get

subTerms ((Asycrypt k x,protocolName), Crypt key msg) = do
      (state, counter) ← get
      put ( Map.insert ((alphaRenaming (Asycrypt k x)
          counter),protocolName) (Crypt key msg) $ state,
          counter+1)
      if isNonVarSubTerm k
              then do subTerms ((k,protocolName),key)
                      subTerms ((x,protocolName),msg)
              else subTerms ((x,protocolName),msg)

subTerms ((Symcrypt k x,protocolName), Scrypt key msg) = do
      (state, counter) ← get
      put ( Map.insert ((alphaRenaming (Symcrypt k x)
          counter),protocolName) (Scrypt key msg) $ state,
          counter+1)
      subTerms ((k,protocolName),key)
      subTerms ((x,protocolName),msg)
```

```
subTerms ((Concat x,protocolName), Conc t) = do
      (state, counter) ← get
      put (Map.insert ((alphaRenaming (Concat x) counter),
          protocolName) (Conc t) $ state, counter+1)
      mapM subTerms (zip (zipName x protocolName) t)
      get

subTerms ((Hash id x,protocolName), HashT t) = do
      (state, counter) ← get
      put (Map.insert ((alphaRenaming (Hash id x) counter),
          protocolName) (HashT t) $ state, counter+1)
      subTerms ((x,protocolName),t)
-------------------------------------------------
-- | function counts non variable subterms of one protocol
protocolNVST :: [((Message,String), MessageType)] → (Map.
    Map (Message,String) MessageType, Int) → (Map.Map (
    Message,String) MessageType, Int)
protocolNVST [] initState = initState
protocolNVST ((msg,msgtype):xs) initState = let newInitState
     = evalState (subTerms (msg,msgtype)) initState
                                          in protocolNVST
                                              xs
                                              newInitState

-- | alpha renaming function
alphaRenaming :: Message → Int → Message
alphaRenaming (Atom (Ident t)) num | (isVariable t) = Atom (
    Ident (t++show num))
                                   | otherwise = (Atom (
                                      Ident t))
alphaRenaming (Atom (Mapping id t)) num = Atom (renameAtomic
     (Mapping id t) num)
alphaRenaming (Symcrypt key t) num = Symcrypt (alphaRenaming
     key num) (alphaRenaming t num)
alphaRenaming (Asycrypt key t) num = Asycrypt (alphaRenaming
     key num) (alphaRenaming t num)
alphaRenaming (Hash id t) num = Hash id (alphaRenaming t num
    )
alphaRenaming (Concat t) num = Concat (map (λ x →
    alphaRenaming x num) t)
```

**Listing A.2:** Part of Format Type Safe module

```
-- | Extension of the theorem. Number of EMPs is equal to
    the depth of encryptions inside a protocol.
depth :: ((Message,String),MessageType) → Int
depth ((Atom _, protocolName), _)= 0
```

```
depth ((Hash id t,protocolName), HashT tt) = depth ((t,
    protocolName),tt)
depth ((Concat t,protocolName), Conc tt) = (foldl (λacc (a,
    a1) → depth (a,a1)) 0 (zip (zipName t protocolName) tt))
depth ((Asycrypt k t,protocolName), Crypt kt tt) = depth ((k
    ,protocolName),kt) + depth ((t,protocolName),tt)
depth ((Symcrypt k t,protocolName), Scrypt kt tt) = (depth
    ((k,protocolName),kt) + depth ((t,protocolName),tt))+1

-- | Uses depth function

depthProtocol :: [((Message,String), MessageType)] → Int
depthProtocol array = maximum (map depth array)

-- | It gives single EMPi, preserving counter value for
    alpha renaming

makeSingleEMP :: [((Message,String),MessageType)] → State
    State' State'
makeSingleEMP [] = do
   (state,counter) ← get
   return (state,counter)

makeSingleEMP (x:xs) = do
     (state, counter) ← get
     let aux = encryptMsg x ("KEY"++(show counter))
      in  put(Map.insert ((alphaRenaming (fst (fst aux))
          counter), (snd(fst aux))) (snd aux) $ state ,
          counter+1)
     makeSingleEMP xs

-- | Now we make EMP set. Monadic computation.
--

computeEMP :: Int → State StateEMP StateEMP

computeEMP 0 = do
            (state, counter) ← get
            return (state, counter)

computeEMP depth = do
                (state, counter) ← get
                let newState = evalState (makeSingleEMP (Map
                    .toList(head(state)))) (Map.empty,
                    counter)
                 in put((fst newState):state, (snd newState)
                     +counter)
```

```
                computeEMP (depth-1)
```

**Listing A.3:** Part of Encrypted Message Pattern module

# Bibliography


[1] D. Dolev and A. C. Yao, "On the security of public key protocols," *Foundations of Computer Science, IEEE Annual Symposium on*, vol. 0, pp. 350–357, 1981.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1." RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746.

[3] V. Cortier, S. Delaune, and J. Delaitre, "Safely composing security protocols," 2008.

[4] S. Mödersheim and T. Groß, "Vertical protocol composition (extended version)," *IBM Research - Zurich, Switzerland, DTU Informatics Denmark*, p. 22, 2011.

[5] J. Heather, G. Lowe, and S. Schneider, "How to prevent type flaw attacks on security protocols," *Department of Computing, School of Electronics, Computing and Mathematics, University of Surrey; Programming Research Group, Oxford University Computing Laboratory and Royal Holloway; University of London.*

[6] G. Wang and S. Qing, *Two New Attacks Against Otway-Rees Protocol*, pp. 137–139. International Academic Publishers, 2000.

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

[8] D. Basin, S. Mödersheim, and L. Viganò, *OFMC: A symbolic model checker for security protocols*, vol. 4, pp. 181–208. Springer-Verlag, 2005.

[9] "Haskell: Fuctional programming language." http://www.haskell.org/haskellwiki/Haskell, 2011.

[10] S. Mödersheim, *An AnB Tutorial*. DTU Informatics, September 2011.

[11] J. Newbern, "All about monads." http://monads.haskell.cz/html/index.html, 2010.

[12] B. O'Sullivan, J. Goerzen, and D. Stewart, *Real World Haskell*. O'Reilly Media, Inc., 1 ed., Dec. 2008.

[13] "Haskell GHC 7.0.4." http://www.haskell.org/ghc/docs/7.0.4/, 2011.

[14] J. A. Clark and J. L. Jacob, "A Survey of Authentication Protocol Literature: Version 1.0," tech. rep., Department of Computer Science, University of York, 1997.

[15] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications," in *Computer Aided Verification* (K. Etessami and S. K. Rajamani, eds.), vol. 3576, ch. 27, pp. 281–285, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[16] "AVISPA - Automated Validation of Internet Security Protocols and Applications." http://www.avispa-project.org/.

[17] "Alex: A lexical analyser generator for Haskell." http://www.haskell.org/alex/.

[18] "Happy - the parser generator for haskell." http://www.haskell.org/happy/.

[19] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2 ed., Sept. 2006.

[20] B. C. Neuman and T. Ts'o, "Kerberos: an authentication service for computer networks," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 33–38, 1994.

[21] M. Satyanarayanan, "Integrating security in a large distributed system," *ACM Trans. Comput. Syst.*, vol. 7, pp. 247–280, Aug. 1989.

[22] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, pp. 993–999, Dec. 1978.

[23] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," *INFORMATION PROCESSING LETTERS*, vol. 56, pp. 131–133, 1995.

[24] "H.530: Symmetric security procedures for h.323 mobility in h.510." http://www.avispa-project.org/library/h.530.html, 2004.

[25] S. Mödersheim, "Diffie-Hellman without Difficulty," *FAST 2011*.