Extending Session Types to Model Security Properties

Julie Tollund

11th December 2018

Contents

1 | Introduction

The purpose of this report, is to establish a foundation for the future thesis by the author, in regards of extending session types to model security properties with the introduction of adversaries.

In the report I will first introduce the motivation for exploring this field, as well as the current research done in the area. Secondly, I will introduce different research areas, all related to the field, and how these work together as background knowledge for the future thesis, by which I will explain further about in the final section of this report.

1.1 Motivation

With IT becoming an ever bigger part of our lives, the need for stronger and better security measurements, has grown with it. In recent years we have seen the introduction of voting machines in the USA and a digitalisation of our hospitals here in Denmark. With this comes the importance of being able to restrict access to ensure acceptable behaviour especially in the presence of malicious adversaries where it becomes paramount. To solve this many researchers have suggested using security protocols to ensure these security guarantees. In this report I will highlight some of the research that has made it easier to program these security protocols, as it can be a complex and error prone task.

A lot of research has gone into communication protocols in the recent years, and fields such as Session Types, has made it a lot easier to build such protocols and ensure they are correct by construction. This idea however falls apart, when we introduce an adversary, as an attacker would be able to block messages from being delivered. This creates the motivation for doing the future thesis of introducing cryptography to Session Types, and with this report, create a basic understanding

of the different fields.

Security measurements has not only been introduced through protocols, but also through physical hardware, adding another layer of authentication. The company Trusted Computing Group introduced the Trusted Platform Module (TPM), a physical chip implemented on the motherboard, that provides a safe space for generating cryptographic keys, which is now used in a lot of modern computers and as a part of the Microsoft BitLocker. The TPM will be used as case in the thesis and is therefore also presented and explained in the following report.

1.2 Current research

The project relies heavily on the research done within the field of Security protocols and Session types. A lot of research has already gone into the field of the TPM specifications, especially by Ryan, Delaune, Kremer *et al.* [1], and will work as a foundation for the TPM's commands and protocols. Furthermore the report will use Cortier and Kremer to formally model security protocols and their goals.

1.3 Intended outcome

The intended outcome of the thesis, will be to take the idea of session types and consider them in an adversarial environment. This will be done by extending session types to model properties, and from this produce protocols that are secure by construction. Furthermore a compiler will be constructed to automate the process of producing these programs afterwards.

2 | Background Study

This section describes the work carried out so far. Most of it will be highlighting research done within the different fields, and showcasing examples of its use. The section ends in a summery of how the different fields come together, and how they can be used further in the coming thesis.

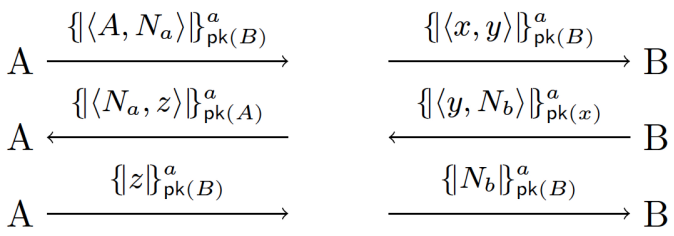
2.1 Security Protocols

Security protocols is an abstract or concrete protocol, that characterise the security related functions and applies cryptographic methods. It describes how the algorithm should be used to ensure the security and integrity of data transmitted. The security protocol is a protocol that runs in an untrusted environment where it assumes channels are untrusted and participants are dishonest. In academic examples, they are often described with the Alice and Bob notation, which will also be used in the following examples to create a better understanding of how security protocols work and can be reasoned about.

2.1.1 Needham-Schroeder Protocol

The Needham-Schroeder Public Key Protocol, was first proposed by Roger Needham and Michael Schroeder in 1978, and will be used as a running example in this and the following two sections.

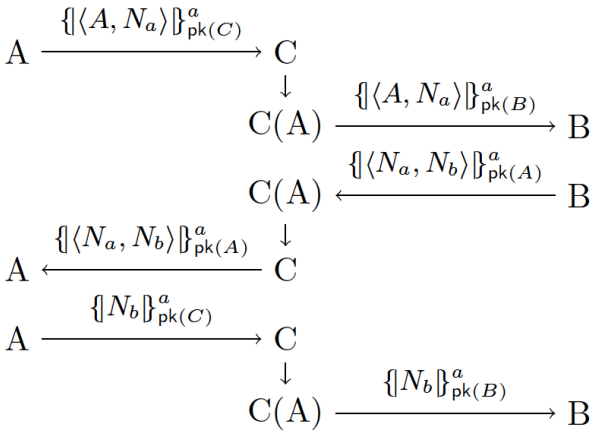
The Needham-Schroeder Public Key Protocol can be illustrated by the before mentioned Alice and Bob notation in the following way, as done by Cortier and Kremer



The A and B each represent Alice and Bob, while the arrows indicates the direction of the sent and received messages, by which are illustrated above each line. The notation $\{|m|\}^a_{pk(B)}$ denotes that the message m is created with an asymmetric encryption of Bob's public key, while the $\langle m_1, m_2\rangle$ illustrates a pairing or concatenation of the two messages. The A and B in the messages each represent the identities of Alice and Bob, while N_a denotes the freshly generated nonce, a random number generated each session. The variables x, y, z are used for the unknown values of the message.

In the first exchange, Alice sends her identity and nonce asymmetrically encrypted with Bob's public key. Bob receives the message, with variables x and y, illustrating the unknown values of the message. Bob decrypts the message to check that it is well-formed. He then generate his own nonce, pair it with Alice's nonce, and then encrypts it with Alice's public key. Alice then decrypts Bob's message, to verify that it contains her previously sent nonce, which proves that Bob received her first message. This way of sending and receiving nonces is often called a *challenge-response* authentication [2], and is also what you see when using passwords, where the challenger asks for a password and then checks that the response is valid.

The gap left between the two participants illustrate the challenge of this protocol in an untrusted network, as an attacker may instigate a *man-in-the-middle* attack. This vulnerability of the protocol, was first described by Gavin Lowe in his paper published in 1995, where also a fix was purposed. Again the Alice and Bob notation is used, but now we introduce an adversary C, as illustrated here by Cortier and Kremer.



If the attacker can persuade A to initiate a session with him, he relay the messages to B, and thus convince him he is communicating with A. Lowe suggest a simple solution to this problem, by adding the identity of the sender in the second message so that $\{|\langle N_a, N_b\rangle|\}^a_{pk(a)}$ would now look as such $\{|\langle N_a, \langle N_b, B\rangle\rangle|\}^a_{pk(a)}$. To highlight this vulnerability more clearly, I will revisit the Needham-Schroeder protocol again in the Applied Pi-Calculus section, where a more detailed description of why this attack is possible will be shown.

2.1.2 Message deduction

Message deduction is a formal way of figuring out wether a message can be deducted from a priori given set of messages by using induction rules and derivation sequences. Most important is the use of *terms* representing messages. This allows for a better illustration of the keys, identities and nonces, but abstract away the exact values, while still keeping the structure as a special labelled graph.

Terms Functions symbols such as *f*, are used to capture cryptographic primitives i.e. encryption and one-way hash functions, each assigned with an associated integer as its arity. Function symbols with arity 0 are seen as a constant, while a finite set of functions symbols is called a *signature*. Terms are build by applying function symbols to a infinite set of names N (used for keys, nonces or identities), variables X and other terms F, and can be defined as such *T(F, X, N)*. As can be seen later, variables are often used to represent unknown variables, e.g. components received from another participant of the protocol.

Inference rules Inference rules uses the following notation $\frac{u_l \quad ... \quad u_n}{u}$ with $u_l, ..., u_n, u$ as terms with variables. Having a set of inference rules is also called an inference system, and often contains both *composition rules* and *decomposition rules*. Below can be seen an inference system for the Dolev-Yao model ($\mathcal{I}_{DY}$) with the composition rules presented first in each line, and the following decomposition rules.

$$\mathcal{I}_{DY} \ : \begin{cases} \dfrac{x, y}{\langle x, y \rangle} \qquad \dfrac{\langle x, y \rangle}{x} \qquad \dfrac{\langle x, y \rangle}{y} \\[2ex] \dfrac{x \ y}{\text{senc}(x, y)} \qquad \dfrac{\text{senc}(x, y) \ y}{x} \\[2ex] \dfrac{x \ y}{\text{aenc}(x, y)} \qquad \dfrac{\text{aenc}(x, \text{pk}(y)) \ y}{x} \end{cases}$$

In the first line we have the rules for concatenation. It shows that anyone have the possibility of concatenating two terms (first rule), while the next two rules show that it is possible to retrieve the individual terms from a concatenation. On the second line we have the rules for symmetric encrypting and decrypting, showing that anyone can decrypt a message if they have the corresponding key. The same goes for the asymmetric encryption and decryption, with the rules shown in the third line.

Derivation sequence By combining inference rule, we are able to derive new messages. An example is given by Cortier and Kremer with the set of messages $S = \{\langle k_1, k_2 \rangle, \langle k_3, a \rangle, \{|n|\}^a_{\langle k_1, k_3 \rangle}\}$, illustrating how the set can be derived by an attacker through the use of the before mentioned inference rules (here represented by a proof tree):

$$\dfrac{\dfrac{\{|n|\}^s_{\langle k_1, k_3 \rangle} \qquad \dfrac{\dfrac{\langle k_1, k_2 \rangle}{k_1} \qquad \dfrac{\langle k_3, a \rangle}{k_3}}{\langle k_1, k_3 \rangle}}{n} \qquad \dfrac{\langle k_3, a \rangle}{a}}{\langle n, a \rangle}$$

From $\langle k1, k2\rangle$ the attacker can use the second rule shown, to learn *k1*. Using the same rule, he is also able to obtain *k3* from $\langle k3, a\rangle$. From these two he is now able to decrypt message n from $\{|n|\}^s_{\langle k1,k3\rangle}$, and thus get the encrypted message sent. For further illustration it is shown that the attacker is also able to get hold of a, which in turn is called that $\langle n, a\rangle$ is *deducible* from the set of messages S.

2.2 Applied π-Calculus

The applied pi-calculus is based upon the language pi-calculus, but offers a more convenient use for modelling security protocols to be specified, by allowing for a more wide variety of complex primitives. It is used for describing and analysing security protocols, as it provides a more intuitive process syntax for detailing the actions of the participants in a protocol [3]. This is done by introducing the before mentioned rich term algebra for modelling the cryptographic operations used in security protocols, where function symbols represent cryptographic protocols.

Tools such as ProVerif [4] uses a syntax closely related to the applied pi-calculus, and offers a way of automated reasoning about the security properties found in cryptographic protocols. The ProVerif tool is however not used for this report, but is mentioned as it is often used when analysing security protocols, and works as motivation for describing Applied pi-calculus, as ProVerif will most likely be used in the following thesis.

2.2.1 Syntax

The applied pi-calculus has two types of processes, the *plain* and *extended* process. First we introduce the grammar for the *plain* process:

$P,\ Q,\ R ::=$	plain processes
$\mid 0$	null process
$\mid P \mid Q$	parallel composition
$\mid\ !P$	replication
$\mid \nu n.P$	name restriction
$\mid$ if $M = N$ then P else Q	conditional
$\mid u(x).P$	message input
$\mid \overline{u}\langle M\rangle.P$	message output

The 0 process is the process that does nothing; $P \mid Q$ is the parallel composition of the processes P and Q executed in parallel; $!P$ is the replication of P that allows for an infinite composition of $P \mid P \mid ...$, which is often used for illustrating an unbound number of sessions; Name restriction $\nu n.P$ acts as a binder which generates a restricted name n inside P. This is often used for capturing fresh random numbers such as nonces and keys, or private channels, which we will see later when we apply it to the Needham-Schroeder public key Protocol; The conditional if $M = N$ then P else Q is like the one we know from normal conditioning in data types, where it behaves as P whenever $M = N$ (representing equality), and as Q otherwise; Last we have the message input and output, where $u(x).P$ expects an input on channel u and binds it to variable x in P, and $\overline{u}\langle M\rangle.P$ outputs term M on channel u and then behaves as P. It should be noted that message input and output will also we written as in*(u, x).P* and out*(u, M).P*, and parallel composition as $P \mid\mid Q$, as done by Cortier and Kremer

The *extended* process uses *active substitutions* to capture the knowledge exposed to the adversarial environment:

$A,\ B,\ C ::=$	extended processes
$\mid P$	plain process
$\mid A \mid B$	parallel composition
$\mid \nu n.A$	name restriction
$\mid \nu x.P$	variable restriction
$\mid \{^{M}/_{x}\}$	active substitution

With *active substitution* we allow for M to be available in the environment through the 'handle' x. In other words, M can now be replaced by x in every process it is

related to, and is only controlled by the variable restriction, i.e. $\nu x.(\{^{M}/_{x}\} \mid P)$ is exactly the same as writing $x = M$ in P.

With terms we use function symbols to capture primitives such as encryption or decryption used by cryptographic protocols. It should be noted that functions with arity 0 are what we define as constants. For terms we apply function symbols to names, variable and other terms as such:

L, M, N, T, U, V ::=	terms
| a, b, c,...,k,...,m, n,..,s	names
| x, y, z	variables
| g(M_1,..,M_l)	function application

These three grammars work as the foundation for applied pi-calculus, which in itself is a fairly simple language but the capabilities of describing security protocols in more detail than presented earlier.

2.2.2 Re-visiting the Needham-Schroeder Protocol

Having established an understanding of the applied pi-calculus and its grammar, we now use it model the previously mentioned Needham-Schroeder public key protocol.

To make a more detailed illustration, the processes has been parametrised to represent the initiator (Alice) and responder (Bob). First we look at the process of Alice (P_A):

$$\begin{aligned} P_A(sk_i, pk_r) \quad \hat{=} \quad & \nu n_a.\ \mathsf{out}(c, \mathsf{aenc}(\langle \mathsf{pk}(sk_i), n_a \rangle, \mathsf{pk}_r)). \\ & \mathsf{in}(c, x). \\ & \mathsf{if}\ \mathsf{fst}(\mathsf{adec}(x, sk_i)) = n_a\ \mathsf{then} \\ & \mathsf{let}\ x_{nb} = \mathsf{snd}(\mathsf{adec}(x, sk_i))\ \mathsf{in} \\ & \mathsf{out}(c, \mathsf{aenc}(x_{nb}, \mathsf{pk}_r)) \end{aligned}$$

The first thing Alice does is to generate a fresh random nonce and binds it to n_a, the process then continues to output the first message of an asymmetric encryption on channel c. Next she waits for a input message on the same channel, and binds the message to variable x. She then checks wether the message matches her previously

sent nonce n_a shown as the conditional. For readability the x is then bound to a local variable n_{xb}, representing the nonce received by the sender (Bob). Last she sends out a message again with an encryption of Bob's nonce and the public key, on channel c.

The dual of the process can now be modelled from the responder's (Bob) point of view:

$$\begin{aligned} P_B(sk_r) \quad \hat{=} \quad & \mathsf{in}(c, y). \\ & \mathsf{let}\ pk_i = \mathsf{fst}(\mathsf{adec}(y, sk_r))\ \mathsf{in} \\ & \mathsf{let}\ y_{na} = \mathsf{snd}(\mathsf{adec}(y, sk_r))\ \mathsf{in} \\ & \nu n_b.\mathsf{out}(c, \mathsf{aenc}(\langle y_{na}, n_b\rangle, pk_i)) \\ & \mathsf{in}(c, z). \\ & \mathsf{if}\ \mathsf{adec}(z, sk_r)) = n_b\ \mathsf{then}\ Q \end{aligned}$$

Combining the two processes we are able to model the Needham-Schroeder public key protocol, done so by Cortier and Kremer, as a whole:

$$\begin{aligned} P^1_{\mathsf{nspk}} \hat{=} \nu sk_a, sk_b.(P_A(sk_a, \mathsf{pk}(sk_b)) \parallel P_B(sk_b) \parallel \\ \mathsf{out}(c, \mathsf{pk}(sk_a)) \parallel \mathsf{out}(c, \mathsf{pk}(sk_b))) \end{aligned}$$

This however represent the naive model of the protocol, so to add the Lowe fix, as mentioned earlier i then report, we need to modify the protocol slightly:

$$\begin{aligned} P^5_{\mathsf{nspk}} \hat{=} !\nu sk_a, sk_b.(!\mathsf{in}(c, x_{pk}).P_A(sk_a, x_{pk}) \parallel !P_B(sk_a) \parallel \\ !\mathsf{in}(c, x_{pk}).P_A(sk_b, x_{pk}) \parallel !P_B(sk_b) \parallel \\ \mathsf{out}(c, \mathsf{pk}(sk_a)) \parallel \mathsf{out}(c, \mathsf{pk}(sk_b))) \end{aligned}$$

As can be seen, the input of the public key by Bob, shown as $\mathsf{pk}(sk_b)$, is now replaced by x_{pk} representing the unknown value of the variable, illustrating how Alice does now know who she is starting a session with. This also highlight the risk of an *reflection attack*, where the intruder tricks the participant to execute a protocol with him/her self. Replication has also been added to the session, as no upper bound has been put for the number of parallel session that may be initiated. Last but not least, we showcase that both session may be started by the same participant. All of this allow for an adversary to create an arbitrary number of instances of P_A and P_B with either the same or different private keys [2].

For the next section, we will look at another way of illustrating protocols that also have its roots from the process calculus.

2.3 Session Types

As with the applied pi-calculus, session types also have its root in the pi-calculus, and can be thought of as types for protocols. The idea behind session types, is to describe the communication protocols as a type that can be statically checked and thus ensure that well-typed programs are well behaved. Essential to session types is the distinction between binary and multiparty communication channels, where the binary sessions allow for only two participants, while the multiparty session types can have zero or more participants. Only the basic definitions of session types will be presented here and will refer the reader to articles such as Hüttel, Lanese, Vasconcelos *et al.* [5] for a deeper introduction.

2.3.1 Binary Session Types

Session types allow for more structure to channel types, by defining the input, output and linear types. With pi-calculus we are only able to keep track of the number of arguments passed through a channel, so preventing that the number of channels send by a participant is different from the recipients expectations. With session type we can define the types passed i.e. *(nat, (nat))* that describes a channel where the recipient expects a pair of values of natural numbers and a channel on which it will reply another natural number [5].

Input and Output types Session types allow for even more refinement, by including information on how channels are used in relation to input and output types. For the input types (processes that can only read from the associated channel) the following notation is used $!(T_1,...,T_n)$, while for output types (processes that can only write on the channel) the following notation is used $?(T_1,...,T_n)$. With the previous mentioned channel we can now define it in more detail $?(nat, !(nat))$ where the channel can only read a pair of nat values and write back a natural number.

Linear types Using the idea from linear logic, session types further introduce *multiplicities* on top of polarities to control the number of times a channel can be used. Using the same example, we can now add $!^1(T_1, ...T_n)$ to the channel to

showcase that it can only be used once for output, while $?^{\omega}(T_1, ...T_n)$ can be used zero or more times for input.

Binary session types As binary session types only allow for exactly two participants, we can introduce the *duality* of a session type. If one participant describe the types of a message exchange as !nat.?bool.end, where one expects to output a string and input a boolean, the *dual* would be ?nat.!bool.end, with .end denoting the end of the protocol. Another important construct presented in session types, is *choice*. Choice allow for a participant to choose between multiple options. An example could be the interaction between a client and an ATM, where the ATM present the client with the option to choose either *deposit* or *withdraw*. From the clients point of view, the options would be presented as $\oplus\{deposit : T_1, withdraw : T_2\}$, while the *dual* seen from the ATM's point of view, would look like $\&\{deposit : T_3, withdraw : T_4\}$. This also mean that the two types T_1 and T_2 are a dual of T_3 and T_3.

2.3.2 Multiparty Session Types

Multiparty session types extend the theory of binary session types to include more than two participants. It does so by describing interactions of a session in a top-down manner as a *global description* of all the messages exchanged, instead of separately looking at the behaviour of each individual channel endpoint, as done so in binary session types. This can ensure protocol conformance and preventing deadlocks, while making it easier to detect errors both manually and by automatic means.

Taking the example made by Hüttel, Lanese, Vasconcelos *et al.* of an interaction between three participants of a Client, ATM and the Bank, we can here show how an implementation of global session types look:

$$\mathsf{Client} \to \mathsf{ATM}(\mathsf{string}).\ \mathsf{rec}\, a.$$
$$\mathsf{Client} \to \mathsf{ATM} \left\{ \begin{array}{l} \mathit{deposit} : \mathsf{Client} \to \mathsf{ATM}(\mathsf{nat}).\ \mathsf{ATM} \to \mathsf{Bank}\{\mathit{deposit} : a\}, \\ \mathit{withdraw} : \mathsf{Client} \to \mathsf{ATM}(\mathsf{nat}). \\ \quad \mathsf{ATM} \to \mathsf{Bank} \left\{ \mathit{withdraw} : \mathsf{ATM} \to \mathsf{Client} \left\{ \begin{array}{l} \mathit{dispense} : a, \\ \mathit{overdraft} : a \end{array} \right\} \right\}, \\ \mathit{balance} : \mathsf{ATM} \to \mathsf{Client}(\mathsf{nat}).\ \mathsf{ATM} \to \mathsf{Bank}\{\mathit{balance} : a\}, \\ \mathit{quit} : \mathsf{ATM} \to \mathsf{Bank}\{\mathit{quit} : \mathsf{end}\}. \end{array} \right\}.$$

First we have that the Client interacts with the ATM, here expecting a string input from the Client, which is one of the key interactions operations of *global types*. The ATM then branches and gives the Client four different options to choose from; *deposit, withdraw, balance* or *quit*, from which the ATM further reports the selected branch to the Bank. Looking at *deposit*, we can tell that the ATM expects a natural number from the client, and then reports the branch picked by the Client, to the Bank. The same happens in *withdraw*, but here the ATM further branches into the selection of *dispence* or *overdraft*.

Thus *global types* helps specify the order of the interaction between the three participants in relation to exchanging messages and the order of requests involved.

2.4 TPM

The Trusted Platform Module (TPM) is a specialised chip storing RSA encryption keys to provide hardware-based security related functions. It is used as a component on an endpoint device and can be seen used in a lot of modern computers for the Windows BitLocker. The TPM contains a RSA key pair of the Endorsement Key (EK) and the owner-specified password. When a user takes ownership of the TPM, a *Storage Root Key* (SRK) is generated. The SRK works as the root of a tree like structure, where keys are stored and later used for encrypting and decrypting data. To each TPM key a 160-bit string is associated called the *authdata*, and works like a password to authorise the use of a key.

2.4.1 The API

The API offers operations related to; *Secure key management and storage*, which generates new keys and impose restrictions on their use; *Platform configuration registers (PCR)* which stores hashes of measurements taken by external software and later lock those by signing them with a specified key. This allows for the TPM to provide *root of trust* for a variety of applications such as:

- *Secure storage*: allows the user to securely store content which is encrypted with a key only available to the TPM
- *Platform authentication*: where a platform can obtain keys by which it can authenticate itself reliably
- *Platform measurement and reporting*: where a platform can create reports of its integrity and configuration state that can be relied on by a remote verifier

The TPM's application program interface (API) offers a wide variety of commands, that e.g. allow the user to load new keys or certify a key by another one. As the TPM offers more than 90 different commands through its API, the report will only focus on a small sample of these. Common for all commands is that they have to be called inside an *authorisation session*, so the user will first have to choose between one of the following sessions:

- Object Independent Authorisation Protocol (OIAP)
- Object Specific Authorisation Protocol (OSAP)

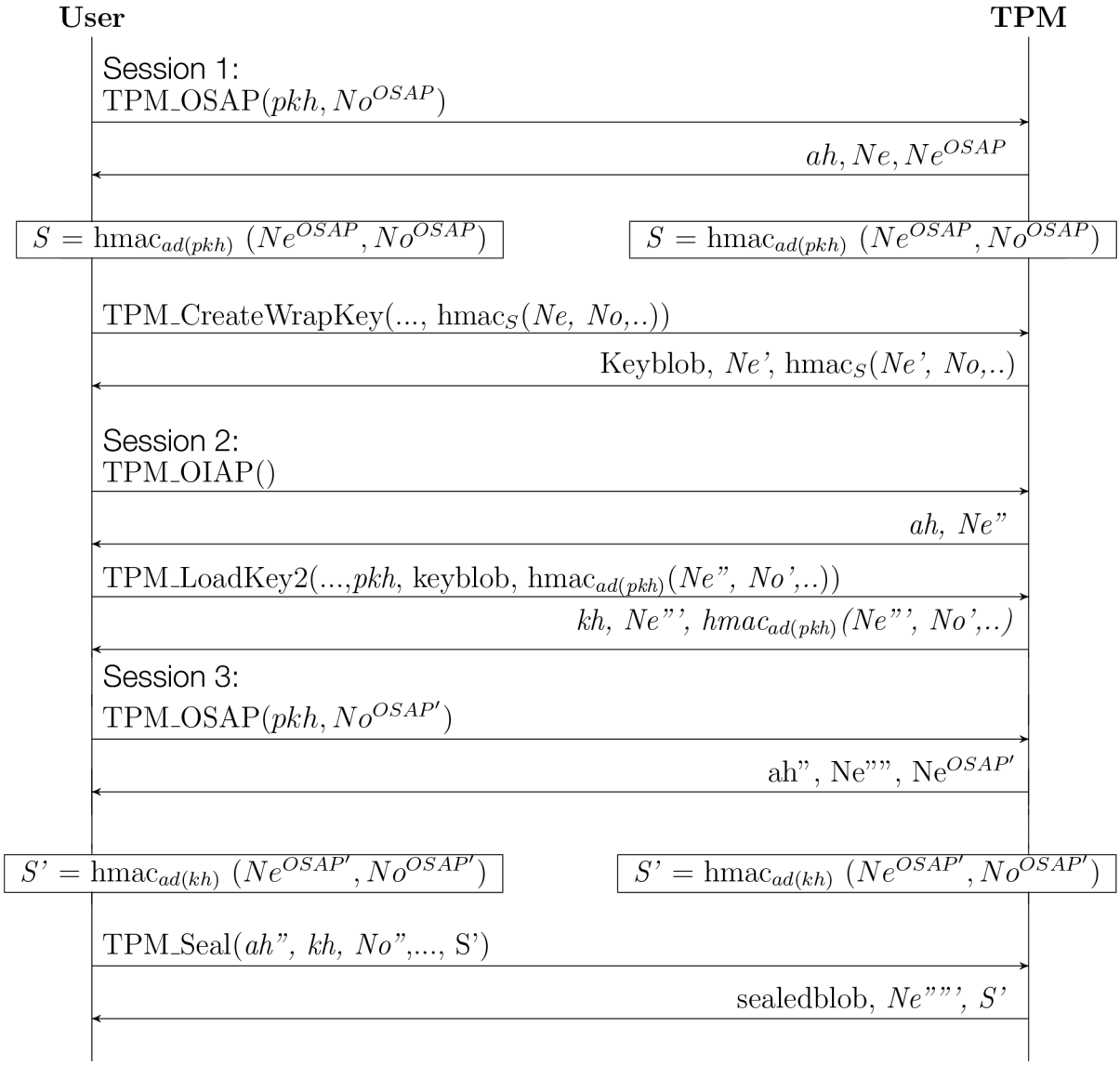
The OIAP creates a session that can manipulate any object, but will only work with certain commands. When setting up the session, the TPM will send back a *session handle* and a fresh *even nonce* as part of its arguments. The OSAP creates a session that can only manipulate a specific object, specified at the session start, so when starting the session, the user will have to send with it the *key handle* of the object and an *odd nonce*. This rotation of nonces, with the user's defined as *odd nonces (No)* and the TPM's as *even nonce (Ne)*, guarantees freshness of the commands and responses, which are then encrypted with an HMAC algorithm and works as a *shared secret* $\text{hmac}(auth, \langle Ne^{OSAP}, No^{OSAP} \rangle)$.

2.4.2 Authorisation

To illustrate the exchange of messages between the TPM and a user, three commands besides the OIAP and OSAP has been chosen to get a better look at the authorisation going on:

- *TPM_CreateWrapKey*: Creates a new key in the storage key tree
- *TPM_LoadKey2*: Loads a key from the tree into the TPM internal memory
- *TPM_Seal*: Uses a key to encrypt data and binding it to a specific PCR value

The following illustration shows the message exchange of how a user starts an OSAP session (1) with the TPM to create a new key in the storage tree. The user then request an OIAP session (2) with the TPM to load the key handle of the just created key, and finishes with a third OSAP session (3) to use the key to seal some arbitrary data. The OSAP sessions are authenticated through the *shared secret*, shown as S and S'.



Session 1: The user sends the request to start an OSAP session with the TPM based on a child key (called parent key *pkh*) of a previous loaded key and the freshly generated odd nonce No^{OSAP}. The TPM respond by sending back a new session authorisation handle *ah* and two new even nonces *Ne* and Ne^{OSAP}. Individually the User and TPM then generate the *shared secret S* derived from the *pkh* and the two nonces, encrypted by a HMAC algorithm. The user can now request the TPM to create a new key by TPM_CreateWrapKey, using the shared secret as authentication and other parameters for creating the key. The TPM checks

the HMAC encryption, creates the key and sends back a *keyblob* consisting of the public key and an encrypted package containing the private key and the new authdata, together with the shared secret. The session then terminates, as the shared secret has been used to create a key.

Session 2: The user request an OIAP session with the TPM, and the TPM answers by sending back an authorisation handle *ah'* and a fresh even nonce *Ne"*. The user then requests to load the previously created key, by calling TPM_LoadKey2 and providing arguments including *pkh*, *keyblob* and a HMAC encryption of the nonces. The TPM checks the encryption, decrypts the *keyblob* and loads the key into its internal memory. It send back the key handle *kh*, a new nonce and the HMAC encryption for authentication. The user can now use this new key handle to encrypt data by using TPM_Seal, as seen in the next session.

Session 3: Again the User requests an OSAP session as in session 1, but this time uses the key handle of the newly loaded key *kh* and authdata *ah"*, together with a new nonce. The *shared secret S'* is generated, and the User now requests to seal some arbitrary data by the TPM_Seal command, together with the shared secret, the key handle and a new nonce. The TPM respond after having checked the HMAC encryption, by sending back a *sealedblob* containing an encrypted package of the sealed data, together with the shared secret. After this the OSAP will end, as the has been used to encrypt the data.

The next section will focus on how to combine the different fields presented, and highlight some of the problematics related to it.

2.5 Evaluation

With each topic now presented, we need to see how they all come together, to form the basis for the future thesis. As the thesis will focus mostly on session types and the TPM in a security related notion I will here demonstrate how these can be put together to form TPM commands with session types. To illustrate the TPM commands, the code of Delaune, Kremer, Ryan *et al.* has been used for reference, and can be found in the appendix. In their example we find the commands:

- *Read*: Read the value of the PCR
- *Quote*: Generate a certificate of an input value
- *CreateWrapKey*: Create wrap key
- *LoadKey2*: Loads wrap key
- *CertifyKey*: Certifies a key
- *Unbind*: Unbinds PCR value
- *Seal*: Seals data
- *Unseal*: Unseals data
- *Extend*: Extends PCR value

On the next page, can be seen how each of these commands are constructed by session types. It should be noted, that in the process of constructing the commands, certain limitations were found within the expressivity of session types such as simplifying conditionals and what to do with cryptographic messages. The original pi-calculus model uses memory cells, which might allow for a better way to express this with session types, e.g. using some concept of state. Another thing session types are missing to model security protocols, is a way to express the violation of security properties, as session types are mainly concerned about correctness and thus missing the security part. These concerns are some of the main points for the motivation of writing the future thesis.

TPM → KeyStorage (parameters)

TPM → Alice : PrivateKey. rec a

$$
\left\{
\begin{array}{ll}
\textit{Read}: & \text{Alice} \rightarrow \text{TPM(bitstring)}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{TPM} \rightarrow \text{Alice(pcr\_value)}.\ \text{rec a} \\
\textit{Quote}: & \text{Alice} \rightarrow \text{TPM(bitstring)}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{TPM} \rightarrow \text{Alice} : \text{CertPCR}.\ \text{rec a} \\
\textit{CreateWrapKey}: & \text{Alice} \rightarrow \text{TPM(data)}.\ \text{PCR} \rightarrow \text{TPm(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{KeyTable} \rightarrow \text{TPM} : \text{KeyLoaded}.\ \text{TPM} \rightarrow \text{Alice(data)}.\ \text{rec a} \\
\textit{LoadKey2}: & \text{Alice} \rightarrow \text{TPM} : \text{LoadKey2}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{KeyTable} \rightarrow \text{TPM} : \text{KeyLoaded}.\ \text{TPM} \rightarrow \text{KeyTable} : \text{KeyLoaded}.\ \text{rec a} \\
\textit{CertifyKey}: & \text{Alice} \rightarrow \text{TPM(data)}\ \text{KeyTable} \rightarrow \text{TPM} : \text{KeyLoaded}.\ \text{TPM} \rightarrow \text{Alice} : \text{Cert}.\ \text{rec a} \\
\textit{Unbind}: & \text{Alice} \rightarrow \text{TPM} : \text{unbind}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{KeyTable} \rightarrow \text{TPM} : \text{KeyLoaded}.\ \text{TPM} \rightarrow \text{Alice} : \text{adec(data)}.\ \text{rec a} \\
\textit{Seal}: & \text{Alice} \rightarrow \text{TPM} : \text{seal}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{KeyTable} \rightarrow \text{TPM} : \text{KeyLoaded}.\ \text{TPM} \rightarrow \text{Alice(data)}.\ \text{rec a} \\
\textit{Unseal}: & \text{Alice} \rightarrow \text{TPM} : \text{unseal}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)}.\ \text{KeyTable} \rightarrow \text{TPM} : \text{KeyLoaded}.\ \text{TPM} \rightarrow \text{Alice} : \text{KeyLoaded}.\ \text{rec a} \\
\textit{Extend}: & \text{Alice} \rightarrow \text{TPM} : \text{extend}.\ \text{PCR} \rightarrow \text{TPM(pcr\_value)}.\ \text{TPM} \rightarrow \text{PCR(pcr\_value)} : \text{hpcr}.\ \text{rec a}
\end{array}
\right\}.
$$

The reader will notice that a lot of the commands look very similar, which proves that lack of expressiveness in session types when used on security protocols.

3 | Future Plan

This section elaborates on the tasks intended for the thesis, building upon the findings in this report. Each of the presented topics provide knowledge on how to construct and reason about communication protocols, and together will form the basis for the future thesis.

3.1 Extending Session Types

In the previous chapter I slightly touched upon applying session types to the TPM commands. As could be deducted from this, global session types still exhibits some limitations of its expressivity, which is the prime motivation for the thesis, thus making one of the first tasks to reason about how to extend session types to model security properties and prove protocols such as the TPM. To start this of, I will be focusing on language design and how to change the language of session types to express all the features we want in a security protocol like the TPM. The second phase will be about constructing an analysis procedure for security properties, as secrecy and authentication are important properties of the TPM. For this I will most likely be using existing tools for automated reasoning about security properties found in security protocols such as ProVerif or Tamarin. It should be noted that Chen and Ryan had to hand-code their analysis in Horn-Clause (in practice deduction rules) instead of using the applied pi-calculus, due to ProVerif not terminating otherwise. Next will be the implementation of a model for the TPM in session types, and finally an evaluation of the analysis procedure against the model that has been build, and comparing it to other approaches. As part of my thesis, I will also be developing a compiler in F# to automate the construction of the protocols.

References

[1] S. Delaune, S. Kremer, M. D. Ryan and G. Steel, "A formal analysis of authentication in the TPM", in *Formal Aspects of Security and Trust - 7th International Workshop, FAST 2010, Pisa, Italy, September 16-17, 2010. Revised Selected Papers*, 2010.

[2] V. Cortier and S. Kremer, "Formal models and techniques for analyzing security protocols: A tutorial", *Foundations and Trends in Programming Languages*, vol. 1, no. 3, pp. 151–267, 2014.

[3] M. D. Ryan and B. Smyth, "Applied pi calculus", 2010.

[4] B. Blanchet, *Proverif*. [Online]. Available: `http://prosecco.gforge.inria.fr/personal/bblanche/proverif/` (visited on 03/12/2018).

[5] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. T. Vieira and G. Zavattaro, "Foundations of session types and behavioural contracts", *ACM Comput. Surv.*, vol. 49, no. 1, 3:1–3:36, 2016.

[6] S. Delaune, S. Kremer, M. D. Ryan and G. Steel, "Formal analysis of protocols based on TPM state registers", in *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, 2011, pp. 66–80.

[7] A. Mukhamedov, A. D. Gordon and M. Ryan, "Towards a verified reference implementation of a trusted platform module", in *Security Protocols XVII, 17th International Workshop, Cambridge, UK, April 1-3, 2009. Revised Selected Papers*, 2009.

[8] S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts and R. Plaga, "Security evaluation of scenarios based on the tcg's TPM specification", in *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, 2007.

[9] L. Chen and M. Ryan, "Attack, solution and verification for shared authorisation data in TCG TPM", in *Formal Aspects in Security and Trust, 6th International Workshop, FAST 2009, Eindhoven, The Netherlands, November 5-6, 2009, Revised Selected Papers*, 2009, pp. 201–216.

[10] V. T. Vasconcelos, "Fundamentals of session types", *Inf. Comput.*, vol. 217, pp. 52–70, 2012.

Appendices

```
(* Bitlocker protocol.

Found in "Formal analysis of protocols based on TPM state registers."
by Stéphanie Delaune, Steve Kremer, Mark D. Ryan, and Graham Steel
in Proceedings of the 24th IEEE Computer Security Foundations Symposium
(CSF'11), pp. 66–82, IEEE Computer Society Press, Cernay-la-Ville, France, June
2011.
*)

(*******************************
***      TPM Declaration      ***
*******************************)

free c:channel.
free pcr:channel [private].

type type_key.

free bindk:type_key.
free sealk:type_key.

type private_key.
type public_key.

type value_pcr.
free nil:value_pcr.

(* Functions of the TPM *)

fun hpcr(value_pcr,bitstring) : value_pcr.
free init : value_pcr.

fun pk(private_key) : public_key.
fun certPCR(private_key,value_pcr,bitstring) : bitstring.
fun certKey(private_key,public_key,value_pcr) : bitstring.

reduc forall sk:private_key, v:value_pcr, d:bitstring ;
check_certPCR(certPCR(sk,v,d),pk(sk)) = (v,d).
reduc forall sk:private_key, v:value_pcr, xpk:public_key ;
check_certKey(certKey(sk,xpk,v),pk(sk)) = (xpk,v).

fun aenc(public_key,bitstring) : bitstring.
reduc forall sk:private_key, d:bitstring; adec(sk,aenc(pk(sk),d)) = d.

free aik:private_key [private].
free srk:private_key [private].
table keyloaded(private_key,public_key,type_key,value_pcr).

fun wrap(public_key,private_key,type_key,bitstring,value_pcr) : bitstring.
```

```
50  reduc forall x_pk:public_key, x_key:private_key, t_key:type_key, data:bitstring,
51  x_pcr:value_pcr;
52    unwrap(wrap(x_pk,x_key,t_key,data, x_pcr)) = (x_pk,x_key,t_key,data,x_pcr)
53  [private].
54
55  fun seal(public_key,bitstring,bitstring,value_pcr) : bitstring.
56  reduc forall x_pk:public_key, d:bitstring, p:bitstring, v:value_pcr;
57  unseal(seal(x_pk,d,p,v)) = (x_pk,d,p,v) [private].
58
59  (*********************************
60  ***     TPM Functionality     ***
61  *********************************)
62
63  (* The commands *)
64  free load: bitstring.
65  free read: bitstring.
66  free quote: bitstring.
67  free wrap_key : bitstring.
68  free certify : bitstring.
69  free unbind : bitstring.
70  free seal_data: bitstring.
71  free unseal_data: bitstring.
72  free extend : bitstring.
73  free reboot : bitstring.
74  free tpm_proof: bitstring [private].
75
76  (* Read the value of the PCR *)
77  let Read =
78    in(c,=read);
79    in(pcr,v:value_pcr);
80    out(pcr,v);
81    out(c,v).
82
83  (* Generate a certificate of an input value. *)
84  let Quote =
85    in(c,(=quote,x:bitstring));
86    in(pcr,v:value_pcr);
87    out(pcr,v);
88    out(c,certPCR(aik,v,x)).
89
90  (* Create Wrap Key *)
91  let CreateWrapKey =
92    in(c,(=wrap_key,x_pk:public_key,t:type_key,v_lock:value_pcr));
93    in(pcr,v_cur:value_pcr);
94    out(pcr,v_cur);
95    get keyloaded(x_key:private_key,=x_pk,t':type_key,v:value_pcr) in
96    if v = nil || v = v_cur then
97    new key[v_cur,v_lock]:private_key;
98    out(c, (pk(key),wrap(x_pk,key,t,tpm_proof,v_lock))).
```

```
 99
100   (* Load wrapped key *)
101   let LoadKey2 =
102     in(c,(=load,x_pk:public_key,x_w:bitstring));
103     let (y_pk:public_key,x_key:private_key,t:type_key,=tpm_proof,x_pcr:value_pcr) =
104   unwrap(x_w) in
105     if pk(x_key) = x_pk then
106     in(pcr,v:value_pcr);
107     out(pcr,v);
108     get keyloaded(x_sk:private_key,=y_pk,t':type_key,v':value_pcr) in
109     if v = v' || v' = nil then
110     insert keyloaded(x_key,x_pk,t,x_pcr).
111
112   (* Certify Key *)
113   let CertifyKey =
114     in(c,(=certify,x_pk:public_key));
115     get keyloaded(x_key:private_key,=x_pk,t:type_key,v:value_pcr) in
116     out(c,certKey(aik,x_pk,v)).
117
118   (* Unbind *)
119   let Unbind =
120     in(c,(=unbind, x_pk:public_key, cypher:bitstring));
121     in(pcr,v:value_pcr);
122     out(pcr,v);
123     get keyloaded(x_sk:private_key,=x_pk,=bindk,v':value_pcr) in
124     if v' = nil || v = v' then
125     out(c,adec(x_sk,cypher)).
126
127   (* Seal *)
128   let Seal =
129     in(c,(=seal_data, d:bitstring, x_pcr:value_pcr, x_pk:public_key));
130     in(pcr,v:value_pcr);
131     out(pcr,v);
132     get keyloaded(x_sk:private_key,=x_pk,=sealk,v':value_pcr) in
133     if v' = nil || v = v' then
134     out(c,seal(x_pk, d, tpm_proof, x_pcr)).
135
136   (* Unseal *)
137   let Unseal =
138     in(c,(=unseal_data, x:bitstring));
139     let (x_pk:public_key,d:bitstring,=tpm_proof,v':value_pcr) = unseal(x) in
140     in(pcr,v:value_pcr);
141     out(pcr,v);
142     get keyloaded(x_sk:private_key,=x_pk,=sealk,v'':value_pcr) in
143     if (v' = nil && v'' = nil) || (v' = nil && v = v'') || (v' = v && v'' = nil) || (v' = v && v''
144   = v) then
145     out(c,d).
146
147   (* Extend *)
```

```
148  let Extend =
149    in(c,(=extend, x:bitstring));
150    in(pcr,v:value_pcr);
151    out(pcr,hpcr(v,x)).
152
153  let Initialisation =
154    insert keyloaded(srk,pk(srk),bindk,nil) | out(c, pk(srk)).
155
156  let Main_TPM =
157    Initialisation | ! (Read | Quote | CreateWrapKey | LoadKey2 | CertifyKey |
158  Unbind | Seal | Unseal | Extend).
159
160  free deny:bitstring.
161
162  (** Alice role **)
163
164  free vmk:bitstring [private].
165  free bios:bitstring.
166  free loader:bitstring.
167
168  fun abs_secret(bitstring,bitstring):bitstring [private].
169
170  let Alice =
171    out(c,(wrap_key,pk(srk),sealk,nil));
172    in(c,(x_pk:public_key,w:bitstring));
173    out(c,(load,x_pk,w));
174    get keyloaded(x_sk:private_key,=x_pk,=sealk,v':value_pcr) in
175    out(c,seal(x_pk,vmk,tpm_proof,hpcr(hpcr(init,bios),loader))).
176
177  let reboot_and_measure_BIOS_and_loader =
178    in(c,(x_bios:bitstring,x_loader:bitstring));
179    in(pcr,v:value_pcr);
180    if x_bios = bios && x_loader = loader
181    then out(pcr,hpcr(hpcr(hpcr(init,x_bios),x_loader),deny))
182    else if x_bios = bios
183    then out(pcr,hpcr(hpcr(init,bios),x_loader))
184    else out(pcr,hpcr(init,x_bios)).
185
186  let first_boot_measure_BIOS_and_loader =
187    in(c,(x_bios:bitstring,x_loader:bitstring));
188    if x_bios = bios && x_loader = loader
189    then out(pcr,hpcr(hpcr(hpcr(init,x_bios),x_loader),deny))
190    else if x_bios = bios
191    then out(pcr,hpcr(hpcr(init,bios),x_loader))
192    else out(pcr,hpcr(init,x_bios)).
193
194  let Main_Process =
195      (! (Alice | reboot_and_measure_BIOS_and_loader) ) | Main_TPM |
196  first_boot_measure_BIOS_and_loader.
```

```
197
198     query attacker(vmk).
199
200     process Main_Process  | ! in(pcr,x:value_pcr); out(pcr,x)
201
```