Extending Session Types to Model Security Properties

Julie Tollund

10th December 2018

Contents

1 | Introduction

The purpose of this report, is to establish a foundation for a 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 further 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.
Voting machines
Confidentiality
Ever expanding field
Tighter restrictions (session types - adversaries)
TPM (new security measurement)

1.2 Current research

The project relies heavily on the research done within the field of Security protocols and Session types.
Mark Ryan's research on the TPM (citation).
ProVerif?

1.3 Intended outcome

"The intended outcome of the thesis, is to take the idea of session type and choreography programming, 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." (Taken from Thesis prep agreement). Compiler.

2 | Work Done

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 usually 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. (The Dolev-Yao model)

2.1.1 Needham-Schroeder Protocol

The Needham-Schroeder Public Key Protocol, was first proposed by Roger Needham and Michael Schroeder in 1978 (ref?), 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

$$
\begin{aligned}
&\mathrm{A} \xrightarrow{\{|\langle A, N_a\rangle|\}^a_{\mathsf{pk}(B)}} \qquad \xrightarrow{\{|\langle x, y\rangle|\}^a_{\mathsf{pk}(B)}} \mathrm{B} \\
&\mathrm{A} \xleftarrow{\{|\langle N_a, z\rangle|\}^a_{\mathsf{pk}(A)}} \qquad \xleftarrow{\{|\langle y, N_b\rangle|\}^a_{\mathsf{pk}(x)}} \mathrm{B} \\
&\mathrm{A} \xrightarrow{\{|z|\}^a_{\mathsf{pk}(B)}} \qquad \xrightarrow{\{|N_b|\}^a_{\mathsf{pk}(B)}} \mathrm{B}
\end{aligned}
$$

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, so a concatenation of the two messages. 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, 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 [7], 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 a fix was also purposed. Again the Alice and Bob notation is used, but now we introduce an adversary C, as illustrated here by Cortier and Kremer.

$$
\begin{array}{lcl}
\mathrm{A} & \xrightarrow{\{|\langle A, N_a \rangle|\}^a_{\mathsf{pk}(C)}} & \mathrm{C} \\
 & & \downarrow \\
 & & \mathrm{C(A)} \xrightarrow{\{|\langle A, N_a \rangle|\}^a_{\mathsf{pk}(B)}} \mathrm{B} \\
 & & \mathrm{C(A)} \xleftarrow{\{|\langle N_a, N_b \rangle|\}^a_{\mathsf{pk}(A)}} \mathrm{B} \\
\mathrm{A} & \xleftarrow{\{|\langle N_a, N_b \rangle|\}^a_{\mathsf{pk}(A)}} & \mathrm{C} \\
\mathrm{A} & \xrightarrow{\{|N_b|\}^a_{\mathsf{pk}(C)}} & \mathrm{C} \\
 & & \downarrow \\
 & & \mathrm{C(A)} \xrightarrow{\{|N_b|\}^a_{\mathsf{pk}(B)}} \mathrm{B}
\end{array}
$$

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 $\{|\langle N_a, \langle N_b, B\rangle\rangle|\}^a_{pk(a)}$

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 through induction rules and derivation sequences. This

TODO: Terms

Inference rules(system)

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 rest being the decomposition rules.

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

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 do so, if they have the corresponding key. The same goes for en asymmetric encryption and decryption, with the rules shown in the third line.

TODO: Derivation sequence (Deduction rules?)
By combining inference rule, we can derive new messages. .... Inference rules can be combined to compute or derive new messages (rewrite), where the corresponding deduction steps can be shown by a proof tree.

2.2 Applied π-Calculus

The applied pi-calculus (ref. Abadi and Fournet, 2001) 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 [1]. This is done by introducing a rich term algebra for modelling the cryptographic operations used in security protocols, where function symbols represent cryptographic protocols.

Tools such as ProVerif [9] 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 will most likely be used in the following thesis.

2.2.1 Syntax

As mentioned, the applied pi-calculus is not restricted to communication names, but offers processes where they output terms representing messages instead. The applied pi-calculus has two two types of process, the *plain* and *extended* processes. First we describe the grammar for the *plain* process, as shown below (fig. ref?).:

$P,\ Q,\ R ::=$	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 NS Protocol; The conditional if $M = N$ then P else Q is how we know from normal conditioning, 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* as done by Cortier and Kremer

Processes are extended with *active substitutions* to capture the knowledge exposed to the adversarial environment: (rewirte)

$A, B, C ::=$	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 $x = M$ 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

Terms represent messages that are exhanged, and are inductively build over a finite set of variables $X = \{x, y, z\}$ and.... Variable can represent any term. Names are used to represent atomic values, such as key and nonces. A term is *ground* if it dies not contain any variables. (rewrite!) TODO: description (maybe first)

2.2.2 Re-visiting the Needham-Schroeder Protocol

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

To make this distinction explicit we parametrize the processes representing the initiator and responder with the keys of the agents who execute the role. (rewrite!!):

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

First Alice generates 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 $n_x b$, representing the nonce received by the sender (Bob). Last she sendt out a message again on channel c.

The dual of the process can now be modelled from the responders 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 as a whole: Cortier and Kremer (Describe P_A and P_B first):

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

This however represent the naive model of the protocol, so to add the Lowe fix, as mentioned earlier i then report, we add the identity to the encryptions: (TODO: needs further explanation)

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

TODO: smooth transition to the next chapter + don't forget reference to article!

2.3 Session Types

As with the applied pi-calculus, session types also have its root in the process 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 checked either at compile-time or runtime, 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.* 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.

Input and Output types
Furthermore it allows 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 write on channels) the following notation is used $!(T_1,...,T_n)$. For output types (processes that can only write on channels) 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 a natural number.

Linear types *duality*

Binary session types

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 an 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.

TODO: short description of choreography programming

2.4 TPM

The Trusted Platform Module (TPM) is a specialised chip that stores RSA encryption keys specific for the host system for hardware authentication. It is used as a component on an endpoint device and is used for the Windows BitLocker. The TPM contains an 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 future keys are stored, by which are used for encrypting 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 and its commands

The API offers operations related to; *Secure key management and storage*, which generates new key 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*: a platform can obtain keys by which it can authenticate itself reliably (rewrite)
- *Platform measurement and reporting*: A platform can create reports of its integrity and configuration state that can be relied on by a remote verifier (rewirte)

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 application program interface (API), the report will only focus on a small sample of these. Each command has 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)

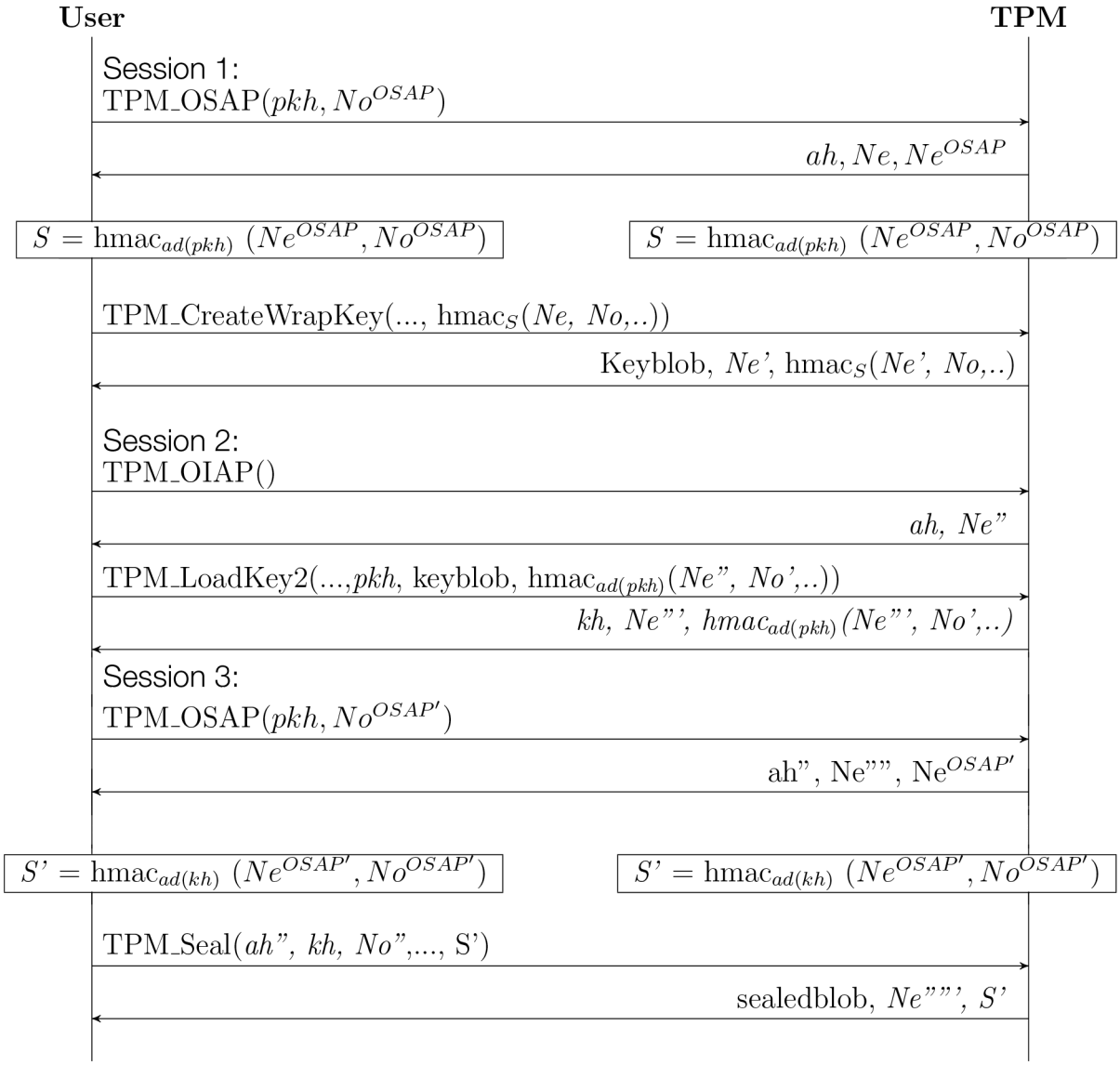
The OIAP creates a session that can manipulate any object, but will only work with certain commands. When setting up the session, the TPM will send back a *session handle* and a fresh *even nonce* as part of its arguments. The OSAP creates a session that can only manipulate a specific object, specified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

2.4.2 Example of commands

To illustrate the exchange of message 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*: Load key from tree into the TPM internal memory
- *TPM_Seal*: Uses key to encrypt data

TODO: describe pkh! In the following illustration, the user starts with creating an OSAP session with the TPM to generate a *shared secret* = S.


User
TPM
Session 1:
TPM_OSAP(pkh, No^{OSAP})
ah, Ne, Ne^{OSAP}
$S = \text{hmac}_{ad(pkh)}\ (Ne^{OSAP}, No^{OSAP})$
$S = \text{hmac}_{ad(pkh)}\ (Ne^{OSAP}, No^{OSAP})$
TPM_CreateWrapKey(..., $\text{hmac}_S(Ne,\ No,..)$)
Keyblob, Ne', $\text{hmac}_S(Ne',\ No,..)$
Session 2:
TPM_OIAP()
$ah,\ Ne"$
TPM_LoadKey2(...,pkh, keyblob, $\text{hmac}_{ad(pkh)}(Ne",\ No',..)$)
$kh,\ Ne"',\ hmac_{ad(pkh)}(Ne"',\ No',..)$
Session 3:
TPM_OSAP($pkh, No^{OSAP'}$)
ah", Ne"", $\text{Ne}^{OSAP'}$
$S' = \text{hmac}_{ad(kh)}\ (Ne^{OSAP'}, No^{OSAP'})$
$S' = \text{hmac}_{ad(kh)}\ (Ne^{OSAP'}, No^{OSAP'})$
TPM_Seal($ah",\ kh,\ No"$,..., S')
sealedblob, $Ne""',\ S'$

2.5 Evaluation

- How they all come together
- Examples of TPM commands with session types

2.5.1 TPM as Session Types

TPM → KeyStorage (parameters)
TPM → Alice : PrivateKey. rec a

$$
\left\{
\begin{array}{ll}
\textit{Read}: & \text{Alice} \to \text{TPM} : \text{read. PCR} \to \text{TPM} : \text{value. TPM} \to \text{PCR} : \text{value. TPM} \to \text{Alice} : \text{value. rec a} \\
\textit{Quote}: & \text{Alice} \to \text{TPM} : \text{quote. PCR} \to \text{TPM} : \text{value. TPM} \to \text{PCR} : \text{value. TPM} \to \text{Alice} : \text{CertPCR. rec a} \\
\textit{CreateWrapKey}: & \text{Alice} \to \text{TPM} : \text{CreateWrapKey. PCR} \to \text{TPm} : \text{value. TPM} \to \text{PCR} : \text{value. KeyTable} \to \text{TPM} : \text{KeyLoaded. } \textbf{?(new key)?}\text{. TPM} \to \text{Alice} : \text{data. rec a} \\
\textit{LoadKey2}: & \text{Alice} \to \text{TPM} : \text{LoadKey2. PCR} \to \text{TPM} : \text{value. TPM} \to \text{PCR} : \text{value. KeyTable} \to \text{TPM} : \text{KeyLoaded. TPM} \to \text{KeyTable} : \text{KeyLoaded. rec a} \\
\textit{CertifyKey}: & \text{Alice} \to \text{TPM} : \text{data. KeyTable} \to \text{TPM} : \text{KeyLoaded. TPM} \to \text{Alice} : \text{Cert. rec a} \\
\textit{Unbind}: & \text{Alice} \to \text{TPM} : \text{unbind. PCR} \to \text{TPM} : \text{value. TPM} \to \text{PCR} : \text{value. KeyTable} \to \text{TPM} : \text{KeyLoaded. TPM} \to \text{Alice} : \text{adec. rec a} \\
\textit{Seal}: & \text{Alice} \to \text{TPM} : \text{data. PCR} \to \text{TPM} : \text{value. TPM} \to \text{PCR} : \text{value. KeyTable} \to \text{TPM} : \text{KeyLoaded. TPM} \to \text{Alice} : \text{data. rec a} \\
\textit{Unseal}: & \text{Alice} \to \text{TPM} : \text{unseal. PCR} \to \text{TPM} : \text{value. TPM} \to \text{PCR} : \text{value. KeyTable} \to \text{TPM} : \text{KeyLoaded. TPM} \to \text{Alice} : \text{KeyLoaded. rec a} \\
\textit{Extend}: & \text{Alice} \to \text{TPM} : \text{extend. PCR} \to \text{TPM} : \text{value. TPm} \to \text{PCR} : \text{hpcr. rec a}
\end{array}
\right\}.
$$

TODO: Description of above (+ code in appendix? for refs.)

3 | Future Plan

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 future thesis.

3.1 Introducing an adversary

3.2 Compiler

As part of my thesis, I will also be developing a compiler in F# for the findings.

Farewell

References

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

[2] 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.

[3] 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.

[4] 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.

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

[7] 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.

[8] 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.

Online references

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