Saarland University
Faculty of Natural Sciences and Technology I
Computer Science Department
Master's Program in Computer Science

Master's Thesis

Security Despite System Compromise with Zero-Knowledge Proofs

submitted by

Martin Peter Grochulla

martin@mgrochulla.de

January 8, 2009

Supervisor	Prof. Dr. Michael Backes
Advisors	Dr. Matteo Maffei
	Cătălin Hriţcu
Reviewers	Prof. Dr. Michael Backes
	Dr. Matteo Maffei

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, January 8, 2009

Martin Peter Grochulla

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, January 8, 2009

Martin Peter Grochulla

Abstract

We present a technique to strengthen protocols against system compromise. A system is compromised when some parties are controlled by the attack and deviate from the protocol. Our approach extends protocols with zero-knowledge proofs to convince the receiver of a message that all principals involved in the protocol session have so far followed the protocol. Zero-knowledge proofs allow parties to prove such statements without revealing those messages that are kept secret by the original protocol. Our approach is able to handle public key cryptography, digital signatures, hashes, and symmetric key cryptography.

Protocols are represented as processes in the applied π-calculus. We propose a technique to automatically transform such processes into processes with zero-knowledge proofs which are safe under system compromise. We use an existing type system to statically analyse the safety of the processes with respect to some given authorisation policy. The original protocols are analysed against external intruders and the ones strengthened with zero-knowledge proofs against compromised participants. A type-checker makes our approach fully automated.

Acknowledgement

I am deeply indebted to my advisors, Dr. Matteo Maffei and Cătălin Hriţcu, for their support during the last year. Their patience, their advice, and their help have allowed me to complete this thesis. They have not only given me the opportunity to gain insight into an interesting research field, but they have also shared their enthusiasm for their work with me. Furthermore I thank Prof. Michael Backes and Dr. Matteo Maffei for reviewing this thesis.

Contents

Chapter 1

Introduction

Communicating with each other is an essential part of our life. We communicate in order to acquire or exchange information. With the emergence of the Internet and its services – like e-mail or online banking – new forms of communications have emerged. Interactive communication is one possible variant of communicating. It is governed by rules describing how individual participants should behave in order to exchange information. The rules governing the communication over the internet are called protocols. There are lots of different protocols for different tasks.

1.1 Cryptographic Protocols

Usually it is not harmful to communicate with each other over the internet without taking precautions. However as soon as confidential data is concerned one has to be careful. An attacker may be interested in learning that confidential data. In order to communicate secret data over the internet where attackers are present, one has to use cryptographic protocols. These protocols try to achieve desirable properties by means of cryptography – for example being able to communicate with a specific participant such that no party learns anything. Other properties of protocols besides secrecy are authenticity, anonymity or non-repudiability.

Verifying protocols for specific purposes is hard, because proving properties a protocol should fulfil is hard. This has to do with the analysis techniques for protocols and the fact that the attacker and his attacks may be arbitrary. An attacker can, for instance, intercept and exchange messages sent over the network. He can also try to run several instances of one protocol and try to exploit flaws of the protocol. This is the case for man-in-the-middle attacks by interleaving parallel executions of the protocol or for replay-attacks by reusing sent messages from different protocol executions.

1.2 Basic Cryptography

Cryptographic protocols use cryptographic primitives. Cryptographic primitives are – for example – encryption schemes, signature schemes, and hashes. An encryption scheme consists of two efficient algorithms called encryption and decryption. Applying the encryption algorithm to a message (also called plaintext) m together with a key k yields a ciphertext c. Applying the decryption algorithm to that ciphertext c together with the key k yields again the plaintext m. Furthermore no efficient algorithm can distinguish a ciphertext obtained by encrypting a plaintext m with k from a ciphertext obtained by encrypting a plaintext m' with k without knowing k. Intuitively no one can tell, what message is contained in a ciphertext unless one possesses the correct decryption key k. These encryption schemes are called symmetric encryp-

tion schemes, since k is used for encryption and decryption. Asymmetric encryption schemes use different keys for encryption and decryption: everyone may know the encryption key k^+ (in this case also called public key) and can encrypt messages, but only the owner of the corresponding decryption key k^- (or secret key) is able to decrypt ciphertexts encrypted with his public key.

The setting in which everyone knows the public keys of all participants, but only the owners know their corresponding secret keys is called public key infrastructure. Signature schemes also use a public key infrastructure. But here the public key is called verification key and the secret key is called signing key. A signature scheme consists of two efficient algorithms sign and verify. Signing a message m with the signing key produces a signature s. Verifying a signature s together with the original message m and the corresponding verification key returns *true*. Furthermore no efficient algorithm can compute a signature s' such that applying the verify algorithm to s' and m with the corresponding verification key returns *true*. Intuitively no one can create a valid signature for a message except the owner of the signing key.

A hash $f(x)$ is the value of a hash function f for a bitstring x. A hash function f is a function that maps bitstrings of arbitrary length to bitstrings of some fixed length n. Given $f(x)$ no one can compute the string x from $f(x)$ efficiently. Intuitively the hash of a string x is something that reveals nothing about x and only the someone who knows x can generate $f(x)$.

Another powerful cryptographic primitive are zero-knowledge proofs ([Gol01]):

1.3 Zero-Knowledge Proofs

A zero-knowledge proof is cryptographic primitive which allows one principal (called prover) to prove the validity of a true statement to another principal (called verifier). What makes a zero-knowledge proof so powerful is the fact that it convinces the verifier of a true statement about some secrets although no secrets are revealed in the proof. Consider for example the situation in which the prover wants to convince the verifier of knowing some secret. The prover could just send the secret to the verifier, which would convince him, but then the secret would be revealed to the verifier. An attacker who controls the communication channels between prover and verifier will be able to learn the secret as well. Furthermore a malicious verifier could execute the zero-knowledge proof protocol with an honest prover in order to learn the secret and then say that he is not convinced. A simple solution to that problem is a zero-knowledge proof in which the prover reveals the hash value of the secret and proves to the verifier that he knows a string such that the hash value of that string is equal to the revealed hash value. If the prover does not know the secret, he will not be able to convince the verifier of knowing the secret. On the other hand revealing the hash of the secret does not reveal anything about the secret itself.

We will use zero-knowledge proofs in order to deal with principal compromise in protocols. In contrast to attackers from outside trying to exploit flaws in the protocol or running several instances of one protocol, principal compromise describes the situation in which a principal (participant) is malicious. A malicious participant which does not behave as intended by the protocol can cause harm because the correctness of the protocol may depend on the correct behaviour of each party.

Chapter 2

Background

Before we present our approach to deal with principal compromise, we first describe a variant of the applied π-calculus in which we represent protocols. We use a variant of the applied π-calculus with constructors and destructors similar to [AB02]. Then we describe how we represent zero-knowledge proofs in the calculus. Finally we present the type system which allows us to check the safety of protocols. We use the applied π-calculus, the representation of zero-knowledge proofs, and the type system as presented in [BHM08a].

2.1 Applied π-calculus

2.1.1 Syntax

A term in the applied π-calculus is either a name, a variable, a constructor application or a tuple. Names can be used to represent secrets, channels, nonces or identifiers. New terms can be built out of names and variables using constructors. For example, the name k_A can represent the secret key of participant A in a protocol. Applying the constructor pk to k_A results in the term $pk(k_A)$ representing the public key of participant A.

$M, N ::=$		terms
	a, b, c, m, n	names
	x, y, z, v, w	variables
	$\langle M_1, \ldots, M_n \rangle$	tuples
	$f\big(M_1, \ldots, M_n\big)$	constructor application

$$f ::= \quad pk^1, enc^2, senc^2, vk^1, sign^2, hash^1, true^0, false^0$$

The constructors *true* and *false* have arity 0; pk, vk and *hash* have arity 1; $enc, senc$, and *sign* have arity 2. *true* and *false* represent the Boolean values; pk yields the public key (encryption key) corresponding to the secret key (decryption key) of an asymmetric encryption scheme; vk yields the verification key to a signing key of a signature scheme; *hash* the hashed value to a given term. *enc* represents the asymmetric encryption of a term with a public key, *senc* represents the symmetric encryption of a term with a symmetric key and *sign* the signature of a term with a signing key.

In order to represent the decryption of an encrypted message or the verification of a signature there are the following destructors:

$$g ::= \quad id^1, eq^2, \wedge^2, \vee^2, dec^2, sdec^2, check^2$$

id is a destructor of arity 1; $eq, \wedge, \vee, dec$, and *check* of arity 2. *id* represents identity; *eq* equality; $\wedge$ conjunction; $\vee$ disjunction; *dec* yields the asymmetric decryption of a term with a decryption key; *sdec* yields the symmetric decryption of a term with a symmetric key; *check* verifies a signature of a signed term with a verification key and yields the term, which has been signed. For convenience we will write $M_1 = M_2$ instead of $eq(M_1, M_2)$, $M_1 \wedge M_2$ instead of $\wedge(M_1, M_2)$, and $M_1 \vee M_2$ instead of $\vee(M_1, M_2)$.

The semantics of destructors is defined by a reduction relation $\Downarrow$. This relation states when an application of a destructor g to some terms $\tilde{N}$ yields another term M. In this case we write $g(\tilde{N}) \Downarrow M$, otherwise the reduction fails and we obtain no term $g(\tilde{N}) \not\Downarrow$ (we use the notation $\tilde{N}$ for a tuple $\langle N_1, \ldots, N_k \rangle$ consisting of several terms – in this case k terms). For the destructors from above we have the following reduction rules:

$$
\begin{array}{ll}
id(M) & \Downarrow M \\
eq(M, M) & \Downarrow true \\
\wedge(true, true) & \Downarrow true \\
\vee(true, M) & \Downarrow true \\
\vee(M, true) & \Downarrow true \\
dec\left(enc\left(M, pk(K)\right), K\right) & \Downarrow M \\
sdec\left(senc\left(M, K\right), K\right) & \Downarrow M \\
check\left(sign\left(M, K\right), vk(K)\right) & \Downarrow M
\end{array}
$$

id represents identity, $id(M)$ can always be reduced to M, while all other destructors cannot always be reduced. Equality, conjunction, and disjunction $eq, \wedge, \vee$ have their canonical meaning. Given an encryption $enc\left(M, pk(K)\right)$ of a term M with an encryption key $pk(K)$ and the corresponding decryption key K, *dec* can be reduced to M. Applying *sdec* to a symmetric encryption $senc(M, K)$ with the key K can be reduced to M. Given a signature $sign\left(M, K\right)$ of a term M with a signing key K and the corresponding verification key $vk(K)$, *check* can be reduced to M. Note that there is no reduction rule for *hash*.

Terms and destructors can be used to build processes. Principals in a protocol are represented by processes in the applied π-calculus. The process $k : \mathtt{out}\left(M, N\right).P$ outputs a message N on channel M and continues with the process P. $k : \mathtt{in}\left(M, x\right).P$ receives a message on channel M. The message will be bound to the variable x and the process continues with process P. $k :!\mathtt{in}\left(M, x\right).P$ is a replicated input. This process behaves like a infinite number of input processes $k : \mathtt{in}\left(M, x\right).P$ running in parallel. Each input and output process has label k. The purpose of the labels is to identify which input process is supposed to receive which term from an output process. Corresponding input and output processes will have the same labels in our protocol description. They are only used by the implementation of our approach and have no computational significance. $\nu n..P$ restricts the name n and then behaves as process P. Restricted names are used to model secrets in the process, while free names are assumed to be public knowledge. Variables are never restricted, instead they are bound in processes. $P|Q$ is a process where processes P and Q are running in parallel. 0 is a process that does nothing. $\mathtt{let}\ x = g(\tilde{M})\ \mathtt{then}\ P\ \mathtt{else}\ Q$ is a destructor evaluation. In this process the destructor g is applied to the terms $\tilde{M}$. If reduction rules can be applied ($g(\tilde{M}) \Downarrow N$) then this process continues with the process P, otherwise the process behaves as process Q. $\mathtt{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathtt{in}\ P$ splits a tuple into its components.

Each participant in a protocol is now represented by a process. Communication is expressed by input and output processes, which send and receive terms. A protocol consisting of several participants is represented by processes running in parallel. Secrets, for example, decryption keys and signing keys are represented by restricted names. Private channels are modelled by using restricted names instead of free names. Encryption and verification keys are output on

public channels. In this case there is no receiver, which means, there is no input process with a label of these output processes. By sending the public and verification keys, the attacker gets access to them as it is the case in a public key infrastructure.

$P, Q, R ::=$		processes
	$k : \mathtt{out}\,(M, N)\,.P$	output
	$k : \mathtt{in}\,(M, x)\,.P$	input
	$k :!\mathtt{in}\,(M, x)\,.P$	replicated input
	$\nu n.P$	restriction
	$P \vert Q$	parallel composition
	0	null process
	$\mathtt{let}\ x = g(\tilde{M})\ \mathtt{then}\ P\ \mathtt{else}\ Q$	destructor evaluation
	$\mathtt{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathtt{in}\ P$	tuple splitting

2.1.2 Semantics

In the following we present rules for structural equivalence $\equiv$. These rules state which processes we consider equivalent up to syntactic rearrangement. We can always add a null process running in parallel to some other process P. A process P running in parallel to a process Q is equivalent to the process Q running in parallel to the process P; a process $P|Q$ running in parallel with a process R is equivalent to the process P running in parallel to the process $Q|R$. The next two rules state that the order of processes in a parallel composition does not play a role. If a name n is not free in a process P ($n \notin fn(P)$, where $fn(P)$ is the set of all free names in process P and $fv(P)$ is the set of all free variables in process P), the process P running in parallel to $\nu n.Q$ is equivalent to P running in parallel to Q, where n is restricted in both processes P and Q. If n would be a free name in P, we would have to rename n in P before applying that rule. By applying that rule we can always order all restrictions on top of the process. Furthermore the order of restrictions of names does not play a role. Finally, $\mathcal{E}[\ ]$ stands for an evaluation context $\mathcal{E}[\ ] = \nu\tilde{a} : \tilde{T}.([\ ]|P)$. This is used to express that all the rules can be applied inside a process:

$$
\begin{aligned}
P|0 &\equiv P \\
P|Q &\equiv Q|P \\
(P|Q)|R &\equiv P|(Q|R) \\
\nu n.(P|Q) &\equiv P|\nu n.Q, && \text{if } n \notin fn(P) \\
\nu n_1.\nu n_2.P &\equiv \nu n_2.\nu n_1.P, && \text{if } n_1 \neq n_2 \\
\mathcal{E}[P] &\equiv \mathcal{E}[Q], && \text{if } P \equiv Q
\end{aligned}
$$

Execution of a protocol represented by a process is described by the rules of internal reduction $\rightarrow$. An output process $k : \mathtt{out}\,(a, M)\,.P$ running in parallel to an input process $k : \mathtt{in}\,(a, x)\,.Q$ can be reduced to P running in parallel to $Q\{M/x\}$, where the output message M is bound to x in Q, if the channels and the labels of the input and output process are the same. If the input process is a replicated input process $k :!\mathtt{in}\,(u, x)\,.Q$ additionally $k :!\mathtt{in}\,(u, x)\,.Q$ runs in parallel to P and $Q\{M/x\}$. For the destructor evaluation there are two cases: either reduction rules can be applied or not. If reduction rules can be applied ($g(\tilde{M}) \Downarrow N$) the process $\mathtt{let}\ x = g(\tilde{M})\ \mathtt{then}\ P\ \mathtt{else}\ Q$ behaves as $P\{N/x\}$, where x is replaced by the result of the destructor evaluation in P. If no reduction rule is applicable, $\mathtt{let}\ x = g(\tilde{M})\ \mathtt{then}\ P\ \mathtt{else}\ Q$ behaves as Q. The tuple splitting $\mathtt{let}\ \tilde{x} = \tilde{M}\ \mathtt{in}\ P$ behaves as P where each x_i from $\tilde{x}$ in P are replaced by the i-th term in the tuple $\tilde{M}$. Again, all these rules can be applied inside a process, which is stated by the last rule.

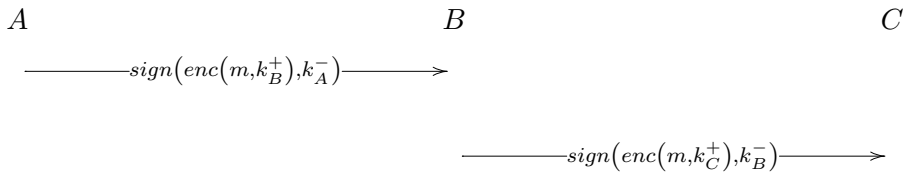
$$
\begin{aligned}
&k\colon \mathtt{out}\,(u, M)\,.P | k\colon \mathtt{in}\,(u, x)\,.Q \to P | Q\{M/x\} \\
&k\colon \mathtt{out}\,(u, M)\,.P | k\colon !\mathtt{in}\,(u, x)\,.Q \to P | Q\{M/x\} | k\colon !\mathtt{in}\,(u, x)\,.Q \\
&\mathtt{let}\ x = g\big(\tilde{M}\big)\ \mathtt{then}\ P\ \mathtt{else}\ Q \to P\{N/x\}, && \text{if } g\big(\tilde{M}\big) \Downarrow N \\
&\mathtt{let}\ x = g\big(\tilde{M}\big)\ \mathtt{then}\ P\ \mathtt{else}\ Q \to Q, && \text{if } g\big(\tilde{M}\big) \not\Downarrow \\
&\mathtt{let}\ \tilde{x} = \tilde{M}\ \mathtt{in}\ P \to P\{\tilde{M}/\tilde{x}\} \\
&\mathcal{E}\big[P\big] \to \mathcal{E}\big[Q\big], && \text{if } P \to Q
\end{aligned}
$$

In order to connect structural equivalence with internal reduction, P can be reduced to Q, if P is structurally equivalent to P', P' can be reduced to Q' and Q' on the other hand is structurally equivalent to Q:

$$
P \to Q, \qquad \text{if } P \equiv P', P' \to Q', Q' \equiv Q
$$

Example

Let us consider an example. Protocols are usually given in a graphical representation:



In this protocol three participants A, B, and C are involved and two messages are sent. The first message is sent from A to B. Here A encrypts a secret m with the (public) encryption key k_B^+ of the receiver – B – and signs the encryption with his own signing key k_A^-. The signature is then sent to participant B. The second message is sent from B to C. B extracts A's secret m from its input, encrypts it with C's encryption key k_C^+, and signs the encryption with its own signing key k_B^-. By following the protocol, C will be able to receive A's original message m.

In the applied π-calculus the protocol will look as follows:

$$
\begin{aligned}
A &\mathrel{\widehat{=}} \nu m.1 : \mathrm{out}\big(ch, sign\big(enc\big(m, pk(k_{BE})\big), k_{AS}\big)\big) \\
B &\mathrel{\widehat{=}} 1 : \mathrm{in}\big(ch, x\big).\mathtt{let}\ x_1 = check(x, vk(k_{AS}))\ \mathtt{then}\ \mathtt{let}\ x_2 = dec(x_1, k_{BE}) \\
&\quad \mathtt{then}\ 2 : \mathrm{out}\Big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\Big) \\
C &\mathrel{\widehat{=}} 2 : \mathrm{in}\big(ch, y\big).\mathtt{let}\ y_1 = check(y, vk(k_{BS}))\ \mathtt{then}\ \mathtt{let}\ y_2 = dec(y_1, k_{CE})\ \mathtt{then}\ Q
\end{aligned}
$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}}\ & \nu k_{AS}.\nu k_{BE}.\nu k_{BS}.\nu k_{CE}. \\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\,.\ p2 : \mathtt{out}\,(ch, pk(k_{CE}))\,. \\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\,.\ p4 : \mathtt{out}\,(ch, vk(k_{BE})) \\
& \big(A \big| B \big| C\big)
\end{aligned}
$$

In our example we model the participants of the protocol as three processes A, B, and C. These processes run in parallel within the scope of some restrictions. The restricted values are the

secret m (which is only known to A, since A has generated it), the decryption (k_{BE}, k_{CE}) and signing keys (k_{AS}, k_{BS}) of the participants. Additionally the encryption and verification keys are output on channel ch. That channel is public, since ch is free. Process A simply outputs the message $sign(enc(m, pk(k_{BE})), k_{AS})$ on the public channel ch. The label of this output is 1. Process B has the corresponding input with label 1. After binding the input to variable x, A's signature is checked and the encryption is decrypted with B decryption key. The content of the encryption is then bound to variable x_2, which is now encrypted with C's public key, signed with B's signing key, and finally sent by the output process with label 2 again on the public channel ch. Process C receives this message as input – it will be bound to variable y – and then checks B's signature and decrypts the encryption. The result – A's secret message – is bound to variable y_2.

Note that we assume the existence of a public key infrastructure: every participant has a secret key for decryption and a signing key for signing messages; all particpants have the public and the verification keys of all other participants. Furthermore all public and verification keys used in the protocol are also output on public channels so that an attacker can use them as well.

2.2 Zero-Knowledge in the Applied π-Calculus

Zero-knowledge proofs are modelled in the applied π-calculus as follows ([BMU08]). There is an additional $zk_{n,m,S}$ constructor:

$M, N ::=$		terms
	a, b, c, m, n	names
	x, y, z, v, w	variables
	$\langle M_1, \ldots, M_n \rangle$	tuples
	$f\big(M_1, \ldots, M_n\big)$	constructor application

$f ::= \quad pk^1, enc^2, senc^2, vk^1, sign^2, hash^1, true^0, false^0, zk^{n+m}_{n,m,S}$

$zk^{n+m}_{n,m,S}$ valid, iff S is an (n, m)-statement

$zk_{n,m,S}$ represents a zero-knowledge proof of the statement S, where the first n arguments are terms from the private component, which are kept secret, and the next m arguments are from the public component, which can be revealed. A statement S is a Boolean formula built from names, variables, placeholders α_i and β_j, constructors, tuples, and destructors for statements. Destructors in statements are the same as in a applied π-calculus process, except that there is no $ver^\sharp$ destructor allowed in a statement. To distinguish between destructors in processes and in statements, a destructor in a statement will be denoted by $g^\sharp$, while the corresponding destructor in the process will be denoted simply by g.

$S ::=$		statements
	a, b, c, m, n	names
	x, y, z, v, w	variables
	α_i, β_j	placeholder
	$\langle S_1, \ldots, S_n \rangle$	tuple
	$f\big(S_1, \ldots, S_n\big)$	constructor
	$g^\sharp\big(S_1, \ldots, S_n\big)$	destructor

$g^\sharp$ is defined, iff $g \neq ver$

A zero-knowledge proof $zk_{n,m,S}$ is valid, if S is an (n,m)-statement, which means for all placeholders α_i and β_j we have $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$. Furthermore two additional destructors are added:

$$g ::= \quad id^1, eq^2, \wedge^2, \vee^2, dec^2, sdec^2, check^2, public_m^1, ver_{n,m,l,S}^{l+1}$$

$public_m$ is a destructor of arity 1 and yields the public component of a zero-knowledge proof. $ver_{n,m,l,S}$ is a destructor of arity $l+1$ and yields the last $m-l$ terms from the public component of the zero-knowledge proof, if and only if the verification of that proof succeeds.

The reduction relation $\Downarrow$ is extend with one reduction rule for *public* and one reduction rule for *ver*:

$$\begin{array}{ll}
id(M) & \Downarrow M \\
eq(M,M) & \Downarrow true \\
\wedge(true, true) & \Downarrow true \\
\vee(true, M) & \Downarrow true \\
\vee(M, true) & \Downarrow true \\
dec\left(enc\left(M, pk(K)\right), K\right) & \Downarrow M \\
check\left(sign\left(M,K\right), vk(K)\right) & \Downarrow M \\
public_m\big(zk_{n,m,S}\big(\tilde{N},\tilde{M}\big)\big) & \Downarrow \tilde{M} \\
ver_{n,m,l,S}\big(zk_{n,m,S}\big(\tilde{N},\tilde{M}\big), M_1, \ldots, M_l\big) & \Downarrow \langle M_{l+1}, \ldots, M_m\rangle \\
 & \quad \text{iff } S\{\tilde{N}/\tilde{\alpha}\}\{\tilde{M}/\tilde{\beta}\} \Downarrow^\sharp true
\end{array}$$

$public_m$ can be reduced to the public component $\tilde{M}$ of a zero-knowledge proof $zk_{n,m,S}(\tilde{N},\tilde{M})$, if the arity of public component is matched. Given a zero-knowledge proof $zk_{n,m,S}(\tilde{N},\tilde{M})$ and l terms, which have to match the first l terms of $zk_{n,m,S}(\tilde{N},\tilde{M})$, $ver_{n,m,l,S}$ verifies that zero-knowledge proof and yields a term consisting of the remaining $m-l$ terms from the public component. The verification of a zero-knowledge proof succeeds if and only if the statement $S\{\tilde{N}/\tilde{\alpha}\}\{\tilde{M}/\tilde{\beta}\}$ from $zk_{n,m,S}(\tilde{N},\tilde{M})$ holds, meaning the statement can be evaluated to *true*: $S\{\tilde{N}/\tilde{\alpha}\}\{\tilde{M}/\tilde{\beta}\} \Downarrow_\sharp true$. Here $\Downarrow_\sharp$ is the evaluation relation for statements and $S\{\tilde{N}/\tilde{\alpha}\}\{\tilde{M}/\tilde{\beta}\}$ means that we replace all placeholders α_i and β_j by the corresponding i-th term from the private component and the j-th term from the public component, respectively. For the evaluation relation $\Downarrow_\sharp$ there are the following rules:

$$M \Downarrow_\sharp M \qquad \frac{\forall i \in [1,n].\ S_i \Downarrow_\sharp M_i}{f(S_1,\ldots,S_n) \Downarrow_\sharp f(M_1,\ldots,M_n)}$$

$$\frac{\forall i \in [1,n].\ S_i \Downarrow_\sharp M_i}{\langle S_1,\ldots,S_n\rangle \Downarrow_\sharp \langle M_1,\ldots,M_n\rangle} \qquad \frac{\forall i \in [1,n].\ S_i \Downarrow_\sharp M_i \qquad g(M_1,\ldots,M_n) \Downarrow M}{g^\sharp(S_1,\ldots,S_n) \Downarrow_\sharp M}$$

The evaluation rule for a destructor $g^\sharp(S_1,\ldots,S_n) \Downarrow_\sharp M$ demands that $g(M_1,\ldots,M_n)$ can be reduced to M. $S \Downarrow_\sharp M$ is only defined, if S does not contain any placeholders.

Processes, structural equivalence $\equiv$ for terms and internal reduction $\rightarrow$ remain as described in 2.1

2.3 Type-Checker

In order to prove security properties of protocols, we use a type-checker which relies on a type system for the applied π-calculus. The type system allows to check statically authorisation policies on protocols represented in the applied π-calculus. If a process representing a protocol

is well-typed, then the protocol is safe in the presence of an arbitrary adversary. For deciding whether a process is well-typed or not, one has to type-check the process. However type-checking a process is lengthy and error-prone, especially for larger protocols or if complex cryptographic operations – such as zero-knowledge proofs – are involved. We use a type-checker for automatically checking whether processes are well-typed and safe.

The type system and the type-checker we use have been presented in [BHM08a]. An implementation of the type-checker is available for download at [BHM08b]. In the following, we will give an overview of the type system. For more detail, we refer the interested reader to the paper.

For the type system we first extend the definition of processes as follows:

$P, Q, R ::=$		processes
	$k : \mathtt{out}\,(M, N)\,.P$	output
	$k : \mathtt{in}\,(M, x)\,.P$	input
	$k :!\mathtt{in}\,(M, x)\,.P$	replicated input
	$\nu n : T.P$	restriction
	$P \mid Q$	parallel composition
	0	null process
	$\mathtt{let}\ x = g(\tilde{M})\ \mathtt{then}\ P\ \mathtt{else}\ Q$	destructor evaluation
	$\mathtt{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathtt{in}\ P$	tuple splitting
	$\mathtt{assume}(C)$	assume formula
	$\mathtt{assert}(C)$	expect formula to hold

Each restricted name n will be annotated by the user with a type T of n. Authorisation policies will be expressed by assumptions $\mathtt{assume}(C)$ and assertions $\mathtt{assert}(C)$ as in [FGM07], where C is a logical formula. Assumptions are used to express global policies of the protocol and events in the processes representing different participants. Assertions are formulas that should hold during or after the execution of the protocol.

The type system relies on the following types:

$T, U ::=$	Un	$\mathsf{SigKey}(T)$	$\mathsf{PubKey}(T)$	$\mathsf{SymKey}(T)$	$\mathsf{Stm}(T)$
	$\mathsf{Private}$	$\mathsf{VerKey}(T)$	$\mathsf{PrivKey}(T)$	$\mathsf{SymEnc}(T)$	$\mathsf{ZKProof}_{n,m,S}(T)$
	$\mathsf{Ch}(T)$	$\mathsf{Signed}(T)$	$\mathsf{PubEnc}(T)$	$\mathsf{Hash}(T)$	
	$\langle \widetilde{x} : \widetilde{T} \rangle \{C\}$				

The types have the following meaning: Un is the type of an untrusted term, the adversary has access to all terms of type Un. $\mathsf{Private}$ is the type of a secret term, an adversary has no access to such a term. $\mathsf{Ch}(T)$ is the type of a channel on which messages of type T can be send. In an untrusted network – such as the internet – all channels are public, which means a participant in a communication can send a message to any other participant. In the type system public channels have type $\mathsf{Ch}(\mathsf{Un})$, while private channels on which messages of type $T \neq \mathsf{Un}$ are sent, have type $\mathsf{Ch}(T)$. $\langle \widetilde{x} : \widetilde{T} \rangle \{C\} = \langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}$ (for some n) is a refinement type (as presented in [BBF+08]). It is the type of a tuple containing n terms $x_1, \ldots, x_n$ of type $T_1, \ldots, T_n$, such that the formula C holds. In this type $x_1, \ldots, x_n$ are bound in the formula C, but not in $T_1, \ldots, T_n$. $\mathsf{SigKey}(T), \mathsf{VerKey}(T)$ and $\mathsf{Signed}(T)$ are the types of a signing key to sign a term of type T, a verification key to check the signature of a term of type T and the signature of a term of type T, respectively. Analogue $\mathsf{PubKey}(T), \mathsf{PrivKey}(T)$, and $\mathsf{PubEnc}(T)$ are the types of a public (encryption) key for encrypting a term of type T, a secret (decryption) key for decrypting an encrypted term of type T, and an encryption of a term of type T. $\mathsf{SymKey}(T)$ is the type for a symmetric key encryption a message of type T and $\mathsf{SymEnc}(T)$ is the type of a

symmetric encryption of a message of type T. $\mathsf{Hash}(T)$ is the type for the hashed value of a term of type T. Finally $\mathsf{Stm}(T)$ is the type of a statement of type T, where T is a refinement type $\langle \widetilde{y} : \widetilde{T} \rangle \{\exists \widetilde{x}.C\}$ of the associated zero-knowledge proof. In the logical formula of the refinement type the arguments of the private component are existentially quantified and the types $\widetilde{T}$ of the arguments $\widetilde{y}$ of the public component are specified. $\mathsf{ZKProof}_{n,m,S}(T)$ is the type of a zero-knowledge proof of the statement S where the private component consists of n terms, the public component consists of m terms and the public component (the m terms that can be revealed) have type T. Types $s_{m,n,S}$ of statements S are annotated by the user.

The typing system relies on the following typing judgements

$\Gamma \vdash \diamond$	well-formed environment
$f : (T_1, \ldots, T_n) \mapsto T$	constructor typing
$g : (T_1, \ldots, T_n) \mapsto T$	destructor typing
$\Gamma \vdash T <: U$	subtyping
$\Gamma \vdash T :: k$	kinding, $k \in \{\mathsf{pub}, \mathsf{tnt}\}$
$\Gamma \vdash M : T$	term typing
$\Gamma \vdash P$	well-typed process

The typing environment Γ consists of bindings of the form $u : T$ (a name or variable u is of type T in the environment Γ) and formulas C. An environment is well-formed ($\Gamma \vdash \diamond$), if all names and variables are bound exactly once, all free names and variables occurring in the types of names and variables are bound beforehand and all free names and variables occurring in a formula are bound in the environment.

Constructors are typed in the following way:

$$
\begin{aligned}
pk &: (\mathsf{PrivKey}(T)) \mapsto \mathsf{PubKey}(T) \\
enc &: (T, \mathsf{PubKey}(T)) \mapsto \mathsf{PubEnc}(T) \\
senc &: (T, \mathsf{SymKey}(T)) \mapsto \mathsf{SymEnc}(T) \\
vk &: (\mathsf{SigKey}(T)) \mapsto \mathsf{VerKey}(T) \\
sign &: (T, \mathsf{SigKey}(T)) \mapsto \mathsf{Signed}(T) \\
hash &: (T) \mapsto \mathsf{Hash}(T) \\
true &: () \mapsto \mathsf{Un} \\
false &: () \mapsto \mathsf{Un}
\end{aligned}
$$

If the term M has type $\mathsf{PrivKey}(T)$, then the public key $pk(M)$ corresponding to private key M is of type $\mathsf{PubKey}(T)$. Analogue if the term M has type $\mathsf{SigKey}(T)$, then the verification key $vk(M)$ corresponding to the signing key M is of type $\mathsf{VerKey}(T)$. Typing an encryption of a term M_1 of type T however is only possible, if the encryption key has type $\mathsf{PubKey}(T)$. Then the encryption $enc(M_1, M_2)$ has type $\mathsf{PubEnc}(T)$. A symmetric encryption has type $\mathsf{SymEnc}(T)$, if the encryption key is of type $\mathsf{SymKey}(T)$ and the message which is encrypted is of type T. Typing the signature of a term M_1 of type T is only possible, if the signing key has type $\mathsf{SigKey}(T)$. Then the signature $sign(M_1, M_2)$ has type $\mathsf{Signed}(T)$. The hashed value $hash(M)$ of a term of type T has type $\mathsf{Hash}(T)$. *true* and *false* are of type Un. We omit the typing rule for the zk constructor and refer to the paper ([BHM08a]).

Destructor are typed in a similar way:

$$\begin{aligned}
eq &: (T, T) \mapsto \mathsf{Un} \\
\wedge &: (\mathsf{Un}, \mathsf{Un}) \mapsto \mathsf{Un} \\
\vee &: (\mathsf{Un}, \mathsf{Un}) \mapsto \mathsf{Un} \\
dec &: (\mathsf{PubEnc}(T), \mathsf{PrivKey}(T)) \mapsto T \\
sdec &: (\mathsf{SymEnc}(T), \mathsf{SymKey}(T)) \mapsto T \\
check &: (\mathsf{Signed}(T), \mathsf{VerKey}(T)) \mapsto T \\
public_m &: (\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\,\{C\})) \mapsto \langle y_1 : T_1, \ldots, y_m : T_m\rangle\{true\}
\end{aligned}$$

$eq(M_1, M_2)$ is of type Un, if M_1 and M_2 are of the same type T (since eq will return *true* or *false* which are of type Un). $\wedge(M_1, M_2)$ and $\vee(M_1, M_2)$ are of type Un if M_1 and M_2 are of type Un. The result of the decryption $dec(M_1, M_2)$ of a term M_1 having type $\mathsf{PubEnc}(T)$ with a term M_2 having type $\mathsf{PrivKey}(T)$ will be of type T. $sdec(M_1, M_2)$ is of type T, if M_1 is of type $\mathsf{SymEnc}(T)$ and M_2 is of type $\mathsf{SymKey}(T)$. The result of the verification of a signature $check(M_1, M_2)$ of a term M_1 having type $\mathsf{Signed}(T)$ with a term M_2 will be of type T, if M_2 is of type $\mathsf{VerKey}(T)$. Finally $public_m(t)$ is of type $\langle y_1 : T_1, \ldots, y_m : T_m\rangle\{true\}$ if M is has type $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\,\{C\})$ of a zero-knowledge proof of statement S with n terms in the private component and m terms $y_1, \ldots, y_m$ of type $T_1, \ldots, T_m$ in the public component. Here we omit the typing rule for verifying zero-knowledge proofs (ver), for details see [BHM08a].

Since communication over the internet is modelled as communication via channels of type $\mathsf{Ch}(\mathsf{Un})$ only messages of type Un could be sent and received over those channels. However this would be too restrictive. Hence one uses subtyping and kinding. The subtyping relation $\Gamma \vdash T <: U$ expresses that a term of type T is a subtype of the type U in Γ, which means a term of type T can be used wherever a term of type U can be used.

Furthermore if a type T is a subtype of Un in Γ, then we say T is public ($\Gamma \vdash T :: \mathsf{pub}$) and can be sent over an untrusted network without compromising the security of the protocol. If however a type T is a supertype of Un in Γ, then T is tainted ($\Gamma \vdash T :: \mathsf{tnt}$) and can be received from an untrusted network without compromising the security of the protocol. The type $\mathsf{Private}$ is neither public nor tainted. For example a channel $\mathsf{Ch}(T)$ is public or tainted, if T is public and tainted. A signature $\mathsf{Signed}(T)$ of a term of type T is always tainted and it is public, if T is public. A asymmetric encryption $\mathsf{PubEnc}(T)$ of a term of type T is always public as well.

Finally we have term typing $\Gamma \vdash M : T$, meaning that the term M is of type T and the typing judgement for a well-typed process $\Gamma \vdash P$, meaning that the process P is well-typed in the environment Γ.

Example

Let us recall the example from 2.1 and see how the type system checks whether the protocol is secure or not.

$$\begin{aligned}
A &\mathrel{\widehat{=}} \nu m.1 : \mathrm{out}\big(ch, sign\big(enc\big(m, pk(k_{BE})\big), k_{AS}\big)\big) \\
B &\mathrel{\widehat{=}} 1 : \mathrm{in}\big(ch, x\big).\texttt{let}\ x_1 = check(x, vk(k_{AS}))\ \texttt{then let}\ x_2 = dec(x_1, k_{BE}) \\
&\qquad \texttt{then}\ 2 : \mathrm{out}\Big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\Big) \\
C &\mathrel{\widehat{=}} 2 : \mathrm{in}\big(ch, y\big).\texttt{let}\ y_1 = check(y, vk(k_{AS}))\ \texttt{then let}\ y_2 = dec(y_1, k_{BE})\ \texttt{then}\ Q
\end{aligned}$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}} & \nu k_{AS}.\ \nu k_{BE}.\ \nu k_{BS}.\ \nu k_{CE}. \\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\ .\ p2 : \mathtt{out}\,(ch, pk(k_{CE}))\ . \\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\ .\ p4 : \mathtt{out}\,(ch, vk(k_{BE})) \\
& \big(A \big| B \big| C\big)
\end{aligned}
$$

m is some secret message generated by A. Hence m will be given the type Private. m is then encrypted with B's public key. So B's public key $pk(k_{BE})$ is a key to encrypt a term of type Private, hence $pk(k_{BE})$ is of type PubKey(Private). Having a look at the typing rule for the constructor pk we see that k_{BE} must then have type PrivKey(Private). The message m (which is of type Private as mentioned) is encrypted with $pk(k_{BE})$ (which is of type PubKey(Private) as mentioned) yielding the term $enc(m, pk(k_{BE}))$ which has (according to the typing rule for the constructor enc) the type PubEnc(Private). The encryption $enc(m, pk(k_{BE}))$ is then signed with A's signing key k_{AS}, which leads to k_{AS} having type SigKey(PubEnc(Private)). The signature (having the type Signed(PubEnc(Private))) is then output on the public channel ch with label 1. The public channel ch has type Ch(Un). By the kinding rules we have that SigKey(PubEnc(Private)) :: pub whenever PubEnc(Private) :: pub and that PubEnc(Private) is always public. Hence SigKey(PubEnc(Private)) <: Un and $sign(enc(m, pk(k_{BE})), k_{AS})$ can be sent over an untrusted network without compromising the security of the protocol.

B now receives the message send by A in the input with label 1. The message will be bound to x and – since x has been received from a public channel – x has type Un. B then checks x with A's verification key $vk(k_{AS})$. By the typing rule for the vk constructor and the fact k_{AS} has type SigKey(PubEnc(Private)), $vk(k_{AS})$ is of type VerKey(PubEnc(Private)). Furthermore the typing rule for the *check* destructor require the first argument to be of type Signed(T) and the second argument to be of type VerKey(T). It follows that in this case T = PubEnc(Private). By the kinding rules Signed(T) :: tnt always holds, so Un <: Signed(PubEnc(Private)) and we can apply the check destructor to the signature x without compromising the security of the protocol. The typing rule for the *check* destructor we then know that x_1 is of type PubEnc(Private). B now decrypts x_1 with his decryption key k_{BE}. By the typing rule for the *dec* destructor we then have that x_2 is of type Private. From now on B repeats with the term x_2 what A has done with her message m and C repeats all steps B has done so far, except now B's signing key and verification and C's public key and private keys are involved.

Finally we get k_{BS} having the same type as k_{AS}, namely SigKey(PubEnc(Private)) and k_{CE} having the same type as k_{BE}, namely PrivKey(Private). The process in the applied π-calculus now is as follows:

$$
\begin{aligned}
A \mathrel{\widehat{=}} & \ \nu m : \mathsf{Private}.1 : \mathrm{out}\big(ch, sign\big(enc\big(m, pk(k_{BE})\big), k_{AS}\big)\big) \\
B \mathrel{\widehat{=}} & \ 1 : \mathrm{in}\big(ch, x\big).\mathtt{let}\ x_1 = check(x, vk(k_{AS}))\ \mathtt{then}\ \mathtt{let}\ x_2 = dec(x_1, k_{BE}) \\
& \quad \mathtt{then}\ 2 : \mathrm{out}\Big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\Big) \\
C \mathrel{\widehat{=}} & \ 2 : \mathrm{in}\big(ch, y\big).\mathtt{let}\ y_1 = check(y, vk(k_{AS}))\ \mathtt{then}\ \mathtt{let}\ y_2 = dec(y_1, k_{BE})\ \mathtt{then}\ Q
\end{aligned}
$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}} & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\mathsf{Private}))\ .\ \nu k_{BE} : \mathsf{PrivKey}(\mathsf{Private}). \\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\mathsf{Private}))\ .\ \nu k_{CE} : \mathsf{PrivKey}(\mathsf{Private}). \\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\ .\ p2 : \mathtt{out}\,(ch, pk(k_{CE}))\ . \\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\ .\ p4 : \mathtt{out}\,(ch, vk(k_{BE})) \\
& \big(A \big| B \big| C\big)
\end{aligned}
$$

The next step is to add a policy the protocol should fulfill. An authorisation policy consists of assumptions and assertions and governs security-related decisions in the protocol. It is expressed in logical formulas. Assumptions are used to express global policies of the protocol and events in the processes representing different participants. Assertions are formulas that should hold during or after the execution of the protocol.

In our case the purpose of the protocol is to send the secret m generated by A to C. To express the policy, we add the assumption $\mathit{Good}(m)$ to the process representing the principal A in order to reflect that m is the secret A wants to send to C. Now we have to change the protocol and the types accordingly: instead of encrypting and signing m, we encrypt and sign a tuple consisting of the message m. This has the following reason: m has still type $\mathsf{Private}$, but since A says that $\mathit{Good}(m)$ holds, we use a tuple $\langle m \rangle$ whose type is $\langle z : \mathit{Private} \rangle \{\mathit{Good}(z)\}$. By $\langle m \rangle : \langle z : \mathit{Private} \rangle \{\mathit{Good}(z)\}$ we mean that the tuple $\langle m \rangle$ consists of one element (in the type of the tuple we refer to that element by z) and that $\mathit{Good}(z)$ holds. Now C has to split the tuple after he has verified the signature and decrypted:

$$
\begin{aligned}
A \mathrel{\widehat{=}} {} & \nu m : \mathsf{Private}.\big(\mathtt{assume}(\mathit{Good}(m)) \big| 1 : \mathrm{out}\big(ch, \mathit{sign}\big(\mathit{enc}\big(\langle m \rangle, \mathit{pk}(k_{BE})\big), k_{AS}\big)\big)\big) \\
B \mathrel{\widehat{=}} {} & 1 : \mathrm{in}\big(ch, x\big).\mathtt{let}\ x_1 = \mathit{check}(x, \mathit{vk}(k_{AS}))\ \mathtt{then\ let}\ x_2 = \mathit{dec}(x_1, k_{BE}) \\
& \mathtt{then}\ 2 : \mathrm{out}\Big(ch, \mathit{sign}\big(\mathit{enc}\big(x_2, \mathit{pk}(k_{CE})\big), k_{BS}\big)\Big) \\
C \mathrel{\widehat{=}} {} & 2 : \mathrm{in}\big(ch, y\big).\mathtt{let}\ y_1 = \mathit{check}(y, \mathit{vk}(k_{AS}))\ \mathtt{then\ let}\ y_2 = \mathit{dec}(y_1, k_{BE})\ \mathtt{then} \\
& \mathtt{then\ let}\ \langle y_3 \rangle = y_2\ \mathtt{then}\ \big(\mathtt{assert}(\mathit{Good}(y_3)) \big| Q\big)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Prot} \mathrel{\widehat{=}} {} & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{Private} \rangle \{\mathit{Good}(z)\})). \\
& \nu k_{BE} : \mathsf{PrivKey}(\langle z : \mathsf{Private} \rangle \{\mathit{Good}(z)\}). \\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{Private} \rangle \{\mathit{Good}(z)\})). \\
& \nu k_{CE} : \mathsf{PrivKey}(\langle z : \mathsf{Private} \rangle \{\mathit{Good}(z)\}). \\
& p1 : \mathtt{out}\,(ch, \mathit{pk}(k_{BE}))\,.\ p2 : \mathtt{out}\,(ch, \mathit{pk}(k_{CE}))\,. \\
& p3 : \mathtt{out}\,(ch, \mathit{vk}(k_{AS}))\,.\ p4 : \mathtt{out}\,(ch, \mathit{vk}(k_{BE})) \\
& \big(A \big| B \big| C\big)
\end{aligned}
$$

The process can be type-checked as in the case without assumptions and assertions. The only differences for the modified process is the changed type of the encrypted message $\langle z : \mathsf{Private} \rangle \{\mathit{Good}(z)\}$ instead of $\mathsf{Private}$. Furthermore the assumption $\mathit{Good}(m)$ is contained in the typing environment Γ. The last difference is that for typing the tuple splitting $\langle y_3 \rangle = y_2$ the formula carried in the type $\langle z : \mathsf{Private} \rangle \{\mathit{Good}(z)\}$ of the tuple y_2, $\mathit{Good}(z)\{y_3/z\}$ is added to the typing environment. Here z is replaced by the variable of the tuple splitting y_3 according to the typing rule for the tuple splitting. Since now $\mathit{Good}(y_3)$ is in the environment the assertion holds which intuitively proves the well-typing of the process. The authorisation policy in this case is: whenever all checks of C succeed C can be sure that the secret message has been generated by A and forwarded by B.

Chapter 3

Principal Compromise in the Applied π-Calculus

The scenario in which participants involved in the execution of a protocol do not behave as intended is called principal compromise or system compromise.

Modelling compromised participants in the applied π-calculus is done in the following way: If principals are compromised in the execution of the protocol, secrecy cannot be guaranteed anymore, since a malicious participant can output all the secrets he learns on a public channel. Hence the type $\mathsf{Private}$ is no more applicable. To account that fact, we have to replace the unconditional type $\mathsf{Private}$ for secrets by conditional types, which we will call $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ in the presence of compromised participants $\tilde{X}$. By giving some message m the type $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ we mean, the message m is of type $\mathsf{Private}$ as long as none of the participants in $\tilde{X}$ is compromised and m is of type Un, if at least one of the participants in $\tilde{X}$ is compromised. We define the type $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ as $\mathsf{Ch}(\langle\rangle\{\tilde{X}saysfalse\})$, where $\tilde{X}saysfalse$ stands for the disjunction $\bigvee_{X\in\tilde{X}} Xsaysfalse$. Furthermore we add for all $X \in \tilde{X}$ the assumption $Xsaysfalse$ if participant X is compromised and we add $\neg Xsaysfalse$, if X is not compromised.

Having a look at the kinding rule for channels we see that $\mathsf{Ch}(T)$ is public and tainted only if T is public and tainted. So $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ is public and tainted if and only if $\langle\rangle\{\tilde{X}saysfalse\}$ is public and tainted. Since $\langle\rangle\{\tilde{X}saysfalse\}$ is a type for empty tuples by the kinding rule for tuples we have that $\langle\rangle\{\tilde{X}saysfalse\}$ is tainted if and only if $\tilde{X}saysfalse$ follows from the environment Γ. If we now assume some $Xsaysfalse$ with $X \in \tilde{X}$ to hold then $\tilde{X}saysfalse$ also holds. This implies that $\langle\rangle\{\tilde{X}saysfalse\}$ is tainted and (since $\langle\rangle\{\tilde{X}saysfalse\}$ is always public) that $\mathsf{Ch}(\langle\rangle\{\tilde{X}saysfalse\})$ is public and tainted. This means that if some participant is compromised then the type $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ is equal to the type Un since Un is public and tainted. On the other hand, if we assume $\neg Xsaysfalse$ to hold for all $X \in \tilde{X}$ then $\langle\rangle\{\tilde{X}saysfalse\}$ is not tainted. Since $\langle\rangle\{\tilde{X}saysfalse\}$ is not tainted, $\mathsf{Ch}(\langle\rangle\{\tilde{X}saysfalse\})$ is neither public nor tainted and hence $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ is equal to the type $\mathsf{Private}$.

By using the type $\mathsf{PrivateUnless}\tilde{\mathsf{X}}$ the types of keys have to be changed accordingly. Finally, if a principal X is compromised, we replace the process representing X by a process, which outputs all secrets of X on some public channels.

Example

We continue with the example from 2.3. Since B has to forward the secret m received from A to C, we consider the case B being compromised. m will now be of type $\mathsf{PrivateUnlessB}$ which stands for the type $\mathsf{Ch}(\langle\rangle\{Bsaysfalse\})$. By giving m the type $\mathsf{PrivateUnlessB}$ we have to change the types of the restricted names (keys) in our example: k_{AS} will now have the

type $\mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\}))$, in the same way k_{BE} will have the type $\mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})$. Because B may exchange the received secret from A and send some other message m' for which $Good(m')$ does not hold, k_{BS} will have the type $\mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}))$ and k_{CE} will have the type $\mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})$. Note that $Good(z) \vee Bsaysfalse$ is equal to $Good(z)$ if $\neg Bsaysfalse$ holds and it is equal to $true$, if $Bsaysfalse$ holds.

In the case that B is not compromised, we have:

$$
\begin{aligned}
A \mathrel{\widehat{=}}\ & \nu m : \mathsf{PrivateUnlessB}.\\
& \big(\mathtt{assume}(Good(m))\big|1 : \mathrm{out}\big(ch, sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)\big)\big)\\
B \mathrel{\widehat{=}}\ & 1 : \mathrm{in}\big(ch, x\big).\\
& \mathtt{let}\ x_1 = check(x, vk(k_{AS}))\ \mathtt{then}\\
& \mathtt{let}\ x_2 = dec(x_1, k_{BE})\ \mathtt{then}\\
& 2 : \mathrm{out}\Big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\Big)\\
C \mathrel{\widehat{=}}\ & 2 : \mathrm{in}\big(ch, y\big).\\
& \mathtt{let}\ y_1 = check(y, vk(k_{AS}))\ \mathtt{then}\\
& \mathtt{let}\ y_2 = dec(y_1, k_{BE})\ \mathtt{then}\\
& \mathtt{let}\ \langle y_3\rangle = y_2\ \mathtt{then}\\
& \big(\mathtt{assert}(Good(y_3))\big|Q\big)
\end{aligned}
$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}}\ & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})).\\
& \nu k_{BE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\}).\\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})).\\
& \nu k_{CE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}).\\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\,.\\
& p2 : \mathtt{out}\,(ch, pk(k_{CE}))\,.\\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\,.\\
& p4 : \mathtt{out}\,(ch, vk(k_{BE}))\,.\\
& \big(\mathtt{assume}\ (\neg Bsaysfalse)\big|A\big|B\big|C\big)
\end{aligned}
$$

With the assumption that $\neg Bsaysfalse$ holds the type $\mathsf{PrivateUnlessB}$ is equal to the type $\mathsf{Private}$ and $Good(z) \vee Bsaysfalse$ is equal to $Good(z)$. Then the example can be successfully type-checked as before and hence the protocol is safe.

However if we assume that B is compromised the same protocol then is as follows:

$$
\begin{aligned}
A \mathrel{\widehat{=}}\ & \nu m : \mathsf{PrivateUnlessB}.\\
& \big(\mathtt{assume}(Good(m))\big|1 : \mathrm{out}\big(ch, sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)\big)\big)\\
B \mathrel{\widehat{=}}\ & b1 : \mathrm{out}\big(ch, k_{BE}\big).\\
& b2 : \mathrm{out}\big(ch, k_{BS}\big)\\
C \mathrel{\widehat{=}}\ & 2 : \mathrm{in}\big(ch, y\big).\\
& \mathtt{let}\ y_1 = check(y, vk(k_{AS}))\ \mathtt{then}\\
& \mathtt{let}\ y_2 = dec(y_1, k_{BE})\ \mathtt{then}\\
& \mathtt{let}\ \langle y_3\rangle = y_2\ \mathtt{then}\\
& \big(\mathtt{assert}(Good(y_3))\big|Q\big)
\end{aligned}
$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}} \; & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})). \\
& \nu k_{BE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\}). \\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})). \\
& \nu k_{CE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}). \\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\,. \\
& p2 : \mathtt{out}\,(ch, pk(k_{CE}))\,. \\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\,. \\
& p4 : \mathtt{out}\,(ch, vk(k_{BE}))\,. \\
& \big(\mathtt{assume}\ Bsaysfalse \big| A \big| B \big| C\big)
\end{aligned}
$$

Note that we have only replaced the process B and the assumption $\neg Bsaysfalse$ by $Bsaysfalse$. In this case the type $\mathsf{PrivateUnlessB}$ is equal to Un, $Good(z) \vee Bsaysfalse$ is equal to $true$ and hence from the kinding rules it follows that $\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}$ is public and tainted. Furthermore the assertion $Good(y_3)$ in process C cannot be fulfilled, since y_2 will have the type $\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}$ which is equal to $\langle z : \mathsf{Un}\rangle\{true\}$. Hence the protocol cannot be type-checked successfully and is not safe under system compromise.

One possibility to overcome the problem of compromised participants is to weaken the policy as suggested in [FGM07]. Here the authors propose to refine the policy such that it indicates on which participants the policy relies on. In our case the policy depends on principal B: If B is compromised the policy is not fulfilled anymore. Instead of weakening the policy our approach will strengthen the protocol in order to be able to fulfil the given policy even in the presence of compromised participants.

Chapter 4

Approach

Our approach will strengthen the protocol by making usage of zero-knowledge proofs. The idea behind our approach is the following: In a usual protocol a principal performs some operations and then sends some message to another principal as specified by the protocol description. Firstly we change the protocol in a way that a principal proves by a zero-knowledge proof that he has performed the operations specified by the protocol description. By using a zero-knowledge proof a principal does not have to reveal his secrets, for instance signing and decryption keys. The proof however will convince other principals that the he knows a correct signing or decryption key. Secondly the principal does not only prove his correct behaviour by a zero-knowledge proof but he has additionally to forward all zero-knowledge proofs he has received. By doing so, the receiver of the zero-knowledge proofs will be able to check whether all principals have followed the protocol.

Before we present an algorithm that transforms a given protocol to strengthen it, let us continue with the example from chapter 3 and demonstrate the idea of our approach. The protocol in the case of B not being compromised is as follows:

$$
\begin{aligned}
A \mathrel{\widehat{=}} & \ \nu m : \mathsf{PrivateUnlessB}. \\
& \ \big(\texttt{assume}(\mathit{Good}(m)) \big| 1 : \mathrm{out}\big(ch, \mathit{sign}\big(\mathit{enc}\big(\langle m \rangle, \mathit{pk}(k_{BE})\big), k_{AS}\big)\big)\big) \\
B \mathrel{\widehat{=}} & \ 1 : \mathrm{in}\big(ch, x\big). \\
& \ \texttt{let}\ x_1 = \mathit{check}(x, \mathit{vk}(k_{AS}))\ \texttt{then} \\
& \ \texttt{let}\ x_2 = \mathit{dec}(x_1, k_{BE})\ \texttt{then} \\
& \ 2 : \mathrm{out}\Big(ch, \mathit{sign}\big(\mathit{enc}\big(x_2, \mathit{pk}(k_{CE})\big), k_{BS}\big)\Big) \\
C \mathrel{\widehat{=}} & \ 2 : \mathrm{in}\big(ch, y\big). \\
& \ \texttt{let}\ y_1 = \mathit{check}(y, \mathit{vk}(k_{AS}))\ \texttt{then} \\
& \ \texttt{let}\ y_2 = \mathit{dec}(y_1, k_{BE})\ \texttt{then} \\
& \ \texttt{let}\ \langle y_3 \rangle = y_2\ \texttt{then} \\
& \ \big(\texttt{assert}(\mathit{Good}(y_3)) \big| Q\big)
\end{aligned}
$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}} \; & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})). \\
& \nu k_{BE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\}). \\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})). \\
& \nu k_{CE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}). \\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\,. \\
& p2 : \mathtt{out}\,(ch, pk(k_{CE}))\,. \\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\,. \\
& p4 : \mathtt{out}\,(ch, vk(k_{BE}))\,. \\
& \big(\mathtt{assume}\ (\neg Bsaysfalse)\big|A\big|B\big|C\big)
\end{aligned}
$$

According to our approach A has to prove to B that she sends $sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)$, where m is the message, she wants to keep secret. Since she wants to keep m secret, she would like to keep $\langle m\rangle$ secret as well. Now A constructs the zero-knowledge proof in the following way: she proves that

- verifying $sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)$ with the verification key $vk(k_{AS})$ corresponding to her signing key k_{AS} results in the message $enc\big(\langle m\rangle, pk(k_{BE})\big)$
- $enc\big(\langle m\rangle, pk(k_{BE})\big)$ is an encryption of the term $\langle m\rangle$ with B's public key $pk(k_{BE})$ (here the term $\langle m\rangle$ will be kept secret)
- the term $\langle m\rangle$ is a tuple consisting of the single term m (here m will be kept secret as well).

The terms which we have to keep secret in the zero-knowledge proof are: $\langle m\rangle$ and m; the terms which may be revealed to anyone are the following terms: $sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)$, $vk(k_{AS}), enc\big(\langle m\rangle, pk(k_{BE})\big)$, and $pk(k_{BE})$. Hence A sends the zero-knowledge proof

$$
\begin{aligned}
& zk_{2,4,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^{\sharp}(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1} \\
& \quad \big(\big\langle m\big\rangle,\ m;\ vk(k_{AS}),\ pk(k_{BE}),\ sign\big(enc\big(\big\langle m\big\rangle, pk(k_{BE})\big), k_{AS}\big),\ enc\big(\big\langle m\big\rangle, pk(k_{BE})\big)\big).
\end{aligned}
$$

The order of the terms in the public component of the zero-knowledge proof is changed such that in the verification of the zero-knowledge proof the verification key for A's signature and the public key of B can be matched and the message from the original protocol is the first message following the terms that are matched.

Now B has to prove that he has followed the protocol as given in the protocol description. He proves that

- verifying $sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)$ with the verification key $vk(k_{BS})$ corresponding to his signing key k_{BS} results in the message $enc\big(x_2, pk(k_{CE})\big)$
- $enc\big(x_2, pk(k_{CE})\big)$ is an encryption of the term x_2 with C's public key $pk(k_{CE})$ (here the term x_2 will be kept secret)
- B has obtained the term x_2 from x_1 by decrypting x_1 with his decryption key k_{BE} (here k_{BE} will be kept secret as well)
- B knows the corresponding decryption key k_{BE} to his public key $pk(k_{BE})$
- B has obtained the term x_1 from x by verifying x with A's verification key $vk(k_{AS})$

The terms which we have to keep secret in the zero-knowledge proof are: x_2 and k_{BE}; the terms which may be revealed to anyone are: $sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big), vk(k_{BS}), enc\big(x_2, pk(k_{CE})\big)$, $pk(k_{CE}), x_1, pk(k_{BE}), x$, and $vk(k_{AS})$. B now sends the zero-knowledge proof

$$zk_{2,8,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)}$$
$$\big(x_2,\ k_{BE};\ vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ x,\ sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big),$$
$$enc\big(x_2, pk(k_{CE})\big),\ x_1\big)$$

and forwards the zero-knowledge proof received from A. Now we have to add the verification of the zero-knowledge proofs to B and C to get:

$$A \mathrel{\widehat{=}} 1 : \mathrm{out}\Big(ch, zk_{2,4,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1}$$
$$\big(\langle m\rangle, m; vk(k_{AS}), pk(k_{BE}), sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big), enc\big(\langle m\rangle, pk(k_{BE})\big)\big)\Big)$$

$$B \mathrel{\widehat{=}} 1 : \mathrm{in}\big(ch, \hat{\hat{x}}\big).$$
$$\texttt{let } \langle \hat{x}_1\rangle = \hat{\hat{x}} \texttt{ in}$$
$$\texttt{let } \langle \hat{x}_{1,1}, \hat{x}_{1,2}, \hat{x}_{1,3}, \hat{x}_{1,4}\rangle = public_4(\hat{x}_1) \texttt{ in}$$
$$\texttt{let } \hat{x}_{1,V} = ver_{2,4,2,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1}\big(\hat{x}_1, vk(k_{AS}),\ pk(k_{BE})\big) \texttt{ then}$$
$$\texttt{let } \langle x, \hat{x}_{1,4}\rangle = \hat{x}_{1,V} \texttt{ in}$$
$$\texttt{let } x_1 = check(x, vk(k_{AS})) \texttt{ then}$$
$$\texttt{let } x_2 = dec\big(x_1, k_{BE}\big)$$
$$\texttt{then } 2 : \mathrm{out}\Big(ch, \big\langle zk_{2,8,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)}$$
$$\big(x_2, k_{BE}; vk(k_{AS}), pk(k_{CE}), pk(k_{BE}), vk(k_{BS}), x,$$
$$sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big), enc\big(x_2, pk(k_{CE})\big), x_1\big), \hat{x}_1\big\rangle\Big)$$

$$C \mathrel{\widehat{=}} 2 : \mathrm{in}\big(ch, \hat{\hat{y}}\big).$$
$$\texttt{let } \langle \hat{y}_1, \hat{y}_2\rangle = \hat{\hat{y}} \texttt{ in}$$
$$\texttt{let } \langle \hat{y}_{2,1}, \hat{y}_{2,2}, \hat{y}_{2,3}, \hat{y}_{2,4}\rangle = public_4(\hat{y}_2) \texttt{ in}$$
$$\texttt{let } \langle \hat{y}_{1,1}, \hat{y}_{1,2}, \hat{y}_{1,3}, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7}\rangle = public_7(\hat{y}_1) \texttt{ in}$$
$$\texttt{let } \hat{y}_{2,V} = ver_{2,4,3,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1}$$
$$\big(\hat{y}_2, vk(k_{AS}),\ pk(k_{BE}),\ \hat{y}_{1,5}\big) \texttt{ then}$$
$$\texttt{let } \hat{y}_{1,V} = ver_{2,8,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)}$$
$$\big(\hat{y}_1, vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ \hat{y}_{1,5}\big) \texttt{ then}$$
$$\texttt{let } \langle \hat{y}_{2,4}\rangle = \hat{y}_{2,V} \texttt{ in}$$
$$\texttt{let } \langle y, \hat{y}_{1,7}, \hat{y}_{1,8}\rangle = \hat{y}_{1,V} \texttt{ in}$$
$$\texttt{let } y_1 = check(y, vk(k_{BS})) \texttt{ then}$$
$$\texttt{let } y_2 = dec\big(y_1, k_{CE}\big) \texttt{ then}$$
$$\texttt{let } \langle y_3\rangle = y_2 \texttt{ in } Q$$

$$
\begin{aligned}
Prot \mathrel{\widehat{=}} \; & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})). \\
& \nu k_{BE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\}). \\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})). \\
& \nu k_{CE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}). \\
& p1 : \mathtt{out}\,(ch, pk(k_{BE}))\,. \\
& p2 : \mathtt{out}\,(ch, pk(k_{CE}))\,. \\
& p3 : \mathtt{out}\,(ch, vk(k_{AS}))\,. \\
& p4 : \mathtt{out}\,(ch, vk(k_{BE}))\,. \\
& \big(\mathtt{assume}\ (\neg Bsaysfalse)\big|A\big|B\big|C\big)
\end{aligned}
$$

Note that C extracts the fourth message from B's zero-knowledge proof and matches that message in both A's and B's zero-knowledge proof to link both proofs. Intuitively the modified process type-checks because C expects to receive two zero-knowledge proofs. The second zero-knowledge proof should convince C that A has followed the protocol and that $Good(m)$ holds, and the first zero-knowledge proof should convince C that B has followed the protocol and that B has used the message he has received from A. For further details see 5.2.

Approach for symmetric key cryptography

Symmetric key cryptography we have to treat differently. Since symmetric keys are both encryption and decryption keys at the same time. In a protocol with symmetric encryption the sender as well as the receiver have to know the symmetric key, which enables especially the receiver of the encryption to generate the zero-knowledge proof of the sender on its own. To exclude that possibility we handle symmetric encryptions in the following way: Whenever we replace a message containing a symmetric encryption by a zero-knowledge proof (proving the correct behaviour of the principal) we extend the zero-knowledge proof by proving additionally that the sender know the corresponding secret key to his public key. This will prevent the receiver from being able to generate the zero-knowledge proof on its own since he does not know the secret key of the sender. We will present an example for symmetric key cryptography in 5.2.

Chapter 5

Algorithm

We present an algorithm that transform protocols according to our approach. Protocols will be expressed in the applied π-calculus as presented in 2.1 with type annotations for restricted names and authorisation policy. We first present the algorithm that transforms the applied π-calculus process. Then we present the transformation of the types.

5.1 Processes

We assume that in the process representing the protocol all names and variables are distinct and for each input process with a label k there exists a corresponding output process with the same label. Our algorithm relies on a public key infrastructure: Every participant has a decryption and a signing key, all participants know all encryption and verification keys of all participants.

Furthermore we assume that each symmetric key is annotated by the private key of the participant using the symmetric key to encrypt a message. For instance, if participant A uses the symmetric key k_{AB} to encrypt m and sends it to B, we will use the annotation $\mathit{privatekey}(k_{AB}) = k_{AE}$.

For the description of the algorithm we will use the following definitions and notations:

- $\mathcal{P}$ denotes the set of all processes, while $P, Q, \ldots \in \mathcal{P}$ denote processes. $\mathcal{I}$ denotes the set of all labels, where $k, k', \ldots \in \mathcal{L}$ are labels. $\mathcal{V}$ denote the set of (input-) variables $(x, y, \ldots \in \mathcal{V})$, $\mathcal{N}$ the set of names $(m, n, \ldots \in \mathcal{N})$, $\mathcal{T}$ the set of terms, Σ the set of substitutions $(\sigma, \sigma_1, \ldots \in \Sigma)$, and $\mathcal{S}$ the set of statements. In order to refer to variables in the original process we will use variables $x, y, \ldots (\in \mathcal{V})$. We will denote variables added by the algorithm by $\hat{x}, \hat{y}, \hat{\hat{x}}, \ldots (\notin \mathcal{V})$

- The algorithm will modify every output for which there exists an input with the same label in the process P. To refer to input and output labels we will use:

$$\mathtt{labels_{in}}: \ \mathcal{P} \to 2^{\mathcal{L}}$$
$$\mathtt{labels_{in}}(P) = \left\{ k \mid \exists C, Q : P \widehat{=} C\left[k : \mathtt{in}\,(u, x)\,.Q\right] \right\}$$

$$\mathtt{labels_{out}}: \ \mathcal{P} \to 2^{\mathcal{L}}$$
$$\mathtt{labels_{out}}(P) = \left\{ k \mid \exists C, Q : P \widehat{=} C\left[k : \mathtt{out}\,(u, M)\,.Q\right] \right\}$$

The labels of the outputs of the encryption and verification keys are $p1, p2, p3$, and $p4$. Two messages are sent, the label for sending and receiving the first message is 1 and for the second message 2.

$$\begin{aligned}\texttt{labels}_{\texttt{in}}\,(Prot) &= \{1,2\}\\ \texttt{labels}_{\texttt{out}}\,(Prot) &= \{p1,p2,p3,p4,1,2\}\end{aligned}$$

- In the process P a message, sent in an output with label k, will be bound to a variable in the input with label k. In order to refer to this variable we use:

$$\texttt{variable}:\ \mathcal{P}\times\mathcal{L}\to\mathcal{V}$$
$$\texttt{variable}\,(P,k) = \begin{cases} x, & \exists C,Q : P \mathrel{\widehat{=}} C\,[k : [!]\texttt{in}\,(u,x)\,.Q] \\ \texttt{undef}, & otherwise \end{cases}$$

The message received in input with label 1 is bound to the variable x, the message received in input 2 is bound to y.

$$\begin{aligned}\texttt{variable}\,(Prot,1) &= x\\ \texttt{variable}\,(Prot,2) &= y\end{aligned}$$

- Analogously to above, we will refer to the channel and the message of the output with label k with:

$$\texttt{output}:\ \mathcal{P}\times\mathcal{L}\to\mathcal{T}\times\mathcal{T}$$
$$\texttt{output}\,(P,k) = \begin{cases} (u,M), & \exists C,Q : P \mathrel{\widehat{=}} C\,[k : \texttt{out}\,(u,M)\,.Q] \\ \texttt{undef}, & otherwise \end{cases}$$

In output 1 and 2 the channels and messages used are the following:

$$\begin{aligned}\texttt{output}\,(Prot,1) &= \big(ch, sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)\big)\\ \texttt{output}\,(Prot,2) &= \big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\big)\end{aligned}$$

- We will refer to the set of restricted values in the output with label k by:

$$\texttt{restricted}:\ \mathcal{P}\times\mathcal{L}\to 2^{\mathcal{N}}$$
$$\texttt{restricted}\,(P,k) = \Big\{ n \Big| \exists C, C', Q, R : P \mathrel{\widehat{=}} C\,[\nu n.Q] \wedge Q \mathrel{\widehat{=}} C'\,[k : \texttt{out}\,(u,M)\,.R] \Big\}$$

No restrictions occur inside a process representing a participant of the protocol, hence for both outputs the restricted values are the same, namely A's message m and the signing and decryption keys of the participants.

$$\begin{aligned}\texttt{restricted}\,(Prot,1) &= \{m, k_{AS}, k_{BE}, k_{BS}, k_{CE}\}\\ \texttt{restricted}\,(Prot,2) &= \{k_{AS}, k_{BE}, k_{BS}, k_{CE}\}\end{aligned}$$

- In order to keep track of the dependency between input messages and output message, we have to keep track of all destructor evaluations and pair splittings occurring before the output. This is done by collecting all destructor evaluations and pair splittings as

substitutions in reversed order of the process. This substitution applied to the output message – considered as a statement – will result in a statement built from input variables, public values, restricted names, constructors, and destructors (because the definition of terms does not allow destructors, but the definition of statements does, we have to work on statements and not on terms). By reversing the order of the single substitutions and exchanging in each substitution the term to be replaced and the term replacing it, backsubstitutions can be performed. In the definition of $\texttt{deseva}(P,k)$ we look for a destructor evaluation or a pair splitting occurring before the output with label k, such that no other destructor evaluation or pair splitting occurs before the found one. This destructor evaluation or pair splitting is transformed to a substitution and added at the end of the destructor evaluations and pair splittings occurring in the continuation process of the found one. Adding it at the end reverses the order of the substitutions in the process.

$$\begin{aligned}
&\texttt{deseva} : \mathcal{P} \times \mathcal{L} \to 2^{\Sigma} \\
&\texttt{deseva}\,(P,k) = \\
&\quad \begin{cases}
\texttt{deseva}\,(Q,k)\,\{g(\tilde{N})/x\}, \\
\qquad \text{if } \exists C, C', Q, Q', Q'' : P \mathrel{\widehat{=}} C\left[\texttt{let } x = g(\tilde{N}) \texttt{ then } Q \texttt{ else } Q'\right] \wedge \\
\qquad\quad Q \mathrel{\widehat{=}} C'\,[k : \texttt{out}\,(u,M)\,.Q''] \wedge C[] \text{ is a } \texttt{deseva}\text{-context} \\
\\
\texttt{deseva}\,(Q,k)\,\{N[1]/x_1\}\ldots\{N[n]/x_n\}, \\
\qquad \text{if } \exists C, C', Q, Q' : P \mathrel{\widehat{=}} C\,[\texttt{let } \langle x_1,\ldots,x_n\rangle = N \texttt{ in } Q] \wedge \\
\qquad\quad Q \mathrel{\widehat{=}} C'\,[k : \texttt{out}\,(u,M)\,.Q'] \wedge C[] \text{ is a } \texttt{deseva}\text{-context} \\
\\
\varepsilon, \qquad \textit{otherwise}
\end{cases}
\end{aligned}$$

We say $C\,[]$ is a $\texttt{deseva}$-context, the hole does not occur in a branch of a destructor evaluation or a tuple splitting.

Before output 1 no destructor evaluations or pair splittings occur. For output 2 the A's message has to be extracted from the input, which results in two destructor applications.

$$\begin{aligned}
&\texttt{deseva}\,(Prot,1) = \varepsilon \\
&\texttt{deseva}\,(Prot,2) = \{dec(x_1,k_{BE})/x_2\}\{check(x,vk(k_{AS}))/x_1\}
\end{aligned}$$

- $\texttt{compile}_{\texttt{output}}$ returns the statements obtained by applying the destructor evaluations at output with label k to the channel and the message of this output:

$$\begin{aligned}
\texttt{compile}_{\texttt{output}} &: \mathcal{P} \times \mathcal{L} \to \mathcal{S} \times \mathcal{S} \\
\texttt{compile}_{\texttt{output}}\,(P,k) &= (u\sigma, M\sigma) \\
\text{with } (u,M) &= \texttt{output}\,(P,k) \\
\sigma &= \texttt{deseva}\,(P,k)
\end{aligned}$$

Applying all substitutions given by the destructor evaluations from above to the channels and output messages, we obtain statements built from constructor and destructor applications, names, and input variables.

$$\texttt{compile}_{\texttt{output}}\left(Prot,1\right) = \Big(ch, sign\Big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\Big)\Big)$$
$$\texttt{compile}_{\texttt{output}}\left(Prot,2\right) = \Big(ch, sign\Big(enc\big(dec\big(check(x, vk(k_{AS})), k_{BE}\big), pk(k_{CE})\big), k_{BS}\Big)\Big)$$

- `graph` returns a list of labels belonging to the inputs which output with label k depends on. One can think of the message flow of a process as a dependency graph: each message sent in the process corresponds to a node, using the label of the input and output process to refer to the nodes, and we have a vertex from node k' to node k'' only if the message in output with label k' depends on the message received in the input with label k''. $\texttt{graph}(P,k)$ will then return the adjacency list of node k.

$$\begin{aligned}
\texttt{graph}' &: \mathcal{P}\times\mathcal{L}\to 2^{\mathcal{L}}\\
\texttt{graph}'\left(P,k\right) &= \big\{k' \big| \exists C, C', Q, R : P \mathrel{\widehat{=}} C\left[k' : [!]\texttt{in}\left(v,x\right).Q\right]\\
&\qquad\qquad \wedge\ Q \mathrel{\widehat{=}} C'\left[k : \texttt{out}\left(u', M'\right).R\right]\\
&\qquad\qquad \wedge\ x \in variables\left(M\right)\big\}\\
\text{with } (u,M) &= \texttt{compile}_{\texttt{output}}\left(P,k\right)
\end{aligned}$$

The first label in the list is the label k. The order of the remaining labels corresponds to the order of the inputs in the process, i. e. the label k_i will occur before the label k_j in the list if and only if there is the input with label k_i first, then there is the input with label k_j and then there is the output with label k:

$$\begin{aligned}
\texttt{graph} &: \mathcal{P}\times\mathcal{L}\to\mathcal{L}^n\\
\texttt{graph}\left(P,k\right) &= \left(k, k_1, \ldots, k_{n-1}\right),\\
\text{with } \texttt{graph}'\left(P,k\right) &= \left\{k_1,\ldots,k_{n-1}\right\},\\
\forall n > j \geqslant i \geqslant 1 &: \nexists C, C', Q, R : \big(P \mathrel{\widehat{=}} C\left[k_j : [!]\texttt{in}\left(u_1,x_1\right).Q\right]\\
&\qquad\qquad \wedge\ Q \mathrel{\widehat{=}} C'\left[k_i : [!]\texttt{in}\left(u_2,x_2\right).R\right]\big)
\end{aligned}$$

Output 1 is the first output in the protocol, hence the output does not depend on other inputs, but output 2 depend on input 1. In this case the transitive closures of `graph` are identical to `graph`.

$$\begin{aligned}
\texttt{graph}\left(Prot,1\right) &= 1\\
\texttt{graph}\left(Prot,2\right) &= (2,1)
\end{aligned}$$

- $\texttt{inputvariables}_{\texttt{all}}$ will return all input variables occurring in the output with label k of process P:

$$\begin{aligned}
\texttt{inputvariables}_{\texttt{all}} &: \mathcal{P}\times\mathcal{L}\to 2^{\mathcal{V}}\\
\texttt{inputvariables}_{\texttt{all}}\left(P,k\right) &= \Big\{\texttt{variable}\left(P,k'\right)\Big|k'\in\left\{k_1,\ldots,k_{n-1}\right\}\Big\}\\
\text{with } \texttt{graph}\left(P,k\right) &= \left(k,k_1,\ldots,k_{n-1}\right)
\end{aligned}$$

There are no inputs before output with label 1. Before output 2, there is an input which is bound to variable x, since x occurs in the compiled output message of output 2 ($\texttt{compile\_output}(Prot, 2)$), this output depends on input 1 ($\texttt{variable}(Prot, 1) = x$).

$$\texttt{inputvariables}_{\texttt{all}}(Prot, 1) = \varnothing$$
$$\texttt{inputvariables}_{\texttt{all}}(Prot, 2) = \{x\}$$

- $\texttt{inputvariables}_{\texttt{pub}}$ returns the same as $\texttt{inputvariables}_{\texttt{all}}$ minus those variables corresponding to inputs of private channels. So $\texttt{inputvariables}_{\texttt{pub}}$ will return all input variables occurring in the output with label k of process P and where the input variables carry messages from public channels:

$$\texttt{inputvariables}_{\texttt{pub}} : \mathcal{P} \times \mathcal{L} \to 2^{\mathcal{V}}$$
$$\texttt{inputvariables}_{\texttt{pub}}(P, k) = \Big\{ \texttt{variable}\left(P, k'\right) \big| k' \in \{k_1, \ldots, k_{n-1}\} \wedge \neg \texttt{private\_channel}\left(P, k'\right) \Big\}$$
$$\text{with } \texttt{graph}(P, k) = (k, k_1, \ldots, k_{n-1})$$

The output of $\texttt{inputvariables}_{\texttt{pub}}$ is the same as the output of $\texttt{inputvariables}_{\texttt{all}}$, since there are no private channels in our example process $Prot$.

$$\texttt{inputvariables}_{\texttt{pub}}(Prot, 1) = \varnothing$$
$$\texttt{inputvariables}_{\texttt{pub}}(Prot, 2) = \{x\}$$

The definition of $\texttt{inputvariables}_{\texttt{pub}}$ relies on the definition of $\texttt{private\_channel}$:

- $\texttt{private\_channel}$ returns a boolean value, whether the channel of the output with label k is a private one or not. A channel u is private if it is restricted and one is not able to extract that channel u out of a message used in the protocol. To find out, whether extracting u from a term used in the protocol we use $\texttt{public}$ and $\texttt{public\_terms}$ as defined below.

$$\texttt{private\_channel} : \mathcal{P} \times \mathcal{L} \to \mathit{Bool}$$
$$\texttt{private\_channel}(P, k) = u \in \texttt{restricted}(P, k) \wedge \neg \texttt{public}(u, \texttt{inputvariables}_{\texttt{pub}}(P, k))$$
$$\text{with } (u, M) = \texttt{compile}_{\texttt{output}}(P, k)$$

Note that this definition is not circular, because of the partial ordering of the labels returned by $\texttt{graph}$.

- $\texttt{public\_terms}(P, k)$ will be used to refer to all terms, which are public in the process P at output with label k. These terms include all terms in outputs which have no corresponding input process (this is, for example, the case for outputs of encryption and verification keys) and all inputvariables of the process P before output k. The set of all public terms will not change during execution of the transformation algorithm (up to renaming of inputvariables).

$$\texttt{public\_terms} : \mathcal{P} \times \mathcal{L} \to 2^{\mathcal{T}}$$
$$\texttt{public\_terms}(P, k) = \big\{ M | \exists C, Q : P \widehat{=} C\left[k' : \texttt{out}(u, M).Q\right] \wedge k' \notin \texttt{labels}_{\texttt{in}}(P) \wedge \neg \texttt{private\_channel}(P, k') \big\} \cup \texttt{inputvariables}_{\texttt{pub}}(P, k)$$

The public terms in the example are:

$$\texttt{public\_terms}\,(Prot, 1) = \{pk(k_{BE}), pk(k_{CE}), vk(k_{AS}), vk(k_{BS})\}$$
$$\texttt{public\_terms}\,(Prot, 2) = \{pk(k_{BE}), pk(k_{CE}), vk(k_{AS}), vk(k_{BS}), x\}$$

- For deciding whether a statement s is public or not, we use `public`, which receives the statement s, a set of public statements pub and a set of restricted names res as inputs. `public` will be used by the statement compilation $\texttt{compile}_{\texttt{statement}}$. While the set pub of public statements will be extended during the compilation of the statement as new public statements will be added, $\texttt{public\_terms}(P, k)$ is used to keep track of all public terms at output k which will not change during statement compilation.

$$\begin{aligned}\texttt{public} &: \mathcal{S} \times 2^{\mathcal{S}} \times \mathcal{L} \to Bool\\ \texttt{public}\,(s, pub, k) &= s \in pub \cup \texttt{public\_terms}(P, k)\\ &\quad \vee\ \texttt{public}'\,(s, pub, k)\end{aligned}$$

$$\begin{aligned}\texttt{public}' &: \mathcal{S} \times 2^{\mathcal{S}} \times \mathcal{L} \to Bool\\ \texttt{public}'\,(u, pub, k) &= u \in pub \cup \texttt{public\_terms}(P, k)\\ &\quad \vee\ u \notin \texttt{restricted}(P, k)\\ \texttt{public}'\left(\langle M_1, \ldots, M_n\rangle, pub, k\right) &= \bigwedge_{i=1}^{n} \texttt{public}(M_i, pub, k)\\ \texttt{public}'\,(pk(M), pub, k) &= \texttt{public}(M, pub, k)\\ \texttt{public}'\,(enc\,(M_1, M_2)\,, pub, k) &= \texttt{public}(M_1, pub, k) \wedge \texttt{public}(M_2, pub, k)\\ \texttt{public}'\,(senc\,(M_1, M_2)\,, pub, k) &= \texttt{public}(M_1, pub, k) \wedge \texttt{public}(M_2, pub, k)\\ \texttt{public}'\,(vk(M), pub, k) &= \texttt{public}(M, pub, k)\\ \texttt{public}'\,(sign\,(M_1, M_2)\,, pub, k) &= \texttt{public}(M_1, pub, k) \wedge \texttt{public}(M_2, pub, k)\\ \texttt{public}'\,(hash\,(M)\,, pub, k) &= \texttt{public}(M, pub, k)\\ \texttt{public}'\,(id\,(M)\,, pub, k) &= \texttt{public}(M, pub, k)\\ \texttt{public}'\,(dec\,(M_1, M_2)\,, pub, k) &= \texttt{public}(M_1, pub, k) \wedge \texttt{public}(M_2, pub, k)\\ \texttt{public}'\,(sdec\,(M_1, M_2)\,, pub, k) &= \texttt{public}(M_1, pub, k) \wedge \texttt{public}(M_2, pub, k)\\ \texttt{public}'\,(check\,(M_1, M_2)\,, pub, k) &= \texttt{public}(M_1, pub, k) \wedge \texttt{public}(M_2, pub, k)\end{aligned}$$

The channels used to send the messages of output 1 and 2 are public.

$$\texttt{private\_channel}\,(Prot, 1) = \texttt{false}$$
$$\texttt{private\_channel}\,(Prot, 2) = \texttt{false}$$

- The statement compilation $\texttt{compile}_{\texttt{statement}}\,(P, k)$ returns a tuple consisting of a statement, a list of private and a list of public messages. The list of private messages forms the private component and the list of public messages forms the public component. The statement, the private, and the public component are used for the generation of the zero-knowledge proof for the output k. The idea behind the zero-knowledge proof is to prove that the participant (represented by a process in the applied π-calculus) has constructed the output according to the protocol specification.

  In order to generate the statement and the private and public component for the output k, we obtain the statement stm from $\texttt{compile}_{\texttt{output}}(P, k)$. stm is the message of output

k, where all variables, which are not input variables, have been substituted by destructor applications. *stm* consists only of names, input variables, constructors, and destructors. In our example we will generate zero-knowledge proofs for outputs 1 and 2 (here, we are only interested in the second component of the result, the first one is the channel, which we don't use here):

$$\texttt{compile}_{\texttt{output}}\,(Prot,1) = \Big(ch, sign\Big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\Big)\Big)$$
$$\texttt{compile}_{\texttt{output}}\,(Prot,2) = \Big(ch, sign\Big(enc\big(dec\big(check(x, vk(k_{AS})), k_{BE}\big), pk(k_{CE})\big), k_{BS}\Big)\Big)$$

Now we would like to compile the statement, the private and public component for

$$sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big) \quad \text{and}$$
$$sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\big).$$

The statement will be compiled by induction on the structure of the input statement $M^{\#}$. Intuitively the statements are compiled the following way. The statement compilation for the message of output 1:

$$\begin{aligned}
&\Big([\![sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)]\!]_{\texttt{true}}\,, (\varepsilon), \Big(sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)\Big)\Big)\\
=&\Big([\![enc\big(\langle m\rangle, pk(k_{BE})\big)]\!]_{\beta_3 = check^{\sharp}(\beta_1,\beta_2)}\,,\\
&\qquad (\varepsilon), \Big(sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big),\ vk(k_{AS}),\ enc\big(\langle m\rangle, pk(k_{BE})\big)\Big)\Big)\\
=&\Big([\![\langle m\rangle]\!]_{\beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check^{\sharp}(\beta_1,\beta_2)}\,,\\
&\qquad \big(\langle m\rangle\big),\\
&\qquad \Big(sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big),\ vk(k_{AS}),\ enc\big(\langle m\rangle, pk(k_{BE})\big),\ pk(k_{BE})\Big)\Big)\\
=&\Big([\![m]\!]_{\langle\alpha_2\rangle = \alpha_1 \wedge \beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check(\beta_1,\beta_2)}\,,\\
&\qquad \big(\langle m\rangle,\ m\big),\\
&\qquad \Big(sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big),\ vk(k_{AS}),\ enc\big(\langle m\rangle, pk(k_{BE})\big),\ pk(k_{BE})\Big)\Big)\\
=&\Big(\langle\alpha_2\rangle = \alpha_1 \wedge \beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check^{\sharp}(\beta_1,\beta_2),\\
&\qquad \big(\langle m\rangle,\ m\big),\\
&\qquad \Big(sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big),\ vk(k_{AS}),\ enc\big(\langle m\rangle, pk(k_{BE})\big),\ pk(k_{BE})\Big)\Big)
\end{aligned}$$

The statement compilation for the message of output 2:

$$\begin{aligned}
&\Big([\![sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\big)]\!]_{\texttt{true}}\,,\\
&\qquad (\varepsilon), \quad \big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\big)\big)\Big)\\
=&\Big([\![enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big)]\!]_{\beta_3 = check^{\sharp}(\beta_1,\beta_2)}\,,\\
&\qquad (\varepsilon), \quad \big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\big),\ vk(k_{BS}),\\
&\qquad\qquad enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big)\big)\Big)
\end{aligned}$$

$$
\begin{aligned}
=\Big(& [\![dec(check(x, vk(k_{AS})), k_{BE})]\!]_{\beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check^\sharp(\beta_1,\beta_2)}, \\
& \big(dec(check(x, vk(k_{AS})), k_{BE})\big), \\
& \Big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\Big),\ vk(k_{BS}), \\
& \ enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big),\ pk(k_{CE})\big)\Big) \\
=\Big(& [\![dec(check(x, vk(k_{AS})), k_{BE})]\!]_{\beta_5 = pk(\alpha_2) \wedge \beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check^\sharp(\beta_1,\beta_2)}, \\
& \big(dec(check(x, vk(k_{AS})), k_{BE}),\ k_{BE}\big), \\
& \Big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\Big),\ vk(k_{BS}), \\
& \ enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big),\ pk(k_{CE}),\ pk(k_{BE})\big)\Big) \\
=\Big(& [\![check(x, vk(k_{AS}))]\!]_{\alpha_1 = dec^\sharp(\beta_6,\alpha_2) \wedge \beta_5 = pk(\alpha_2) \wedge \beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check^\sharp(\beta_1,\beta_2)}, \\
& \big(dec(check(x, vk(k_{AS})), k_{BE}),\ k_{BE}\big), \\
& \Big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\Big),\ vk(k_{BS}), \\
& \ enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big),\ pk(k_{CE}),\ pk(k_{BE}) \\
& \ check(x, vk(k_{AS}))\big)\Big) \\
=\Big(& [\![x]\!]_{\beta_6 = check^\sharp(\beta_8,\beta_7) \wedge \alpha_1 = dec^\sharp(\beta_6,\alpha_2) \wedge \beta_5 = pk(\alpha_2) \wedge \beta_3 = enc(\alpha_1,\beta_4) \wedge \beta_3 = check^\sharp(\beta_1,\beta_2)}, \\
& \big(dec(check(x, vk(k_{AS})), k_{BE}),\ k_{BE}\big), \\
& \Big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\Big),\ vk(k_{BS}), \\
& \ enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big),\ pk(k_{CE}),\ pk(k_{BE}), \\
& \ check(x, vk(k_{AS})),\ vk(k_{AS}),\ x\big)\Big) \\
=\Big(& \beta_6 = check^\sharp(\beta_8, \beta_7) \wedge \alpha_1 = dec^\sharp(\beta_6, \alpha_2) \wedge \beta_5 = pk(\alpha_2) \\
& \wedge \beta_3 = enc(\alpha_1, \beta_4) \wedge \beta_3 = check^\sharp(\beta_1, \beta_2), \\
& \big(dec(check(x, vk(k_{AS})), k_{BE}),\ k_{BE}\big), \\
& \Big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\Big),\ vk(k_{BS}), \\
& \ enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big),\ pk(k_{CE}),\ pk(k_{BE}), \\
& \ check(x, vk(k_{AS})),\ vk(k_{AS}),\ x\big)\Big)
\end{aligned}
$$

As stated above, the statement is compiled by induction on the structure of the input statement $M\sigma$, where M is obtained from $\texttt{output}(P, k)$ and all destructor evaluations $\{M_1/x_1\} \ldots \{M_n/x_n\} = \sigma = \texttt{deseva}\,(P, k)$ are applied to M.

$$
\begin{aligned}
\texttt{compile}_{\texttt{statement}} &: \mathcal{P} \times \mathcal{L} \to \mathcal{S} \times \big(\mathcal{S} \times \ldots \times \mathcal{S}\big) \times \big(\mathcal{S} \times \ldots \times \mathcal{S}\big) \\
\texttt{compile}_{\texttt{statement}}\,(P, k) &= \big(S', sec', pub'\big) \\
&\text{with } (u, M) = \texttt{output}\,(P, k)\,, \\
&\qquad \sigma = \texttt{deseva}\,(P, k)\,,
\end{aligned}
$$

$$
\begin{aligned}
&\{M_1/x_1\}\dots\{M_n/x_n\} = \sigma,\\
&\quad\big(S_0, s_0, sec_0, pub_0\big) = \big([\![M\sigma]\!]_{\mathtt{true}}, \beta_1, \varepsilon, M\sigma\big),\\
&\quad\big(S_i, s_i, sec_i, pub_i\big) = \begin{cases}
\big(S_{i-1}, s_{i-1}, sec_{i-1}, pub_{i-1}\big),\\
\qquad\text{if } x_i \in sec_{i-1} \cup pub_{i-1},\\
\big([\![M_i]\!]_{S_{i-1}}, \beta_{|pub_{i-1}|+1}, sec_{i-1}, (pub_{i-1}, x_i)\big),\\
\qquad\text{if } x_i \notin sec_{i-1} \cup pub_{i-1} \wedge \mathtt{public}\big(x_i, pub_{i-1}, k\big)\\
\big([\![M_i]\!]_{S_{i-1}}, \alpha_{|sec_{i-1}|+1}, (sec_{i-1}, x_i), pub_{i-1}\big),\\
\qquad\text{if } x_i \notin sec_{i-1} \cup pub_{i-1} \wedge \neg\mathtt{public}\big(x_i, pub_{i-1}, k\big),
\end{cases}\\
&\qquad\text{for } 0 < i \leqslant n
\end{aligned}
$$

Here $\big([\![M\sigma]\!]_S, s, sec, pub\big)$ means the compilation of statement $M\sigma$, where s is the placeholder for $M\sigma$; S is the formula compiled so far and *sec* and *pub* are the private and public component, which have been compiled so far. After the statement for the output message has been compiled, the statement is extended by the destructor evaluations which have not been taken into consideration. The compilation starts by adding the statement $M\sigma$ to the public component *pub*, β_1 as placeholder for the statement and *true* as formula. Depending on the structure of $S^{\#}$ the following cases may occur:

If $S^{\#}$ is a name or a variable u, nothing has to be done, since the name or the variable has already been added to one of the list in the previous compilation step. We keep the compiled statement S along with the private and public messages *sec* and *pub*:

$$\big([\![u]\!]_S, s, sec, pub\big) = (S, s, sec, pub)$$

If $S^{\#}$ is a tuple $\langle M_1, \dots, M_n\rangle$, there are two cases: if the tuple is public, we add all components $M_1, \dots, M_n$ to the public messages and then compile the statements for each single component, otherwise we add all components $M_1, \dots, M_n$ to the private messages and compile the statements for each single component.

$$
\begin{aligned}
&\big([\![\langle M_1, \dots, M_n\rangle]\!]_S, s, sec, pub\big) =\\
&\quad\begin{cases}
\big(s = \langle\beta_{|pub|+1}, \dots, \beta_{|pub|+n}\rangle \wedge S_n, s_n, sec_n, pub_n\big)\\
\text{with } \big(S_1, s_1, sec_1, pub_1\big) = \big([\![M_1]\!]_S, \beta_{|pub|+1}, sec, (pub, M_1, \dots, M_n)\big),\\
\qquad\big(S_2, s_2, sec_2, pub_2\big) = \big([\![M_2]\!]_{S_1}, \beta_{|pub|+2}, sec_1, pub_1\big),\\
\qquad\qquad\vdots\\
\qquad\big(S_n, s_n, sec_n, pub_n\big) = \big([\![M_n]\!]_{S_{n-1}}, \beta_{|pub|+n}, sec_{n-1}, pub_{n-1}\big),\\
\qquad\text{if } \mathtt{public}\big(\langle M_1, \dots, M_n\rangle, pub, k\big)\\
\\
\big(s = \langle\alpha_{|sec|+1}, \dots, \alpha_{|sec|+n}\rangle \wedge S_n, s_n, sec_n, pub_n\big)\\
\text{with } \big(S_1, s_1, sec_1, pub_1\big) = \big([\![M_1]\!]_S, \alpha_{|sec|+1}, (sec, M_1, \dots, M_n), pub\big),\\
\qquad\big(S_2, s_2, sec_2, pub_2\big) = \big([\![M_2]\!]_{S_1}, \alpha_{|sec|+2}, sec_1, pub_1\big),\\
\qquad\qquad\vdots\\
\qquad\big(S_n, s_n, sec_n, pub_n\big) = \big([\![M_n]\!]_{S_{n-1}}, \alpha_{|sec|+n}, sec_{n-1}, pub_{n-1}\big),\\
\qquad\text{if } \neg\mathtt{public}\big(\langle M_1, \dots, M_n\rangle, pub, k\big)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is a encryption key $pk(M)$, there are three different cases. If the encryption key is public, there is nothing to compile, since $pk(M)$ is then contained in the list of public messages. If this is not the case, we have to look whether the corresponding decryption key M is public or not. Depending on that, we add M to the private or the public component and continue by compiling the statement for M.

$$
\begin{aligned}
&(\llbracket pk(M) \rrbracket_S, s, sec, pub) = \\
&\qquad \begin{cases}
(S, s, sec, pub), \\
\qquad \text{if } \texttt{public}\big(pk(M), pub, k\big) \\
\Big(\llbracket M \rrbracket_{s=pk(\beta_{|pub|+1}) \wedge S}, \beta_{|pub|+1}, sec, (pub, M)\Big), \\
\qquad \text{if } \neg\texttt{public}\big(pk(M), pub, k\big) \wedge \texttt{public}\big(M, pub, k\big) \\
\Big(\llbracket M \rrbracket_{s=pk(\alpha_{|sec|+1}) \wedge S}, \alpha_{|sec|+1}, (sec, M), pub\Big), \\
\qquad \text{if } \neg\texttt{public}\big(pk(M), pub, k\big) \wedge \neg\texttt{public}\big(M, pub, k\big)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is an asymmetric encryption $enc\,(M_1, M_2)$ of message M_1 with encryption key M_2 we have four cases: both message M_1 and encryption key M_2 are public; M_1 is secret and M_2 is public; M_1 is public and M_2 is secret or both M_1 and M_2 are secret. In all cases, first M_2 is added to the corresponding list of messages and the statement for it is compiled, then the same is done for M_1.

$$
\begin{aligned}
&\big(\llbracket enc\,(M_1, M_2) \rrbracket_S, s, sec, pub\big) = \big(\llbracket M_1 \rrbracket_{S''}, s'', sec'', pub''\big) \\
&\text{with } (S', s', sec', pub') = \\
&\qquad \begin{cases}
\big(\llbracket M_2 \rrbracket_S, \beta_{|pub|+1}, sec, (pub, M_2)\big), \\
\qquad \text{if } \texttt{public}(M_2, pub, k) \\
\big(\llbracket M_2 \rrbracket_S, \alpha_{|sec|+1}, (sec, M_2), pub\big), \\
\qquad \text{if } \neg\texttt{public}(M_2, pub, k)
\end{cases} \\
&(S'', s'', sec'', pub'') = \\
&\qquad \begin{cases}
\big(s = enc(\beta_{|pub''|+1}, s') \wedge S', \beta_{|pub'|+1}, sec', (pub', M_1)\big), \\
\qquad \text{if } \texttt{public}(M_1, pub, k) \\
\big(s = enc(\alpha_{|sec''|+1}, s') \wedge S', \alpha_{|sec'|+1}, (sec', M_1), pub'\big), \\
\qquad \text{if } \neg\texttt{public}(M_1, pub, k)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is a symmetric encryption $senc\,(M_1, M_2)$ of message M_1 with symmetric key M_2 we have the same cases as for the asymetric encryption. The only difference in the compilation is for the case of M_2 being secret. Then we additionally use the annotation for the symmetric key M_2 and prove that the sender of $senc\,(M_1, M_2)$ knows the secret key for his public key.

$$
\begin{aligned}
&\big(\llbracket senc\,(M_1, M_2) \rrbracket_S, s, sec, pub\big) = \big(\llbracket M_1 \rrbracket_{S'''}, s''', sec''', pub'''\big) \\
&\text{with } (S', s', sec', pub') = \\
&\qquad \begin{cases}
\big(\llbracket M_2 \rrbracket_S, \beta_{|pub|+1}, sec, (pub, M_2)\big), \\
\qquad \text{if } \texttt{public}(M_2, pub, k) \\
\big(\llbracket M_2 \rrbracket_S, \alpha_{|sec|+1}, (sec, M_2), pub\big), \\
\qquad \text{if } \neg\texttt{public}(M_2, pub, k)
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&(S'', s'', sec'', pub'') = \\
&\big(\beta_{|pub'|+1} = pk(\alpha_{|sec'|+1}) \wedge S', \beta_{|pub'|+1}, \\
&\qquad (sec', privatekey(M_2)), (pub', pk(privatekey(M_2)))\big) \\
&(S''', s''', sec''', pub''') = \\
&\qquad\qquad \begin{cases}
\big(s = senc(\beta_{|pub''|+1}, s') \wedge S'', \beta_{|pub''|+1}, sec'', (pub'', M_1)\big), \\
\qquad \text{if } \texttt{public}(M_1, pub, k) \\
\big(s = senc(\alpha_{|sec''|+1}, s') \wedge S'', \alpha_{|sec''|+1}, (sec'', M_1), pub''\big), \\
\qquad \text{if } \neg\texttt{public}(M_1, pub, k)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is a verification key, the same cases apply as for an encryption key.

$$
\begin{aligned}
&(\llbracket vk(M) \rrbracket_S, s, sec, pub) = \\
&\qquad\qquad \begin{cases}
(S, s, sec, pub), \\
\qquad \text{if } \texttt{public}\big(vk(M), pub, k\big) \\
\Big(\llbracket M \rrbracket_{s=vk(\beta_{|pub|+1}) \wedge S}, \beta_{|pub|+1}, sec, (pub, M)\Big), \\
\qquad \text{if } \neg\texttt{public}\big(vk(M), pub, k\big) \wedge \texttt{public}\big(M, pub, k\big) \\
\Big(\llbracket M \rrbracket_{s=vk(\alpha_{|sec|+1}) \wedge S}, \alpha_{|sec|+1}, (sec, M), pub\Big), \\
\qquad \text{if } \neg\texttt{public}\big(vk(M), pub, k\big) \wedge \neg\texttt{public}\big(M, pub, k\big)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is a hash $hash\,(M)$ of M there are two cases: M is either public or secret. We add M to the corresponding list and extend the statement S. We do not continue compiling the statement for M, since the hash does not reveal anything about M. In the same way the statement should not reveal anything about M.

$$
\begin{aligned}
&(\llbracket hash\,(M) \rrbracket_S, s, sec, pub) = \\
&\qquad\qquad \begin{cases}
\big(s = hash(\beta_{|pub|+1}) \wedge S, \beta_{|pub|+1}, sec, (pub, M)\big), \\
\qquad \text{if } \texttt{public}\big(M, pub, k\big) \\
\big(s = hash(\alpha_{|sec|+1}) \wedge S, \alpha_{|sec|+1}, (sec, M), pub\big), \\
\qquad \text{if } \neg\texttt{public}\big(M, pub, k\big)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is a signature $sign\,(M_1, M_2)$ of a message M_1 with signing key M_2 we do the following: If the corresponding verification key $vk(M_2)$ is public, we add the verification key to the public component. Then we add the message M_1 to the public or private component depending on whether M_1 is public or not. We extend the statement by proving that verifying the signature $sign\,(M_1, M_2)$ with the verification key $vk(M_2)$ results in the message M_1. Finally we continue the compilation with M_1. If on the other hand the verification key $vk(M_2)$ is not public, we compile the statement as for an asymetric encryption: we add M_2 to the corresponding component, compile the statement for M_2. Then we add M_1 to the corresponding component and finally continue compiling the statement for M_1.

$$\big(\,[\![sign\,(M_1,M_2)]\!]_S\,,s,sec,pub\big) = \big(\,[\![M_1]\!]_{S'''}\,,s''',sec''',pub'''\big)$$
with $(S',s',sec',pub') =$
$$\begin{cases} \big(\,[\![M_2]\!]_S\,,\beta_{|pub|+1},sec,(pub,M_2)\big), \\ \qquad \text{if } \texttt{public}(M_2,pub,k) \wedge \neg\texttt{public}(vk(M_2),pub,k) \\ \big(\,[\![M_2]\!]_S\,,\alpha_{|sec|+1},(sec,M_2),pub\big), \\ \qquad \text{if } \neg\texttt{public}(M_2,pub,k) \wedge \neg\texttt{public}(vk(M_2),pub,k) \\ \big(S,\beta_{|pub|+1},sec,(pub,vk(M_2))\big), \\ \qquad \text{if } \neg\texttt{public}(vk(M_2),pub,k) \end{cases}$$
$(S'',s'',sec'',pub'') =$
$$\begin{cases} \big(S',\beta_{|pub'|+1},sec',(pub',M_1)\big), \\ \qquad \text{if } \texttt{public}(M_1,pub,k) \\ \big(S',\alpha_{|sec'|+1},(sec',M_1),pub'\big), \\ \qquad \text{if } \neg\texttt{public}(M_1,pub,k) \end{cases}$$
$(S''',s''',sec''',pub''') =$
$$\begin{cases} \big(s'' = check^\sharp(s,s') \wedge S',s'',sec'',pub''\big), \\ \qquad \text{if } \texttt{public}(vk(M_2),pub,k) \\ \big(s = sign(s'',s') \wedge S',s'',sec'',pub''\big), \\ \qquad \text{if } \neg\texttt{public}(vk(M_2),pub,k) \end{cases}$$

If $S^\#$ is a decryption $dec\,(M_1,M_2)$ of M_1 with decryption key M_2, we have four different cases. Again we have all combinations of M_1 and M_2 being private or public. Depending on that, M_2 is added to the corresponding list, the statement for M_2 is compiled. Then the statement is extended to prove the connection between the decryption and the encryption key. Finally M_1 is added to the corresponding list and the statement compilation is continued with M_1.

$$\big(\,[\![dec\,(M_1,M_2)]\!]_S\,,s,sec,pub\big) = \big(\,[\![M_1]\!]_{S'''}\,,s''',sec''',pub'''\big)$$
with $(S',s',sec',pub') =$
$$\begin{cases} \big(\,[\![M_2]\!]_S\,,\beta_{|pub|+1},sec,(pub,M_2)\big), \\ \qquad \text{if } \texttt{public}(M_2,pub,k) \\ \big(\,[\![M_2]\!]_S\,,\alpha_{|sec|+1},(sec,M_2),pub\big), \\ \qquad \text{if } \neg\texttt{public}(M_2,pub,k) \end{cases}$$
$(S'',s'',sec'',pub'') =$
$$\begin{cases} \big(\beta_{|pub'|+1} = pk(s') \wedge S',\beta_{|pub'|+1},sec',(pub',pk(M_2))\big), \\ \qquad \text{if } \texttt{public}(pk(M_2),pub,k) \\ \big(\alpha_{|sec'|+1} = pk(s') \wedge S',\alpha_{|sec'|+1},(sec',pk(M_2)),pub'\big), \\ \qquad \text{if } \neg\texttt{public}(pk(M_2),pub,k) \end{cases}$$

$$(S''', s''', sec''', pub''') =$$
$$\begin{cases} \big(s = dec\left(\beta_{|pub''|+1}, s'\right) \wedge S'', \beta_{|pub''|+1}, sec'', (pub'', M_1)\big), \\ \qquad \text{if } \texttt{public}(M_1, pub, k) \\ \big(s = dec\left(\alpha_{|sec''|+1}, s'\right) \wedge S'', \alpha_{|sec''|+1}, (sec'', M_1), pub''\big), \\ \qquad \text{if } \neg\texttt{public}(M_1, pub, k) \end{cases}$$

If $S^{\#}$ is a symmetric decryption $sdec\,(M_1, M_2)$ of M_1 with decryption key M_2, we have four different cases: all combinations of M_1 and M_2 being private or public. Depending on that, M_1 and M_2 are added to the corresponding lists and the statements for M_2 and M_1 are compiled.

$$\big(\,[\![sdec\,(M_1, M_2)]\!]_S\,, s, sec, pub\big) = \big(\,[\![M_1]\!]_{S''}\,, s'', sec'', pub''\big)$$
$$\text{with } (S', s', sec', pub') =$$
$$\begin{cases} \big(\,[\![M_2]\!]_S\,, \beta_{|pub|+1}, sec, (pub, M_2)\big), \\ \qquad \text{if } \texttt{public}(M_2, pub, k) \\ \big(\,[\![M_2]\!]_S\,, \alpha_{|sec|+1}, (sec, M_2), pub\big), \\ \qquad \text{if } \neg\texttt{public}(M_2, pub, k) \end{cases}$$
$$(S'', s'', sec'', pub'') =$$
$$\begin{cases} \big(s = sdec\left(\beta_{|pub'|+1}, s'\right) \wedge S', \beta_{|pub'|+1}, sec', (pub', M_1)\big), \\ \qquad \text{if } \texttt{public}(M_1, pub, k) \\ \big(s = sdec\left(\alpha_{|sec'|+1}, s'\right) \wedge S', \alpha_{|sec'|+1}, (sec', M_1), pub'\big), \\ \qquad \text{if } \neg\texttt{public}(M_1, pub, k) \end{cases}$$

If $S^{\#}$ is a verification $check\,(M_1, M_2)$ of a signature M_1 with verification key M_2, we have four different cases. As above, we have all combinations of M_1 and M_2 begin private or public.

$$\big(\,[\![check\,(M_1, M_2)]\!]_S\,, s, sec, pub\big) = \big(\,[\![M_1]\!]_{S''}\,, s'', sec'', pub''\big)$$
$$\text{with } (S', s', sec', pub') =$$
$$\begin{cases} \big(\,[\![M_2]\!]_S\,, \beta_{|pub|+1}, sec, (pub, M_2)\big), \\ \qquad \text{if } \texttt{public}(M_2, pub, k) \\ \big(\,[\![M_2]\!]_S\,, \alpha_{|sec|+1}, (sec, M_2), pub\big), \\ \qquad \text{if } \neg\texttt{public}(M_2, pub, k) \end{cases}$$
$$(S'', s'', sec'', pub'') =$$
$$\begin{cases} \big(s = check^{\sharp}(\beta_{|pub'|+1}, s') \wedge S', \beta_{|pub'|+1}, sec', (pub', M_1)\big), \\ \qquad \text{if } \texttt{public}(M_1, pub, k) \\ \big(s = check^{\sharp}(\alpha_{|sec'|+1}, s') \wedge S', \alpha_{|sec'|+1}, (sec', M_1), pub'\big), \\ \qquad \text{if } \neg\texttt{public}(M_1, pub, k) \end{cases}$$

Compiling the statements for output 1 and 2 we obtain the statements, the private and the public components. The messages in the public components will be reordered in the compilation of the zero-knowledge proof.

$$
\begin{aligned}
&\texttt{compile}_{\texttt{statement}}\big(Prot, 1\big)\\
&= \Big(\big\langle\alpha_2\big\rangle = \alpha_1 \wedge \beta_3 = enc(\alpha_1, \beta_4) \wedge \beta_3 = check^\sharp(\beta_1, \beta_2),\\
&\quad \Big(\big\langle m\big\rangle,\ m\Big)\\
&\quad \Big(sign\big(enc\big(\big\langle m\big\rangle, pk(k_{BE})\big), k_{AS}\big), vk(k_{AS}), enc\big(\big\langle m\big\rangle, pk(k_{BE})\big), pk(k_{BE})\Big)\Big)\\
&\texttt{compile}_{\texttt{statement}}\big(Prot, 2\big)\\
&= \Big(\beta_6 = check^\sharp(\beta_8, \beta_7) \wedge \alpha_1 = dec^\sharp(\beta_6, \alpha_2) \wedge \beta_5 = pk(\alpha_2)\\
&\quad \wedge \beta_3 = enc(\alpha_1, \beta_4) \wedge \beta_3 = check^\sharp(\beta_1, \beta_2),\\
&\quad \big(dec(check(x, vk(k_{AS})), k_{BE}),\ k_{BE}\big),\\
&\quad \big(sign\big(enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big), k_{BS}\Big),\ vk(k_{BS}),\\
&\quad\ enc\big(dec(check(x, vk(k_{AS})), k_{BE}), pk(k_{CE})\big),\ pk(k_{CE}),\ pk(k_{BE}),\\
&\quad\ check(x, vk(k_{AS})),\ vk(k_{AS}),\ x\big)\Big)
\end{aligned}
$$

- Given a process P and a label k, $\texttt{compile}_{\texttt{zk}}(P, k)$ returns a tuple consisting of the label k; the length of the private and public component of the compiled zero-knowledge proof; the statement of this proof; a tuple consisting of the zero-knowledge proof with forwarded proofs, which replaces the message in the original process; the length of a list of terms to be matched in the verification of the compiled zero-knowledge proof and that list.

$$\texttt{compile}_{\texttt{zk}} : \mathcal{P} \times \mathcal{L} \to \mathcal{L} \times \mathbb{N} \times \mathbb{N} \times \mathcal{S} \times \mathcal{T} \times \mathbb{N} \times \mathcal{T}$$

Basically the statement compilation $\texttt{compile}_{\texttt{statement}}$ returns the statement S', the private and public component sec', pub' (constisting of statements) for the zero-knowledge proof. For constructing a zero-knowledge proof, first we have to backsubstitute the statements in sec' and pub' to get terms sec and pub'':

$$
\begin{aligned}
\big(S', sec', pub'\big) &= \texttt{compile}_{\texttt{statement}}\,(P, k)\\
\sigma &= \texttt{deseva}\,(P, k)\\
sec &= sec'\sigma^{-1}\\
pub'' &= pub'\,\sigma^{-1}
\end{aligned}
$$

To avoid mixing up preliminary results of $\texttt{compile}_{\texttt{zk}}(Prot, 1)$ and $\texttt{compile}_{\texttt{zk}}(Prot, 2)$, we will add the label (1), (2) respectively, to the index to distinguish those results. For $\texttt{compile}_{\texttt{statement}}(Prot, 1)$ we have $\sigma_{(1)} = \varepsilon$ and hence $\sigma_{(1)}^{-1} = \varepsilon$. Performing the (empty) back substitution for the result of $\texttt{compile}_{\texttt{statement}}(Prot, 1)$, we get

$$
\begin{aligned}
S'_{(1)} &= \big\langle\alpha_2\big\rangle = \alpha_1 \wedge \beta_3 = enc(\alpha_1, \beta_4) \wedge \beta_3 = check^\sharp(\beta_1, \beta_2),\\
sec_{(1)} &= \Big(\big\langle m\big\rangle,\ m\Big),\\
pub''_{(1)} &= \Big(sign\big(enc\big(\big\langle m\big\rangle, pk(k_{BE})\big), k_{AS}\big),\ vk(k_{AS}),\\
&\qquad enc\big(\big\langle m\big\rangle, pk(k_{BE})\big),\ pk(k_{BE})\Big).
\end{aligned}
$$

But for $\texttt{compile}_{\texttt{statement}}(Prot, 2)$ we have

$$\begin{aligned}\sigma_{(2)} &= \{dec(x_1, k_{BE})/x_2\}\{check(x, vk(k_{AS}))/x_1\},\\ \sigma_{(2)}^{-1} &= \{x_1/check(x, vk(k_{AS}))\}\{x_2/dec(x_1, k_{BE})\}.\end{aligned}$$

Performing that back substitution for the result of $\texttt{compile}_{\texttt{statement}}(Prot, 2)$, we get

$$\begin{aligned}S'_{(2)} &= \beta_6 = check^\sharp(\beta_8, \beta_7) \wedge \alpha_1 = dec^\sharp(\beta_6, \alpha_2) \wedge \beta_5 = pk(\alpha_2)\\ &\quad \wedge \beta_3 = enc(\alpha_1, \beta_4) \wedge \beta_3 = check^\sharp(\beta_1, \beta_2),\\ sec_{(2)} &= \Big(x_2,\ k_{BE}\Big),\\ pub''_{(2)} &= \Big(sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big),\ vk(k_{BS}),\ enc\big(x_2, pk(k_{CE})\big),\\ &\quad pk(k_{CE}),\ pk(k_{BE}),\ x_1,\ vk(k_{AS}),\ x\Big)\end{aligned}$$

However for the verification of the zero-knowledge proof terms of the public component have to be matched. According to the reduction rule for the *ver* destructor only terms at the beginning of the private component of a zero-knowledge proof can be matched. Therefore the order of the messages in the public component pub'' has to be changed. Public terms returned by $\texttt{public\_terms}(P, k)$ will be matched in the verification, since they are available to everyone. We will denote all public terms in pub'' by $\tilde{L}$ and put them at the beginning of the public component pub. Furthermore we will put the message M sent in the original process as first message behind the terms $\tilde{L}$:

$$\begin{aligned}\tilde{L} &= \Big(\big(pub'' \setminus \{M\}\big) \cap \texttt{public\_terms}(P, k)\Big)\\ pub &= \Big(\tilde{L}, M, pub'' \setminus \big(\texttt{public\_terms}(P, k) \cup \{M\}\big)\Big)\\ (u, M) &= \texttt{output}\,(P, k)\end{aligned}$$

For rearranging the terms we have

$$\begin{aligned}(u_{(1)}, M_{(1)}) &= \big(ch, sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)\big)\\ (u_{(2)}, M_{(2)}) &= \big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\big)\\ \texttt{public\_terms}(Prot, 1) &= \{pk(k_{BE}), pk(k_{CE}), vk(k_{AS}), vk(k_{BS})\}\\ \texttt{public\_terms}(Prot, 2) &= \{pk(k_{BE}), pk(k_{CE}), vk(k_{AS}), vk(k_{BS}), x\}\end{aligned}$$

resulting in

$$\begin{aligned}\tilde{L}_{(1)} &= \Big(vk(k_{AS}),\ pk(k_{BE})\Big)\\ \tilde{L}_{(2)} &= \Big(vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ x\Big)\\ pub_{(1)} &= \Big(vk(k_{AS}),\ pk(k_{BE}),\ sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big),\\ &\quad enc\big(\langle m\rangle, pk(k_{BE})\big)\Big)\\ pub_{(2)} &= \Big(vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ x,\\ &\quad sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big),\ enc\big(x_2, pk(k_{CE})\big),\\ &\quad check(x, vk(k_{AS}))\Big)\end{aligned}$$

In the same way the placeholders in the statement have to be changed. First we compute the length i of the private component *sec*, the length j of the public component *pub*, and the length l of the list of public terms $\tilde{L}$:

$$\begin{aligned} i &= \left|sec\right| \\ j &= \left|pub\right| \\ l &= \left|\tilde{L}\right| \end{aligned}$$

That results in

$$\begin{aligned} i_{(1)} &= 2, \qquad j_{(1)} = 4, \qquad l_{(1)} = 2 \\ i_{(2)} &= 2, \qquad j_{(2)} = 8, \qquad l_{(2)} = 5 \end{aligned}$$

Then we substitute the β's in the statement S' from $\texttt{compile}_{\texttt{statement}}$. We have to pay attention to avoid capturing wrong messages from the reordered public component *pub*. This can be done for instance by first increment the index i of each β_i by j (since there are only j public messages, no message will be captured) and then replace each β_{i+j} (which has been the placeholder for the ith message in pub'') by the position of the message in *pub*:

$$\begin{aligned} S &= S'\sigma' \\ \sigma' &= \{\beta_{1+j}/\beta_1\} \ldots \{\beta_{j+j}/\beta_j\}\{\beta_{l_1}/\beta_{1+j}\} \ldots \{\beta_{l_j}/\beta_{j+j}\} \\ \text{such that } pub[l_i] &= pub''[i], \qquad \text{for } i = 1, \ldots j \end{aligned}$$

The substitutions now is as follows:

$$\begin{aligned} \sigma'_{(1)} = {} & \{\beta_5/\beta_1\}\{\beta_6/\beta_2\}\{\beta_7/\beta_3\}\{\beta_8/\beta_4\} \\ & \{\beta_3/\beta_5\}\{\beta_1/\beta_6\}\{\beta_4/\beta_7\}\{\beta_2/\beta_8\} \\ \sigma'_{(2)} = {} & \{\beta_9/\beta_1\}\{\beta_{10}/\beta_2\}\{\beta_{11}/\beta_3\}\{\beta_{12}/\beta_4\} \\ & \{\beta_{13}/\beta_5\}\{\beta_{14}/\beta_6\}\{\beta_{15}/\beta_7\}\{\beta_{16}/\beta_8\} \\ & \{\beta_6/\beta_9\}\{\beta_1/\beta_{10}\}\{\beta_7/\beta_{11}\}\{\beta_2/\beta_{12}\} \\ & \{\beta_3/\beta_{13}\}\{\beta_8/\beta_{14}\}\{\beta_4/\beta_{15}\}\{\beta_5/\beta_{16}\} \end{aligned}$$

Applying them to the statements we obtain the same statements, but with changed order of messages in the public component:

$$\begin{aligned} S_{(1)} &= \left\langle \alpha_2 \right\rangle = \alpha_1 \wedge \beta_4 = enc(\alpha_1, \beta_2) \wedge \beta_4 = check^\sharp(\beta_3, \beta_1) \\ S_{(2)} &= \beta_8 = check^\sharp(\beta_5, \beta_4) \wedge \alpha_1 = dec^\sharp(\beta_8, \alpha_2) \wedge \beta_3 = pk(\alpha_2) \\ &\quad \wedge \beta_7 = enc(\alpha_1, \beta_2) \wedge \beta_7 = check^\sharp(\beta_6, \beta_1) \end{aligned}$$

Furthermore all inputs have to be forwarded. In the transformed protocol inputs will be zero-knowledge proofs. If zero-knowledge proofs are sent over private channels, we have to encrypt them to avoid revealing secrets. Let $k_1, \ldots, k_n$ be the labels of the inputs (where $k_1 = k$ by definition of $\texttt{graph}(P, k)$) and x_i the input variable of input k_i. The zero-knowledge proof of input k_i will be bound to the variable x_i. If that zero-knowledge proof x_i of input k_i has been sent over a private channel ($\texttt{private\_channel}(P, k_i)$) and we forward that over a public channel ($\neg\texttt{private\_channel}(P, k)$), we have to encrypt it:

$enc\big(x_i, pk(\hat{k}_i)\big)$. For that encryption we will use a new public key $pk(\hat{k}_i)$ for encrypting the zero-knowledge proof from input k_i. Otherwise we forward the input x_i unencrypted. We use $\hat{x}_i$ to refer to the (possibly) encrypted zero-knowledge proof x_i.

$$\begin{aligned}
\texttt{graph}\,(P,k) &= \big(k_1,\ldots,k_n\big)\\
x_i &= \texttt{variable}\,(P,k_i)\,,\ 1<i\leqslant n\\
x'_i &= \begin{cases} enc\left(\hat{x}_i, pk(\hat{k}_i)\right), & \texttt{private_channel}\,(P,k_i)\\ & \quad\wedge\neg\texttt{private_channel}\,(P,k)\\ \hat{x}_i, & \text{otherwise,} \qquad\qquad 1<i\leqslant n \end{cases}
\end{aligned}$$

Note that there is nothing to forward in output 1. Hence we get

$$\begin{aligned}
\texttt{graph}\,(Prot,1) &= 1\\
\texttt{graph}\,(Prot,2) &= (2,1)\\
x_{2\,(2)} &= x\\
x'_{2\,(2)} &= \hat{x}
\end{aligned}$$

Finally $\texttt{compile}_{\texttt{zk}}(P,k)$ returns a tuple consisting of the label k; the length i of the private component of the zero-knowledge proofs generated for the message of output k; the length j of the public component of that proof; the statement S of that proof; the tuple $\big\langle zk_{i,j,S}\,(sec,pub)\,, x'_2,\ldots,x'_n\big\rangle$ consisting the generated zero-knowledge proof for output k, $zk_{i,j,S}\,(sec,pub)$, and the forwarded proofs $x'_2,\ldots,x'_n$; the length l of the list of public terms $\tilde{L}$ and this list $\tilde{L}$:

$$\begin{aligned}
\texttt{compile}_{\texttt{zk}}\,(P,k) = \Big(&k,\ i,\ j,\ S,\ \big\langle zk_{i,j,S}\,(sec,pub)\,, x'_2,\ldots,x'_n\big\rangle,\ l+1,\\
&\big(\tilde{L},\texttt{variable}(P,k)\big)\Big)
\end{aligned}$$

Note that we append the variable $\texttt{variable}(P,k)$ of the input k of the original protocol. This results in the list of public terms $\big(\tilde{L},\texttt{variable}(P,k)\big)$ and its length $l+1$. This is done to be able to link different zero-knowledge proofs and will be exploited for compiling the verification of zero-knowledge proofs by $\texttt{compile}_{\texttt{ver}}$.

After applying the backsubstitution, reordering the public components from the statement compilation, adapting the statement we obtain the following results from the compilation of the zero-knowledge proofs.

$$\begin{aligned}
&\texttt{compile}_{\texttt{zk}}\,(Prot,1)\\
&= \Big(1,2,4,\big\langle\alpha_2\big\rangle = \alpha_1\wedge\beta_4 = enc(\alpha_1,\beta_2)\wedge\beta_4 = check^{\sharp}(\beta_3,\beta_1),\\
&\quad \big\langle zk_{2,4,\langle\alpha_2\rangle=\alpha_1\wedge\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^{\sharp}(\beta_3,\beta_1)}\\
&\quad \big(\big\langle m\big\rangle,\ m;\ vk(k_{AS}),\ pk(k_{BE}),\\
&\quad sign\big(enc\big(\big\langle m\big\rangle,pk(k_{BE})\big),k_{AS}\big),\ enc\big(\big\langle m\big\rangle,pk(k_{BE})\big)\big)\big\rangle,\\
&\quad 3,\big(vk(k_{AS}),\ pk(k_{BE}),\ x\big)\Big)
\end{aligned}$$

$$
\begin{aligned}
&\texttt{compile}_{\texttt{zk}}\,(Prot, 2)\\
&= \Big(2, 2, 8, \beta_8 = check^\sharp(\beta_5, \beta_4) \wedge \alpha_1 = dec^\sharp(\beta_8, \alpha_2) \wedge \beta_3 = pk(\alpha_2)\\
&\qquad\qquad \wedge \beta_7 = enc(\alpha_1, \beta_2) \wedge \beta_7 = check^\sharp(\beta_6, \beta_1),\\
&\quad \big\langle zk_{2,8,\beta_8 = check^\sharp(\beta_5,\beta_4) \wedge \alpha_1 = dec^\sharp(\beta_8,\alpha_2) \wedge \beta_3 = pk(\alpha_2) \wedge \beta_7 = enc(\alpha_1,\beta_2) \wedge \beta_7 = check^\sharp(\beta_6,\beta_1)}\\
&\quad \big(x_2,\ k_{BE};\ vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ x,\\
&\quad sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big),\ enc\big(x_2, pk(k_{CE})\big),\ x_1\big), \hat{x}\big\rangle,\\
&\quad 6, \big(vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ x,\ y\big)\Big)
\end{aligned}
$$

For replacing the message M in the original process only the tuple $\langle zk_{i,j,S}\,(sec, pub)\,, x'_2, \ldots, x'_n\rangle$ containing the compiled zero-knowledge proof and the other forwarded proofs is used. All other components will be used for the verification of the zero-knowledge proofs. In summary $\texttt{compile}_{\texttt{zk}}$ is defined as:

$$
\begin{aligned}
\texttt{compile}_{\texttt{zk}} &: \mathcal{P} \times \mathcal{L} \to \mathcal{L} \times \mathbb{N} \times \mathbb{N} \times \mathcal{S} \times \mathcal{T} \times \mathbb{N} \times \mathcal{T}\\
\texttt{compile}_{\texttt{zk}}\,(P, k) &= \Big(k, i, j, S, \big\langle zk_{i,j,S}\,(sec, pub)\,, x'_2, \ldots, x'_n\big\rangle,\\
&\qquad\qquad l + 1, \big(\tilde{L}, \texttt{variable}(P, k)\big)\Big)\\
\text{with } \big(S', sec', pub'\big) &= \texttt{compile}_{\texttt{statement}}\,(P, k)\\
(u, M) &= \texttt{output}\,(P, k)\\
\sigma &= \texttt{deseva}\,(P, k)\\
sec &= sec'\sigma^{-1}\\
pub'' &= pub'\,\sigma^{-1}\\
\tilde{L} &= \Big(\big(pub'' \setminus \{M\}\big) \cap \texttt{public\_terms}(P, k)\Big)\\
pub &= \Big(\tilde{L}, M, pub'' \setminus \big(\texttt{public\_terms}(P, k) \cup \{M\}\big)\Big)\\
i &= \big|sec\big|\\
j &= \big|pub\big|\\
l &= \big|\tilde{L}\big|\\
S &= S'\sigma'\\
\sigma' &= \{\beta_{1+j}/\beta_1\} \ldots \{\beta_{j+j}/\beta_j\}\{\beta_{l_1}/\beta_{1+j}\} \ldots \{\beta_{l_j}/\beta_{j+j}\}\\
\text{such that } pub[l_i] &= pub''[i], \qquad \text{for } i = 1, \ldots j\\
\texttt{graph}\,(P, k) &= \big(k_1, \ldots, k_n\big)\\
x_i &= \texttt{variable}\,(P, k_i)\,,\ 1 < i \leqslant n\\
x'_i &= \begin{cases}
enc\left(\hat{x}_i, pk(\hat{k}_i)\right), & \texttt{private\_channel}\,(P, k_i)\\
& \quad \wedge \neg\texttt{private\_channel}\,(P, k)\,,\ 1 < i \leqslant n\\
\hat{x}_i, & \text{otherwise}
\end{cases}
\end{aligned}
$$

- Given a process P, a continuation process Q, and a label k, $\texttt{compile}_{\texttt{ver}}(P, Q, k)$ returns a process, which replaces the continuation process Q in the original process P.

$$\texttt{compile}_{\texttt{ver}} : \mathcal{P} \times \mathcal{P} \times \mathcal{L} \to \mathcal{P}$$

$\texttt{compile}_{\texttt{ver}}$ basically inserts a process between the input k and the continuation process Q. As stated above we use $\texttt{compile}_{\texttt{zk}}$ to replace the term M of an output k' by a tuple $\langle \hat{x}_1, \ldots, \hat{x}_n \rangle$, where each $\hat{x}_p$ is a zero-knowledge proof – either compiled for that specific output k' (as it is the case for $\hat{x}_1$) or forwarded (as it is the case for the other ones). In order to process the transformed output, several steps have to be performed and inserted before continuing the protocol with the process Q.

$$\texttt{compile}_{\texttt{ver}}\,(P, Q, k) = \ldots$$

The tuple $\hat{\hat{x}}$ received from the input k will be split into its components $\hat{\hat{x}}_1, \ldots, \hat{\hat{x}}_n$. Note that the algorithm will change input variables (see `transform`), since the content of the corresponding output will be changed by $\texttt{compile}_{\texttt{zk}}$. The input variable x in the original process will be changed to $\hat{\hat{x}}$ in the transformed process. Therefore we do not split x, but $\hat{\hat{x}}$. To find out how many components the input tuple will consist of, we use the transitive closure of `graph`: $tc(\texttt{graph})$ will return a list of labels $(k_1, \ldots, k_n)$, where we have $k = k_1$ and each $\hat{\hat{x}}_p$ will be the zero-knowledge proof compiled for output k_p when executing the process.

$$\begin{aligned} &\texttt{let } \left\langle \hat{\hat{x}}_1, \ldots, \hat{\hat{x}}_n \right\rangle = \hat{\hat{x}} \texttt{ in} \\ &\text{with } x := \texttt{variable}(P, k) \\ tc(\texttt{graph})\,(P, k) &= (k_1, \ldots, k_n) \end{aligned}$$

For our example we then have the following preliminary results:

$$\begin{aligned} \texttt{variable}(Prot, 1) &= x \\ \texttt{variable}(Prot, 2) &= y \\ tc(\texttt{graph})\,(Prot, 1) &= 1 \\ tc(\texttt{graph})\,(Prot, 2) &= (2, 1) \\ \texttt{compile}_{\texttt{ver}}(Prot, Q, 1) &= \texttt{let } \langle \hat{\hat{x}}_1 \rangle = \hat{\hat{x}} \texttt{ in} \\ &\quad \ldots \\ \texttt{compile}_{\texttt{ver}}(Prot, Q, 2) &= \texttt{let } \langle \hat{\hat{y}}_1, \hat{\hat{y}}_2 \rangle = \hat{\hat{y}} \texttt{ in} \\ &\quad \ldots \end{aligned}$$

In the next step we decrypt the zero-knowledge proofs $\hat{\hat{x}}_1, \ldots, \hat{\hat{x}}_n$ resulting in the decrypted zero-knowledge proofs $\hat{x}_1, \ldots, \hat{x}_n$. If they have been encrypted before forwarding, otherwise we apply the id destructor. According to our definition of $\texttt{compile}_{\texttt{zk}}$ the first zero-knowledge proof is never encrypted, hence we do not have to decrypt it.

$$\begin{aligned} &\texttt{let } \hat{x}_1 = N_1 \texttt{ in} \\ &\ldots \\ &\texttt{let } \hat{x}_n = N_n \texttt{ in} \\ \text{with } N_p = &\begin{cases} dec\left(\hat{\hat{x}}_p, \hat{k}_p\right), & \texttt{private\_channel}\,(P, k_p) \wedge p \neq 1 \\ id\left(\hat{\hat{x}}_p\right), & \texttt{otherwise}, \end{cases} \qquad 1 \leqslant p \leqslant n \end{aligned}$$

We have that for $\texttt{compile}_{\texttt{ver}}(Prot, 1)$, $p = 1$ and for $\texttt{compile}_{\texttt{ver}}(Prot, 2)$, p ranges from 1 to 2. Now we get:

$$
\begin{aligned}
N_{1\,(1)} &= id(\hat{\hat{x}}_1) \\
N_{1\,(2)} &= id(\hat{\hat{y}}_1) \\
N_{2\,(2)} &= id(\hat{\hat{y}}_2) \\
\texttt{compile}_{\texttt{ver}}(Prot, Q, 1) &= \ldots \\
&\quad \texttt{let } \hat{x}_1 = id(\hat{\hat{x}}_1) \texttt{ in} \\
&\quad \ldots \\
\texttt{compile}_{\texttt{ver}}(Prot, Q, 2) &= \ldots \\
&\quad \texttt{let } \hat{y}_1 = id(\hat{\hat{y}}_1) \texttt{ in} \\
&\quad \texttt{let } \hat{y}_2 = id(\hat{\hat{y}}_2) \texttt{ in} \\
&\quad \ldots
\end{aligned}
$$

For the verification of the zero-knowledge proofs $\hat{x}_1, \ldots, \hat{x}_n$, we will need some messages from the public components $public_{j_1}(\hat{x}_1), \ldots, public_{j_n}(\hat{x}_n)$ of the proofs. We extract them from the proofs by applying the *public* destructor to each $\hat{x}_p$ and then split the tuple obtained from the application of the *public* destructor. To do so, we need the arity j_p of each public component of the zero-knowledge proof $\hat{x}_p$, which we get from $\texttt{compile}_{\texttt{zk}}$ (we omit the components of the result which are not necessary for this step). We denote the mth message of the public component of the pth proof by $\hat{x}_{p,m}$.

$$
\begin{aligned}
&\texttt{let } \langle \hat{x}_{n,1}, \ldots, \hat{x}_{n,j_n} \rangle = public_{j_n}(\hat{x}_n) \texttt{ then} \\
&\ldots \\
&\texttt{let } \langle \hat{x}_{1,1}, \ldots, \hat{x}_{1,j_1} \rangle = public_{j_1}(\hat{x}_1) \texttt{ then} \\
\text{with } \texttt{compile}_{\texttt{zk}}(P, k_p) &= \big(p, \ldots, j_p, \ldots, \ldots, \ldots, \ldots\big), \qquad 1 \leqslant p \leqslant n
\end{aligned}
$$

Extracting the messages from the public components in our example is done in the following way:

$$
\begin{aligned}
j_{1\,(1)} &= 4, \qquad j_{1\,(2)} = 8 \qquad j_{2\,(2)} = j_{1\,(1)} = 4 \\
\texttt{compile}_{\texttt{ver}}(Prot, Q, 1) &= \ldots \\
&\quad \texttt{let } \langle \hat{x}_{1,1}, \hat{x}_{1,2}, \hat{x}_{1,3}, \hat{x}_{1,4} \rangle = public_4(\hat{x}_1) \texttt{ then} \\
&\quad \ldots \\
\texttt{compile}_{\texttt{ver}}(Prot, Q, 2) &= \ldots \\
&\quad \texttt{let } \langle \hat{y}_{2,1}, \hat{y}_{2,2}, \hat{y}_{2,3}, \hat{y}_{2,4} \rangle = public_4(\hat{y}_2) \texttt{ then} \\
&\quad \texttt{let } \langle \hat{y}_{1,1}, \hat{y}_{1,2}, \hat{y}_{1,3}, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7}, \hat{y}_{1,8} \rangle = public_8(\hat{y}_1) \texttt{ then} \\
&\quad \ldots
\end{aligned}
$$

The fourth step is the verification of each zero-knowledge proof $ver_{i_p, j_p, l_p, S_p}\big(\hat{x}_p, \tilde{L}_p\big)$. While basically all messages which we have to use in the matching part of the verification are collected in each list $(M_{p,1}, \ldots, M_{p,l_p})$ generated by $\texttt{compile}_{\texttt{zk}}$, the variables in those lists have to be changed. This is necessary in order to link different zero-knowledge proofs:

For generation the input variables of a process representing a participant have been used, while now those input variables have to be extracted from forwarded zero-knowledge proofs. For changing the variable names in $(M_{p,1},\ldots,M_{p,l_p})$ we use the substitution σ_p: If a term $M_{p,l}$ in $(M_{p,1},\ldots,M_{p,l_p})$ is a input variable in the process with output k_p, then we substitute that term with $\hat{x}_{p,l}$. If a term $M_{p',l'}$ in $(M_{p',1},\ldots,M_{p',l_{p'}})$ as $M_{p',l'}$ is a variable in the process with output $k_{p'}$ and occurs in $(M_{p,1},\ldots,M_{p,l_p})$ as $M_{p,l}$ then we substitute $M_{p,l}$ by $\hat{x}_{p,l}$. We perform the substitutions σ_p an each list $(M_{p,1},\ldots,M_{p,l_p})$, except for σ_1: here we only perform them on the shortened list $(M_{1,1},\ldots,M_{1,l_1-1})$ (M_{1,l_p} will be assigned to the original input variable in the next step).

Furthermore the message $\hat{x}_{1,l_1}$ of the public component of the first zero-knowledge proof is assigned to the variable x which has been the input variable in the original process. Then we continue with the continuation process Q.

$$
\begin{aligned}
&\texttt{let } \hat{x}_{n,V} = ver_{i_n,j_n,l_n,S_n}\left(\hat{x}_n,\tilde{L}_n\right) \texttt{ then}\\
&\ldots\\
&\texttt{let } \hat{x}_{2,V} = ver_{i_2,j_2,l_2,S_2}\left(\hat{x}_2,\tilde{L}_2\right) \texttt{ then}\\
&\texttt{let } \hat{x}_{1,V} = ver_{i_1,j_1,l_1-1,S_1}\left(\hat{x}_1,\tilde{L}_1\right) \texttt{ then}\\
&\texttt{let } \langle \hat{x}_{n,l_n+1},\ldots \hat{x}_{n,j_n}\rangle = \hat{x}_{n,V} \texttt{ in}\\
&\ldots\\
&\texttt{let } \left\langle \hat{x}_{2,l_2+1},\ldots \hat{x}_{2,j_2}\right\rangle = \hat{x}_{2,V} \texttt{ in}\\
&\texttt{let } \left\langle x, \hat{x}_{n,l_1+1},\ldots \hat{x}_{n,j_1}\right\rangle = \hat{x}_{1,V} \texttt{ in } Q\\
\text{with } \texttt{compile}_{\texttt{zk}}(P,k_p) &= \big(p,i_p,j_p,S_p,\ldots,l_p,\big(M_{p,1},\ldots,M_{p,l_p}\big)\big), \qquad 1\leqslant p\leqslant n\\
\tilde{L}_p &= \begin{cases}\big(M_{1,1},\ldots,M_{1,l_1-1}\big)\sigma_1, & p=1\\ (M_{p,1},\ldots,M_{p,l_p})\sigma_p, & p\neq 1\end{cases}\\
\sigma_p &= \{\hat{x}_{p,l}/M_{p,l}\} \text{ where } M_{p,l}\in \texttt{inputvariables}_{\texttt{pub}}(P,k_p)\\
&\quad \{\hat{x}_{p',l'}/M_{p,l}\} \text{ where } M_{p,l} = \texttt{inputvariables}_{\texttt{pub}}(P,k_{p'})\\
&\qquad\qquad \wedge M_{p',l'} = M_{p,l}
\end{aligned}
$$

With $\texttt{inputvariables}_{\texttt{pub}}(Prot,1) = \varnothing$ and $\texttt{inputvariables}_{\texttt{pub}}(Prot,2) = \{x\}$ we get for this step:

$$
\begin{aligned}
\big(M_{1,1\,(1)},\ldots,M_{1,3\,(1)}\big) &= \big(vk(k_{AS}),\ pk(k_{BE}),\ x\big)\\
\big(M_{1,1\,(2)},\ldots,M_{1,6\,(2)}\big) &= \big(vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ x,\ y\big)\\
\big(M_{2,1\,(2)},\ldots,M_{2,3\,(2)}\big) &= \big(vk(k_{AS}),\ pk(k_{BE}),\ x\big)\\
\sigma_{1\,(1)} &= \varepsilon\\
\sigma_{1\,(2)} &= \{\hat{y}_{1,5}/x\}\\
\sigma_{2\,(2)} &= \{\hat{y}_{1,5}/x\}\\
\tilde{L}_{1\,(1)} &= \big(vk(k_{AS}),\ pk(k_{BE})\big)\\
\tilde{L}_{1\,(2)} &= \big(vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ \hat{y}_{1,5}\big)\\
\tilde{L}_{2\,(2)} &= \big(vk(k_{AS}),\ pk(k_{BE}),\ \hat{y}_{1,5}\big)
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{compile}_{\mathtt{ver}}(Prot, Q, 1) = \ldots \\
&\qquad\qquad \mathtt{let}\ \hat{x}_{1,V} = ver_{2,4,2,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1} \\
&\qquad\qquad\qquad\qquad (\hat{x}_1, vk(k_{AS}),\ pk(k_{BE}))\ \mathtt{then} \\
&\qquad\qquad \mathtt{let}\ \langle x, \hat{x}_{1,4}\rangle = \hat{x}_{1,V}\ \mathtt{in}\ Q \\
&\mathtt{compile}_{\mathtt{ver}}(Prot, Q, 2) = \ldots \\
&\qquad\qquad \mathtt{let}\ \hat{y}_{2,V} = ver_{2,4,3,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1} \\
&\qquad\qquad\qquad\qquad (\hat{y}_2, vk(k_{AS}),\ pk(k_{BE}),\ \hat{y}_{1,5})\ \mathtt{then} \\
&\qquad\qquad \mathtt{let}\ \hat{y}_{1,V} = ver_{2,8,5,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)} \\
&\qquad\qquad\qquad\qquad (\hat{y}_1, vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ \hat{y}_{1,5})\ \mathtt{then} \\
&\qquad\qquad \mathtt{let}\langle \hat{y}_{2,4}\rangle = \hat{y}_{2,V}\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \langle y, \hat{y}_{1,7}, \hat{y}_{1,8}\rangle = \hat{y}_{1,V}\ \mathtt{in}\ Q
\end{aligned}
$$

The processes which will be inserted after receiving the inputs 1 and 2 are the following (here we abbreviate the continuation process with Q)

$$
\begin{aligned}
&\mathtt{compile}_{\mathtt{ver}}\,(Prot, Q, 1) = \\
&\qquad\qquad \mathtt{let}\ \langle \hat{x}_1\rangle = \hat{\hat{x}}\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \langle \hat{x}_{1,1}, \hat{x}_{1,2}, \hat{x}_{1,3}, \hat{x}_{1,4}\rangle = public_4(\hat{x}_1)\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \hat{x}_{1,V} = ver_{2,4,2,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1} \\
&\qquad\qquad\qquad\qquad \big(\hat{x}_1, vk(k_{AS}),\ pk(k_{BE})\big)\ \mathtt{then} \\
&\qquad\qquad \mathtt{let}\ \langle x, \hat{x}_{1,4}\rangle = \hat{x}_{1,V}\ \mathtt{in}\ Q \\
&\mathtt{compile}_{\mathtt{ver}}\,(Prot, Q, 2) = \\
&\qquad\qquad \mathtt{let}\ \langle \hat{y}_1, \hat{y}_2\rangle = \hat{\hat{y}}\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \langle \hat{y}_{2,1}, \hat{y}_{2,2}, \hat{y}_{2,3}, \hat{y}_{2,4}\rangle = public_4(\hat{y}_2)\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \langle \hat{y}_{1,1}, \hat{y}_{1,2}, \hat{y}_{1,3}, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7}\rangle = public_7(\hat{y}_1)\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \hat{y}_{2,V} = ver_{2,4,3,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1} \\
&\qquad\qquad\qquad\qquad \big(\hat{y}_2, vk(k_{AS}),\ pk(k_{BE}),\ \hat{y}_{1,5}\big)\ \mathtt{then} \\
&\qquad\qquad \mathtt{let}\ \hat{y}_{1,V} = ver_{2,8,5,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)} \\
&\qquad\qquad\qquad\qquad \big(\hat{y}_1, vk(k_{BS}),\ pk(k_{CE}),\ pk(k_{BE}),\ vk(k_{AS}),\ \hat{y}_{1,5}\big)\ \mathtt{then} \\
&\qquad\qquad\quad \langle \hat{y}_{2,4}\rangle = \hat{y}_{2,V}\ \mathtt{in} \\
&\qquad\qquad \mathtt{let}\ \langle y, \hat{y}_{1,7}, \hat{y}_{1,8}\rangle = \hat{y}_{1,V}\ \mathtt{in}\ Q
\end{aligned}
$$

Summing up all steps we get:

$$
\begin{aligned}
\texttt{compile}_{\texttt{ver}} &: \mathcal{P} \times \mathcal{P} \times \mathcal{L} \to \mathcal{P} \\
\texttt{compile}_{\texttt{ver}}\,(P, Q, k) &= \texttt{let } \left\langle \hat{\hat{x}}_1, \ldots, \hat{\hat{x}}_n \right\rangle = \hat{\hat{x}} \texttt{ in} \\
&\qquad \texttt{let } \hat{x}_2 = N_2 \texttt{ in} \ldots \texttt{let } \hat{x}_n = N_n \texttt{ in} \\
&\qquad \texttt{let } \langle \hat{x}_{n,1}, \ldots, \hat{x}_{n,j_n} \rangle = \mathit{public}_{j_n}\,(\hat{x}_n) \texttt{ then} \\
&\qquad \ldots \\
&\qquad \texttt{let } \langle \hat{x}_{1,1}, \ldots, \hat{x}_{1,j_1} \rangle = \mathit{public}_{j_1}\,(\hat{x}_1) \texttt{ then} \\
&\qquad \texttt{let } \hat{x}_{n,V} = \mathit{ver}_{i_n,j_n,l_n,S_n}\left(\hat{x}_n, \tilde{L}_n\right) \texttt{ then} \\
&\qquad \ldots \\
&\qquad \texttt{let } \hat{x}_{2,V} = \mathit{ver}_{i_2,j_2,l_2,S_2}\left(\hat{x}_2, \tilde{L}_2\right) \texttt{ then} \\
&\qquad \texttt{let } \hat{x}_{1,V} = \mathit{ver}_{i_1,j_1,l_1-1,S_1}\left(\hat{x}_1, \tilde{L}_1\right) \texttt{ then} \\
&\qquad \texttt{let } \langle \hat{x}_{n,l_n+1}, \ldots \hat{x}_{n,j_n} \rangle = \hat{x}_{n,V} \texttt{ in} \\
&\qquad \ldots \\
&\qquad \texttt{let } \left\langle \hat{x}_{2,l_2+1}, \ldots \hat{x}_{2,j_2} \right\rangle = \hat{x}_{2,V} \texttt{ in} \\
&\qquad \texttt{let } \left\langle x, \hat{x}_{n,l_1+1}, \ldots \hat{x}_{n,j_1} \right\rangle = \hat{x}_{1,V} \texttt{ in } Q \\
&\qquad \text{with } x := \texttt{variable}(P, k) \\
\mathit{tc}(\texttt{graph})\,(P, k) &= (k_1, \ldots, k_n) \\
N_p &= \begin{cases} \mathit{dec}\left(\hat{\hat{x}}_p, \hat{k}_p\right), & \texttt{private\_channel}\,(P, k_p) \wedge p \neq 1 \\ \mathit{id}\left(\hat{\hat{x}}_p\right), & \texttt{otherwise}, \end{cases} \qquad 1 \leqslant p \leqslant n \\
\texttt{compile}_{\texttt{zk}}\,(P, k_p) &= \big(p, i_p, j_p, S_p, \ldots, l_p, (M_{p,1}, \ldots, M_{p,\,l_p})\big), \qquad 1 \leqslant p \leqslant n \\
\tilde{L}_p &= \begin{cases} (M_{1,1}, \ldots, M_{1,\,l_1-1})\sigma_1, & p = 1 \\ (M_{p,1}, \ldots, M_{p,\,l_p})\sigma_p, & p \neq 1 \end{cases} \\
\sigma_p &= \{\hat{x}_{p,\,l}/M_{p,\,l}\} \text{ where } M_{p,\,l} \in \texttt{inputvariables}_{\texttt{pub}}(P, k_p) \\
&\qquad \{\hat{x}_{p',\,l'}/M_{p,\,l}\} \text{ where } M_{p,\,l} = \texttt{inputvariables}_{\texttt{pub}}(P, k_{p'}) \\
&\qquad\qquad \wedge\, M_{p',\,l'} = M_{p,\,l}
\end{aligned}
$$

- The transformation algorithm `transform` returns the compiled process. The algorithm compiles for every output the zero-knowledge proof, if there is an input with matching label, and forwards the other zero-knowledge proofs. It changes the variables of all inputs (since now tuple of zero-knowledge proofs are bound to them) and inserts the verification of all received zero-knowledge proofs for inputs, where there is a corresponding output with matching label. Additionally private keys – which are restricted – are added to the process. The corresponding encryption keys are output to allow for the decryption of

zero-knowledge proofs received from private channels.

$$\texttt{transform} : \mathcal{P} \rightarrow \mathcal{P}$$

$$\texttt{transform}\,(P) = \nu\tilde{k}.\texttt{out}\left(ch_k, p\tilde{k}(k)\right)\left(\texttt{transform}'\,(P,P)\right)$$

$$\texttt{transform}' : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$$

$$\texttt{transform}'\,(P, k : \texttt{out}\,(u, M)\,.Q) = \begin{cases} k : \texttt{out}\,(u, M_k)\,.\texttt{transform}'\,(P,Q)\,, \\ \qquad \text{if } ' \in \texttt{labels}_{\texttt{in}}(P) \cap \texttt{labels}_{\texttt{out}}(P) \\ k : \texttt{out}\,(u, M)\,.\texttt{transform}'\,(P,Q)\,, \\ \qquad \text{otherwise} \end{cases}$$

$$\texttt{transform}'\,(P, k : [!]\texttt{in}\,(u, x)\,.Q) = \begin{cases} k : [!]\texttt{in}\left(u, \hat{\hat{x}}_{in}\right).\texttt{compile}_{\texttt{ver}}\,(P, \texttt{transform}'\,(P,Q)\,, k)\,, \\ \qquad \text{if } k \in \texttt{labels}_{\texttt{in}}(P) \cap \texttt{labels}_{\texttt{out}}(P) \\ k : [!]\texttt{in}\,(u, x)\,.\texttt{transform}\,(Q)\,, \\ \qquad \text{otherwise} \end{cases}$$

$$\texttt{transform}'\,(P, \nu n.Q) = \nu n.\texttt{transform}'\,(P,Q)$$

$$\texttt{transform}'\left(P, Q|Q'\right) = \texttt{transform}'\,(P,Q)\,|\texttt{transform}'\left(P,Q'\right)$$

$$\texttt{transform}'\,(P, 0) = 0$$

$$\texttt{transform}'\big(P, \texttt{let } x = g(\tilde{M}) \texttt{ then } Q \texttt{ else } Q'\big) = \texttt{let } x = g(\tilde{M}) \texttt{ then transform}'\,(P,Q) \texttt{ else transform}'\left(P,Q'\right)$$

$$\texttt{transform}'\,(P, \texttt{let } \langle x_1, \ldots, x_n\rangle = M \texttt{ in } Q) = \texttt{let } \langle x_1, \ldots, x_n\rangle = M \texttt{ in transform}'\,(P,Q)$$

$$\texttt{transform}'\,(P, \texttt{assume } C) = \texttt{assume } C$$

$$\texttt{transform}'\,(P, \texttt{assert } C) = \texttt{assert } C$$

with $\left(k, i_k, j_k, S_k, M_k, l_k, \tilde{L}_k\right) = \texttt{compile}_{\texttt{zk}}\,(P,k)$

$$\texttt{labels}_{\texttt{out}}\,(P) = \left(k_1, \ldots, k_n\right),$$

$$\tilde{k} = \left\{\hat{k}_i | \texttt{private\_channel}\,(P, k_i)\right\},$$

$$p\tilde{k}(k) = \left\{pk(k) | k \in \tilde{k}\right\},$$

$$ch_k \notin free(\texttt{transform}'(P,P))$$

5.2 Types

The algorithm changes messages which are sent in the process, hence the types in the process are also changed.

The type of a zero-knowledge proofs $zk_{i,j,S}\,(n_1,\ldots,n_i;m_1,\ldots,m_j)$ depends on the type of the public component of the proof. The type of the public component is a dependant tuple type. The formula for this dependent tuple type is obtained by substituting the placeholders in the statement by existentially quantified variable $x_1,\ldots,x_i$ for messages from the private component, variables $y_1,\ldots,y_j$ used for typing the singe terms of the public component, and a conjunction of formulas. The conjunction consists of all formulas which occur in assumptions preceding the output k and which contain at least one name which is restricted in the output k, since we want to obtain the formulas containing restricted names. For typing a single term m_k of the public component of a zero-knowledge proof we use the typing environment $\Gamma \vdash m_k : T'_k$.

$$
\begin{aligned}
f_\Gamma^{zk} &: \mathcal{P}\times\mathcal{I}\times\mathcal{T}\to type_{\mathcal{T}}\\
f_\Gamma^{zk}\Big(P,k,zk_{i,j,S}\big(n_1,\ldots,n_i;m_1,\ldots,m_j\big)\Big) &=\\
&\qquad \mathsf{ZKProof}_{i,j,S}(\langle y_1:T'_1,\ldots,y_j:T'_j\rangle\{\exists x_1,\ldots,x_i : S\sigma\wedge\textstyle\bigwedge_{i=1}^{m}F_i\sigma\})\\
&\text{with } \Gamma\vdash m_k:T'_k \qquad k=1,\ldots,j,\\
&\qquad \sigma=\{x_1,\ldots,x_i/\alpha_1,\ldots,\alpha_i\}\{y_1,\ldots,y_j/\beta_1,\ldots,\beta_j\}\\
\{F_1,\ldots F_m\} &= \{F\big|\exists C,Q,Q' : P\mathrel{\widehat{=}} C\left[\mathtt{assume}\ (F)|Q\right]\wedge Q\mathrel{\widehat{=}} C\left[k:\mathtt{out}\,(u,M)\,.Q'\right]\\
&\qquad\qquad \wedge \mathit{free}(F)\cap\mathtt{restricted}(P,k)\neq\emptyset\}
\end{aligned}
$$

According to the definition of $\mathtt{compile_{zk}}$ the message M of the output k in the original process P will be transformed into a tuple of zero-knowledge proofs. To find out how many zero-knowledge proofs the tuple will consist of, we use the transitive closure of `graph`: $tc(\mathtt{graph})$ will return a list of labels $(k_1,\ldots,k_n)$, where we have $k=k_1$. The corresponding message M' in the output k of the transformed process $P'=\mathtt{transform}(P)$ will have the following type:

$$
\begin{aligned}
f_\Gamma^{type} &: \mathcal{P}\times\mathcal{I}\times\mathcal{T}\to type_{\mathcal{T}}\\
f_\Gamma^{type}(P',k,M') &= \langle y_1:T_1,\ldots,y_n:T_n\rangle\\
\text{with } (u',M') &= \mathtt{output}(P',k)\\
tc(\mathtt{graph})(P,k) &= (k_1,\ldots,k_n)\\
M' &= \langle M'_1,\ldots,M'_n\rangle\\
f_\Gamma^{zk}(P,k_i,M_i) &= T'_i, \qquad i=1,\ldots,j,\\
T_i &= \begin{cases}
\mathsf{PubEnc}(T'_i), & \mathtt{private\_channel}(P,k_i)\\
& \qquad\wedge\neg\mathtt{private\_channel}(P,k)\\
T'_i, & \text{otherwise,} \qquad\qquad 1\leqslant i\leqslant n
\end{cases}
\end{aligned}
$$

For our example we get the following types for the zero-knowledge proofs: For

$$
\begin{aligned}
&zk_{2,4,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1}\\
&\qquad\big(\langle m\rangle,m;vk(k_{AS}),pk(k_{BE}),sign\big(enc\big(\langle m\rangle,pk(k_{BE})\big),k_{AS}\big),enc\big(\langle m\rangle,pk(k_{BE})\big)\big)
\end{aligned}
$$

we get

$$\mathsf{ZKProof}_{2,4,\beta_4=enc(\alpha_1,\beta_2)\wedge\beta_4=check^\sharp(\beta_3,\beta_1)\wedge\langle\alpha_2\rangle=\alpha_1}$$
$$\Big(\langle y_1 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})),$$
$$y_2 : \mathsf{PubKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\}),$$
$$y_3 : \mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})),$$
$$y_4 : \mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})\rangle$$
$$\{\exists x_1, x_2 : y_4 = enc(x_1, y_2) \wedge y_4 = check^\sharp(y_3, y_1) \wedge \langle x_2\rangle = x_1 \wedge Good(x_2)\}\Big)$$

and for

$$zk_{2,8,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)}$$
$$\big(x_2, k_{BE}; vk(k_{AS}), pk(k_{CE}), pk(k_{BE}), vk(k_{BS}), x,$$
$$sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big), enc\big(x_2, pk(k_{CE})\big), x_1\big)$$

we get

$$\mathsf{ZKProof}_{2,8,\beta_8=check^\sharp(\beta_5,\beta_4)\wedge\alpha_1=dec^\sharp(\beta_8,\alpha_2)\wedge\beta_3=pk(\alpha_2)\wedge\beta_7=enc(\alpha_1,\beta_2)\wedge\beta_7=check^\sharp(\beta_6,\beta_1)}$$
$$\Big(\langle y_1 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})),$$
$$y_2 : \mathsf{PubKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}),$$
$$y_3 : \mathsf{PubKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})),$$
$$y_4 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})),$$
$$y_5 : \mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})),$$
$$y_6 : \mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\})),$$
$$y_7 : \mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z) \vee Bsaysfalse\}),$$
$$y_8 : \mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{Good(z)\})\rangle$$
$$\{\exists x_1, x_2 : y_8 = check^\sharp(y_5, y_4) \wedge x_1 = dec^\sharp(y_8, x_2) \wedge y_3 = pk(x_2)$$
$$\wedge\, y_7 = enc(x_1, y_2) \wedge y_7 = check^\sharp(y_6, y_1)\}\Big)$$

Example

For our example from above

$$A \mathrel{\widehat{=}} 1 : \mathrm{out}\Big(ch, sign\big(enc\big(\langle m\rangle, pk(k_{BE})\big), k_{AS}\big)\Big)$$
$$B \mathrel{\widehat{=}} 1 : \mathrm{in}\big(ch, x\big).\texttt{let}\ x_1 = check(x, vk(k_{AS}))\ \texttt{then let}\ x_2 = dec(x_1, k_{BE})$$
$$\texttt{then}\ 2 : \mathrm{out}\Big(ch, sign\big(enc\big(x_2, pk(k_{CE})\big), k_{BS}\big)\Big)$$
$$C \mathrel{\widehat{=}} 2 : \mathrm{in}\big(ch, y\big).\texttt{let}\ y_1 = check(y, vk(k_{AS}))\ \texttt{then let}\ y_2 = dec(y_1, k_{BE})\ \texttt{then}\ Q$$

$$Prot \mathrel{\widehat{=}} \Big(\nu m.\nu k_{AS}.\nu k_{BE}.\nu k_{BS}.\nu k_{CE}.$$
$$\texttt{out}\,(ch, pk(k_{BE}))\,.\texttt{out}\,(ch, pk(k_{CE}))\,.\texttt{out}\,(ch, vk(k_{AS}))\,.\texttt{out}\,(ch, vk(k_{BE}))$$
$$\big(A|B|C\big)\Big)$$

our algorithm generates the following protocol:

$$
\begin{aligned}
A \mathrel{\widehat{=}} {} & \nu m : \mathsf{PrivateUnlessB}. \\
& \big(\mathtt{assume}(\mathit{Good}(m))\big|1 : \mathrm{out}\big(\mathit{ch}, \mathit{zk}_{2,4,S_1} \\
& \quad \big(\langle m\rangle, m; \mathit{vk}(k_{AS}), \mathit{pk}(k_{BE}), \mathit{sign}\big(\mathit{enc}\big(\langle m\rangle, \mathit{pk}(k_{BE})\big), k_{AS}\big), \mathit{enc}\big(\langle m\rangle, \mathit{pk}(k_{BE})\big)\big)\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
B \mathrel{\widehat{=}} {} & 1 : \mathrm{in}\big(\mathit{ch}, \hat{\hat{x}}\big). \\
& \mathtt{let}\ \big\langle \hat{x}_1 \big\rangle = \hat{\hat{x}}\ \mathtt{in} \\
& \mathtt{let}\ \big\langle \hat{x}_{1,1}, \hat{x}_{1,2}, \hat{x}_{1,3}, \hat{x}_{1,4} \big\rangle = \mathit{public}_4(\hat{x}_1)\ \mathtt{in} \\
& \mathtt{let}\ \hat{x}_{1,V} = \mathit{ver}_{2,4,2,S_1}\big(\hat{x}_1, \mathit{vk}(k_{AS}),\ \mathit{pk}(k_{BE})\big)\ \mathtt{then} \\
& \mathtt{let}\ \langle x, \hat{x}_{1,4} \rangle = \hat{x}_{1,V}\ \mathtt{in} \\
& \mathtt{let}\ x_1 = \mathit{check}(x, \mathit{vk}(k_{AS}))\ \mathtt{then} \\
& \mathtt{let}\ x_2 = \mathit{dec}\big(x_1, k_{BE}\big) \\
& \mathtt{then}\ 2 : \mathrm{out}\Big(\mathit{ch}, \big\langle \mathit{zk}_{2,8,S_2}\big(x_2, k_{BE}; \mathit{vk}(k_{AS}), \mathit{pk}(k_{CE}), \mathit{pk}(k_{BE}), \mathit{vk}(k_{BS}), x, \\
& \quad \mathit{sign}\big(\mathit{enc}\big(x_2, \mathit{pk}(k_{CE})\big), k_{BS}\big), \mathit{enc}\big(x_2, \mathit{pk}(k_{CE})\big), x_1\big), \hat{x}_1 \big\rangle\Big)
\end{aligned}
$$

$$
\begin{aligned}
C \mathrel{\widehat{=}} {} & 2 : \mathrm{in}\big(\mathit{ch}, \hat{\hat{y}}\big). \\
& \mathtt{let}\ \big\langle \hat{y}_1, \hat{y}_2 \big\rangle = \hat{\hat{y}}\ \mathtt{in} \\
& \mathtt{let}\ \big\langle \hat{y}_{2,1}, \hat{y}_{2,2}, \hat{y}_{2,3}, \hat{y}_{2,4} \big\rangle = \mathit{public}_4(\hat{y}_2)\ \mathtt{in} \\
& \mathtt{let}\ \big\langle \hat{y}_{1,1}, \hat{y}_{1,2}, \hat{y}_{1,3}, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7} \big\rangle = \mathit{public}_7(\hat{y}_1)\ \mathtt{in} \\
& \mathtt{let}\ \hat{y}_{2,V} = \mathit{ver}_{2,4,3,S_1}\big(\hat{y}_2, \mathit{vk}(k_{AS}),\ \mathit{pk}(k_{BE}),\ \hat{y}_{1,5}\big)\ \mathtt{then} \\
& \mathtt{let}\ \hat{y}_{1,V} = \mathit{ver}_{2,8,5,S_2}\big(\hat{y}_1, \mathit{vk}(k_{BS}),\ \mathit{pk}(k_{CE}),\ \mathit{pk}(k_{BE}),\ \mathit{vk}(k_{AS}),\ \hat{y}_{1,5}\big)\ \mathtt{then} \\
& \mathtt{let}\ \langle \hat{y}_{2,4} \rangle = \hat{y}_{2,V}\ \mathtt{in} \\
& \mathtt{let}\ \langle y, \hat{y}_{1,7}, \hat{y}_{1,8} \rangle = \hat{y}_{1,V}\ \mathtt{in} \\
& \mathtt{let}\ y_1 = \mathit{check}(y, \mathit{vk}(k_{BS}))\ \mathtt{then} \\
& \mathtt{let}\ y_2 = \mathit{dec}\big(y_1, k_{CE}\big)\ \mathtt{then} \\
& \mathtt{let}\ \big\langle y_3 \big\rangle = y_2\ \mathtt{in}\ \big(\mathtt{assert}(\mathit{Good}(y_3))\big|Q\big)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Prot}' \mathrel{\widehat{=}} {} & \nu k_{AS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{\mathit{Good}(z)\})). \\
& \nu k_{BE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{\mathit{Good}(z)\}). \\
& \nu k_{BS} : \mathsf{SigKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB}\rangle\{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\})). \\
& \nu k_{CE} : \mathsf{PrivKey}(\langle z : \mathsf{PrivateUnlessB}\rangle\{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\}). \\
& \quad \mathtt{out}\,(\mathit{ch}, \mathit{pk}(k_{BE}))\,.\mathtt{out}\,(\mathit{ch}, \mathit{pk}(k_{CE}))\,.\mathtt{out}\,(\mathit{ch}, \mathit{vk}(k_{AS}))\,.\mathtt{out}\,(\mathit{ch}, \mathit{vk}(k_{BE})) \\
& \quad \big(A\big|B\big|C\big)
\end{aligned}
$$

where S_1 and S_2 are the statements of the zero-knowledge proofs. We have

$$
\begin{aligned}
S_1 = {} & \beta_4 = \mathit{enc}(\alpha_1, \beta_2) \wedge \beta_4 = \mathit{check}^\sharp(\beta_3, \beta_1) \wedge \langle \alpha_2 \rangle = \alpha_1 : \\
& \mathsf{Stm}\Big(\langle y_1 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})), \\
& \quad y_2 : \mathsf{PubKey}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\}), \\
& \quad y_3 : \mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})), \\
& \quad y_4 : \mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})\rangle \\
& \quad \{\exists x_1, x_2 : y_4 = \mathit{enc}(x_1, y_2) \wedge y_4 = \mathit{check}^\sharp(y_3, y_1) \wedge \langle x_2 \rangle = x_1 \wedge \mathit{Good}(x_2)\}\Big) \\
S_2 = {} & \beta_8 = \mathit{check}^\sharp(\beta_5, \beta_4) \wedge \alpha_1 = \mathit{dec}^\sharp(\beta_8, \alpha_2) \wedge \beta_3 = \mathit{pk}(\alpha_2) \\
& \wedge \beta_7 = \mathit{enc}(\alpha_1, \beta_2) \wedge \beta_7 = \mathit{check}^\sharp(\beta_6, \beta_1) : \\
& \mathsf{Stm}\Big(\langle y_1 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\})), \\
& \quad y_2 : \mathsf{PubKey}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\}), \\
& \quad y_3 : \mathsf{PubKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})), \\
& \quad y_4 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})), \\
& \quad y_5 : \mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})), \\
& \quad y_6 : \mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\})), \\
& \quad y_7 : \mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\}), \\
& \quad y_8 : \mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})\rangle \\
& \quad \{\exists x_1, x_2 : y_8 = \mathit{check}^\sharp(y_5, y_4) \wedge x_1 = \mathit{dec}^\sharp(y_8, x_2) \wedge y_3 = \mathit{pk}(x_2) \\
& \quad \wedge y_7 = \mathit{enc}(x_1, y_2) \wedge y_7 = \mathit{check}^\sharp(y_6, y_1)\}\Big)
\end{aligned}
$$

For verifying the first zero-knowledge proof $\hat{y}_2$ we have: $\mathit{vk}(k_{AS})$ is of type $\mathsf{VerKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\}))$ and $\mathit{pk}(k_{BE})$ is of type $\mathsf{PubKey}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})$. From the statement S_1 of the zero-knowledge proof we can conclude that y_4 has type $\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})$ (by the typing rule for *enc* and $\beta_4 = \mathit{enc}(\alpha_1, \beta_2)$). By the typing rule for *check* and $\beta_4 = \mathit{check}^\sharp(\beta_3, \beta_1)$ we can conclude that the y_3 has type $\mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\}))$. Since y_4 has type $\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})$, x_1 is of type $\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\}$. Then by $\langle \alpha_2 \rangle = \alpha_1$, x_2 is of type $\mathsf{PrivateUnlessB}$ and $\mathit{Good}(x_2)$ holds. This justifies the type for the statement of the zero-knowledge proof generated by A. Similar reasoning justifies the type for the statement of the zero-knowledge proofs generated by B: From $\beta_8 = \mathit{check}^\sharp(\beta_5, \beta_4)$, the types of y_5 and y_4 and the typing rule for *check* we obtain the type $\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\})$ for y_8; from $\beta_3 = \mathit{pk}(\alpha_2)$, the type of y_3 and the typing rule for *pk* we have that x_2 has type $\mathsf{PrivKey}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\}))$, and from $\alpha_1 = \mathit{dec}^\sharp(\beta_8, \alpha_2)$, the type of x_2 and the typing rule for *dec* we have that x_1 has type $\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z)\}$. From $\beta_7 = \mathit{enc}(\alpha_1, \beta_2)$, the type for y_2, the typing rule for *enc* and the subtyping rule for tuples we obtain the type $\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\})$ for y_7; from $\beta_7 = \mathit{check}^\sharp(\beta_6, \beta_1)$, the type of y_7 and y_1 and the typing rule for *check* we obtain the type $\mathsf{Signed}(\mathsf{PubEnc}(\langle z : \mathsf{PrivateUnlessB} \rangle \{\mathit{Good}(z) \vee \mathit{Bsaysfalse}\}))$ for y_6.

The strengthened protocol type-checks and is hence safe even if B is compromised. This is because C receives two zero-knowledge proofs: one generated by B ($\hat{y}_1$) and one generated by A and forwarded by B to C ($\hat{y}_2$). From the formula of S_2 we know, that $\exists x_1, x_2 : y_8 = \mathit{check}^\sharp(y_5, y_4) \wedge x_1 = \mathit{dec}^\sharp(y_8, x_2) \wedge y_3 = \mathit{pk}(x_2) \wedge y_7 = \mathit{enc}(x_1, y_2) \wedge y_7 = \mathit{check}^\sharp(y_6, y_1)$ and from S_1 we know, that $\exists x'_1, x'_2 : y'_4 = \mathit{enc}(x'_1, y'_2) \wedge y'_4 = \mathit{check}^\sharp(y'_3, y'_1) \wedge \langle x'_2 \rangle = x'_1 \wedge \mathit{Good}(x'_2)$. We

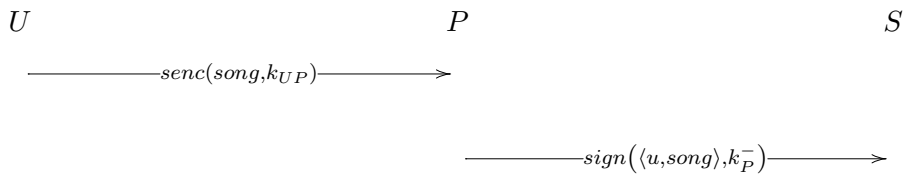
know from the process that $y'_2 = y_3$ (both are placeholders for $pk(k_{BE})$), $y'_1 = y_4$ (placeholders for $vk(k_{AS})$), and $y'_3 = y_5$ (in both verifications we use the same term). Now $y_8 = check^\sharp(y_5, y_4)$ and $y'_4 = check^\sharp(y'_3, y'_1)$ imply that $y'_4 = y_8$. Furthermore $x_1 = dec^\sharp(y_8, x_2) \wedge y_3 = pk(x_2)$ and $y'_4 = enc(x'_1, y'_2)$ imply that $x'_1 = x_1$. From the formula of S_2 we know that y_7 is an encryption of a term $x'_1 = x_1$, x_1 is a tuple consisting of one message x_2 for which $Good(x_2)$ hold, and y_6 is a signature of y_7. Hence the assertion $Good(y_3)$ holds – after C has checked the signature, decrypted the encryption, and split the remaining tuple.

We do not prove that the presented algorithm is type-preserving, in example well-typed processes will be transformed processes which are well-typed as well. Instead we use a type-checker to check whether the transformed process is safe or not. However from our experience and the experiments we have made we are confident that the algorithm preserves types.

5.3 Approach for Symmetric Key Cryptography

As explained in 4 in the case of symmetric key cryptography we have to generate zero-knowledge proofs differently: a zero-knowledge proof proving that some publicly known term is a symmetric encryption of some message with some symmetric key is not sufficient, since both owners of that symmetric key can generate such a zero-knowledge proof. Hence we rely on a public key infrastructure and prove additionally that the sender know the secret key corresponding to the sender's public key.

Let us consider the following protocol from [FGM07]:



A user U wants to download a song from the store S. The user sends a symmetric encryption of the song with the symmetric key for communication between the user U and a proxy P. The proxy P obtains the song the user wants to downlad by decrypting the symmetric encryption. Then the proxy signs a tuple consisting of the users identification u and the song with the proxy's signing key. We assume that the user is registered and that he wants to order the song. The global policy is that whenever a user orders a song and the user is registered then the user may download that song.

$$
\begin{aligned}
U \mathrel{\widehat{=}} {} & \nu song : \mathsf{Un}. \\
& \big(\texttt{assume}\ Order(u, song) | 1 : \mathrm{out}\big(senc\big(\langle song \rangle, k_{UP}\big)\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
P \mathrel{\widehat{=}} {} & 1 : \mathrm{in}\big(ch, x\big). \\
& \texttt{let}\ x_1 = sdec\big(x, k_{UP}\big)\ \texttt{then} \\
& \texttt{let}\ \big\langle x_2 \big\rangle = x_1\ \texttt{in} \\
& 2 : \mathrm{out}\big(sign\big(\big\langle u, x_2 \big\rangle, k_{PS}\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
S \mathrel{\widehat{=}}\ & 2 : \mathrm{in}\big(ch, y\big). \\
& \texttt{let } y_1 = \mathit{check}\big(y, \mathit{vk}(k_{PS})\big) \texttt{ then} \\
& \texttt{let } \big\langle y_2, y_3 \big\rangle = y_1 \texttt{ in} \\
& \texttt{assert } \mathit{CanDownload}(y_2, y_3).
\end{aligned}
$$

$$
\mathit{policy} \mathrel{\widehat{=}}\ \texttt{assume } \forall u, s : \mathit{Order}(u, s) \wedge \mathit{Registered}(u) \Rightarrow \mathit{CanDownload}(u, s).
$$

$$
\begin{aligned}
\mathit{Prot} \mathrel{\widehat{=}}\ & \nu k_{UP} : \mathsf{SymKey}(\langle s : \mathsf{Un} \rangle \{\mathit{Order}(u, s)\}). \\
& \nu k_{PS} : \mathsf{SigKey}(\langle u : \mathsf{Un}, s : \mathsf{Un} \rangle). \\
& p1 : \mathrm{out}\big(ch, \mathit{vk}(k_{PS})\big). \\
& \big(\texttt{assume } \mathit{Registered}(u) \,\big|\, \mathit{policy} \,\big|U\big|P\big|S\big).
\end{aligned}
$$

For the sake of readability we have omitted the decryption key k_{PE} and encryption key $pk(k_{PE})$ of P.

$$
\begin{aligned}
U \mathrel{\widehat{=}}\ & \nu \mathit{song} : \mathsf{Un} \\
& \big(\texttt{assume } \mathit{Order}(u, \mathit{song}) | 1 : \mathrm{out}\big(ch, \big\langle zk_{4,2,S_1} \\
& \quad \big(k_{UP}, k_{UE}, \langle \mathit{song} \rangle, \mathit{song}; pk(k_{UE}), \mathit{senc}\big(\langle \mathit{song} \rangle, k_{UP}\big)\big)\big\rangle\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
P \mathrel{\widehat{=}}\ & 1 : \mathrm{in}\big(ch, \hat{\hat{x}}\big). \\
& \texttt{let } \big\langle \hat{x}_1 \big\rangle = \hat{\hat{x}} \texttt{ in} \\
& \texttt{let } \hat{x}_{1,P} = \mathit{public}_2(\hat{x}_1) \texttt{ then} \\
& \texttt{let } \big\langle \hat{x}_{1,1}, \hat{x}_{1,2} \big\rangle = \hat{x}_{1,P} \texttt{ in} \\
& \texttt{let } \hat{x}_{1,V} = \mathit{ver}_{4,2,1,S_1}\big(\hat{x}_1, pk(k_{UE})\big) \texttt{ then} \\
& \texttt{let } \big\langle x \big\rangle = \hat{x}_{1,V} \texttt{ in} \\
& \texttt{let } x_1 = \mathit{sdec}\big(x, k_{UP}\big) \texttt{ then} \\
& \texttt{let } \big\langle x_2 \big\rangle = x_1 \texttt{ in} \\
& 2 : \mathrm{out}\big(ch, \big\langle zk_{1,7,S_2} \\
& \quad \big(k_{UP}; \mathit{vk}(k_{PS}), x, \mathit{sign}\big(\langle u, x_2 \rangle, k_{PS}\big), \langle u, x_2 \rangle, \\
& \quad u, x_2, x_1\big), \hat{x}_1 \big\rangle\big)
\end{aligned}
$$

$$
\begin{aligned}
S \mathrel{\widehat{=}}\ & 2 : \mathrm{in}\big(ch, \hat{\hat{y}}\big). \\
& \texttt{let } \big\langle \hat{y}_1, \hat{y}_2 \big\rangle = \hat{\hat{y}} \texttt{ in} \\
& \texttt{let } \hat{y}_{2,P} = \mathit{public}_2(\hat{y}_2) \texttt{ then} \\
& \texttt{let } \big\langle \hat{y}_{2,1}, \hat{y}_{2,2} \big\rangle = \hat{y}_{2,P} \texttt{ in} \\
& \texttt{let } \hat{y}_{1,P} = \mathit{public}_7(\hat{y}_1) \texttt{ then} \\
& \texttt{let } \big\langle \hat{y}_{1,1}, \hat{y}_{1,2}, \hat{y}_{1,3}, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7} \big\rangle = \hat{y}_{1,P} \texttt{ in} \\
& \texttt{let } \hat{y}_{2,V} = \mathit{ver}_{4,2,2,S_1}\big(\hat{y}_2, pk(k_{UE}), \hat{y}_{1,2}\big) \texttt{ then} \\
& \texttt{let } \hat{y}_{1,V} = \mathit{ver}_{1,7,2,S_2}\big(\hat{y}_1, \mathit{vk}(k_{PS}), \hat{y}_{1,2}\big) \texttt{ then}
\end{aligned}
$$

$$
\begin{array}{l}
\texttt{let } \langle \varepsilon \rangle = \hat{y}_{2,V} \texttt{ in} \\
\texttt{let } \langle y, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7} \rangle = \hat{y}_{1,V} \texttt{ in} \\
\texttt{let } y_1 = \mathit{check}\big(y, \mathit{vk}(k_{PS})\big) \texttt{ then} \\
\texttt{let } \langle y_2, y_3 \rangle = y_1 \texttt{ in} \\
\texttt{assert } \mathit{CanDownload}(y_2, y_3)
\end{array}
$$

$$
\mathit{policy} \mathrel{\widehat{=}} \texttt{assume } \forall u, s : \mathit{Order}(u, s) \wedge \mathit{Registered}(u) \Rightarrow \mathit{CanDownload}(u, s).
$$

$$
\begin{array}{rl}
\mathit{Prot}' \mathrel{\widehat{=}} & \nu k_{UP} : \mathsf{SymKey}(\langle s : \mathsf{Un} \rangle \{\mathit{Order}(u, s)\}). \\
& \nu k_{UE} : \mathsf{PrivKey}(\mathsf{Un}). \\
& \nu k_{PS} : \mathsf{SigKey}(\langle u : \mathsf{Un}, s : \mathsf{Un} \rangle). \\
& p1 : \mathrm{out}\big(\mathit{ch}, \mathit{pk}(k_{UE})\big). \\
& p2 : \mathrm{out}\big(\mathit{ch}, \mathit{vk}(k_{PS})\big). \\
& \big(\texttt{assume } \mathit{Registered}(u) \,\big|\, \mathit{policy} \,\big|U\big|P\big|S\big).
\end{array}
$$

The statements S_1 and S_2 of the zero-knowledge proofs are

$$
\begin{array}{l}
S_1 = \alpha_3 = \langle \alpha_4 \rangle \wedge \beta_2 = \mathit{senc}(\alpha_3, \alpha_1) \wedge \beta_1 = \mathit{pk}(\alpha_2) : \\
\quad \mathsf{Stm}\Big(\big\langle y_1 : \mathsf{PubKey}(\mathsf{Un}), \\
\qquad y_2 : \mathsf{SymEnc}(\langle s : \mathsf{Un} \rangle \{\mathit{Order}(u, s)\}) \\
\qquad \big\{\exists x_1, x_2, x_3, x_4 : x_3 = \langle x_4 \rangle \wedge y_2 = \mathit{senc}(x_3, x_1) \wedge y_1 = \mathit{pk}(x_2) \wedge \mathit{Order}(u, x_4)\big\}\Big) \\
S_2 = \beta_7 = \mathit{sdec}(\beta_2, \alpha_1) \wedge \beta_7 = \langle \beta_6 \rangle \wedge \beta_4 = \langle \beta_5, \beta_6 \rangle \wedge \beta_4 = \mathit{check}(\beta_3, \beta_1) : \\
\quad \mathsf{Stm}\Big(\big\langle y_1 : \mathsf{VerKey}(\langle u : \mathsf{Un}, s : \mathsf{Un} \rangle), \\
\qquad y_2 : \mathsf{SymEnc}(\langle s : \mathsf{Un} \rangle \{\mathit{Order}(u, s)\}), \\
\qquad y_3 : \mathsf{Signed}(\langle u : \mathsf{Un}, s : \mathsf{Un} \rangle), \\
\qquad y_4 : \langle u : \mathsf{Un}, s : \mathsf{Un} \rangle, \\
\qquad y_5 : \mathsf{Un}, \\
\qquad y_6 : \mathsf{Un}, \\
\qquad y_7 : \langle s : \mathsf{Un} \rangle \{\mathit{Order}(u, s)\} \\
\qquad \big\{\exists x_1 : y_7 = \mathit{sdec}(y_2, x_1) \wedge y_7 = \langle y_6 \rangle \wedge y_4 = \langle y_5, y_6 \rangle \wedge y_4 = \mathit{check}(y_3, y_1)\big\}\Big)
\end{array}
$$

5.4 Implementation

We implemented the presented algorithm. Our tool is written in Moscow ML, simple dialect of Standard ML and comprises approximately 1600 lines of code. It is available at [imp].

The implementation consist of three parts: the algorithm described above, a compiler for compiling protocols into LaTeX-code, and a compiler for compiling protocols into the format of the type-checker. The implementation applies our approach to a given process and generates

two output files: one file containing the original and the strengthened protocol in LaTeX-format, and one file containing the modified process in the input format for the type-checker.

Chapter 6

Conclusion

6.1 Summary

We have presented an automated technique to strengthen protocols in order to make them resistant to principal compromise. In our approach we transform protocols by adding zero-knowledge proofs. By using a zero-knowledge proof a principal is able to prove to others that he has followed the protocol but at the same time he does not have to reveal any of his secrets. Furthermore zero-knowledge proofs are forwarded by principals to prove the correct behaviour of all participants involved in the protocol run so far. Our approach assumes a public key infrastructure. It is applicable to most protocols and offers a solution to strengthen them against compromised principals. We have implemented our approach in order to automatically strengthen protocols, the implementation is available at [imp].

We represent protocols in the applied π-calculus. For proving the safety of the transformed protocols we use a type system. For the type system the restricted values of the process are annotated by types. Authorisation policies are added to the process as well. With the results of [BHM08a] we are able type-check zero-knowledge proofs in the applied π-calculus and we are able to statically analyse the safety of a protocol with respect to an authorisation policy. With the type-checker available at [BHM08b] we are able to check the protocols generated by our algorithm for safety in an automated way. We have applied our algorithm to two example and type-checked the strengthened protocols generated by our algorithm.

6.2 Future Work

One topic for future work is to generate efficient protocols with zero-knowledge proofs. In the approach we presented one zero-knowledge proof is generated for each message sent by a participant in the protocol. Further zero-knowledge proofs are forwarded by participants. This means that not only a participant has to verify one zero-knowledge proof for the message in the original protocols, but he has also to verify all forwarded zero-knowledge proofs. For large protocols where lots of messages are exchanged between the participants our algorithm will generate protocols which are much larger than the original protocols. Those protocols are not as efficient as they could be.

Since we rely on a type system to prove the safety of a protocol with respect to some authorisation policy an important topic is to prove the preservation of typing. Proving that in case of no compromised participants our algorithm generates well-typed processes and that in case of compromised participants the processes are well-typed as well shows that our technique indeed strengthens protocols, if the process representing the original protocol is well-typed.

Bibliography

[AB02] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *Proc. 29th Symposium on Principles of Programming Languages (POPL)*, pages 33–44. ACM Press, 2002. 2

[BBF+08] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 17–32. IEEE Computer Society Press, 2008. 2.3

[BHM08a] M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 357–370. ACM, 2008. 2, 2.3, 6.1

[BHM08b] M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. Implementation available at `http://www.infsec.cs.uni-sb.de/projects/zk-typechecker`, 2008. 2.3, 6.1

[BMU08] M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 202–215. IEEE Computer Society Press, 2008. 2.2

[FGM07] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 31–45. IEEE Computer Society Press, 2007. 2.3, 3, 5.3

[Gol01] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001. 1.2

[imp] Implementation available at `http://www.mgrochulla.de/index.html`. 5.4, 6.1