Achieving Security Despite Compromise Using Zero-knowledge

Michael Backes[1,2], Martin P. Grochulla[1], Cătălin Hriţcu[1], and Matteo Maffei[1]
[1]Saarland University, Saarbrücken, Germany
[2]MPI-SWS

April 26, 2009

Abstract

One of the important challenges when designing and analyzing cryptographic protocols is the enforcement of security properties in the presence of compromised participants. This paper presents a general technique for strengthening cryptographic protocols in order to satisfy authorization policies despite participant compromise. The central idea is to automatically transform the original cryptographic protocols by adding non-interactive zero-knowledge proofs. Each participant proves that the messages sent to the other participants are generated in accordance to the protocol. The zero-knowledge proofs are forwarded to ensure the correct behavior of all participants involved in the protocol, without revealing any secret data. We use an enhanced type system for zero-knowledge to verify that the transformed protocols conform to their authorization policy even if some participants are compromised. Finally, we developed a tool that automatically generates ML implementations of protocols based on zero-knowledge proofs. The protocol transformation, the verification, and the generation of protocol implementations are fully automated.

1 Introduction

A central challenge in the design of security protocols for modern applications is devising protocols that satisfy strong security properties. Ideally, the designer should only have to consider restricted security threats (e.g., honest-but-curious participants); automated tools should then strengthen the original protocols so that they withstand stronger attacks (e.g., malicious participants). In this paper, we automatically strengthen protocols so that they withstand attacks even in the presence of compromised participants. The notion of "security despite compromise" [14] captures the intuition that *an invalid authorization decision by an uncompromised participant should only arise if participants on which the decision logically depends are compromised.* The impact of participant compromise should be thus apparent from the policy, without having to study the details of the protocol.

Zero-knowledge proofs[1] [17, 15] are a natural candidate for strengthening protocols so that they achieve security despite compromise since they allow the participants to prove that they correctly generated the messages they send, without revealing any secret data. Zero-knowledge proofs go beyond the traditional understanding of cryptography that only ensures secrecy and authenticity of a communication. This primitive's unique security features, combined with the recent advent of efficient cryptographic implementations of zero-knowledge proofs for special

[1]A zero-knowledge proof combines two seemingly contradictory properties. First, it is a proof of a statement that cannot be forged, i.e., it is impossible, or at least computationally infeasible, to produce a zero-knowledge proof of a wrong statement. Second, a zero-knowledge proof does not reveal any information besides the bare fact that the statement is valid.

classes of problems, have paved the way for their deployment in modern applications. For instance, zero-knowledge proofs can guarantee authentication yet preserve the anonymity of protocol participants, as in the Civitas electronic voting protocol [10], or the Pseudo Trust protocol [19], or they can prove the reception of a certificate from a trusted server without revealing the actual content, as in the Direct Anonymous Attestation (DAA) protocol [9]. Although highly desirable, there is no computer-aided support for using zero-knowledge proofs in the design of security protocols: in the aforementioned applications, these primitives were used in the design by leading security researchers, and still security vulnerabilities in some of those protocols were subsequently discovered [24, 5].

1.1 Our Contributions

We present a general technique for strengthening security protocols in order to satisfy authorization policies despite participant compromise, as well as an enhanced type system for verifying that the strengthened protocols are indeed in conformance with their policy even if some participants are compromised.

The central idea is to automatically transform the original security protocols by including non-interactive zero-knowledge proofs. Each participant proves that the messages sent to the other participants are generated in accordance to the protocol. The zero-knowledge proofs are forwarded to ensure the correct behaviour of all participants involved in the protocol, without revealing any secret data[2]. Our approach is general and can strengthen any protocol based on public-key encryption, digital signatures, hashes, and symmetric encryption. Moreover, the transformation automatically derives proper type annotations for the strengthened protocol provided that the original protocol is augmented with type annotations. In general, this frees protocol designers from inspecting the strengthened protocol to conduct a successful security analysis, and only requires them to properly design the original, simpler protocol.

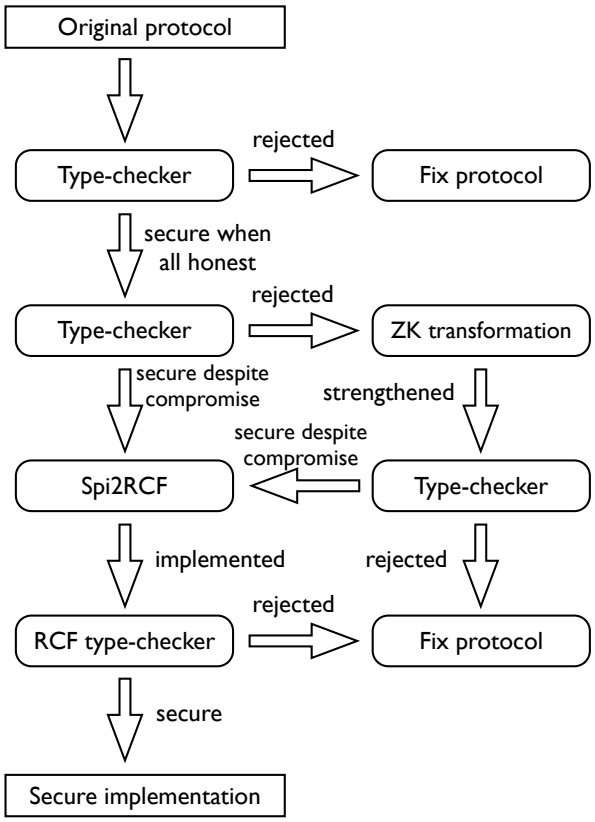
The type system extends our previous type system for zero-knowledge [4] to the setting of participant compromise. In particular, instead of relying on unconditionally secure types, we give a precise characterization of when a type is compromised in the form of a logical formula. We use refinement types that contain such logical formulas together with union types to express type information that is conditioned by a participant not being compromised. We use intersection and union types to infer very precise type information about the secret witnesses of zero-knowledge proofs. These improvements lead to a much more fine-grained analysis that can deal with compromised participants, but also increase the overall precision and expressivity of the type system.

Finally, we developed a tool that automatically implements protocols based on zero-knowledge proofs in RCF [7], a core calculus of ML. These implementations can be linked either against a concrete cryptographic library, or against a symbolic one where the cryptographic primitives are considered fully reliable building blocks. We use another type-checker to verify the security of the generated RCF implementation with respect to the symbolic cryptographic library. The overall workflow for generating secure ML implementations is depicted in Figure 1.

In general, this translation validation approach has the advantage that even if we apply drastic optimizations, or completely reimplement the transformation, we do not need to redo any proofs. While a direct proof of correctness of the translation would provide stronger guarantees for any generated protocol implementation without relying on any validator, this far from trivial proof would need to be redone every time the transformation is changed, e.g., when

[2]The compromised participants can leak any data they receive so, while our transformation preserves secrecy in the uncompromised setting, secrecy cannot be enforced when some participants are compromised. What our transformation enforces in case of partial compromise is conformance to the authorization policy.

Figure 1 Workflow for Generating Secure ML Implementations



applying any optimization. We believe that the added benefits of having such a direct proof are greatly outweighed by the amount of work necessary to create it and keep it up-to-date as the transformation evolves.

1.2 Related Work

Security despite compromised participants was introduced by Fournet et al. [14]. The authors observe that in order to fix a protocol that is not secure despite compromise one can either weaken the authorization policy to document all dependencies between participants or correct the specification of the protocol in order to avoid such dependencies. We take the latter approach. Our current work provides a systematic technique for removing dependencies between participants and achieving security despite compromise.

The closest work to ours is by Corin et al. [11], who automatically compile high-level specifications of multi-party protocols based on *session types* (and not involving cryptography) into cryptographic implementations that are secure despite participant compromise. The generated cryptographic implementations are efficient and are guaranteed to adhere to the original specification even if some of the participants are compromised. The original transformation did not consider secrecy (all messages were assumed to be public) or payload binding (the generated protocols were susceptible to message substitution attacks), but these limitations have recently been addressed by the authors [8]. While the original transformation was proven correct [11], the more recent one [8] relies on a type-checker [7] for verifying that each of the generated cryptographic implementations is secure.

The main difference with respect to [11, 8] is that our translation takes a cryptographic protocol as input, not a higher-level specification of a multi-party protocol. This is conceptually

different and has the advantage of providing an effective way to strengthen *existing* cryptographic protocols. Furthermore, our approach may in principle allow the original protocol and the strengthened one to *interoperate*, assuming the former has a flexible enough message format. On the other hand, the transformation proposed in [8] directly returns executable protocol implementations, while the implementations we generate use functions for creating and verifying zero-knowledge proofs that currently have to be implemented by hand[3].

The idea of *strengthening security protocols with zero-knowledge proofs* was used by Goldreich et al. to transform secure multi-party computation protocols that are secure against honest-but-curious adversaries into protocols secure against compromised participants [16, 15]. They work in the setting of computational cryptography, while we work in the symbolic one. Another important difference is that their solution requires broadcast communication for the generated protocol, while we only need point-to-point communication and preserve the message flow of the original protocol.

Katz and Yung [18], and later Cortier et al. [12] proposed *transformations* from protocols secure against passive, eavesdropping attackers to protocols secure against active attacks. Bellare et al. transform a protocol that is secure when the communication between parties is authenticated into one that is secure even if this assumption is not satisfied [6]. Datta et al. [13] propose a methodology for modular development of protocols where security properties are added to a protocol through generic transformations. Abadi et al. [2] give a compiler for protocols using secure channels into implementations that use cryptography.

Translation validation, as introduced by Pnueli et al. [22], is an accepted technique for detecting compiler bugs and preventing incorrect code from being run. Since the validator is usually developed independently from the compiler and uses very different algorithms, translation validation significantly increases the confidence of the user in the compilation process. The validator can use a variety of techniques, from program analysis and symbolic execution [20, 26], to model checking and theorem proving [22, 27]. In the current paper we use a type-checker to validate the results of our translation.

Type systems are particularly salient tools to statically and automatically enforce authorization policies on abstract protocol specifications [14] and on concrete protocol implementations [7]. Type systems require little human effort and provide security proofs for an unbounded number of protocol executions. Furthermore, the analysis is modular, compositional, and usually guaranteed to terminate. Fournet et al. [14] proposed a type system that can be used to analyze conformance with authorization policies in the presence of compromised participants. Our previous type system for zero-knowledge [4] extends the one by Fournet et al. to handle zero-knowledge; however, it crucially relies on unconditionally secure types, which makes it unsuitable for dealing with security despite compromise. In the current work we remove this limitation by using a logical characterization of when a type is compromised, and by extending the type system with union and intersection types [21].

1.3 Outline

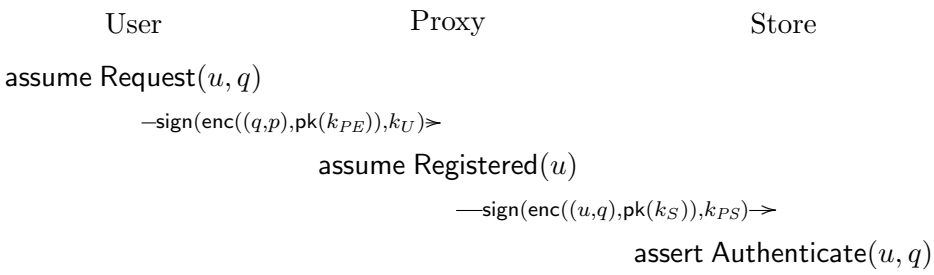
Section 2 uses an illustrative example to explain our technique. Section 3 describes the generic transformation and applies it to this example. Our enhanced type system for zero-knowledge is presented in Section 4. Section 2.5 describes two methods that can be used to handle symmetric encryption. Section 5 presents our tool for automatically generating ML implementations of protocols based on zero-knowledge proofs. Section 6 concludes and provides directions for future work. Appendix A describes the syntax and semantics of the calculus we use to specify both

[3] A symbolic version of these functions is still generated automatically and can be used for verification and debugging.

the original and the transformed protocols. Appendix B presents our enhanced type system for zero-knowledge. Appendix C describes the transformation algorithm in full detail. Finally, Appendix D lists the complete code of the example using asymmetric cryptography, before and after the transformation. The implementation of the transformation [3] and that of the type-checker [4] are available online.

2 Illustrative Example

This section reviews authorization policies, introduces the problem of participant compromise, and illustrates the fundamental ideas of our protocol transformation. As a running example, we consider a simple protocol involving a user, a proxy, and an online store. This protocol is inspired by a protocol proposed by Fournet et al. [14]. The main difference is that we use asymmetric cryptography in the first message, while the original protocol uses symmetric encryption. In the long version of this paper [3] we discuss how our transformation handles symmetric encryption and apply these ideas to the original protocol by Fournet et al. [14].



In this protocol, the user u sends a query q and a password p to the proxy. This data is first encrypted with the public key $\mathsf{pk}(k_{PE})$ of the proxy and then signed with u's signing key k_U. The proxy verifies the signature and decrypts the message, checks that the password is correct, and sends the user's name and the query to the online store. This data is first encrypted with the public key $\mathsf{pk}(k_S)$ of the store and then signed with the signing key k_{PS} of the proxy.

2.1 Authorization Policies and Safety

As proposed in [14, 4], we model the security goal of this protocol as an authorization policy. The fundamental idea is to decorate security-related protocol events by predicates and to express the security property of interest as a logical formula over such predicates. Predicates are split into *assumptions* and *assertions*, and we say that a protocol is *safe* if and only if in all protocol executions each assertion is entailed by the assumptions made earlier in the execution and by the authorization policy. If a protocol is safe when executed in parallel with an arbitrary attacker, then we say that the protocol is *robustly safe*. The protocol above is decorated with two assumptions and one assertion: the assumption $\mathsf{Request}(u, q)$ states that the user u is willing to send a query q, the assumption $\mathsf{Registered}(u)$ states that the user u is registered in the system, and the assertion $\mathsf{Authenticate}(u, q)$ states that the online store authenticates the query q sent by user u.

The goal of this protocol is that the online store authenticates the query q as coming from u only if u has indeed sent query q and u is registered in the system. This is formulated as the following authorization policy:

$$\forall u, q.\mathsf{Request}(u, q) \wedge \mathsf{Registered}(u) \Rightarrow \mathsf{Authenticate}(u, q) \tag{1}$$

We want $\mathsf{Authenticate}(u, q)$ to be entailed in all executions of the protocol that reach the assert. Since the only way to obtain this predicate is by using rule (1), which only applies if the assertions $\mathsf{Request}(u, q)$ and $\mathsf{Registered}(u)$ have been previously executed, this policy enforces that the store authenticates q only if a registered user requested q. Please note that all authorization decisions that are not explicitly allowed by the policy are disallowed.

2.2 Security Despite Compromised Participants

If all the participants are honest then the protocol above is robustly safe with respect to authorization policy (1). The intuitive reason is that: (i) the messages exchanged in the protocol cannot be forged by the attacker, since they are digitally signed; (ii) the user sends the first message to the proxy only after assuming $\mathsf{Request}(u, q)$; and (iii) the proxy sends the second message to the store only after receiving the first message and assuming $\mathsf{Registered}(u)$.

We now investigate what happens if some of the participants are compromised. We model the compromise of a participant v by (a) revealing all her secrets to the attacker; (b) removing the code of v, since it is controlled by the attacker; and (c) introducing the assumption $\mathsf{Compromised}(v)$. Since the attacker can impersonate v and send messages on her behalf without assuming any predicate, we make the convention that for each assumption F in the code of v we have a rule of the form $\mathsf{Compromised}(v) \Rightarrow F$ in the authorization policy. In our example we have two such additional rules:

$$\mathsf{Compromised}(user) \Rightarrow \forall q.\mathsf{Request}(u, q) \tag{2}$$

$$\mathsf{Compromised}(proxy) \Rightarrow \forall u.\mathsf{Registered}(u). \tag{3}$$

With these additional rules the protocol is robustly safe even when the user is compromised, since the only way for the attacker to interact with the honest proxy is to follow the protocol and, by impersonating the user, to authenticate a query with a valid password. This is, however, harmless since the attacker is just following the protocol. The protocol is vacuously safe if the store is compromised, since no assertion has to be justified; moreover, it is safe if both the proxy and the user are compromised, since in this case the two hypotheses of rule (1) are always entailed.

Therefore the only interesting case is when the proxy is compromised and the other participants are not. In this case, we introduce the assumption $\mathsf{Compromised}(proxy)$, which by (3) implies that $\forall u.\mathsf{Registered}(u)$. Still, the compromised proxy might send a message to the store without having received any query from the user, which would lead to an $\mathsf{Authenticate}(u, q)$ assertion that is not logically entailed by the preceding assumptions. Notice that the only way to infer $\mathsf{Authenticate}(u, q)$ is using rule (1), and this requires that both $\mathsf{Request}(u, q)$ and $\mathsf{Registered}(u)$ hold. However, since the user did not issue a request, the $\mathsf{Request}(u, q)$ predicate is not entailed in the system.

As suggested in [14], we could document the attack by weakening the authorization policy. This could be achieved by introducing a new rule stating that if the proxy is compromised, then $\forall u, q.\ \mathsf{Request}(u, q)$ holds. In this paper we take a different approach and, instead of weakening the authorization policy and accepting the attack, we propose a general methodology to strengthen any protocol so that such attacks are prevented.

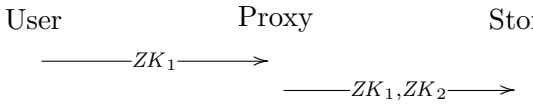
2.3 Symbolic Representation of Zero-knowledge

Before illustrating our approach, we briefly recap the technique introduced in [5] to symbolically represent zero-knowledge proofs. Zero-knowledge proofs are expressed as terms of the form

$\mathsf{zk}_S(\widetilde{M};\widetilde{N})$[4]. The statement S of the zero-knowledge proof is a Boolean formula built over cryptographic operations and the place-holders α_i and β_j that refer to the terms M_i and N_j, respectively. The terms M_i form the *private component* of the proof and will never be revealed, while the terms N_i form the *public component* of the proof and are revealed to the verifier. The verification of a zero-knowledge proof succeeds if and only if the statement obtained after the instantiation of the place-holders, $S\{\widetilde{M}/\widetilde{\alpha}\}\{\widetilde{N}/\widetilde{\beta}\}$, holds true. For instance, $\mathsf{zk}_{\mathsf{check}(\alpha_1,\beta_1)\rightsquigarrow\alpha_2}(\mathsf{sign}(m,k),m;\mathsf{vk}(k))$ is a zero-knowledge proof showing the knowledge of a signature that can be successfully checked with the verification key $\mathsf{vk}(k)$. Notice that the proof reveals neither the signature $\mathsf{sign}(m,k)$ nor the message m. We refer the interested reader to Appendix A.3 for more detail.

2.4 Strengthening the Protocol

The central idea of our technique is to replace each message exchanged in the protocol with a non-interactive zero-knowledge proof showing that the message has been correctly generated. Additionally, zero-knowledge proofs are forwarded by each participant in order to allow the others to independently check that all the participants have followed the protocol. For instance, the protocol considered before is transformed as follows:



$$S_1 \triangleq \mathsf{enc}((\alpha_1,\alpha_2),\beta_2) = \beta_4 \wedge \mathsf{check}(\beta_3,\beta_1) \rightsquigarrow \beta_4$$

$$ZK_1 \triangleq \mathsf{zk}_{S_1}\big(\overbrace{q,p}^{\alpha_1,\alpha_2};\overbrace{\mathsf{vk}(k_U)}^{\beta_1},\overbrace{\mathsf{pk}(k_{PE})}^{\beta_2},\overbrace{\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big)}^{\beta_3},\overbrace{\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)}^{\beta_4}\big)$$

$$S_2 \triangleq \mathsf{check}(\beta_5,\beta_4) \rightsquigarrow \beta_9 \wedge \mathsf{dec}(\beta_9,\alpha_3) \rightsquigarrow (\alpha_1,\alpha_2) \wedge \beta_3 = \mathsf{pk}(\alpha_3) \wedge \beta_7 = \mathsf{enc}((\beta_8,\alpha_1),\beta_2) \wedge \mathsf{check}(\beta_6,\beta_1) \rightsquigarrow \beta_7$$

$$ZK_2 \triangleq \mathsf{zk}_{S_2}\big(\overbrace{q,p,k_{PE}}^{\alpha_1,\alpha_2,\alpha_3};\overbrace{\mathsf{vk}(k_{PS})}^{\beta_1},\overbrace{\mathsf{pk}(k_S)}^{\beta_2},\overbrace{\mathsf{pk}(k_{PE})}^{\beta_3},\overbrace{\mathsf{vk}(k_U)}^{\beta_4},\overbrace{\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big)}^{\beta_5},$$
$$\overbrace{\mathsf{sign}\big(\mathsf{enc}\big((u,q),\mathsf{pk}(k_S)\big),k_{PS}\big)}^{\beta_6},\overbrace{\mathsf{enc}\big((u,q),\mathsf{pk}(k_S)\big)}^{\beta_7},\overbrace{u}^{\beta_8},\overbrace{\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)}^{\beta_9}\big)$$

The first zero-knowledge proof states that the message $\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big)$ sent by the user complies with the protocol specification: the verification of this message with the user's verification key succeeds ($\mathsf{check}(\beta_3,\beta_1) \rightsquigarrow \beta_4$) and the result is the encryption of the query and the password with the proxy's encryption key ($\mathsf{enc}((\alpha_1,\alpha_2),\beta_2) = \beta_4$). We model proofs of knowledge, so the user proves to know the secret query α_1 and the secret password α_2, not just that they exist.

The public component contains only messages that were public in the original protocol. The query and the password are placed in the private component since they were encrypted in the original protocol and could be secrets[5]. Furthermore, notice that the statement S_1 simply describes the operations performed by the user, except for the signature generation which is replaced by the signature verification (this is necessary to preserve the secrecy of the signing key).

[4]Here and throughout the paper, we write $\widetilde{M}$ to denote a sequence of terms $M_1,\ldots,M_n$.

[5]We need to ensure that no secret messages are leaked by the transformation, at least in case all participants are honest.

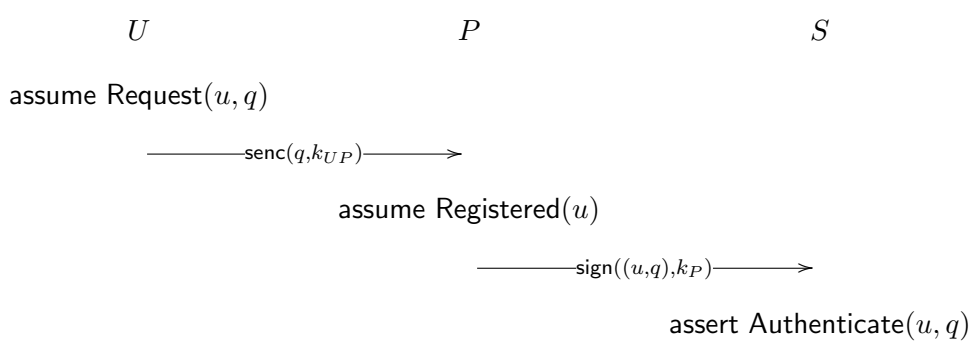
In general, the statement of the generated zero-knowledge proof is computed as the conjunction of the individual operations performed to produce the output message.

The second zero-knowledge proof states that the message β_5 received from the user complies with the protocol, i.e., it is the signature ($\mathsf{check}(\beta_5, \beta_4) \rightsquigarrow \beta_9$)) of an encryption of two secret terms α_1 and α_2 ($\mathsf{dec}(\beta_9, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2)$). The zero-knowledge proof additionally ensures that the message β_6 sent by the proxy is the signature ($\mathsf{check}(\beta_6, \beta_1) \rightsquigarrow \beta_7$) of an encryption of the user's name and the query α_1 received from the user ($\beta_7 = \mathsf{enc}((\beta_8, \alpha_1), \beta_2)$). The statement S_2 guarantees that the query α_1 signed by the user is the same as the one signed by the proxy. Also notice that the proof does not reveal the secret password α_2 received from the user.

The resulting protocol is secure despite compromise, since a compromised proxy can no longer cheat the store by pretending to have received a query from the user. The query will be authenticated only if the store can verify the two zero-knowledge proofs sent by the proxy, and the semantics of these proofs ensures that the proxy is able to generate a valid proof only it has previously received the query from the user. More precisely, the protocol is robustly safe with respect to the authorization policy (1) since (i) the signature of the user cannot be forged by the attacker (we recall that the user is not compromised), (ii) the user outputs the signature only after assuming $\mathsf{Request}(u, q)$; and (iii) the proxy (compromised or not) can generate the second zero-knowledge proof only if it has previously received a signed query from the user. If we compare these properties of the protocol with the ones ensuring the robust safety of the original protocol, which are listed at the beginning of Section 2.2, we can immediately see that the we no longer rely on the integrity of the proxy but just on the cryptographic semantics of zero-knowledge proofs.

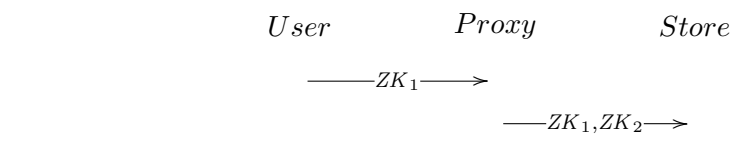
2.5 Symmetric Encryption

Our technique can be easily extended to deal with symmetric key cryptography. As an example, we choose the protocol from [14].



The main difference from the protocol considered so far is that the first message is encrypted with a symmetric-key and the second message is signed but not encrypted. The authorization policy for this protocol is the same as the one for the original protocol, and also in this case the authorization policy is satisfied if all participants are honest, but it is violated if the proxy is compromised. It is not obvious how to strengthen the protocol in order to enforce the authorization policy in the case of participant compromise. The problem is that the proxy cannot reveal the secret key k_{UP} shared with the user, and proving to the store that the query was encrypted with some secret key does not add anything to the security of the protocol, since the ciphertext might have been generated by the proxy as well. In other words, we need a way to guarantee the *authenticity* of the ciphertext. We propose two different techniques to achieve

that. A first solution is to assume a public-key infrastructure and to let the user prove the knowledge of her private key. The protocol obtained after the transformation is as follows:



$$
\begin{array}{lcl}
S_1 & \triangleq & \beta_2 = \mathsf{senc}(\alpha_3, \alpha_1) \wedge \beta_1 = \mathsf{pk}(\alpha_2) : \\
ZK_1 & \triangleq & \mathsf{zk}_{S_1}\big(\overbrace{k_{UP}, k_U, q}^{\alpha_1,\alpha_2,\alpha_3}; \overbrace{\mathsf{pk}(k_U)}^{\beta_1}, \overbrace{\mathsf{senc}(q, k_{UP})}^{\beta_2}\big) \\
S_2 & \triangleq & \mathsf{sdec}(\beta_2, \alpha_1) \rightsquigarrow \beta_5 \wedge \mathsf{ver}(\beta_3, \beta_1) \rightsquigarrow (\beta_4, \beta_5) \\
ZK_2 & \triangleq & \mathsf{zk}_{S_2}\big(\overbrace{k_{UP}}^{\alpha_1}; \overbrace{\mathsf{vk}(k_P)}^{\beta_1}, \overbrace{\mathsf{senc}(q, k_{UP})}^{\beta_2}, \overbrace{\mathsf{sign}((u, q), k_P)}^{\beta_3}, \overbrace{u}^{\beta_4}, \overbrace{q}^{\beta_5}\big)
\end{array}
$$

The first zero-knowledge proof ensures that the prover knows the private part of the user's key-pair and binds this proof of knowledge to the ciphertext. This binding guarantees the *authenticity* of the ciphertext. Note that we model non-malleable zero-knowledge proofs, i.e., it is impossible to change the ciphertext in the proof without redoing the proof from scratch, which requires the knowledge of the user's private key. The second zero-knowledge proof ensures that the query is the same as the one encrypted in the ciphertext received from the user. This can be checked by the store by verifying the two zero-knowledge proofs and checking that the symmetric ciphertext in the public component of ZK_1 matches the one in the public component of ZK_2. The protocol above is robustly safe with respect to the authorization policy (1) since (*i*) the first zero-knowledge proof cannot be forged by the attacker (we recall that the user is not compromised), (*ii*) the user u outputs this proof only after assuming $\mathsf{Request}(u, q)$; and (*iii*) the store accepts the authentication request from the user only if the proxy can prove that the query is obtained by the decryption of a ciphertext (ZK_2) *and* this ciphertext is bound to the proof of knowledge of u's private key (ZK_1).

The first zero-knowledge proof guarantees the authenticity of the query but clearly reveals the identity of the user. In order to guarantee authenticity without breaking the anonymity of the user, we adopt a technique proposed in [19] to achieve authentication in anonymous peer-to-peer protocols. The basic idea is to associate to each user a public pseudonym $\mathsf{hash}(n)$, where n is a secret known only to the user. This pseudonym replaces the user's identifier in the protocol. In order to authenticate a message, the user has just to prove the knowledge of the secret n. Our transformation supports this technique as well. The first zero-knowledge proof is of the following form:

$$
\begin{array}{lcl}
S_1 & \triangleq & \beta_2 = \mathsf{senc}(\alpha_3, \alpha_1) \wedge \beta_1 = \mathsf{hash}(\alpha_2) : \\
ZK_1 & \triangleq & \mathsf{zk}_{S_1}\big(\overbrace{k_{UP}, n, q}^{\alpha_1,\alpha_2,\alpha_3}; \overbrace{\mathsf{hash}(n)}^{\beta_1}, \overbrace{\mathsf{senc}(q, k_{UP})}^{\beta_2}\big)
\end{array}
$$

The first zero-knowledge proof ensures that the user knows the secret n associated to her public pseudonym $\mathsf{hash}(n)$ and, similarly to the previous protocol, binds this proof of knowledge to the ciphertext. The second zero-knowledge proof is similar to the one in previous protocol. The resulting protocol is robustly safe with respect to the authorization policy (1).

3 Transformation

This section presents our transformation for strengthening cryptographic protocols with zero-knowledge proofs in order to achieve security despite compromise. Given the space constraints, we only provide a high-level overview of the transformation and show the transformation in detail when applied to the protocol from Section 2. The complete technical description of the transformation can be found in the long version of this paper [3].

3.1 High-level Overview of the Transformation

Our transformation takes as input a spi calculus process (see Appendix A) that is a parallel composition of participants which communicate by sending messages to each other on public channels. We require that there is at most one sender and one receiver on each public channel, and that the induced message flow graph is acyclic.

The transformation relies on the information obtained from two static analyses of the original protocol. The first is *secrecy analysis*, which for each output message returns the secret values and the public values occurring in that message. For instance, restricted names that are not sent in clear are regarded as private, together with signing and decryption keys, and variables storing the result of a decryption. Free names, public keys, and verification keys are considered public. This information is used when generating zero-knowledge proofs to determine which terms should occur in the private component and which ones in the public component.

Second, we use a *data-dependency analysis* that for each output performed by a participant computes the previous inputs on which the output depends (directly or indirectly), as well as the cryptographic operations performed by the participant to obtain the output values from the input ones. This is done by analyzing the sequential code of each protocol participant and recording all performed cryptographic operations as a substitution that can be applied to the output term. Among others, this information is used to determine what zero-knowledge proofs have to be forwarded together with the proof replacing an output message. In our example the message output by the proxy contains a query received from the user, so in the transformed protocol the proxy needs to forward the zero-knowledge proof received from the user.

Once the static analyses are performed the transformation proceeds in three steps: The first step is *zero-knowledge proof generation*, which replaces each output message by the corresponding zero-knowledge proof together with the proofs that have to be forwarded. The zero-knowledge proof generation is defined by induction on the structure of the term output in the original protocol, but where the variables whose values are computed by the participant from received inputs are replaced by the symbolic representation provided by the data-dependency analysis. The second step is *zero-knowledge verification generation*, which after each input introduces additional checks that verify all the received zero-knowledge proofs. The shape we require for the original process allows the transformation to easily determine which output corresponds to each input, and therefore which zero-knowledge proofs need to be checked after each input. The last step is the *transformation of types*. We assume that the user provides typing annotations for the original protocol from which our algorithm generates typing annotations for the strengthened protocol. Types are discussed in Section 4.

Please note that the transformation only affects the structure of the messages sent and processed by the participants, and leaves unchanged: the top-level structure, the message flow, as well as the authorization policy, assumptions and assertions of the original protocol. Therefore showing that the transformed protocol is well-typed (and therefore robustly safe) implies that it conforms to the original authorization policy (even if some participants are compromised). In the following we illustrate the first two steps of the transformation on the example from Section 2.

3.2 Transforming the Example

The spi calculus process specifying the expected behavior of the user in the example from Section 2 is as follows:

$$\begin{array}{ll} \mathsf{new}\ q.(& \mathsf{assume}\ \mathsf{Request}(u,q)\ \mid \\ & \mathsf{out}(c_1, \mathsf{sign}(\mathsf{enc}((q,p), \mathsf{pk}(k_{PE})), k_U)) \quad) \end{array}$$

The names p, k_U, k_{PE} are bound by top-level restrictions. Secrecy analysis returns

$$\text{secret values}: \quad q, p, k_U \qquad\qquad \text{public values}: \quad \mathsf{pk}(k_{PE}), \mathsf{vk}(k_U), c_1$$

In the following, we show how to automatically construct the private component *sec*, the public component *pub*, and the statement S of the zero-knowledge proof ZK_1 replacing the output term $\mathsf{sign}(\mathsf{enc}((q,p), \mathsf{pk}(k_{PE})), k_U)$. We start with the true statement and empty private and public components. Since the considered term is a signature, we add to the statement the conjunct $\mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3$, proving that the term $\mathsf{enc}((q,p), \mathsf{pk}(k_{PE}))$ has been signed by the user. In order to preserve the secrecy of the signing key, we prove that the verification of the signature with the user's verification key $\mathsf{vk}(k_U)$ succeeds and returns $\mathsf{enc}((q,p), \mathsf{pk}(k_{PE}))$. Notice that the signature is added to the public component (since it is sent on a public channel) together with the verification key and the ciphertext.

$$\begin{array}{lcl} \mathit{Term} & = & \underline{\mathsf{sign}(\mathsf{enc}((q,p), \mathsf{pk}(k_{PE})), k_U)} \\ S & = & \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3 \\ \mathit{sec} & = & \varepsilon \\ \mathit{pub} & = & \mathsf{sign}(\mathsf{enc}((q,p), \mathsf{pk}(k_{PE})), k_U),\ \mathsf{vk}(k_U), \mathsf{enc}((q,p), \mathsf{pk}(k_{PE})) \end{array}$$

The remaining part of the statement is obtained from the nested encryption. The statement is extended to prove that the message signed by the user is the encryption of two secret terms. The names q and p are added to the private component, while the proxy's public key $\mathsf{pk}(k_{PE})$ is added to the public component.

$$\begin{array}{lcl} \mathit{Term} & = & \mathsf{sign}(\underline{\mathsf{enc}((q,p), \mathsf{pk}(k_{PE}))}, k_U) \\ S & = & \mathsf{enc}((\alpha_1, \alpha_2), \beta_4) = \beta_3 \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3 \\ \mathit{sec} & = & p, q \\ \mathit{pub} & = & \mathsf{sign}(\mathsf{enc}((q,p), \mathsf{pk}(k_{PE})), k_U),\ \mathsf{vk}(k_U), \mathsf{enc}((q,p), \mathsf{pk}(k_{PE})),\ \mathsf{pk}(k_{PE}) \end{array}$$

Finally, the terms in the public component are re-arranged so that public terms occur in the beginning followed by the original output message. The statement is changed accordingly. This rearrangement of public terms is necessary for the verification of the zero-knowledge proofs: the semantics requires the messages checked for equality by the verifier be contained in the first part of the public component (see Appendix A.3). The result of this rearrangement is the zero-knowledge proof ZK_1 shown in Section 2.4.

We now illustrate how the the zero-knowledge proof ZK_2 is generated. The spi calculus specification of the proxy is as follows:

$$\begin{array}{l} (\mathsf{assume}\ \mathsf{Registered}(u)\ \mid \\ \mathsf{in}(c_1, x).\mathsf{let}\ x_1 = \mathsf{check}(x, \mathsf{vk}(k_U))\ \mathsf{then} \\ \quad \mathsf{let}\ (x_2, x_3) = \mathsf{dec}(x_1, k_{PE})\ \mathsf{then}\ \mathsf{out}(c_2, \mathsf{sign}(\mathsf{enc}((u, x_2), \mathsf{pk}(k_S)), k_{PS}))\ ) \end{array}$$

The secrecy analysis returns

$$\begin{array}{ll} \text{secret values:} & x_2, x_3, k_{PE}, k_{PS} \\ \text{public values:} & c_1, c_2, x, x_1, \mathsf{vk}(k_{PS}), \mathsf{pk}(k_S), \mathsf{pk}(k_{PE}), \mathsf{vk}(k_U), u \end{array}$$

The generation of the statement for the output term $\mathsf{sign}(\mathsf{enc}((u, x_2), \mathsf{pk}(k_S)), k_{PS})$ is similar to the generation of the statement for the message output by the user. In this case, however, only x_2 is added to the private component, while u is added to the public component. The statement proves that the proxy has signed an encryption of the pair consisting of the name u and of a secret term.

$$\begin{array}{lcl} S & = & \mathsf{enc}((\beta_5, \alpha_1), \beta_4) = \beta_3 \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3 \\ sec & = & x_2 \\ pub & = & \mathsf{sign}(\mathsf{enc}((u, x_2), \mathsf{pk}(k_S)), k_{PS}),\ \mathsf{vk}(k_{PS}),\ \mathsf{enc}((u, x_2), \mathsf{pk}(k_S)), \mathsf{pk}(k_S), u \end{array}$$

As opposed to the term output by the user, the signature output by the proxy contains a variable x_2, whose value is computed by the proxy from the input x. A honest proxy has to compute x_2 by checking whether x is a valid signature made with k_U and in case this succeeds decrypting the result with k_{PE}. This information is returned by the data-dependency analysis, so we add conjuncts $\mathsf{check}(x, \mathsf{vk}(k_U)) \rightsquigarrow x_1$ and $\mathsf{dec}(x_1, k_{PE}) \rightsquigarrow (x_2, x_3)$ to the generated statement:

$$\begin{array}{lcl} S & = & \mathsf{check}(\beta_8, \beta_9) \rightsquigarrow \beta_6 \wedge \mathsf{dec}(\beta_6, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2) \wedge \beta_7 = \mathsf{pk}(\alpha_3) \\ & & \quad \wedge \mathsf{enc}((\beta_5, \alpha_1), \beta_4) = \beta_3 \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3 \\ sec & = & x_2, x_3, k_{PE} \\ pub & = & \mathsf{sign}(\mathsf{enc}((u, x_2), \mathsf{pk}(k_S)), k_{PS}),\ \mathsf{vk}(k_{PS}),\ \mathsf{enc}((u, x_2), \mathsf{pk}(k_S)), \\ & & \mathsf{pk}(k_S),\ u,\ x_1,\ \mathsf{pk}(k_{PE}), x,\ \mathsf{vk}(k_U) \end{array}$$

The statement guarantees that the query x_2 and the secret password are obtained by the decryption of a ciphertext ($\mathsf{dec}(\beta_6, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2)$) signed by the user ($\mathsf{check}(\beta_8, \beta_9) \rightsquigarrow \beta_6$). The equality $\beta_7 = \mathsf{pk}(\alpha_3)$ guarantees that the private key used in the decryption corresponds to the public key of the proxy. Notice that u, $\mathsf{pk}(k_S)$, and the ciphertext x_1 are inserted into the public component. The proxy's decryption key and the password x_3 are instead included in the private component. The password is in the private component since it was obtained by decrypting a ciphertext with the proxy's private key. The zero-knowledge proof ZK_2 shown in Section 2.4 is obtained by rearranging the terms in the public component, as previously discussed. Finally, the signature output by the proxy in the original protocol is replaced by the zero-knowledge proof ZK_2 together with the zero-knowledge proof ZK_1 received from the user, on which the values in ZK_2 depend. The code for the original protocol and the code obtained after the transformation is shown in Appendix D, while Appendix C describes the transformation algorithm in full detail.

4 Type System for Zero-knowledge

We use an enhanced type system for zero-knowledge to statically verify the security despite compromise of the protocols generated by our transformation. This type system extends our previous one [4] to the setting of compromised participants. In particular, instead of relying on unconditionally secure types, we give a precise characterization of when a type is compromised in the form of a logical formula (Section 4.3). Furthermore, we add union and intersection types to [4] (Section 4.4). The union types are used together with refinement types (i.e., types that contain logical formulas) to express type information that is conditioned by a participant being uncompromised (Section 4.5). Additionally, we use intersection and union types to infer very precise type information about the secret witnesses of zero-knowledge proofs (Section 4.6).

4.1 Basic Types

Messages are given security-related types. The syntax of types is reported in Table 1. Type Un (untrusted) describes messages possibly known to the adversary, while messages of type Private

Table 1 Basic types

$$
\begin{array}{rl}
T\ ,U ::= & \top\ ,\mathsf{Un}\ ,\mathsf{Ch}(T)\ ,\mathsf{Pair}(x:T,U)\ ,\{x:T\mid C\} \\
& \mathsf{SigKey}(T)\ ,\mathsf{VerKey}(T)\ ,\mathsf{Signed}(T) \\
& \mathsf{PrivKey}(T)\ ,\mathsf{PubKey}(T)\ ,\mathsf{PubEnc}(T) \\
& \mathsf{Hash}(T)\ ,\mathsf{SymKey}(T)\ ,\mathsf{SymEnc}(T)
\end{array}
$$

Notations: $\mathsf{Private} \triangleq \mathsf{Ch}(\top)$ and $\bot \triangleq \{x:\mathsf{Un}\mid \mathsf{false}\}$.
$\{C\} \triangleq \{x:\mathsf{Un}\mid C\}$ and $\{\!|\, C \,|\!\} \triangleq \{x:\top\mid C\}$, where in both cases $x \notin fv(C)$.

are not revealed to the adversary. Channels carrying messages of type T are given type $\mathsf{Ch}(T)$. So $\mathsf{Ch}(\mathsf{Un})$ is the type of a public channel where the attacker can read and write messages.

Pairs are given dependent types of the form $\mathsf{Pair}(x:T,U)$, where the type U of the second component of the pair can depend on the value x of the first component (For non-dependent pair types we omit the unused variable and write $\mathsf{Pair}(T,U)$.). As in [14, 7] we use refinement types to associate logical formulas to messages. The refinement type $\{x:T\mid C\}$ contains all messages M of type T for which the formula $C\{M/x\}$ is entailed by the current environment. For instance, $\{x:\mathsf{Un}\mid \mathsf{Good}(x)\}$ is the type of all public messages M for which the predicate $\mathsf{Good}(M)$ holds.

Additionally, we consider types for the different cryptographic primitives. For digital signatures, $\mathsf{SigKey}(T)$ and $\mathsf{VerKey}(T)$ denote the types of the signing and verification keys for messages of type T, while $\mathsf{Signed}(T)$ is the type of signed messages of type T. A key of type $\mathsf{SigKey}(T)$ can only be used to sign messages of type T, where the type T is in general annotated by the user. Similarly, $\mathsf{PubKey}(T)$ and $\mathsf{PrivKey}(T)$ denote the types of public encryption keys and of decryption keys for messages of type T, while $\mathsf{PubEnc}(T)$ is the type of a public-key encryption of a message of type T. Finally, the type $\mathsf{SymEnc}(T)$ is given to encryptions done using symmetric keys of type $\mathsf{SymKey}(T)$. In all these cases the type T is usually a refinement type conveying a logical formula. For instance, $\mathsf{SigKey}(\{x:\mathsf{Private}\mid \mathsf{Good}(x)\})$ is the type of keys that can only be used to sign private messages for which we know that the Good predicate holds.

As usual, a typing environment Γ is a set of bindings between names (or variables) and types.

4.1.1 Typing the Original Example (Uncompromised Setting)

We illustrate the type system on the original protocol from Section 2. Since the query q the user sends to the proxy is not secret, but authentic, we give it type $\{\mathsf{Un}\mid \mathsf{Request}(u,x_q)\}$. The password p is of course secret and is given type $\mathsf{Private}$. The payload sent by the user, the pair (q,p), can therefore be typed to $T_1 = \mathsf{Pair}(\{x_q:\mathsf{Un}\mid \mathsf{Request}(u,x_q)\}, x_p:\mathsf{Private})$. The public key of the proxy $\mathsf{pk}(k_{PE})$ is used to encrypt messages of type T_1 so we give it type $\mathsf{PubKey}(T_1)$. Similarly, the signing key of the user k_U is used to sign the term $\mathsf{enc}((q,p),\mathsf{pk}(k_{PE}))$, so we give it the type $\mathsf{SigKey}(\mathsf{PubEnc}(T_1))$, while the corresponding verification key $\mathsf{vk}(k_U)$ has type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1))$. Once the proxy verifies the signature using $\mathsf{vk}(k_U)$, decrypts the result using k_{PE}, and splits the pair into q and p it can be sure not only that q is of type Un and p is of type $\mathsf{Private}$, but also that $\mathsf{Request}(u,q)$ holds, i.e., the user has indeed issued a request.

In a very similar way, the signing key of the proxy k_{PS} is given type $\mathsf{SigKey}(\mathsf{PubEnc}(T_2))$, where T_2 is the dependent pair type $\langle x_u:\mathsf{Un}, x_q:\mathsf{Un}\rangle\{\mathsf{Request}(x_u,x_q)\wedge\mathsf{Registered}(x_u)\}$, which conveys the conjunction of two logical predicates. If the store successfully checks the signature using $\mathsf{vk}(k_{PS})$ the resulting message will have type $\mathsf{PubEnc}(T_2)$. Since k_S has type $\mathsf{PubKey}(T_2)$ it can be used to decrypt this message and obtain the user name u and the query q, for which $\mathsf{Request}(u,q)\wedge\mathsf{Registered}(u)$ holds. By the authorization policy given in Section 2, this logically

implies $\mathsf{Authenticate}(u, q)$. The authentication request is thus justified by the policy, so if all participants are honest the original protocol is secure (robustly safe).

4.2 Subtyping and Kinding

All messages sent to and received from an untrusted channel have type Un, since such channels are considered under the complete control of the adversary. However, a system in which only names and variables of type Un could be communicated over the untrusted network would be too restrictive, e.g., encryptions could not be sent over the network. We therefore consider a *subtyping relation* on types, which allows a term of a subtype to be used in all contexts that require a term of a supertype. This preorder is used to compare types with the special type Un. In particular, we allow messages having a type T that is a subtype of Un, denoted $T <: \mathsf{Un}$, to be sent over the untrusted network, and we say that the type T has *kind public* in this case. Similarly, we allow messages of type Un that are received from the untrusted network to be used as messages of type U, provided that $\mathsf{Un} <: U$, and in this case we say that type U has *kind tainted*.

For example, in our type system the types $\mathsf{PubKey}(T)$ and $\mathsf{VerKey}(T)$ are always public, meaning that public-key encryption keys as well as signature verification keys can always be sent over an untrusted channel without compromising the security of the protocol. On the other hand, $\mathsf{PrivKey}(T)$ is public only if T is also public, since sending to the adversary a private key that decrypts confidential messages will most likely compromise the security of the protocol. Finally, type $\mathsf{Private}$ is neither public nor tainted, while type Un is always public and tainted.

In the following, we address in more detail some interesting subtyping and kinding rules related to refinement types, the top type, and the bottom type, which are reported in Table 2. The type $\top$ is a supertype of any type (by Sub-Top), while the refinement type $\{x : T \mid C\}$ is a subtype of any T' that is a supertype of T (Sub-Refine-Left). Notice that $\{x : T \mid C\}$ is also a subtype of T, since subtyping is reflexive (by Sub-Refl). The bottom type $\bot$ is encoded as $\{x : \mathsf{Un} \mid \mathsf{false}\}$. This is the empty type that is a subtype of all types (by Sub-Refine-Empty), since in any typing environment $\neg\mathsf{false}$ is logically entailed. Since type $\top$ is also a supertype of Un it is tainted, but it is not public. Dually, type $\bot$ is public but not tainted.

A useful property of refinement types in our type system is that $\{x : T \mid C\}$ is equivalent by subtyping[6] to T in an environment in which the formula $\forall x.C$ holds, and equivalent to type $\bot$ in case the formula $\forall x.\neg C$ holds. Together with the property that $\{x : T \mid C\}$ is always a subtype of T and the transitivity of subtyping, this implies that a refinement type $\{x : T \mid C\}$ is public if T is public (since $\{x : T \mid C\} <: T <: \mathsf{Un}$) or if the formula $\forall x.\neg C$ is entailed by the environment (since $\{x : T \mid C\} <:> \bot <: \mathsf{Un}$). Conversely, type $\{x : T \mid C\}$ is tainted if T is tainted and additionally $\forall x.C$ holds (cf. Kind-Refine-Tnt). In the following, we use $\{\!|\, C \,|\!\}$ to denote the refinement type $\{x : \top \mid C\}$, where x does not appear in C. Note that, since we assume a classical authorization logic that fulfills the law of excluded middle, type $\{\!|\, C \,|\!\}$ is either equivalent to $\top$ (if C holds), or to $\bot$ (if $\neg C$ holds).

4.3 Logical Characterization of Kinding and Subtyping

We capture the precise conditions that a typing environment needs to satisfy in order for a type to be public (or tainted) as a logical formula. We define a function pub that given any type T returns a formula which is entailed by a typing environment if and only if type T is public in this environment. In a similar way tnt provides a logical characterization of taintedness. Since

[6] We call types T and U equivalent by subtyping if both $T <: U$ and $U <: T$.

Table 2 Rules for refinement types and top

Subtyping

$$\textsc{Sub-Pub-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T <: U} \qquad \textsc{Sub-Refl}\ \frac{\Gamma \vdash T}{\Gamma \vdash T <: T} \qquad \textsc{Sub-Top}\ \frac{\Gamma \vdash T}{\Gamma \vdash T <: \top}$$

$$\textsc{Sub-Refine-Left}\ \frac{\Gamma \vdash \{x : T \mid C\} \qquad \Gamma \vdash T <: T'}{\Gamma \vdash \{x : T \mid C\} <: T'} \qquad \textsc{Sub-Refine-Empty}\ \frac{\Gamma \vdash \{x : T \mid C\} \qquad \Gamma, x : T \models \neg C}{\Gamma \vdash \{x : T \mid C\} <: T'}$$

$$\textsc{Sub-Refine-Right}\ \frac{\Gamma \vdash T <: T' \qquad \Gamma, x : T \models C}{\Gamma \vdash T <: \{x : T' \mid C\}}$$

Kinding

$$\textsc{Kind-Top-Tnt}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \top :: \mathsf{tnt}} \qquad \textsc{Kind-Top-Pub}\ \frac{\Gamma \models \mathsf{false}}{\Gamma \vdash \top :: \mathsf{pub}} \qquad \textsc{Kind-Refine-Pub}\ \frac{\Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash \{x : T \mid C\} :: \mathsf{pub}} \qquad \textsc{Kind-Refine-Empty-Pub}\ \frac{\Gamma, x : T \models \neg C}{\Gamma \vdash \{x : T \mid C\} :: \mathsf{pub}}$$

$$\textsc{Kind-Refine-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt} \qquad \Gamma, x : T \models C}{\Gamma \vdash \{x : T \mid C\} :: \mathsf{tnt}}$$

Notation: The judgments $\Gamma \vdash T$ and $\Gamma \vdash \diamond$ rule the well-formedness of types and typing environments, while $\Gamma \models C$ states that C is logically entailed from the formulas occurring in the refinement types in Γ.

in the current type system we do not have any unconditionally secure types these two functions are total.

Since type Un is always public and tainted and since true is valid in any environment, we have $\mathsf{pub}(\mathsf{Un}) = \mathsf{tnt}(\mathsf{Un}) = \mathsf{true}$. Conversely, type $\mathsf{Private}$ is neither public and nor tainted. However, in an inconsistent environment, in which false is entailed, it is harmless (and useful) to consider type $\mathsf{Private}$ to be both public and tainted[7], so equivalent to Un. So we have that $\mathsf{pub}(\mathsf{Private}) = \mathsf{tnt}(\mathsf{Private}) = \mathsf{false}$.

In order for a channel type $\mathsf{Ch}(T)$ to be public the type of the payload needs to be both public and tainted, so $\mathsf{pub}(\mathsf{Ch}(T)) = \mathsf{pub}(T) \wedge \mathsf{tnt}(T)$. As intuitively explained at the end of the previous subsection, for refinement types we have $\mathsf{pub}(\{x : T \mid C\}) = \mathsf{pub}(T) \vee \forall x.\neg C$ and $\mathsf{tnt}(\{x : T \mid C\}) = \mathsf{tnt}(T) \wedge \forall x.C$. The types of the cryptographic primitives are also easy to handle. For instance, $\mathsf{pub}(\mathsf{PubKey}(T)) = \mathsf{pub}(\mathsf{VerKey}(T)) = \mathsf{true}$, $\mathsf{pub}(\mathsf{PrivKey}(T)) = \mathsf{pub}(T)$ and $\mathsf{pub}(\mathsf{SigKey}(T)) = \mathsf{tnt}(T)$.

In a similar way, we define a function $\mathsf{sub}(T, U)$ that precisely captures the conditions under which T is a subtype of U. The precise definition is given in Table 19 in Appendix B.

4.4 Union and Intersection Types

[7] In an inconsistent environment all assertions are justified (false implies everything). Furthermore, in such an environment all types become equivalent to Un, so by an argument similar to "opponent typability" any well-formed protocol is also well-typed.

Table 3 Typing rules for Union and Intersection Types

Term and Process Typing

$$\textsc{And}\ \frac{\Gamma \vdash M : T \quad \Gamma \vdash M : U}{\Gamma \vdash M : T \wedge U} \qquad \textsc{Sub}\ \frac{\Gamma \vdash M : T \quad \Gamma \vdash T <: T'}{\Gamma \vdash M : T'}$$

$$\textsc{Proc-Case}\ \frac{\Gamma \vdash M : T \vee U \qquad \Gamma, x : T \vdash P \qquad \Gamma, x : U \vdash P}{\Gamma \vdash \mathsf{case}\ x = M\ \mathsf{in}\ P}$$

Subtyping

$$\textsc{Sub-And-LB}\ \frac{\Gamma \vdash T_i <: U}{\Gamma \vdash T_1 \wedge T_2 <: U} \qquad \textsc{Sub-And-Greatest}\ \frac{\Gamma \vdash T <: U_1 \qquad \Gamma \vdash T <: U_2}{\Gamma \vdash T <: U_1 \wedge U_2} \qquad \textsc{Sub-Or-Smallest}\ \frac{\Gamma \vdash T_1 <: U \qquad \Gamma \vdash T_2 <: U}{\Gamma \vdash T_1 \vee T_2 <: U}$$

$$\textsc{Sub-Or-UB}\ \frac{\Gamma \vdash T <: U_i}{\Gamma \vdash T <: U_1 \vee U_2}$$

Kinding

$$\textsc{Kind-And-Pub}\ \frac{\Gamma \vdash T_i :: \mathsf{pub}}{\Gamma \vdash T_1 \wedge T_2 :: \mathsf{pub}} \qquad \textsc{Kind-And-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt} \qquad \Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T \wedge U :: \mathsf{tnt}} \qquad \textsc{Kind-Or-Pub}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash U :: \mathsf{pub}}{\Gamma \vdash T \vee U :: \mathsf{pub}}$$

$$\textsc{Kind-Or-Tnt}\ \frac{\Gamma \vdash T_i :: \mathsf{tnt}}{\Gamma \vdash T_1 \vee T_2 :: \mathsf{tnt}}$$

We extend the type system from [4] with union and intersection types [21]. The typing and subtyping rules are reported in Table 3.

A message has type $T \wedge U$ if and only if it has both type T and type U (cf. And). Intuitively, the set of messages of type $T \wedge U$ is the intersection between the set of messages of type T and the set of messages of type U. Also, if a message has type T then it also has type $T \vee U$ for any U (cf. Sub and Sub-Or-UB). However if all we know about a message is that it has type $T \vee U$ we cannot safely perform operations on this message that are specific to type T but not to U. The code that uses this message has to be sufficiently "parametric" so that it type-checks both if the message has type T and if it has type U.

The type $T_1 \wedge T_2$ is a subtype of U if T_1 is a subtype of U or if T_2 is a subtype of U (cf. Sub-And-LB), while T is a subtype of $U_1 \wedge U_2$ if it is subtype of both U_1 and U_2 (cf. Sub-And-Greatest). Similarly, $T_1 \wedge T_2$ is public if T or U are public, while $T \wedge U$ is tainted if both T and U are tainted. The rules for intersection types are dual. Some useful properties of union and intersection types are that $T \wedge \top$ and $T \vee \bot$ are equivalent by subtyping to T; $T \wedge \bot$ is equivalent to $\bot$; and $T \vee \top$ is equivalent to $\top$.

More important, union types can be used together with refinement types to express conditional type information. For instance the type $\mathsf{Private} \vee \{\!|\, C \,|\!\}$ is private only in an environment in which the formula C does not hold (e.g., $\mathsf{Private} \vee \{\!|\, \mathsf{false} \,|\!\} <:> \mathsf{Private} \vee \bot <:> \mathsf{Private}$). In case C holds the type is equivalent to $\top$ (e.g., $\mathsf{Private} \vee \{\!|\, \mathsf{true} \,|\!\} <:> \mathsf{Private} \vee \top <:> \top$), so in

this case $\mathsf{Private} \vee \{\!|\, C \,|\!\}$ carries no valuable type information. This technique is most useful in conjunction with the logical characterization of kinding from Section 4.3. For instance, the type $T \vee \{\!|\, \neg\mathsf{pub}(T) \,|\!\}$ is equivalent by subtyping to T if T is a secret type, and to $\top$ if T is public (this will be exploited in Section 4.6). Similarly, we can express type information that is conditioned by a participant being uncompromised. Assuming that the predicate $\mathsf{Compromised}(v)$ is entailed by the environment if and only if v is compromised, then we can define a type $\mathsf{PrivateUnlessP}$ that is private if and only if the proxy is uncompromised as $\{\mathsf{Private} \mid \neg\mathsf{Compromised}(\mathit{proxy})\} \vee \{\mathsf{Un} \mid \mathsf{Compromised}(\mathit{proxy})\}$ (this will be exploited in Section 4.5).

Intersection types are used together with union types to combine type information, in order to infer more precise types for the secret witnesses of a zero-knowledge proof.

4.5 Compromising Participants

As explained in Section 2.2 when a participant v is compromised all its secrets are revealed to the attacker and the predicate $\mathsf{Compromised}(v)$ is added to the environment. However, we need to make the types of p's secrets public, in order to be able to reveal them to the attacker. For instance, in the protocol from Section 2, when compromising the proxy the type of the decryption key k_{PE} needs to be made public. However, once we replace the type annotation of this key from $\mathsf{PrivKey}(T_1)$ to Un, other types need to be changed as well. The type of the signing key of the user k_U is used to sign an encryption done with $\mathsf{pk}(k_{PE})$, so one could change the type of k_U from $\mathsf{SigKey}(\mathsf{PubEnc}(T_1))$ to $\mathsf{SigKey}(\mathsf{PubEnc}(\mathsf{Un}))$, which is actually equivalent to Un. This type would be, however, weaker than necessary. The fact that the store is compromised does not affect the fact that the user assumes $\mathsf{Request}(u, q)$, so we can give k_U type $\mathsf{SigKey}(\mathsf{PubEnc}(T_1'))$, where $T_1' = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, x_p : \mathsf{Un})$. Similar changes need to be done manually for the other type annotations, resulting in a specification that differs from the original uncompromised one only with respect to the type annotations.

However, having two different specifications that need to be kept in sync would be error prone. As proposed by Fournet et al. [14], we use only one set of type annotations for both the uncompromised and the compromised scenarios, containing types that are secure only under the condition that certain participants are uncompromised.

4.5.1 Typing the Original Example (Compromised Setting)

We illustrate this on our running example. The type of the payload sent by the user, which used to be $T_1 = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, x_p : \mathsf{Private})$, is now changed to $T_1^* = \mathsf{Pair}(\{x_q : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\}, x_p : \mathsf{PrivateUnlessP})$. In the uncompromised setting $\neg\mathsf{Compromised}(\mathit{proxy})$ is entailed in the system, type $\mathsf{PrivateUnlessP}$ is equivalent to $\mathsf{Private}$, and T_1^* is equivalent to T_1. However, if the proxy is compromised then the predicate $\mathsf{Compromised}(\mathit{proxy})$ is entailed, $\mathsf{PrivateUnlessP}$ is equivalent to Un and T_1^* is equivalent to T_1'. Using this we can give k_U type $\mathsf{SigKey}(\mathsf{PubEnc}(T_1^*))$ and k_{PE} type $\mathsf{PrivKey}(T_1^*)$.

In the uncompromised setting, the payload sent by the proxy has type $T_2 = \langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\}$. However, once the proxy is compromised, the attacker can replace this payload with a message of his choice, so the type of this payload becomes Un. In order to be able to handle both scenarios we give this payload type $T_2^* = \{T_2 \mid \neg\mathsf{Compromised}(\mathit{proxy})\} \vee \{\mathsf{Un} \mid \mathsf{Compromised}(\mathit{proxy})\}$. The types of k_{PS} and k_S are updated accordingly.

With these changed annotations in place we can still successfully type-check the example protocol in the case all participants are honest (see Section 4.1), but in addition we can also try to check the protocol in case the proxy is corrupted. The latter check will however fail since the store is going to obtain a payload of type T_2^*. Since the proxy is compromised, T_2^* is equivalent

to Un, and provides no guarantees that could justify the authentication of the request. This is not surprising since, as explained in Section 2.2, the original protocol is not secure if the proxy is compromised.

4.6 Type-checking Zero-knowledge

As first done in [4], we give a zero-knowledge proof $\mathsf{zk}_S(\widetilde{N};\widetilde{M})$ a type of the form $\mathsf{ZKProof}_S(\widetilde{y}:\widetilde{T};\exists\widetilde{x}.C)$. This type lists the types of the arguments in the public component and contains a logical formula of the form $\exists x_1,\ldots,x_n.C$, where the arguments in the private component are existentially quantified. The type system guarantees that $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ is entailed by the environment. For instance, the proof ZK_1 sent by the user to the proxy in the strengthened protocol, which was defined in Section 2.4 as:

$$\mathsf{zk}_{S_1}\big(\overbrace{q,p}^{\alpha_1,\alpha_2};\overbrace{\mathsf{vk}(k_U)}^{\beta_1},\overbrace{\mathsf{pk}(k_{PE})}^{\beta_2},\overbrace{\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big)}^{\beta_3},\overbrace{\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)}^{\beta_4}\big)$$

where $S_1 \triangleq \mathsf{enc}((\alpha_1,\alpha_2),\beta_2) = \beta_4 \wedge \mathsf{check}(\beta_3,\beta_1) \rightsquigarrow \beta_4$, is given type:

$$\begin{array}{lll}\mathsf{ZKProof}_{S_1}(& y_1{:}\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*)), & y_2{:}\mathsf{PubKey}(T_1^*),\\ & y_3{:}\mathsf{Signed}(\mathsf{PubEnc}(T_1^*)), & y_4{:}\mathsf{PubEnc}(T_1^*); \quad \exists x_1,x_2.C_1)\end{array}$$

where $C_1 = \mathsf{enc}((x_1,x_2),y_2) = y_4 \wedge \mathsf{Red}(\mathsf{check}(y_3,y_1),y_4) \wedge \mathsf{Request}(u,x_2)$. There is a direct correspondence between the four values in the public component, and the types given to the variables y_1 to y_4. Also, the first two conjuncts in C_1 directly correspond to the statement S_1. It is always safe to include the proved statement in the formula being conveyed by the zero-knowledge type, since the verification of the proof succeeds only if the statement is valid.

However, very often conveying the statement alone does not suffice to type-check the examples we have tried, since the statement only talks about terms and does not mention any logical predicate. The predicates are dependent on the particular protocol and policy, and are automatically inferred by our transformation. For instance, in our example the original message from the user to the proxy was conveying the predicate $\mathsf{Request}(u,q)$, so this predicate is added by the transformation to the formula C_1. Our type-checker verifies that these additional predicates are indeed justified by the statement and by the types of the public components checked for equality by the verifier of the proof.

4.6.1 Typing the Strengthened Example (Compromised Setting)

We illustrate this by type-checking the store in the strengthened protocol in case the proxy is compromised. We start with ZK_1, the zero-knowledge proof created by the user, intended to be forwarded by the (actually compromised) proxy and then verified by the store. The first two public messages in ZK_1, $\mathsf{vk}(k_U)$ and $\mathsf{pk}(k_{PE})$, are checked for equality against the values the store already has. If the verification of ZK_1 succeeds, the store knows that y_1 and y_2 have indeed type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ and $\mathsf{PubKey}(T_1^*)$, respectively. However, since the proof is received from an untrusted source, it could have been generated by the attacker, so the other public components, y_3 and y_4, are given type Un. For the private components x_1 and x_2 the store has no information whatsoever, so he gives them type $\top$. Using this initial type information and the fact that the statement $\mathsf{enc}((x_1,x_2),y_2) = y_4 \wedge \mathsf{check}(y_3,y_1) \rightsquigarrow y_4$ holds, the type-checker tries to infer additional information.

Since y_1 has type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ and $\mathsf{check}(y_3,y_1) \rightsquigarrow y_4$ holds, we infer that y_4 has type $\mathsf{PubEnc}(T_1^*) \vee \{\!|\,\mathsf{tnt}(\mathsf{PubEnc}(T_1^*))\,|\!\}$, i.e., y_4 has type $\mathsf{PubEnc}(T_1^*)$ under the condition that the type $\mathsf{PubEnc}(T_1^*)$ is not tainted. If this type was tainted then the type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$

is equivalent to Un. However, this is not the case since the user is not compromised. So the new type inferred for y_4 is equivalent to $\mathsf{PubEnc}(T_1^*) \vee \bot$ and therefore to $\mathsf{PubEnc}(T_1^*)$. Since y_4 also has type Un from before, the most precise type we can give to it is the intersection type $\mathsf{PubEnc}(T_1^*) \wedge \mathsf{Un}$. Since $\mathsf{PubEnc}(T_1^*)$ is public this happens to be equivalent to just $\mathsf{PubEnc}(T_1^*)$. Since y_4 has type $\mathsf{PubEnc}(T_1^*)$ and $\mathsf{enc}((x_1, x_2), y_2) = y_4$ we can infer that (x_1, x_2) has type $T_1^* \vee \{\!|\, \mathsf{tnt}(T_1^*) \,|\!\}$. Since the user is not compromised $\mathsf{tnt}(T_1^*) = \mathsf{false}$ so (x_1, x_2) has type T_1^*. This implies that the predicate $\mathsf{Request}(u, x_2)$ holds, and thus justifies the type annotation automatically generated by the transformation.

The proof ZK_2 is easier to type-check since its type just contains S_2, but no additional predicates. This means that its successful verification only conveys certain relations between terms. These relations are, however, critical for linking the different messages. Most importantly, they ensure that the query received in ZK_2 is the same as variable x_2 in ZK_1 for which $\mathsf{Request}(u, x_2)$ holds by the verification of ZK_1, as explained above. Since the proxy is compromised the predicate $\mathsf{Registered}(u)$ holds. So in the strengthened protocol the authentication decision of the store is indeed justified by the authorization policy, even if the proxy is compromised.

5 Generating Implementations

We have developed a tool, called Spi2RCF, that automatically implements protocols based on zero-knowledge proofs. The tool takes as input protocols in the same variant of the spi calculus as used by the translation from Section 3 (see Appendix A). It generates reference implementations in RCF [7], a core calculus of ML. These implementations can be linked either against a concrete cryptographic library, or against a symbolic one where the cryptographic primitives are considered fully reliable building blocks and represented using a mechanism for dynamic sealing.

The transformation works as follows. In a pre-processing stage all the nested applications of cryptographic primitives are eliminated using let statements, and the protocol is put into a normal form [23]. The constructors and destructors from the spi calculus are then translated into calls to the corresponding RCF functions.

Types are also translated, and this is in some cases challenging, since certain primitive types from the spi calculus have no direct correspondent in RCF, where the types of cryptographic primitives are encoded using more primitive types. For instance, the type of the user's signing key k_U in the original protocol from Section 2 is $\mathsf{SigKey}(\mathsf{PubEnc}(T_1^*))$ where $T_1^* = \mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{PrivateUnlessP} \mid \mathsf{Request}(u, x_q)\})$. However in RCF the type PubEnc does not exist and encrypted messages are given type Un. Still we encode this type as $\mathsf{SK}\,\langle\{x : \mathsf{Un} \mid \exists y_u, y_q.\ \mathit{encrypted}(k_{PE}, (y_u, y_q), x) \ \wedge \mathsf{Request}(u, y_q)\}\rangle$, where *encrypted* is a predicate that is obtained when calling the encryption and decryption functions, and for which the following injectivity property holds:

$$\forall k, m_1, m_2, c. \quad \mathit{encrypted}(k, m_1, c) \wedge \quad \mathit{encrypted}(k, m_2, c) \Rightarrow m_1 = m_2,$$

corresponding to the determinism of decryption. The $\mathsf{Request}$ predicate is now conveyed by the signature rather than by the encryption, still using the *encrypted* predicate we can link it to the message obtained by the decryption. Note however that this translation only works for transferring logical formulas, and types such as $\mathsf{PubEnc}(\mathsf{Private})$ or $\mathsf{PubEnc}(\mathsf{SymKey}(T))$ cannot be faithfully translated.

A previous version of the transformation that did not include zero-knowledge and considered weaker types for the spi calculus primitives (e.g., encryption returned Un not $\mathsf{PubEnc}(T)$) was proved to preserve typability, and therefore the security properties of the input protocol [25].

As the example above suggests this is not always the case for the current transformation. Nevertheless, the current transformation is a lot more useful since it applies to much more protocols (with the weaker types the original protocol in Section 2 would for instance be rejected). As done for the translation in Section 3, we use a type-checker to validate the security of the generated RCF implementation [7] with respect to the symbolic cryptographic library.

6 Conclusions and Future Work

We have presented a general automated technique to strengthen cryptographic protocols in order to make them resistant to compromised participants. Our approach relies on non-interactive zero-knowledge proofs that are used by each honest participant to prove that she has generated the messages sent to other participants in accordance to the protocol, without revealing any of her secrets. The zero-knowledge proofs are forwarded to ensure the correct behaviour of all participants involved in the protocol. We prove the safety despite compromise of the transformed protocols by using an enhanced type system that can automatically analyze protocols using zero-knowledge in the setting of participant compromise. Finally, we developed a tool called Spi2RCF that automatically implements protocols based on zero-knowledge proofs in a core calculus of ML.

We plan to work on the efficiency of the transformationfrom Section 3. For instance, we can apply techniques similar to the ones proposed in [11, 8] to reduce the number of zero-knowledge proofs forwarded by each participant. Our approach is specifically suited to reason about optimizations since we use a type-checker to check if the transformed process is robustly safe despite compromised participants, and another type-checker to verify that this property carries over to the generated ML implementation. This translation validation approach has the advantage that even if we apply drastic optimizations, or completely reimplement the transformation, we do not need to redo any proofs.

Finally, while Spi2RCF can generate protocol implementations that can link against a symbolic as well as a concrete zero-knowledge library, the task of concretely implementing the employed cryptographic zero-knowledge proof systems is still manual and non-trivial at the moment. We plan to investigate automating this process.

Acknowledgments. We are grateful to Cédric Fournet and Andrew D. Gordon for the useful discussions. We thank the ARSPA-WITS and CSF reviewers for their comments on preliminary versions of this article. Cătălin Hriţcu is supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science. Matteo Maffei is partially supported by the initiative for excellence of the German federal government and by MIUR project "SOFT".

References

[1] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. *Information and Computation*, 174(1):37–83, 2002.

[3] M. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise with zero-knowledge. Long version and implementation available at `http://www.infsec.cs.uni-sb.de/projects/zk-compromise/`, 2009.

[4] M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. In *15th Proc. ACM Conference on Computer and Communications Security*, pages 357–370. ACM Press, 2008. Long version and implementation available at: `http://www.infsec.cs.uni-sb.de/projects/zk-typechecker/`.

[5] 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.

[6] M. Bellare, R. Canetti, and H. Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 419–428, 1998.

[7] 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.

[8] K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *Proc. 22th IEEE Symposium on Computer Security Foundations (CSF)*, 2009. To appear.

[9] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 132–145. ACM Press, 2004.

[10] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *Proc. 29th IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society Press, 2008.

[11] R. Corin, P.-M. Dénielou, C. Fournet, K. Bhargavan, and J. J. Leifer. Secure implementations of typed session abstractions. In *Proc. 20th IEEE Symposium on Computer Security Foundations (CSF)*, pages 170–186. IEEE Computer Society Press, 2007.

[12] V. Cortier, B. Warinschi, and E. Zălinescu. Synthesizing secure protocols. In *Proc. 12th European Symposium On Research In Computer Security (ESORICS)*, pages 406–421. Springer-Verlag, 2007.

[13] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13(3):423–482, 2005.

[14] 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.

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

[16] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – or – a completeness theorem for protocols with honest majority. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

[17] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):690–728, 1991. Online available at `http://www.wisdom.weizmann.ac.il/~oded/X/gmw1j.pdf`.

[18] J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology: CRYPTO '03*, pages 110–125. Springer-Verlag, 2003.

[19] L. Lu, J. Han, L. Hu, J. Huai, Y. Liu, and L. M. Ni. Pseudo trust: Zero-knowledge based authentication in anonymous peer-to-peer protocols. In *Proc. 2007 IEEE International Parallel and Distributed Processing Symposium*, page 94. IEEE Computer Society Press, 2007.

[20] G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[21] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.

[22] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166. Springer-Verlag, 1998.

[23] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Higher-Order and Symbolic Computation*, 6(3):289–360, 1993.

[24] B. Smyth, L. Chen, and M. D. Ryan. Direct anonymous attestation: ensuring privacy with corrupt administrators. In *Proceedings of the Fourth European Workshop on Security and Privacy in Ad hoc and Sensor Networks*, pages 218–231. Springer-Verlag, 2007.

[25] T. Tarrach. Spi2F# – a prototype code generator for security protocols. Bachelor's Thesis, Saarland University, October 2008.

[26] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proc. 35th Symposium on Principles of Programming Languages (POPL)*, pages 17–27. ACM Press, 2008.

[27] L. Zuck, A. Pnueli, and R. Leviahtan. Validation of optimizing compilers. Technical report, Weizmann Institute of Science Technical Report MCS01-12, August 2001.

A Spi Calculus with Constructors, Destructors and Zero-knowledge

In this paper we consider a variant of the spi calculus with arbitrary constructors and destructors similar to the ones in [1, 14], and extended with zero-knowledge proofs [4]. This appendix overviews the syntax and semantics of the calculus.

In the following we identify any phrase ϕ of syntax up to consistent renaming of bound names and variables. We say that ϕ is closed if it does not have any free variables. We write $\phi\{\phi'/x\}$ for the outcome of the capture-avoiding substitution of ϕ' for each free occurrence of x in ϕ.

A.1 Constructors and Terms

Constructors are function symbols that are used to build terms. The set of constructors we consider in this paper[8]includes pk that yields the public encryption key corresponding to a decryption key; enc for public-key encryption; vk that yields the verification key corresponding to a signing key; sign for digital signatures; and hash for hashes.

The set of *terms* (Table 4), ranged over by K, L, M and N, is the free algebra built from names (a, b, c, m, n, and k), variables (x, y, z, v, and w), pairs ((M_1, M_1)), and constructors applied to other terms ($f(M_1, \ldots, M_n)$). We let u range over both names and variables.

Table 4 Terms and constructors

K, L, M, N	::=	terms
	a, b, c, m, n, k	names
	x, y, z, v, w	variables
	(M, N)	pair
	$f(M_1, \ldots, M_n)$	constructor application (f of arity n)

$f ::= \mathsf{pk}^1, \mathsf{enc}^2, \mathsf{vk}^1, \mathsf{sign}^2, \mathsf{hash}^1, \mathsf{senc}^2, \mathsf{ok}^0, \mathsf{zk}^{n+m}_{n,m,S}, \alpha^0_i, \beta^0_i$

Note: $\mathsf{zk}_{n,m,S}$ is only defined when S is an (n, m)-statement.

Notation: We write $\langle M_1, \ldots, M_n\rangle$ to mean $(M_1, (M_2, \ldots, (M_n, \mathsf{ok})))$.

A.2 Destructors

Destructors are partial functions that processes can apply to terms, and are ranged over by g (Table 5). The semantics of destructors is specified by the reduction relation $\Downarrow$ (Table 6): given the terms $M_1, \ldots, M_n$ as arguments, the destructor g can either succeed and provide a term N as a result (which we denote as $g(M_1, \ldots, M_n) \Downarrow N$) or it can fail (denoted as $g(M_1, \ldots, M_n) \not\Downarrow$). The dec destructor decrypts an encrypted message given the corresponding decryption key. The check destructor checks a signed message using a verification key, and if this succeeds returns the message without the signature.

Table 5 Syntax of destructors

g	::=	$\mathsf{id}^1, \mathsf{dec}^2, \mathsf{check}^2, \mathsf{sdec}^2, \mathsf{public}^1_m, \mathsf{ver}^{l+1}_{n,m,l,S}$

Note: $\mathsf{ver}^{l+1}_{n,m,l,S}$ is only defined when S is an (n, m)-statement and $l \in [1, m]$.

[8]Our type-checker supports arbitrary constructors and destructors in a generic way.

Table 6 Semantics of destructors $g(M_1, \ldots, M_n) \Downarrow N$

$\mathsf{id}(M)$	$\Downarrow$	M
$\mathsf{dec}(\mathsf{enc}(M, \mathsf{pk}(K)), K)$	$\Downarrow$	M
$\mathsf{check}(\mathsf{sign}(M, K), \mathsf{vk}(K))$	$\Downarrow$	M
$\mathsf{sdec}(\mathsf{senc}(M, K), K)$	$\Downarrow$	M
$\mathsf{public}_m(\mathsf{zk}_{n,m,S}(\widetilde{N}, \widetilde{M}))$	$\Downarrow$	$\langle \widetilde{M} \rangle$

$\mathsf{ver}_{n,m,l,S}(\mathsf{zk}_{n,m,S}(\widetilde{N}, M_1, \ldots, M_l, \ldots, M_m), M_1, \ldots, M_l) \Downarrow \langle M_{l+1}, \ldots, M_m \rangle$ iff $[\![S\{\widetilde{N}/\widetilde{\alpha}\}\{\widetilde{M}/\widetilde{\beta}\}]\!] = \mathsf{true}$

Notation: $\widetilde{M} = M_1, \ldots, M_n$.
Notation: We write $g(M_1, \ldots, M_n) \not\Downarrow$ if none of the rules above applies, i.e., the destructor fails.

A.3 Representing Zero-knowledge Proofs

Constructing Zero-knowledge Proofs. As proposed in [4], a non-interactive zero-knowledge proof of a statement S is represented as a term of the form $\mathsf{zk}_{n,m,S}(N_1, \ldots, N_n; M_1, \ldots, M_m)$, where $N_1, \ldots, N_n$ and $M_1, \ldots, M_m$ are two sequences of terms. The proof keeps the terms in $N_1, \ldots, N_n$ secret, while the terms $M_1, \ldots, M_m$ are revealed.

Statements. The *statements* conveyed by zero-knowledge proofs are special Boolean formulas ranged over by S. Statements are formed using equalities between terms, a special $\rightsquigarrow$ predicate capturing the destructor reduction relation, as well as conjunctions and disjunctions of such basic statements. The syntax of *statements* is given in Table 7.

Table 7 Syntax of statements

$S, P ::=$		statements
	$g(M_1, \ldots, M_n) \rightsquigarrow N$	destructor reduction
	$M = N$	term equality
	$S_1 \wedge S_2$	conjunction
	$S_1 \vee S_2$	disjunction
	true	
	false	

The statement S used in a term $\mathsf{zk}_{n,m,S}(N_1, \ldots, N_n; M_1, \ldots, M_m)$ is called an (n, m)-*statement*. It does not contain names or variables, and uses the placeholders α_i and β_j, with $i \in [1, n]$ and $j \in [1, m]$, to refer to the secret terms N_i and public terms M_j. For instance, the zero-knowledge term $\mathsf{zk}_{1,2,(\mathsf{dec}(\mathsf{enc}(\beta_1,\beta_2),\alpha_1)) \rightsquigarrow \beta_1}(\ k\ ;\ m, \mathsf{pk}(k)\ )$ proves the knowledge of the decryption key k corresponding to the public encryption key $\mathsf{pk}(k)$. More precisely, the statement reads: "There exists a secret key k such that the decryption of the ciphertext $\mathsf{enc}(m, \mathsf{pk}(k))$ with this key yields m". As mentioned before, m and $\mathsf{pk}(k)$ are revealed by the proof while k is kept secret.

Verifying Zero-knowledge Proofs. The destructor $\mathsf{ver}_{n,m,l,S}$ verifies the validity of a zero-knowledge proof. It takes as arguments a proof together with l terms that are matched against the first l arguments in the public component of the proof. If the proof is valid, then $\mathsf{ver}_{n,m,l,S}$ returns the other $m - l$ public arguments. A proof is valid if and only if the statement obtained by substituting all α_i's and β_j's in S with the corresponding values N_i and M_j is valid.

The semantics of statements is defined in Table 8. The semantics of the $\rightsquigarrow$ predicate is defined in terms of the reduction relation for destructors, unless the destructor is ver in which

Table 8 Semantics of statements $[\![S]\!] \in \{\mathsf{true}, \mathsf{false}\}$

$$
\begin{array}{rcl}
[\![g(\widetilde{M}) \rightsquigarrow N]\!] & = & \begin{cases} \mathsf{true} & \text{if } g \neq \mathsf{ver} \text{ and } g(\widetilde{M}) \Downarrow N \\ \mathsf{false} & \text{otherwise} \end{cases} \\
[\![M = N]\!] & = & \begin{cases} \mathsf{true} & \text{if } M = N \\ \mathsf{false} & \text{otherwise} \end{cases} \\
[\![S_1 \wedge S_2]\!] & = & [\![S_1]\!] \wedge [\![S_2]\!] \\
[\![S_1 \vee S_2]\!] & = & [\![S_1]\!] \vee [\![S_2]\!] \\
[\![\mathsf{true}]\!] & = & \mathsf{true} \\
[\![\mathsf{false}]\!] & = & \mathsf{false}
\end{array}
$$

case the statement is simply false[9].

A.4 Processes

Additional to the processes from [14] and [4], we have an if process that tests two terms for equality ($\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$) and an elimination construct for union types ($\mathsf{case}\ x = M\ \mathsf{in}\ P$).

As in [14], the processes $\mathsf{assume}\ C$ and $\mathsf{assert}\ C$, where C is a logical formula, are used to express authorization policies, and do not have any computational significance. Assumptions are used to mark security-related events in processes, and also to express global authorization policies. Assertions specify logical formulas that are supposed to be entailed at run-time by the currently active assumptions.

Table 9 Syntax of processes

P, Q, R ::=		processes
	$\mathsf{out}(M, N).P$	output
	$\mathsf{in}(M, x).P$	input
	$!\mathsf{in}(M, x).P$	replicated input
	$\mathsf{new}\ a : T.P$	restriction
	$P \mid Q$	parallel composition
	$\mathbf{0}$	null process
	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$	destructor evaluation
	$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ P$	pair splitting
	$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	equality check
	$\mathsf{case}\ x = M\ \mathsf{in}\ P$	elimination of unions
	$\mathsf{assume}\ C$	assume formula
	$\mathsf{assert}\ C$	expect formula to hold

Notation: We use $\mathsf{let}\ \langle \widetilde{x} \rangle = M\ \mathsf{in}\ P$ to denote $\mathsf{let}\ (x_1, y_1) = M\ \mathsf{in}\ \mathsf{let}\ (x_2, y_2) = y_1\ \mathsf{in}\ \ldots \mathsf{let}\ (x_n, y_n) = y_{n-1}\ \mathsf{in}\ \mathsf{if}\ y_n = \mathsf{ok}\ \mathsf{then}\ P$, where $\widetilde{y}$ are fresh variables.

[9] This is necessary to avoid circularity, since the ver destructor can also appear inside statements.

A.5 Authorization Logic

Our calculus and type system are largely independent of the exact choice of authorization logic. We assume that the logical entailment relation $A \models C$ is expansive, monotonous, idempotent and closed under substitution of terms for variables. The equality in the logic needs to be an equivalence relation, and the logic must allow replacing equals by equals. The spi calculus terms form a free algebra in the logic. The logic needs to support function symbols and pairs; however, they do not need to be first-class, it is enough if we can encode them faithfully (e.g., it is easy to encode pairs in first-order logic). The destructor reduction relation has to be faithfully encoded in the logic (e.g., as a binary predicate satisfying corresponding axioms). Finally, the logic is assumed to include some of the usual logical connectives of classical first-order logic with their canonical meaning: true, false, $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\exists$, and $\forall$. Note that, unlike in the earlier versions of this type system [4], **we assume that the logic is classical** and not intuitionistic – the law of the excluded middle is an explicit prerequisite on the logic.

Proposition A.1 (Logical Entailment: Assumptions)

Expansivity: *$C \in A$ implies that $A \models C$;*

Monotonicity: *$A \models C$ and $A \subseteq A'$ then $A' \models C$;*

Idempotence: *$A \models A'$ and $A \cup A' \models C$ then $A \models C$;*

Substitution: *$A \models C$ then $A\sigma \models C\sigma$;*

Reflexivity: *$\models M = M$;*

Symmetry: *If $A \models N = M$ then $A \models M = N$;*

Transitivity: *If $A \models N = M$ and $A \models M = L$ then $A \models N = L$;*

Replacement: *$A \models M = N$ and $A \models C\{M/x\}$ imply that also $A \models C\{N/x\}$;*

Statements: *$[\![S]\!] = \mathsf{true}$ if and only if $\emptyset \models S$.*

Pairs: *If $A \models (M, N) = (M', N')$ then $A \models M = M'$ and $A \models N = N'$;*

True: *$\models \mathsf{true}$;*

False: *$\mathsf{false} \models A$;*

Equals-True: *$A \models C = \mathsf{true}$ if and only if $A \models C$;*

And: *$A \models C \wedge C'$ if and only if $A \models C$ and $A \models C'$;*

Or: *$A \models C \vee C'$ if and only if $A \models C$ or $A \models C'$;*

Implication: *$A \models C' \Rightarrow C$ if and only if $A, C' \models C$;*

Contradiction: *If $A \models C$ and $A \models \neg C$ then $A \models \mathsf{false}$;*

Excluded middle: *$\models C \vee \neg C$;*

Existential: *If $A \models C\{M/x\}$ then $A \models \exists x.C$;*

Universal: *Assuming that $x \notin A$ we have $A \models \forall x.C$ if and only if $A \models C$.*

In our implementation we consider classical first-order logic with equality as the authorization logic and we use various automated theorem provers to discharge the proof obligations generated by our type system.

A.6 Operational Semantics

The semantics of the calculus is standard and is defined by the usual structural equivalence ($P \equiv Q$) and an internal reduction relation ($P \to Q$). *Structural equivalence* relates the processes that are considered equivalent up to syntactic re-arrangement. It is the smallest equivalence relation satisfying the rules in Table 10.

Table 10 Structural equivalence $P \equiv Q$

(Eq-Zero-Id)	$P \mid \mathbf{0}$	$\equiv$	P
(Eq-Par-Comm)	$P \mid Q$	$\equiv$	$Q \mid P$
(Eq-Par-Assoc)	$(P \mid Q) \mid R$	$\equiv$	$P \mid (Q \mid R)$
(Eq-Scope)	$\mathsf{new}\ a : T.(P \mid Q)$	$\equiv$	$P \mid \mathsf{new}\ a : T.Q$, *if* $a \notin fn(P)$
(Eq-Bind-Swap)	$\mathsf{new}\ a_1 : T_1.\mathsf{new}\ a_2 : T_2.P$	$\equiv$	$\mathsf{new}\ a_2 : T_2.\mathsf{new}\ a_1 : T_1.P$, *if* $a_1 \neq a_2$
(Eq-Ctxt)	$\mathcal{E}[P]$	$\equiv$	$\mathcal{E}[Q]$, *if* $P \equiv Q$

Where $\mathcal{E}$ stands for an evaluation context, i.e., a context of the form $\mathcal{E} = \mathsf{new}\ \widetilde{a} : \widetilde{T}.([\,] \mid P)$.

Internal reduction is the smallest relation on closed processes satisfying the rules in Table 11.

Table 11 Internal reduction $P \to Q$

(Red-I/O)	$\mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q$	$\to$	$P \mid Q\{M/x\}$
(Red-!I/O)	$\mathsf{out}(a, M).P \mid !\mathsf{in}(a, x).Q$	$\to$	$P \mid Q\{M/x\} \mid !\mathsf{in}(a, x).Q$
(Red-Destr)	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$	$\to$	$P\{N/x\}$, *if* $g(\widetilde{M}) \Downarrow N$
(Red-Else)	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$	$\to$	Q, *if* $g(\widetilde{M}) \not\Downarrow$
(Red-Split)	$\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ P$	$\to$	$P\{M/x\}\{N/y\}$
(Red-If)	$\mathsf{if}\ M = M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	$\to$	P
(Red-Else)	$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	$\to$	Q, *if* $M \neq N$
(Red-CASE)	$\mathsf{case}\ x = M\ \mathsf{in}\ P$	$\to$	$P\{M/x\}$
(Red-Ctxt)	$\mathcal{E}[P]$	$\to$	$\mathcal{E}[Q]$, *if* $P \to Q$
(Red-Eq)	$P \to Q$, *if* $P \equiv P', P' \to Q'$, and $Q' \equiv Q$		

A.7 Robust Safety

A process is safe if and only if all its assertions are entailed by the active assumptions in every protocol execution.

Definition A.2 ***(Safety)*** *A closed process P is* safe *if and only if for every C and Q such that $P \to^* \mathsf{new}\ \widetilde{a} : \widetilde{T}.(\mathsf{assert}\ C \mid Q)$, there exists an evaluation context $\mathcal{E} = \mathsf{new}\ \widetilde{b} : \widetilde{U}.[\,] \mid Q'$ such that $Q \equiv \mathcal{E}[\mathsf{assume}\ C_1 \mid \ldots \mid \mathsf{assume}\ C_n]$, $fn(C) \cap \widetilde{b} = \emptyset$, and we have that $\{C_1, \ldots, C_n\} \models C$.*

A process is robustly safe if it is safe when run in parallel with an arbitrary opponent. Our type system guarantees that if a process is well-typed, then it is also robustly safe.

Definition A.3 ***(Opponent)*** *A closed process is an* opponent *if it does not contain any* assert *and if the only type occurring therein is* Un.

Definition A.4 ***(Robust Safety)*** *A closed process P is* robustly safe *if and only if $P \mid O$ is safe for every opponent O.*

B Type System for Zero-knowledge

Table 12 lists the types of our type system, while in Table 13 we list the typing judgements. The well-formed environment judgement is defined in Table 14.

The kinding rules are given in Table 16, and the subtyping rules in Table 18.

Tables 20 and 21 are devoted to the types of the constructors and destructors considered in this paper. In Table 22 we define the term typing judgement. Table 23 lists the rules for typing processes, which are defined using the auxiliary statement verification predicate (Table 26) and the environment extraction relation (Table 24).

Table 12 Types

$T, U ::=$	$\top$	$T \wedge U$	Un	$\mathsf{SigKey}(T)$	$\mathsf{PubKey}(T)$	$\mathsf{Hash}(T)$	$\mathsf{Stm}(\widetilde{y} : \widetilde{T}, \exists \widetilde{x}.C)$
		$T \vee U$	$\mathsf{Ch}(T)$	$\mathsf{VerKey}(T)$	$\mathsf{PrivKey}(T)$	$\mathsf{SymKey}(T)$	$\mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C)$
		$\{x : T \mid C\}$	$\mathsf{Pair}(x : T, U)$	$\mathsf{Signed}(T)$	$\mathsf{PubEnc}(T)$	$\mathsf{SymEnc}(T)$	

Notations: Let $\mathsf{Private}$ denote the type $\mathsf{Ch}(\top)$, and type $\bot$ denote $\{x : \mathsf{Un} \mid \mathsf{false}\}$.
Let $\{C\}$ denote the "OK"-type $\{x : \mathsf{Un} \mid C\}$, and $\{\!|\, C \,|\!\}$ denote the refinement type $\{x : \top \mid C\}$, where in both cases $x \notin fv(C)$.
Let $\widetilde{x} : \widetilde{T}$ denote $x_1 : T_1, \ldots, x_n : T_n$ for some n.
Let $\langle \widetilde{x} : \widetilde{T} \rangle \{C\}$ denote the type $\mathsf{Pair}(x_1 : T_1, \ldots \mathsf{Pair}(x_n : T_n, \{C\}))$ where we additionally require that the variables $\widetilde{x}$ are not bound in $\widetilde{T}$ (they can be bound in C though).

Table 13 Typing Judgements

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

Notation: We use $\Gamma \vdash \mathcal{J}$ to denote a judgement where $\mathcal{J} \in \{\diamond, T, C, T :: k, T <: U, M : T, P\}$

Table 14 Well-formed environments and types $\Gamma \vdash \diamond$ and $\Gamma \vdash T$

ENV-EMPTY
$\emptyset \vdash \diamond$

ENV-BINDING
$$\frac{\Gamma \vdash \diamond \qquad u \notin dom(\Gamma) \qquad free(T) \subseteq dom(\Gamma)}{\Gamma, u : T \vdash \diamond}$$

TYPE
$$\frac{\Gamma \vdash \diamond \qquad free(T) \subseteq dom(\Gamma)}{\Gamma \vdash T}$$

Definition: $dom(\emptyset) = \emptyset$; $dom(\Gamma, u : T) = dom(\Gamma), u$

Convention: We will very often disregard the order in $dom(\Gamma)$ and use it as a set.

Table 15 Entailed formula $\Gamma \models C$

$$\frac{\Gamma \vdash \diamond \qquad \mathit{free}(C) \subseteq \mathit{dom}(\Gamma) \qquad \mathit{forms}(\Gamma) \models C}{\Gamma \models C}\ \textsc{Entailed}$$

Definition:

$$\begin{aligned}
&\mathit{forms}(y : \{x : T \mid C\}) = \{C\{y/x\}\} \cup \mathit{forms}(y : T)\\
&\mathit{forms}(y : T_1 \wedge T_2) = \mathit{forms}(y : T_1) \cup \mathit{forms}(y : T_2)\\
&\mathit{forms}(y : T_1 \vee T_2) = \{C_1 \vee C_2 \mid C_1 \in \mathit{forms}(y : T_1), C_2 \in \mathit{forms}(y : T_2)\}\\
&\mathit{forms}(\Gamma_1, \Gamma_2) = \mathit{forms}(\Gamma_1) \cup \mathit{forms}(\Gamma_2)\\
&\mathit{forms}(\Gamma) = \emptyset, \text{ otherwise}
\end{aligned}$$

Table 16 Kinding ($k \in \{\mathsf{pub}, \mathsf{tnt}\}$) $\Gamma \vdash T :: k$

$$\textsc{Kind-Top-Tnt}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \top :: \mathsf{tnt}} \qquad \textsc{Kind-Top-Pub}\ \frac{\Gamma \models \mathsf{false}}{\Gamma \vdash \top :: \mathsf{pub}} \qquad \textsc{Kind-And-Pub1}\ \frac{\Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash T \wedge U :: \mathsf{pub}} \qquad \textsc{Kind-And-Pub2}\ \frac{\Gamma \vdash U :: \mathsf{pub}}{\Gamma \vdash T \wedge U :: \mathsf{pub}}$$

$$\textsc{Kind-And-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt} \qquad \Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T \wedge U :: \mathsf{tnt}} \qquad \textsc{Kind-Or-Pub}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash U :: \mathsf{pub}}{\Gamma \vdash T \vee U :: \mathsf{pub}} \qquad \textsc{Kind-Or-Tnt1}\ \frac{\Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash T \vee U :: \mathsf{tnt}}$$

$$\textsc{Kind-Or-Tnt2}\ \frac{\Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T \vee U :: \mathsf{tnt}} \qquad \textsc{Kind-Refine-Pub}\ \frac{\Gamma \vdash \{x : T \mid C\} \qquad \Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash \{x : T \mid C\} :: \mathsf{pub}}$$

$$\textsc{Kind-Refine-Empty-Pub}\ \frac{\Gamma \vdash \{x : T \mid C\} \qquad \Gamma, x : T \models \neg C}{\Gamma \vdash \{x : T \mid C\} :: \mathsf{pub}} \qquad \textsc{Kind-Refine-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt} \qquad \Gamma, x : T \models C}{\Gamma \vdash \{x : T \mid C\} :: \mathsf{tnt}} \qquad \textsc{Kind-Un}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Un} :: k}$$

$$\textsc{Kind-Chan}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{Ch}(T) :: k} \qquad \textsc{Kind-Pair}\ \frac{\Gamma \vdash T :: k \qquad \Gamma, x : T \vdash T' :: k}{\Gamma \vdash \mathsf{Pair}(x : T, T') :: k} \qquad \textsc{Kind-Signed-Pub}\ \frac{\Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash \mathsf{Signed}(T) :: \mathsf{pub}}$$

$$\textsc{Kind-Signed-Tnt}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{Signed}(T) :: \mathsf{tnt}} \qquad \textsc{Kind-SigKey}\ \frac{\Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{SigKey}(T) :: k} \qquad \textsc{Kind-VerKey-Pub}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{VerKey}(T) :: \mathsf{pub}} \qquad \textsc{Kind-VerKey-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{VerKey}(T) :: \mathsf{tnt}}$$

$$\textsc{Kind-PubEnc-Pub}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{PubEnc}(T) :: \mathsf{pub}} \qquad \textsc{Kind-PubEnc-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{PubEnc}(T) :: \mathsf{tnt}} \qquad \textsc{Kind-PubKey-Pub}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{PubKey}(T) :: \mathsf{pub}}$$

$$\textsc{Kind-PubKey-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash \mathsf{PubKey}(T) :: \mathsf{tnt}} \qquad \textsc{Kind-PrivKey}\ \frac{\Gamma \vdash T :: \mathsf{pub}}{\Gamma \vdash \mathsf{PrivKey}(T) :: k} \qquad \textsc{Kind-Hash-Pub}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{Hash}(T) :: \mathsf{pub}} \qquad \textsc{Kind-Hash-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{Hash}(T) :: \mathsf{tnt}}$$

$$\textsc{Kind-SymKey}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash T :: \mathsf{tnt}}{\Gamma \vdash \mathsf{SymKey}(T) :: k} \qquad \textsc{Kind-SymEnc}\ \frac{\Gamma \vdash T}{\Gamma \vdash \mathsf{SymEnc}(T) :: k} \qquad \textsc{Kind-Stm}\ \frac{\Gamma \models \mathsf{false}}{\Gamma \vdash \mathsf{Stm}(\widetilde{y} : \widetilde{T}, \exists \widetilde{x}.C) :: k}$$

$$\textsc{Kind-Zk}\ \frac{\forall i \in [1, m].\ \Gamma \vdash T_i :: k \qquad \text{if } k = \mathsf{tnt} \text{ then } \Gamma, \widetilde{y} : \widetilde{T} \models \exists \widetilde{x}.C \text{ else } \Gamma, \widetilde{y} : \widetilde{T} \vdash \{\exists \widetilde{x}.C\}}{\Gamma \vdash \mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C) :: k}$$

Derived rules:

$$\textsc{Kind-Ok-Pub}\ \frac{\Gamma \vdash \{C\}}{\Gamma \vdash \{C\} :: \mathsf{pub}} \qquad \textsc{Kind-Ok-Tnt}\ \frac{\Gamma \models C}{\Gamma \vdash \{C\} :: \mathsf{tnt}} \qquad \textsc{Kind-Tuple-Pub}\ \frac{\forall i.\ \Gamma \vdash T_i :: \mathsf{pub} \qquad \Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\}}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} :: \mathsf{pub}}$$

$$\textsc{Kind-Tuple-Tnt}\ \frac{\forall i.\ \Gamma \vdash T_i :: \mathsf{tnt} \qquad \Gamma, \widetilde{x} : \widetilde{T} \models C}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} :: \mathsf{tnt}}$$

Property: If $\Gamma \models \mathsf{false}$ then for all T and k if $\Gamma \vdash T$ then $\Gamma \vdash T :: k$.

Table 17 Logical Characterization of Kinding pub and tnt

$\mathsf{pub}(\top) = \mathsf{false}$	$\mathsf{tnt}(\top) = \mathsf{true}$
$\mathsf{pub}(T \wedge U) = \mathsf{pub}(U) \vee \mathsf{pub}(T)$	$\mathsf{tnt}(T \wedge U) = \mathsf{tnt}(U) \wedge \mathsf{tnt}(T)$
$\mathsf{pub}(T \vee U) = \mathsf{pub}(U) \wedge \mathsf{pub}(T)$	$\mathsf{tnt}(T \vee U) = \mathsf{tnt}(U) \vee \mathsf{tnt}(T)$
$\mathsf{pub}(\{x : T \mid C\}) = \mathsf{pub}(T) \vee \forall x.\neg C$	$\mathsf{tnt}(\{x : T \mid C\}) = \mathsf{tnt}(T) \wedge \forall x.\ C$
$\mathsf{pub}(\mathsf{Un}) = \mathsf{true}$	$\mathsf{tnt}(\mathsf{Un}) = \mathsf{true}$
$\mathsf{pub}(\mathsf{Ch}(T)) = \mathsf{pub}(T) \wedge \mathsf{tnt}(T)$	$\mathsf{tnt}(\mathsf{Ch}(T)) = \mathsf{pub}(T) \wedge \mathsf{tnt}(T)$
$\mathsf{pub}(\mathsf{Pair}(x : T, U)) = \mathsf{pub}(T) \wedge \forall x.\ \mathsf{pub}(U)$	$\mathsf{tnt}(\mathsf{Pair}(x : T, U)) = \mathsf{tnt}(T) \wedge \forall x.\ \mathsf{tnt}(U)$
$\mathsf{pub}(\mathsf{Signed}(T)) = \mathsf{pub}(T)$	$\mathsf{tnt}(\mathsf{Signed}(T)) = \mathsf{true}$
$\mathsf{pub}(\mathsf{SigKey}(T)) = \mathsf{tnt}(T)$	$\mathsf{tnt}(\mathsf{SigKey}(T)) = \mathsf{tnt}(T)$
$\mathsf{pub}(\mathsf{VerKey}(T)) = \mathsf{true}$	$\mathsf{tnt}(\mathsf{VerKey}(T)) = \mathsf{tnt}(T)$
$\mathsf{pub}(\mathsf{PubEnc}(T)) = \mathsf{true}$	$\mathsf{tnt}(\mathsf{PubEnc}(T)) = \mathsf{tnt}(T)$
$\mathsf{pub}(\mathsf{PubKey}(T)) = \mathsf{true}$	$\mathsf{tnt}(\mathsf{PubKey}(T)) = \mathsf{pub}(T)$
$\mathsf{pub}(\mathsf{PrivKey}(T)) = \mathsf{pub}(T)$	$\mathsf{tnt}(\mathsf{PrivKey}(T)) = \mathsf{pub}(T)$
$\mathsf{pub}(\mathsf{Hash}(T)) = \mathsf{true}$	$\mathsf{tnt}(\mathsf{Hash}(T)) = \mathsf{tnt}(T)$
$\mathsf{pub}(\mathsf{SymKey}(T)) = \mathsf{pub}(T) \wedge \mathsf{tnt}(T)$	$\mathsf{tnt}(\mathsf{SymKey}(T)) = \mathsf{pub}(T) \wedge \mathsf{tnt}(T)$
$\mathsf{pub}(\mathsf{SymEnc}(T)) = \mathsf{true}$	$\mathsf{tnt}(\mathsf{SymEnc}(T)) = \mathsf{true}$
$\mathsf{pub}(\mathsf{ZKProof}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C)) = \wedge_i \mathsf{pub}(T_i)$	$\mathsf{tnt}(\mathsf{ZKProof}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C)) = \wedge_i \mathsf{tnt}(T_i) \wedge \exists \widetilde{x}.C$
$\mathsf{pub}(\mathsf{Stm}(\widetilde{y} : \widetilde{T}, \exists \widetilde{x}.C)) = \mathsf{false}$	$\mathsf{tnt}(\mathsf{Stm}(\widetilde{y} : \widetilde{T}, \exists \widetilde{x}.C)) = \mathsf{false}$

Property: Assuming that $\Gamma \vdash T$ then $\Gamma \vdash T :: k$ if and only if $\Gamma \models k(T)$, where $k \in \{\mathsf{pub}, \mathsf{tnt}\}$.

Table 18 Subtyping $\Gamma \vdash T <: U$

$$\textsc{Sub-Pub-Tnt}\ \frac{\Gamma \vdash T :: \mathsf{pub} \qquad \Gamma \vdash U :: \mathsf{tnt}}{\Gamma \vdash T <: U} \qquad \textsc{Sub-Refl}\ \frac{\Gamma \vdash T}{\Gamma \vdash T <: T} \qquad \textsc{Sub-Top}\ \frac{\Gamma \vdash T}{\Gamma \vdash T <: \top} \qquad \textsc{Sub-And1}\ \frac{\Gamma \vdash U \qquad \Gamma \vdash T <: T'}{\Gamma \vdash T \wedge U <: T'}$$

$$\textsc{Sub-And2}\ \frac{\Gamma \vdash T \qquad \Gamma \vdash U <: T'}{\Gamma \vdash T \wedge U <: T'} \qquad \textsc{Sub-And3}\ \frac{\Gamma \vdash T <: T_1 \qquad \Gamma \vdash T <: T_2}{\Gamma \vdash T <: T_1 \wedge T_2} \qquad \textsc{Sub-Or1}\ \frac{\Gamma \vdash T_1 <: T \qquad \Gamma \vdash T_2 <: T}{\Gamma \vdash T_1 \vee T_2 <: T}$$

$$\textsc{Sub-Or2}\ \frac{\Gamma \vdash U \qquad \Gamma \vdash T' <: T}{\Gamma \vdash T' <: T \vee U} \qquad \textsc{Sub-Or3}\ \frac{\Gamma \vdash T \qquad \Gamma \vdash T' < U}{\Gamma \vdash T' <: T \vee U} \qquad \textsc{Sub-Refine-Left}\ \frac{\Gamma \vdash \{x : T \mid C\} \qquad \Gamma \vdash T <: T'}{\Gamma \vdash \{x : T \mid C\} <: T'}$$

$$\textsc{Sub-Refine-Empty}\ \frac{\Gamma \vdash \{x : T \mid C\} \qquad \Gamma, x : T \models \neg C}{\Gamma \vdash \{x : T \mid C\} <: T'} \qquad \textsc{Sub-Refine-Right}\ \frac{\Gamma \vdash T <: T' \qquad \Gamma, x : T \models C}{\Gamma \vdash T <: \{x : T' \mid C\}} \qquad \textsc{Sub-Chan-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{Ch}(T) <: \mathsf{Ch}(U)}$$

$$\textsc{Sub-Pair-Cov}\ \frac{\Gamma \vdash T_1 <: U_1 \qquad \Gamma, x : T_1 \vdash T_2 <: U_2}{\Gamma \vdash \mathsf{Pair}(x : T_1, T_2) <: \mathsf{Pair}(x : U_1, U_2)} \qquad \textsc{Sub-Signed-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{Signed}(T) <: \mathsf{Signed}(U)} \qquad \textsc{Sub-SigKey-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{SigKey}(T) <: \mathsf{SigKey}(U)}$$

$$\textsc{Sub-VerKey-Cov}\ \frac{\Gamma \vdash T <: U}{\Gamma \vdash \mathsf{VerKey}(T) <: \mathsf{VerKey}(U)} \qquad \textsc{Sub-PubEnc-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{PubEnc}(T) <: \mathsf{PubEnc}(U)} \qquad \textsc{Sub-PubKey-Con}\ \frac{\Gamma \vdash U <: T}{\Gamma \vdash \mathsf{PubKey}(T) <: \mathsf{PubKey}(U)}$$

$$\textsc{Sub-PrivKey-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{PrivKey}(T) <: \mathsf{PrivKey}(U)} \qquad \textsc{Sub-Hash-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{Hash}(T) <: \mathsf{Hash}(U)} \qquad \textsc{Sub-SymKey-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{SymKey}(T) <: \mathsf{SymKey}(U)}$$

$$\textsc{Sub-SymEnc-Inv}\ \frac{\Gamma \vdash T <:> U}{\Gamma \vdash \mathsf{SymEnc}(T) <: \mathsf{SymEnc}(U)} \qquad \textsc{Sub-Zk-Cov}\ \frac{\forall i.\ \Gamma \vdash T_i <: U_i \qquad \Gamma, \widetilde{y} : \widetilde{T}, \_ : \{\exists \widetilde{x}.C\} \models \exists \widetilde{x}.C'}{\Gamma \vdash \mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C) <: \mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}.C')}$$

Derived rules:

$$\textsc{Sub-Refine}\ \frac{\Gamma \vdash T <: T' \qquad \Gamma, x : \{x : T \mid C\} \models C'}{\Gamma \vdash \{x : T \mid C\} <: \{x : T' \mid C'\}} \qquad \textsc{Sub-Ok}\ \frac{\Gamma, \_ : \{C\} \models C'}{\Gamma \vdash \{C\} <: \{C'\}}$$

$$\textsc{Sub-Tuple}\ \frac{\forall i.\ \Gamma \vdash T_i <: U_i \qquad \Gamma, \widetilde{x} : \widetilde{T}, \_ : \{C\} \models C'}{\Gamma \vdash \langle \widetilde{x} : \widetilde{T} \rangle \{C\} <: \langle \widetilde{x} : \widetilde{U} \rangle \{C'\}}$$

Notation: $\Gamma \vdash T <:> U$ iff $\Gamma \vdash T <: U$ and $\Gamma \vdash U <: T$.
Property: If $\Gamma \models \mathsf{false}$ then for all T and U if $\Gamma \vdash T$ and $\Gamma \vdash U$ then $\Gamma \vdash T <: U$.

Table 19 Logical Characterization of Subtyping sub

$$\mathsf{sub}(T, T) = \mathsf{true}$$

$$\mathsf{sub}(T, \top) = \mathsf{true}$$

$$\mathsf{sub}(U_1 \vee U_2, T_1 \wedge T_2) = (\mathsf{sub}(U_1 \vee U_2, T_1) \wedge \mathsf{sub}(U_1 \vee U_2, T_2)) \\ \vee (\mathsf{sub}(U_1, T_1 \wedge T_2) \wedge \mathsf{sub}(U_2, T_1 \wedge T_2)) \vee (\mathsf{pub}(U_1 \vee U_2) \wedge \mathsf{tnt}(T_1 \wedge T_2))$$

$$\mathsf{sub}(\{x : T \mid C\}, T_1 \wedge T_2) = (\mathsf{sub}(\{x : T \mid C\}, T_1) \wedge \mathsf{sub}(\{x : T \mid C\}, T_2)) \\ \vee \mathsf{sub}(T, T_1 \wedge T_2) \vee \forall x.\neg C \vee (\mathsf{pub}(\{x : T \mid C\}) \wedge \mathsf{tnt}(T_1 \wedge T_2))$$

$$\mathsf{sub}(U_1 \wedge U_2, T_1 \wedge T_2) = (\mathsf{sub}(U_1 \wedge U_2, T_1) \wedge \mathsf{sub}(U_1 \wedge U_2, T_2)) \\ \vee \mathsf{sub}(U_1, T_1 \wedge T_2) \vee \mathsf{sub}(U_2, T_1 \wedge T_2) \vee (\mathsf{pub}(U_1 \wedge U_2) \wedge \mathsf{tnt}(T_1 \wedge T_2))$$

$$\mathsf{sub}(T, T_1 \wedge T_2) = (\mathsf{sub}(T, T_1) \wedge \mathsf{sub}(T, T_2)) \vee (\mathsf{pub}(T) \wedge \mathsf{tnt}(T_1 \wedge T_2))$$

$$\mathsf{sub}(T_1 \vee T_2, \{x : T \mid C\}) = (\mathsf{sub}(T_1, \{x : T \mid C\}) \wedge \mathsf{sub}(T_2, \{x : T \mid C\})) \\ \vee (\mathsf{sub}(T_1 \vee T_2, T) \wedge \forall x.C) \vee (\mathsf{pub}(T_1 \vee T_2) \wedge \mathsf{tnt}(\{x : T \mid C\}))$$

$$\mathsf{sub}(T_1 \vee T_2, U_1 \vee U_2) = (\mathsf{sub}(T_1, U_1 \vee U_2) \wedge \mathsf{sub}(T_2, U_1 \vee U_2)) \\ \vee \mathsf{sub}(T_1 \vee T_2, U_1) \vee \mathsf{sub}(T_1 \vee T_2, U_2) \vee (\mathsf{pub}(T_1 \vee T_2) \wedge \mathsf{tnt}(U_1 \vee U_2))$$

$$\mathsf{sub}(T_1 \vee T_2, T) = (\mathsf{sub}(T_1, T) \wedge \mathsf{sub}(T_2, T)) \vee (\mathsf{pub}(T_1 \vee T_2) \wedge \mathsf{tnt}(T))$$

$$\mathsf{sub}(\{x : T \mid C\}, \{x : T' \mid C'\}) = (\mathsf{sub}(T, T') \wedge \forall x.\ C \Rightarrow C') \vee (\mathsf{pub}(\{x : T \mid C\}) \wedge \mathsf{tnt}(\{x : T' \mid C'\}))$$

$$\mathsf{sub}(\{x : T \mid C\}, T_1 \vee T_2) = \mathsf{sub}(T, T_1 \vee T_2) \vee (\forall x.\neg C) \vee \mathsf{sub}(\{x : T \mid C\}, T_1) \\ \vee \mathsf{sub}(\{x : T \mid C\}, T_2) \vee (\mathsf{pub}(\{x : T \mid C\}) \wedge \mathsf{tnt}(T_1 \vee T_2))$$

$$\mathsf{sub}(\{x : T \mid C\}, T') = \mathsf{sub}(T, T') \vee (\forall x.\neg C) \vee (\mathsf{pub}(\{x : T \mid C\}) \wedge \mathsf{tnt}(T'))$$

$$\mathsf{sub}(T_1 \wedge T_2, \{x : T \mid C\}) = (\mathsf{sub}(T_1 \wedge T_2, T) \wedge \forall x.\ C) \vee \mathsf{sub}(T_1, \{x : T \mid C\}) \\ \vee \mathsf{sub}(T_2, \{x : T \mid C\}) \vee (\mathsf{pub}(T_1 \wedge T_2) \wedge \mathsf{tnt}(\{x : T \mid C\}))$$

$$\mathsf{sub}(T', \{x : T \mid C\}) = (\mathsf{sub}(T', T) \wedge \forall x.\ C) \vee (\mathsf{pub}(T') \wedge \mathsf{tnt}(\{x : T \mid C\}))$$

$$\mathsf{sub}(U_1 \wedge U_2, T_1 \vee T_2) = \mathsf{sub}(U_1 \wedge U_2, T_1) \vee \mathsf{sub}(U_1 \wedge U_2, T_2) \\ \vee \mathsf{sub}(U_1, T_1 \vee T_2) \vee \mathsf{sub}(U_2, T_1 \vee T_2) \vee (\mathsf{pub}(U_1 \wedge U_2) \wedge \mathsf{tnt}(T_1 \vee T_2))$$

$$\mathsf{sub}(T_1 \wedge T_2, T) = \mathsf{sub}(T_1, T) \vee \mathsf{sub}(T_2, T) \vee (\mathsf{pub}(T_1 \wedge T_2) \wedge \mathsf{tnt}(T))$$

$$\mathsf{sub}(T, T_1 \vee T_2) = \mathsf{sub}(T, T_1) \vee \mathsf{sub}(T, T_2) \vee (\mathsf{pub}(T) \wedge \mathsf{tnt}(T_1 \vee T_2))$$

$$\mathsf{sub}(\mathsf{Ch}(T), \mathsf{Ch}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{Ch}(T)) \wedge \mathsf{tnt}(\mathsf{Ch}(U)))$$

$$\mathsf{sub}(\mathsf{Pair}(x : T, U), \mathsf{Pair}(x : T', U')) = (\mathsf{sub}(T, T') \wedge \forall x.\ \mathsf{sub}(U, U')) \vee (\mathsf{pub}(\mathsf{Pair}(x : T, U)) \wedge \mathsf{tnt}(\mathsf{Pair}(x : T', U')))$$

$$\mathsf{sub}(\mathsf{Signed}(T), \mathsf{Signed}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{Signed}(T)) \wedge \mathsf{tnt}(\mathsf{Signed}(U)))$$

$$\mathsf{sub}(\mathsf{SigKey}(T), \mathsf{SigKey}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{SigKey}(T)) \wedge \mathsf{tnt}(\mathsf{SigKey}(U)))$$

$$\mathsf{sub}(\mathsf{VerKey}(T), \mathsf{VerKey}(U)) = \mathsf{sub}(T, U) \vee (\mathsf{pub}(\mathsf{VerKey}(T)) \wedge \mathsf{tnt}(\mathsf{VerKey}(U)))$$

$$\mathsf{sub}(\mathsf{PubEnc}(T), \mathsf{PubEnc}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{PubEnc}(T)) \wedge \mathsf{tnt}(\mathsf{PubEnc}(U)))$$

$$\mathsf{sub}(\mathsf{PubKey}(T), \mathsf{PubKey}(U)) = \mathsf{sub}(U, T) \vee (\mathsf{pub}(\mathsf{PubKey}(T)) \wedge \mathsf{tnt}(\mathsf{PubKey}(U)))$$

$$\mathsf{sub}(\mathsf{PrivKey}(T), \mathsf{PrivKey}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{PrivKey}(T)) \wedge \mathsf{tnt}(\mathsf{PrivKey}(U)))$$

$$\mathsf{sub}(\mathsf{Hash}(T), \mathsf{Hash}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{Hash}(T)) \wedge \mathsf{tnt}(\mathsf{Hash}(U)))$$

$$\mathsf{sub}(\mathsf{SymKey}(T), \mathsf{SymKey}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{SymKey}(T)) \wedge \mathsf{tnt}(\mathsf{SymKey}(U)))$$

$$\mathsf{sub}(\mathsf{SymEnc}(T), \mathsf{SymEnc}(U)) = (\mathsf{sub}(T, U) \wedge \mathsf{sub}(U, T)) \vee (\mathsf{pub}(\mathsf{SymEnc}(T)) \wedge \mathsf{tnt}(\mathsf{SymEnc}(U)))$$

$$\mathsf{sub}(\mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C), \mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}.C')) = \wedge_i \mathsf{sub}(T_i, U_i) \wedge \exists \widetilde{x}.C \Rightarrow \exists \widetilde{x}.C' \\ \vee (\mathsf{pub}(\mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C)) \wedge \mathsf{tnt}(\mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}.C')))$$

$$\mathsf{sub}(T, U) = \mathsf{pub}(T) \wedge \mathsf{tnt}(U), \text{ otherwise}$$

Note: In the definition of sub each case applies only if none of the previous ones apply (like in most functional programming languages). 32

Property: Assuming that $\Gamma \vdash T$ and $\Gamma \vdash U$ then $\Gamma \vdash T <: U$ if and only if $\Gamma \models \mathsf{sub}(T, U)$.

Table 20 Typing Constructors $f : (T_1, \ldots, T_n) \mapsto U$

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

Table 21 Typing Destructors $g : (T_1, \ldots, T_n) \mapsto U$

$$
\begin{aligned}
\mathsf{dec} &: (\mathsf{PubEnc}(T), \mathsf{PrivKey}(T)) \mapsto T \\
\mathsf{check} &: (\mathsf{Signed}(T), \mathsf{VerKey}(T)) \mapsto T \\
\mathsf{sdec} &: (\mathsf{SymEnc}(T), \mathsf{SymKey}(T)) \mapsto T \\
\mathsf{public}_{n,m} &: (\mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{T}; \exists \widetilde{x}.C)) \mapsto \langle \widetilde{y} : \widetilde{T} \rangle \{\mathsf{true}\}
\end{aligned}
$$

Table 22 Typing Terms $\Gamma \vdash M : T$

$$\textsc{Env}\ \frac{\Gamma \vdash \diamond \qquad u : T \in \Gamma}{\Gamma \vdash u : T} \qquad \textsc{Sub}\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash M : T'} \qquad \textsc{And}\ \frac{\Gamma \vdash M : T \qquad \Gamma \vdash M : U}{\Gamma \vdash M : T \wedge U}$$

$$\textsc{Or1}\ \frac{\Gamma \vdash M : T}{\Gamma \vdash M : T \vee U} \qquad \textsc{Or2}\ \frac{\Gamma \vdash M : U}{\Gamma \vdash M : T \vee U} \qquad \textsc{Refine}\ \frac{\Gamma \vdash M : T \qquad \Gamma \models C\{M/x\}}{\Gamma \vdash M : \{x : T \mid C\}}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2\{M_1/x\}}{\Gamma \vdash (M_1, M_2) : \mathsf{Pair}(x : T_1, T_2)}$$

$$\textsc{Constr}\ \frac{f : (T_1, \ldots, T_n) \mapsto T \qquad f \neq \mathsf{zk} \qquad \forall i \in [1, n].\ \Gamma \vdash M_i : T_i}{\Gamma \vdash f(M_1, \ldots, M_n) : T}$$

$$\textsc{ZK}\ \frac{\begin{array}{c}\Gamma(s_{n,m,S}) = \mathsf{Stm}(y_1 : T_1, \ldots, y_m : T_m, \exists x_1, \ldots, x_n.C) \\ \forall i \in [1, n].\ \Gamma \vdash N_i : U_i \qquad \Gamma \vdash \langle M_1, \ldots, M_m \rangle : \langle y_1 : T_1, \ldots, y_m : T_m \rangle\{C\{\widetilde{N}/\widetilde{x}\}\}\end{array}}{\begin{array}{c}\Gamma \vdash \mathsf{zk}_{n,m,S}(N_1, \ldots, N_n; M_1, \ldots, M_m) : \\ \mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m \rangle\{\exists x_1, \ldots, x_n.C\})\end{array}}$$

$$\textsc{ZK-Un}\ \frac{\Gamma(s^{un}_{n,m,S}) = \mathsf{Stm}(\widetilde{y} : \widetilde{\mathsf{Un}}, \mathsf{true}) \qquad \forall i \in [1, n].\ \Gamma \vdash N_i : \mathsf{Un} \qquad \forall j \in [1, m].\ \Gamma \vdash M_i : \mathsf{Un}}{\Gamma \vdash \mathsf{zk}_{n,m,S}(N_1, \ldots, N_n; M_1, \ldots, M_m) : \mathsf{ZKProof}_{n,m,S}(\langle y_1 : \mathsf{Un}, \ldots, y_m : \mathsf{Un} \rangle\{\exists x_1, \ldots, x_n.\mathsf{true}\})}$$

Derived rules

$$\textsc{Tuple}\ \frac{\forall i \in [1, n].\ \Gamma \vdash M_i : T_i \qquad \Gamma \models C\{\widetilde{M}/\widetilde{x}\}}{\Gamma \vdash \langle M_1, \ldots, M_n \rangle : \langle x_1 : T_1, \ldots, x_n : T_n \rangle\{C\}}$$

Notation: We write $\Gamma(u)$ for the type T such that $\Gamma = \Gamma_1, u : T, \Gamma_2$ for some Γ_1 and Γ_2.

Table 23 Typing Processes $\Gamma \vdash P$

$$\textsc{Proc-Out}\ \frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(M, N).P} \qquad \textsc{Proc-(Repl)-In}\ \frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma, x : T \vdash P}{\Gamma \vdash [!]\mathsf{in}(M, x).P} \qquad \textsc{Proc-Stop}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

$$\textsc{Proc-New}\ \frac{T \in \{\mathsf{Un}, \mathsf{Ch}(U), \mathsf{SigKey}(U), \mathsf{PrivKey}(U)\} \qquad \Gamma, a : T \vdash P}{\Gamma \vdash \mathsf{new}\ a : T.P} \qquad \textsc{Proc-Par}\ \frac{\Gamma, \_ : \{\overline{P}\} \vdash Q \qquad \Gamma, \_ : \{\overline{Q}\} \vdash P}{\Gamma \vdash P \mid Q}$$

$$\textsc{Proc-Des}\ \frac{\begin{array}{c} g : (T_1, \ldots, T_n) \mapsto T \qquad g \neq \mathsf{ver} \qquad \forall i \in [1, n].\ \Gamma \vdash M_i : T_i \\ \Gamma, x : T, \_ : \{\mathsf{Red}(g(M_1, \ldots, M_n), x)\} \vdash P \\ \Gamma, \_ : \{\neg \exists x.\ \mathsf{Red}(g(M_1, \ldots, M_n), x)\} \vdash Q \end{array}}{\Gamma \vdash \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\textsc{Proc-Ver}\ \frac{\begin{array}{c} \Gamma \vdash N : \mathsf{ZKProof}_{n,m,S}(\widetilde{y} : \widetilde{U}; \exists \widetilde{x}.C') \qquad \Gamma(s_{n,m,S}) = \mathsf{Stm}(\widetilde{y} : \widetilde{T}, \exists \widetilde{x}.C) \\ \forall i \in [1, l].\ \Gamma \vdash M_i : T_i \qquad \mathsf{verify\text{-}stm}(S, \Gamma, n, m, l, \widetilde{y} : \widetilde{U}, \widetilde{y} : \widetilde{T}, \exists \widetilde{x}.C)\ \text{holds} \\ \Gamma, x : \langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m \rangle \{\exists \widetilde{x}.\ C\{M_i/y_i\}_{i \in [1,l]}\} \vdash P \qquad \Gamma \vdash Q \end{array}}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(N, M_1, \ldots, M_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\textsc{Proc-Ver-Un}\ \frac{\begin{array}{c} \Gamma(s^{un}_{n,m,S}) = \mathsf{Stm}(\widetilde{y} : \widetilde{\mathsf{Un}}, \mathsf{true}) \\ \Gamma \vdash N : \mathsf{Un} \qquad \forall i \in [1, l].\ \Gamma \vdash M_i : \mathsf{Un} \qquad \Gamma, x : \mathsf{Un} \vdash P \qquad \Gamma \vdash Q \end{array}}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(N, M_1, \ldots, M_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\textsc{Proc-Pair-Split}\ \frac{\Gamma \vdash M : \mathsf{Pair}(x : T, U) \qquad \Gamma, x : T, y : U, \_ : \{(x, y) = M\} \vdash P}{\Gamma \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ P}$$

$$\textsc{Proc-If}\ \frac{\Gamma \vdash N : T \qquad \Gamma \vdash M : U \qquad \Gamma, \_ : \{N = M \wedge \mathsf{non\text{-}disj}(T, U)\} \vdash P \qquad \Gamma, \_ : \{\neg(N = M)\} \vdash Q}{\Gamma \vdash \mathsf{if}\ N = M\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

$$\textsc{Proc-Case}\ \frac{\Gamma \vdash M : T \vee U \qquad \Gamma, x : T \vdash P \qquad \Gamma, x : U \vdash P}{\Gamma \vdash \mathsf{case}\ x = M\ \mathsf{in}\ P} \qquad \textsc{Proc-Assume}\ \frac{\Gamma \vdash \{C\}}{\Gamma \vdash \mathsf{assume}\ C} \qquad \textsc{Proc-Assert}\ \frac{\Gamma \models C}{\Gamma \vdash \mathsf{assert}\ C}$$

Derived rules:

$$\textsc{Proc-Tuple-Split}\ \frac{\begin{array}{c} \Gamma \vdash M : \langle y_1 : T_1, \ldots, y_n : T_n \rangle \{C\} \\ \Gamma, x_1 : T_1, \ldots, x_n : T_n, \_ : \{\langle x_1, \ldots, x_n \rangle = M \wedge C\{\widetilde{x}/\widetilde{y}\}\} \vdash P \end{array}}{\Gamma \vdash \mathsf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathsf{in}\ P}$$

Table 24 Formula Extraction $\overline{P}$

$\overline{\mathsf{new}\ a : T.P} = \exists a.\ \overline{P}$ $\quad$ $\overline{P \mid Q} = \overline{P} \wedge \overline{Q}$ $\quad$ $\overline{\mathsf{assume}\ C} = C$ $\quad$ $\overline{P} = \mathsf{true}$, otherwise

Table 25 Logical Characterization of Type Disjointness $\mathsf{non\text{-}disj}(T_1, T_2)$

$$\mathsf{non\text{-}disj}(T, \mathsf{Private}) = \mathsf{non\text{-}disj}(\mathsf{Private}, T) = \neg\mathsf{pub}(T)$$
$$\mathsf{non\text{-}disj}(T, U) = \mathsf{true}, \text{ otherwise}$$

Property: If there exists N so that $\Gamma \vdash N : T$ and $\Gamma \vdash N : U$ then $\Gamma \models \mathsf{non\text{-}disj}(T, U)$.

Table 26 Statement Verification

$\mathsf{verify\text{-}stm}(S, \Gamma, n, m, l, y_1 : U_1, \ldots, y_m : U_m, y_1 : T_1, \ldots, y_m : T_m, \exists x_1, \ldots, x_n.C)$ holds

if $\exists \Gamma''.\ [|S\{\widetilde{x}/\widetilde{\alpha}\}\{\widetilde{y}/\widetilde{\beta}\}|]_{\Gamma'}^{y_{l+1}:T_{l+1},\ldots,y_m:T_m;C} \mapsto \Gamma'',\ \Gamma'' \models C$ and $\forall j \in [l+1, m].\ \Gamma'' \vdash y_j : T_j$

where $\Gamma' = \Gamma, \widetilde{x} : \widetilde{\top}, y_1 : T_1, \ldots, y_l : T_l, y_{l+1} : U_{l+1}, \ldots, y_m : U_m, z_C : \top$ and $\Gamma' \vdash \diamond$

Let v range over $\widetilde{x} \cup \widetilde{y}$. We write $[|S|]_{\Gamma'}$ to denote $[|S|]_{\Gamma'}^{y_{l+1}:T_{l+1},\ldots,y_m:T_m;C}$

$[|S_1 \wedge S_2|]_{\Gamma'} \ \mapsto\ \Gamma_{12} \wedge \Gamma_{21}$, if $[|S_2|]_{\Gamma'} \mapsto \Gamma_2, [|S_1|]_{\Gamma_2} \mapsto \Gamma_{12}, [|S_1|]_{\Gamma'} \mapsto \Gamma_1$, and $[|S_2|]_{\Gamma_1} \mapsto \Gamma_{21}$

$[|S_1 \vee S_2|]_{\Gamma'} \ \mapsto\ \Gamma_1 \vee \Gamma_2$, if $[|S_1|]_{\Gamma'} \mapsto \Gamma_1$ and $[|S_2|]_{\Gamma'} \mapsto \Gamma_2$

$[|\mathsf{check}(v_M, v_K) \rightsquigarrow v_N|]_{\Gamma'} \ \mapsto\ \Gamma'[v_N : \{\!|\, (\mathsf{tnt}(T^*) \wedge \neg\mathsf{sub}(\Gamma'(v_M), \mathsf{Signed}(T^*))) \,|\!\} \vee T^*,$
$y_{k\in[l+1,m]} : \{\!|\, \mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \,|\!\} \vee T_k,$
$z_C : \{\!|\, (\mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \vee C) \ \wedge\ \mathsf{Red}(\mathsf{check}(v_M, v_K), v_N) \,|\!\}]$, if $\Gamma' \vdash v_K : \mathsf{VerKey}(T^*)$

$[|\mathsf{dec}(v_M, v_K) \rightsquigarrow v_N|]_{\Gamma'} \ \mapsto\ \Gamma'[v_N : \{\!|\, \mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \,|\!\} \vee T^*, y_{k\in[l+1,m]} : \{\!|\, \mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \,|\!\} \vee T_k,$
$z_C : \{\!|\, (\mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \vee C) \wedge \mathsf{Red}(\mathsf{dec}(v_M, v_K), v_N) \,|\!\}]$, if $\Gamma' \vdash v_M : \mathsf{PubEnc}(T^*)$

$[|v_M = \mathsf{hash}(v_N)|]_{\Gamma'} \ \mapsto\ \Gamma'[v_N : \{\!|\, \mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \,|\!\} \vee T^*, y_{k\in[l+1,m]} : \{\!|\, \mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \,|\!\} \vee T_k,$
$z_C : \{\!|\, (\mathsf{non\text{-}disj}(T^*, \mathsf{Un}) \vee C) \wedge v_M = \mathsf{hash}(v_N) \,|\!\}]$, if $\Gamma' \vdash v_M : \mathsf{Hash}(T^*)$

$[|v_N = \mathsf{enc}(v_M, v_K)|]_{\Gamma'} \ \mapsto\ \Gamma'[v_M : \{\!|\, \mathsf{tnt}(T^*) \,|\!\} \vee T^*, v_K : \{\!|\, \mathsf{tnt}(T^*) \,|\!\} \vee \mathsf{PubKey}(T^*),$
$z_C : \{\!|\, v_N = \mathsf{enc}(v_M, v_K) \,|\!\}]$, if $\Gamma' \vdash v_N : \mathsf{PubEnc}(T^*)$

$[|(v_M, v_L) = v_N|]_{\Gamma'} \ \mapsto\ \Gamma'[v_M : T, v_L : U\{v_M/x\}, z_C : \{\!|\, (v_M, v_L) = v_N \,|\!\}]$, if $\Gamma' \vdash v_N : \mathsf{Pair}(x : T, U)$

$[|\langle\widetilde{v_M}\rangle = v_N|]_{\Gamma'} \ \mapsto\ \Gamma'[\widetilde{v_M} : \widetilde{U}], z_C : \{\!|\, C^*\{\widetilde{v_M}/\widetilde{z}\} \wedge \langle\widetilde{v_M}\rangle = v_N \,|\!\}]$, if $\Gamma'(v_N) = \langle\widetilde{z} : \widetilde{U}\rangle\{C^*\}$

$[|S|]_{\Gamma'} \ \mapsto\ \Gamma'[v_C : \{\!|\, S \,|\!\}]$

Definitions: For all Γ_1 and Γ_2 such that $dom(\Gamma_1) = dom(\Gamma_2)$ (same variables and names in the same order), we define $\Gamma_1 \ \wedge\ \Gamma_2$ and $\Gamma_1 \ \vee\ \Gamma_2$ as follows:

$$\begin{array}{lcllcl} \emptyset \ \wedge\ \emptyset & = & \emptyset & (\Gamma_1, u : T) \ \wedge\ (\Gamma_2, u : U) & = & (\Gamma_1 \ \wedge\ \Gamma_2), u : T \wedge U \\ \emptyset \ \vee\ \emptyset & = & \emptyset & (\Gamma_1, u : T) \ \vee\ (\Gamma_2, u : U) & = & (\Gamma_1 \ \vee\ \Gamma_2), u : T \vee U \end{array}$$

Definition: $\Gamma[v : U] = \Gamma', v : T \wedge U, \Gamma''$; if $\Gamma = \Gamma', v : T, \Gamma''$

Note: Note that $[|S|]_{\Gamma'}$ is no longer a function but a relation.

C Transformation Algorithm

We present an algorithm that transform protocols according to our approach. Protocols will be expressed in the extensible spi calculus presented in Appendix A. We first present the algorithm that transforms untyped spi calculus processes (Section C.1). Then we present the transformation of the types (Section C.2).

C.1 Transforming 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:

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

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

  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.

$$\begin{aligned} \mathtt{labels_{in}}(\mathit{Prot}) &= \{1, 2\} \\ \mathtt{labels_{out}}(\mathit{Prot}) &= \{p1, p2, p3, p4, 1, 2\} \end{aligned}$$

- 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:

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

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

$$\texttt{variable}\,(Prot, 1) = x$$
$$\texttt{variable}\,(Prot, 2) = y$$

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

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

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

$$\texttt{output}\,(Prot, 1) = \big(ch, \mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big)\big)$$
$$\texttt{output}\,(Prot, 2) = \big(ch, \mathsf{sign}\big(\mathsf{enc}\big((u, x_3), \mathsf{pk}(k_S)\big), k_{PS}\big)\big)$$

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

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

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.

$$\texttt{restricted}\,(Prot, 1) = \{q, p, k_U, k_{PE}, k_{PS}, k_S\}$$
$$\texttt{restricted}\,(Prot, 2) = \{p, k_U, k_{PE}, k_{PS}, k_S\}$$

- 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.

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

We say $C\,[]$ is a `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.

$$\begin{aligned}
&\texttt{deseva}\,(Prot, 1) = \varepsilon \\
&\texttt{deseva}\,(Prot, 2) = \{x_3/x_1[2]_2\}\{x_2/x_1[1]_2\}\{\mathsf{check}(x, \mathsf{vk}(k_U))/x_1\}
\end{aligned}$$

- $\texttt{compile}_{\texttt{output}}$ 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.

$$\begin{aligned}
&\texttt{compile}_{\texttt{output}}\,(Prot, 1) = \Big(ch, \mathsf{sign}\big(\mathsf{enc}\big((q,p), \mathsf{pk}(k_{PE})\big), k_U\big)\Big) \\
&\texttt{compile}_{\texttt{output}}\,(Prot, 2) = \Big(ch, \mathsf{sign}\big(\mathsf{enc}\big((u, \mathsf{dec}\big(\mathsf{check}(x, \mathsf{vk}(k_U)), k_{PE}\big)[1]_2), \mathsf{pk}(k_S)\big), k_{PS}\big)\Big)
\end{aligned}$$

- `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.

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

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:

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

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`.

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

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

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

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 generated output message of output 2 ($\texttt{compile\_output}(\mathit{Prot}, 2)$), this output depends on input 1 ($\texttt{variable}(\mathit{Prot}, 1) = x$).

$$\begin{aligned}
\texttt{inputvariables}_{\texttt{all}}\,(\mathit{Prot}, 1) &= \varnothing \\
\texttt{inputvariables}_{\texttt{all}}\,(\mathit{Prot}, 2) &= \{x\}
\end{aligned}$$

- $\texttt{inputvariables}_{\texttt{pub}}$ 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:

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

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 `private_channel`:

- `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 `public` and `public_terms` as defined below.

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

  Note that this definition is not circular, because of the partial ordering of the labels returned by `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).

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

  The public terms in the example are:

$$\texttt{public\_terms}\,(Prot, 1) = \{\mathsf{pk}(k_{PE}), \mathsf{pk}(k_S), \mathsf{vk}(k_U), \mathsf{vk}(k_{PS})\}$$
$$\texttt{public\_terms}\,(Prot, 2) = \{\mathsf{pk}(k_{PE}), \mathsf{pk}(k_S), \mathsf{vk}(k_U), \mathsf{vk}(k_{PS}), x\}$$

- 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 generation $\texttt{generate}_{\texttt{statement}}$. While the set pub of public statements will be extended during the generation 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 generation.

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

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

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

$$\begin{aligned}
\texttt{private_channel}\,(\mathit{Prot}, 1) &= \mathsf{false} \\
\texttt{private_channel}\,(\mathit{Prot}, 2) &= \mathsf{false}
\end{aligned}$$

- The statement generation $\texttt{generate}_{\texttt{statement}}\,(P, k)$ 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):

$$\begin{aligned}
\texttt{compile}_{\texttt{output}}\,(\mathit{Prot}, 1) &= \Big(ch, \mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big)\Big) \\
\texttt{compile}_{\texttt{output}}\,(\mathit{Prot}, 2) &= \Big(ch, \mathsf{sign}\big(\mathsf{enc}\big((u, \mathsf{dec}\big(\mathsf{check}(x, \mathsf{vk}(k_U)), k_{PE}\big)[1]_2), \mathsf{pk}(k_S)\big), k_{PS}\big)\Big)
\end{aligned}$$

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

$$\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big) \quad \text{and}$$
$$\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big).$$

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

$$\begin{aligned}
&\Big(\ [\![\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big)]\!]_{\mathsf{true}}\,,(\varepsilon),\Big(\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big)\Big)\Big)\\
=&\Big(\ [\![\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)]\!]_{\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,,\\
&\quad(\varepsilon),\Big(\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big),\ \mathsf{vk}(k_U),\ \mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)\Big)\Big)\\
=&\Big(\ [\![(q,p)]\!]_{\beta_3=\mathsf{enc}(\alpha_1,\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,,\\
&\quad\big((q,p)\big),\\
&\quad\Big(\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big),\ \mathsf{vk}(k_U),\ \mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),\ \mathsf{pk}(k_{PE})\Big)\Big)\\
=&\Big(\beta_3=\mathsf{enc}((\alpha_1,\alpha_2),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3,\\
&\quad(q,p),\\
&\quad\Big(\mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big),\ \mathsf{vk}(k_U),\ \mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),\ \mathsf{pk}(k_{PE})\Big)\Big)
\end{aligned}$$

The statement generation for the message of output 2:

$$\begin{aligned}
&\Big(\ [\![\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big)]\!]_{\mathsf{true}}\,,\\
&\quad(\varepsilon),\quad\big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big)\big)\Big)\\
=&\Big(\ [\![\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big)]\!]_{\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,,\\
&\quad(\varepsilon),\quad\big(\mathsf{sign}\big((u,\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big)[1]_2),k_{PS}\big),\ \mathsf{vk}(k_{PS}),\\
&\qquad\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big)\big)\Big)\\
=&\Big(\ [\![(u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2)]\!]_{\beta_3=\mathsf{enc}(\alpha_1,\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3)}\,,\\
&\quad\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2)\big),\\
&\quad\big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}),\\
&\quad\ \mathsf{enc}\big(\mathsf{dec}((u,\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S)\big)\Big)\\
=&\Big(\ [\![\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2]\!]_{\beta_3=\mathsf{enc}((\beta_5,\alpha_1),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,,\\
&\quad\big(\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2\big),\\
&\quad\big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}),\\
&\quad\ \mathsf{enc}\big(\mathsf{dec}((u,\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u\big),\Big)
\end{aligned}$$

$$
\begin{aligned}
=\Big(& [\![\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})]\!]_{\beta_3=\mathsf{enc}((\beta_5,\alpha_1),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,, \\
& \big(\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2,\ \mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[2]_2\big), \\
& \big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}), \\
& \quad \mathsf{enc}\big(\mathsf{dec}((u,\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u\big),\Big) \\
=\Big(& [\![\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})]\!]_{\beta_6=\mathsf{pk}(\alpha_3)\wedge\beta_3=\mathsf{enc}((\beta_5,\alpha_1),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,, \\
& \big(\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2,\ \mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[2]_2,\ k_{PE}\big), \\
& \big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}), \\
& \quad \mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u,\ \mathsf{pk}(k_{PE})\big)\Big) \\
=\Big(& [\![\mathsf{check}(x,\mathsf{vk}(k_U))]\!]_{\mathsf{dec}(\beta_7,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_6=\mathsf{pk}(\alpha_3)\wedge\beta_3=\mathsf{enc}((\beta_5,\alpha_1),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,, \\
& \big(\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2,\ \mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[2]_2,\ k_{PE}\big), \\
& \big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}), \\
& \quad \mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u,\ \mathsf{pk}(k_{PE}) \\
& \quad \mathsf{check}(x,\mathsf{vk}(k_U))\big)\Big) \\
=\Big(& [\![x]\!]_{\mathsf{check}(\beta_9,\beta_8)\rightsquigarrow\beta_7\wedge\mathsf{dec}(\beta_7,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_6=\mathsf{pk}(\alpha_3)\wedge\beta_3=\mathsf{enc}((\beta_5,\alpha_1),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3}\,, \\
& \big(\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2,\ \mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[2]_2,\ k_{PE}\big), \\
& \big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S),\ u,\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}), \\
& \quad \mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}), \\
& \quad \mathsf{check}(x,\mathsf{vk}(k_U)),\ \mathsf{vk}(k_U),\ x\big)\Big) \\
=\Big(& \mathsf{check}(\beta_9,\beta_8)\rightsquigarrow\beta_7\wedge\mathsf{dec}(\beta_7,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_6=\mathsf{pk}(\alpha_3) \\
& \quad \wedge\beta_3=\mathsf{enc}((\beta_5,\alpha_1),\beta_4)\wedge\mathsf{check}(\beta_1,\beta_2)\rightsquigarrow\beta_3, \\
& \big(\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2,\ \mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[2]_2,\ k_{PE}\big), \\
& \big(\mathsf{sign}\big(\mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),k_{PS}\big),\ \mathsf{vk}(k_{PS}), \\
& \quad \mathsf{enc}\big((u,\mathsf{dec}(\mathsf{check}(x,\mathsf{vk}(k_U)),k_{PE})[1]_2),\mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u,\ \mathsf{pk}(k_{PE}), \\
& \quad \mathsf{check}(x,\mathsf{vk}(k_U)),\ \mathsf{vk}(k_U),\ x\big)\Big)
\end{aligned}
$$

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

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

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

Here $\big([\![M\sigma]\!]_S, s, sec, pub\big)$ means the generation of statement $M\sigma$, where s is the placeholder for $M\sigma$; S is the formula generated so far and *sec* and *pub* are the private and public component, which have been generated so far. After the statement for the output message has been generated, the statement is extended by the destructor evaluations which have not been taken into consideration. The generation 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 generation step. We keep the generated 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 generate the statements for each single component, otherwise we add all components $M_1, \dots, M_n$ to the private messages and generate the statements for each single component.

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

Similarly we generate the statement for the ith component of a tuple M consisting of n elements:

$$
\begin{aligned}
&(\llbracket M[i]_n \rrbracket_S, s, sec, pub) = \\
&\quad \begin{cases}
\big(S_n\{\langle \beta_{|pub|+1}, \ldots, \beta_{|pub|+n} \rangle / s\}, s_n, sec_n, pub_n\big) \\
\text{with } \big(S_1, s_1, sec_1, pub_1\big) = \big(\, \llbracket M[1]_n \rrbracket_S, \beta_{|pub|+1}, sec \setminus \{s\}, (pub, M[1]_n, \ldots, M[n]_n)\big), \\
\qquad \big(S_2, s_2, sec_2, pub_2\big) = \big(\, \llbracket M[2]_n \rrbracket_{S_1}, \beta_{|pub|+2}, sec_1, pub_1\big), \\
\qquad\qquad \vdots \\
\qquad \big(S_n, s_n, sec_n, pub_n\big) = \big(\, \llbracket M[n]_n \rrbracket_{S_{n-1}}, \beta_{|pub|+n}, sec_{n-1}, pub_{n-1}\big), \\
\qquad \text{if } \texttt{public}\big(M, pub, k\big) \\
\\
\big(S_n\{\langle \alpha_{|sec|+1}, \ldots, \alpha_{|sec|+n} \rangle / s\}, s_n, sec_n, pub_n\big) \\
\text{with } \big(S_1, s_1, sec_1, pub_1\big) = \big(\, \llbracket M[1]_n \rrbracket_S, \alpha_{|sec|+1}, (sec, M[1]_n, \ldots, M[n]_n), pub\big), \\
\qquad \big(S_2, s_2, sec_2, pub_2\big) = \big(\, \llbracket M[2]_n \rrbracket_{S_1}, \alpha_{|sec|+2}, sec_1, pub_1\big), \\
\qquad\qquad \vdots \\
\qquad \big(S_n, s_n, sec_n, pub_n\big) = \big(\, \llbracket M[n]_n \rrbracket_{S_{n-1}}, \alpha_{|sec|+n}, sec_{n-1}, pub_{n-1}\big), \\
\qquad \text{if } \neg\texttt{public}\big(M, pub, k\big)
\end{cases}
\end{aligned}
$$

If $S^{\#}$ is a encryption key $\mathsf{pk}(M)$, there are three different cases. If the encryption key is public, there is nothing to generated, since $\mathsf{pk}(M)$ 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.

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

If $S^{\#}$ is an asymmetric encryption $\mathsf{enc}(M_1, M_2)$ 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 generated, then the same is done for M_1.

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

If $S^{\#}$ is a symmetric encryption $\mathsf{senc}(M_1, M_2)$ of message M_1 with symmetric key M_2 we have the same cases as for the asymetric encryption. The only difference in the generation 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 $\mathsf{senc}(M_1, M_2)$ knows the secret key for his public key.

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

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

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

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

If $S^{\#}$ is a hash $\mathsf{hash}(M)$ 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.

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

If $S^{\#}$ is a signature $\mathsf{sign}(M_1, M_2)$ of a message M_1 with signing key M_2 we do the following: If the corresponding verification key $\mathsf{vk}(M_2)$ 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 $\mathsf{sign}(M_1, M_2)$ with the verification key $\mathsf{vk}(M_2)$ results in the message M_1. Finally we continue the generation with M_1. If on the other hand the verification key $\mathsf{vk}(M_2)$ is not public, we generate the statement as for an asymetric encryption: we add M_2 to the corresponding component, generate the statement for M_2. Then we add M_1 to the corresponding component and finally continue compiling the statement for M_1.

$\big(\,[\![\mathsf{sign}(M_1,M_2)]\!]_S\,,s,sec,pub\big) = \big(\,[\![M_1]\!]_{S'''}\,,s''',sec''',pub'''\big)$
with $(S',s',sec',pub') =$

$$\begin{cases} \big(\,[\![M_2]\!]_S\,,\beta_{|pub|+1},sec,(pub,M_2)\big), \\ \qquad \text{if } \mathtt{public}(M_2,pub,k) \wedge \neg\mathtt{public}(\mathsf{vk}(M_2),pub,k) \\ \big(\,[\![M_2]\!]_S\,,\alpha_{|sec|+1},(sec,M_2),pub\big), \\ \qquad \text{if } \neg\mathtt{public}(M_2,pub,k) \wedge \neg\mathtt{public}(\mathsf{vk}(M_2),pub,k) \\ \big(S,\beta_{|pub|+1},sec,(pub,\mathsf{vk}(M_2))\big), \\ \qquad \text{if } \mathtt{public}(\mathsf{vk}(M_2),pub,k) \end{cases}$$

$(S'',s'',sec'',pub'') =$

$$\begin{cases} \big(S',\beta_{|pub'|+1},sec',(pub',M_1)\big), \\ \qquad \text{if } \mathtt{public}(\mathsf{sign}(M_1,M_2),pub,k) \vee \mathtt{public}(M_1,pub,k) \\ \big(S',\alpha_{|sec'|+1},(sec',M_1),pub'\big), \\ \qquad \text{if } \neg\mathtt{public}(\mathsf{sign}(M_1,M_2),pub,k) \wedge \neg\mathtt{public}(M_1,pub,k) \end{cases}$$

$(S''',s''',sec''',pub''') =$

$$\begin{cases} \big(\mathsf{check}(s,s') \rightsquigarrow s'' \wedge S',s'',sec'',pub''\big), \\ \qquad \text{if } \mathtt{public}(\mathsf{vk}(M_2),pub,k) \\ \big(s = \mathsf{sign}(s'',s') \wedge S',s'',sec'',pub''\big), \\ \qquad \text{if } \neg\mathtt{public}(\mathsf{vk}(M_2),pub,k) \end{cases}$$

If $S^{\#}$ is a decryption $\mathsf{dec}(M_1,M_2)$ 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 generated. 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 generation is continued with M_1.

$\big(\,[\![\mathsf{dec}(M_1,M_2)]\!]_S\,,s,sec,pub\big) = \big(\,[\![M_1]\!]_{S'''}\,,s''',sec''',pub'''\big)$
with $(S',s',sec',pub') =$

$$\begin{cases} \big(\,[\![M_2]\!]_S\,,\beta_{|pub|+1},sec,(pub,M_2)\big), \\ \qquad \text{if } \mathtt{public}(M_2,pub,k) \\ \big(\,[\![M_2]\!]_S\,,\alpha_{|sec|+1},(sec,M_2),pub\big), \\ \qquad \text{if } \neg\mathtt{public}(M_2,pub,k) \end{cases}$$

$(S'',s'',sec'',pub'') =$

$$\begin{cases} \big(\beta_{|pub'|+1} = \mathsf{pk}(s') \wedge S',\beta_{|pub'|+1},sec',(pub',\mathsf{pk}(M_2))\big), \\ \qquad \text{if } \mathtt{public}(\mathsf{pk}(M_2),pub,k) \\ \big(\alpha_{|sec'|+1} = \mathsf{pk}(s') \wedge S',\alpha_{|sec'|+1},(sec',\mathsf{pk}(M_2)),pub'\big), \\ \qquad \text{if } \neg\mathtt{public}(\mathsf{pk}(M_2),pub,k) \end{cases}$$

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

If $S^{\#}$ is a symmetric decryption $\mathsf{sdec}(M_1, M_2)$ 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 generated.

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

If $S^{\#}$ is a verification $\mathsf{check}(M_1, M_2)$ 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.

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

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 generation of the zero-knowledge proof.

$$
\begin{aligned}
&\mathtt{generate}_{\mathtt{statement}}\big(Prot, 1\big)\\
&= \Big(\beta_3 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_4) \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3,\\
&\quad (q, p)\\
&\quad \big(\mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big),\ \mathsf{vk}(k_U),\ \mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big),\ \mathsf{pk}(k_{PE})\big)\Big)\\
&\mathtt{generate}_{\mathtt{statement}}\big(Prot, 2\big)\\
&= \Big(\mathsf{check}(\beta_9, \beta_8) \rightsquigarrow \beta_7 \wedge \mathsf{dec}(\beta_7, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2) \wedge \beta_6 = \mathsf{pk}(\alpha_3)\\
&\quad \wedge \beta_3 = \mathsf{enc}((\beta_5, \alpha_1), \beta_4) \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3,\\
&\quad \big(\mathsf{dec}(\mathsf{check}(x, \mathsf{vk}(k_U)), k_{PE})[1],\ \mathsf{dec}(\mathsf{check}(x, \mathsf{vk}(k_U)), k_{PE})[2],\ k_{PE}\big),\\
&\quad \big(\mathsf{sign}\big(\mathsf{enc}\big((u, \mathsf{dec}(\mathsf{check}(x, \mathsf{vk}(k_U)), k_{PE})[1]), \mathsf{pk}(k_S)\big), k_{PS}\big),\ \mathsf{vk}(k_{PS}),\\
&\quad\ \mathsf{enc}\big((u, \mathsf{dec}(\mathsf{check}(x, \mathsf{vk}(k_U)), k_{PE})[1]), \mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u,\ \mathsf{pk}(k_{PE}),\\
&\quad\ \mathsf{check}(x, \mathsf{vk}(k_U)),\ \mathsf{vk}(k_U),\ x\big)\Big)
\end{aligned}
$$

- Given a process P and a label k, $\mathtt{generate}_{\mathtt{zk}}(P, k)$ returns a tuple consisting of the label k; the length of the private and public component of the generated 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 generated zero-knowledge proof and that list.

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

Basically the statement generation $\mathtt{generate}_{\mathtt{statement}}$ 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'':

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

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

$$
\begin{aligned}
S'_{(1)} &= \beta_3 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_4) \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3,\\
sec_{(1)} &= \big(q, p\big),\\
pub''_{(1)} &= \big(\mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big),\ \mathsf{vk}(k_U),\\
&\qquad \mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big),\ \mathsf{pk}(k_{PE})\big).
\end{aligned}
$$

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

$$\begin{aligned}\sigma_{(2)} &= \{x_3/x_1[2]_2\}\{x_2/x_1[1]_2\}\{\mathsf{check}(x, \mathsf{vk}(k_U))/x_1\},\\ \sigma_{(2)}^{-1} &= \{x_1/\mathsf{check}(x, \mathsf{vk}(k_U))\}\{x_1[1]_2/x_2\}\{x_1[2]_2/x_3\}.\end{aligned}$$

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

$$\begin{aligned}S'_{(2)} &= \mathsf{check}(\beta_9, \beta_8) \rightsquigarrow \beta_7 \wedge \mathsf{dec}(\beta_7, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2) \wedge \beta_6 = \mathsf{pk}(\alpha_3)\\ &\quad \wedge \beta_3 = \mathsf{enc}((\beta_5, \alpha_1), \beta_4) \wedge \mathsf{check}(\beta_1, \beta_2) \rightsquigarrow \beta_3,\\ sec_{(2)} &= \big(x_2,\ x_3,\ k_{PE}\big),\\ pub''_{(2)} &= \big(\mathsf{sign}\big(\mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big), k_{PS}\big),\ \mathsf{vk}(k_{PS}),\\ &\qquad \mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big),\ \mathsf{pk}(k_S),\ u,\ \mathsf{pk}(k_{PE}),\\ &\qquad x_1,\ \mathsf{vk}(k_U),\ x\big)\end{aligned}$$

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

$$\begin{aligned}(u_{(1)}, M_{(1)}) &= \big(ch, \mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big)\big)\\ (u_{(2)}, M_{(2)}) &= \big(ch, \mathsf{sign}\big(\mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big), k_{PS}\big)\big)\\ \texttt{public\_terms}(Prot, 1) &= \{\mathsf{pk}(k_{PE}), \mathsf{pk}(k_S), \mathsf{vk}(k_U), \mathsf{vk}(k_{PS})\}\\ \texttt{public\_terms}(Prot, 2) &= \{\mathsf{pk}(k_{PE}), \mathsf{pk}(k_S), \mathsf{vk}(k_U), \mathsf{vk}(k_{PS}), x\}\end{aligned}$$

resulting in

$$\begin{aligned}\tilde{L}_{(1)} &= \Big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE})\Big)\\ \tilde{L}_{(2)} &= \Big(\mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ x\Big)\\ pub_{(1)} &= \Big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ \mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big),\\ &\qquad \mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big)\Big)\\ pub_{(2)} &= \Big(\mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ x,\\ &\qquad \mathsf{sign}\big(\mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big), k_{PS}\big),\ \mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big),\ x_1\Big)\end{aligned}$$

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

$$\begin{aligned} i_{(1)} &= 2, \qquad j_{(1)} = 4, \qquad l_{(1)} = 2 \\ i_{(2)} &= 3, \qquad j_{(2)} = 9, \qquad l_{(2)} = 5 \end{aligned}$$

Then we substitute the β's in the statement S' from $\texttt{generate}_{\texttt{statement}}$. 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:

$$\begin{aligned} \sigma'_{(1)} = {} & \{\beta_5/\beta_1\}\{\beta_6/\beta_2\}\{\beta_7/\beta_3\}\{\beta_8/\beta_4\} \\ & \{\beta_3/\beta_5\}\{\beta_1/\beta_6\}\{\beta_4/\beta_7\}\{\beta_2/\beta_8\} \\ \sigma'_{(2)} = {} & \{\beta_{10}/\beta_1\}\{\beta_{11}/\beta_2\}\{\beta_{12}/\beta_3\}\{\beta_{13}/\beta_4\} \\ & \{\beta_{14}/\beta_5\}\{\beta_{15}/\beta_6\}\{\beta_{16}/\beta_7\}\{\beta_{17}/\beta_8\}\{\beta_{18}/\beta_9\} \\ & \{\beta_6/\beta_{10}\}\{\beta_1/\beta_{11}\}\{\beta_7/\beta_{12}\}\{\beta_2/\beta_{13}\} \\ & \{\beta_8/\beta_{14}\}\{\beta_3/\beta_{15}\}\{\beta_9/\beta_{16}\}\{\beta_4/\beta_{17}\}\{\beta_5/\beta_{18}\} \end{aligned}$$

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

$$\begin{aligned} S_{(1)} &= \beta_4 = \mathsf{enc}((\alpha_1, \alpha_2), \beta_2) \wedge \mathsf{check}(\beta_3, \beta_1) \rightsquigarrow \beta_4 \\ S_{(2)} &= \mathsf{check}(\beta_5, \beta_4) \rightsquigarrow \beta_9 \wedge \mathsf{dec}(\beta_9, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2) \wedge \beta_3 = \mathsf{pk}(\alpha_3) \\ & \quad \wedge \beta_7 = \mathsf{enc}((\beta_8, \alpha_1), \beta_2) \wedge \mathsf{check}(\beta_6, \beta_1) \rightsquigarrow \beta_7 \end{aligned}$$

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:

$\mathsf{enc}\big(x_i, \mathsf{pk}(\hat{k}_i)\big)$. For that encryption we will use a new public key $\mathsf{pk}(\hat{k}_i)$ 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.

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

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

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

Finally $\mathtt{generate}_{\mathtt{zk}}(P,k)$ 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 $\big\langle \mathsf{zk}_{i,j,S}(sec,pub), x'_2,\ldots,x'_n\big\rangle$ consisting the generated zero-knowledge proof for output k, $\mathsf{zk}_{i,j,S}sec,pub$, and the forwarded proofs $x'_2,\ldots,x'_n$; the length l of the list of public terms $\tilde{L}$ and this list $\tilde{L}$:

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

Note that we append the variable $\mathtt{variable}(P,k)$ of the input k of the original protocol. This results in the list of public terms $\big(\tilde{L},\mathtt{variable}(P,k)\big)$ 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 $\mathtt{generate}_{\mathtt{ver}}$.

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

$$\begin{aligned}
&\mathtt{generate}_{\mathtt{zk}}\,(Prot,1)\\
&= \Big(1,2,4,\beta_4 = \mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4,\\
&\quad \big\langle \mathsf{zk}_{2,4,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4}\\
&\quad \big(q,\ p;\ \mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ \mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big),\\
&\quad \mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)\big)\big\rangle,\\
&\quad 3,\big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ x\big)\Big)
\end{aligned}$$

$$
\begin{aligned}
&\texttt{generate}_{\texttt{zk}}\,(Prot, 2)\\
&= \Big(2, 3, 9, \mathsf{check}(\beta_5, \beta_4) \rightsquigarrow \beta_9 \wedge \mathsf{dec}(\beta_9, \alpha_3) \rightsquigarrow (\alpha_1, \alpha_2) \wedge \beta_3 = \mathsf{pk}(\alpha_3)\\
&\quad \wedge \beta_7 = \mathsf{enc}((\beta_8, \alpha_1), \beta_2) \wedge \mathsf{check}(\beta_6, \beta_1) \rightsquigarrow \beta_7,\\
&\quad \Big\langle \mathsf{zk}_{3,9,\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9\wedge\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_3=\mathsf{pk}(\alpha_3)\ \wedge\beta_7=\mathsf{enc}((\beta_8,\alpha_1),\beta_2)\wedge\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7}\\
&\quad \big(x_2,\ x_3,\ k_{PE}; \mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ x,\\
&\quad \mathsf{sign}\big(\mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big), k_{PS}\big),\ \mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big),\ u,\ x_1\big), \hat{x}\Big\rangle,\\
&\quad 6, \big(\mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ x,\ y\big)\Big)
\end{aligned}
$$

For replacing the message M in the original process only the tuple $\langle \mathsf{zk}_{i,j,S}(sec, pub), x'_2, \ldots, x'_n \rangle$ containing the generated 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 $\texttt{generate}_{\texttt{zk}}$ is defined as:

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

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

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

$\texttt{generate}_{\texttt{ver}}$ basically inserts a process between the input k and the continuation process Q. As stated above we use $\texttt{generate}_{\texttt{zk}}$ 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 generated 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.

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

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 $\texttt{generate}_{\texttt{zk}}$. 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 generated for output k_p when executing the process.

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

For our example we then have the following preliminary results:

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

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 $\texttt{generate}_{\texttt{zk}}$ the first zero-knowledge proof is never encrypted, hence we do not have to decrypt it.

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

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

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

For the verification of the zero-knowledge proofs $\hat{x}_1, \ldots, \hat{x}_n$, we will need some messages from the public components $\mathsf{public}_{j_1}(\hat{x}_1), \ldots, \mathsf{public}_{j_n}(\hat{x}_n)$ 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 $\texttt{generate}_{\texttt{zk}}$ (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 $\hat{x}_{p,m}$.

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

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

$$
\begin{aligned}
j_{1\,(1)} = 4, \qquad j_{1\,(2)} = 9 \qquad j_{2\,(2)} = j_{1\,(1)} = 4 \\
\texttt{generate}_{\texttt{ver}}(Prot, Q, 1) &= \ldots \\
&\quad \mathsf{let}\ \langle \hat{x}_{1,1}, \hat{x}_{1,2}, \hat{x}_{1,3}, \hat{x}_{1,4} \rangle = \mathsf{public}_4(\hat{x}_1)\ \mathsf{then} \\
&\quad \ldots \\
\texttt{generate}_{\texttt{ver}}(Prot, Q, 2) &= \ldots \\
&\quad \mathsf{let}\ \langle \hat{y}_{2,1}, \ldots, \hat{y}_{2,4} \rangle = \mathsf{public}_4(\hat{y}_2)\ \mathsf{then} \\
&\quad \mathsf{let}\ \langle \hat{y}_{1,1}, \ldots, \hat{y}_{1,9} \rangle = \mathsf{public}_9(\hat{y}_1)\ \mathsf{then} \\
&\quad \ldots
\end{aligned}
$$

The fourth step is the verification of each zero-knowledge proof $\mathsf{ver}_{i_p, j_p, l_p, S_p}\big(\hat{x}_p, \tilde{L}_p\big)$. 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 $\texttt{generate}_{\texttt{zk}}$, 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.

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

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

$$
\begin{aligned}
\big(M_{1,1\,(1)},\ldots,M_{1,3\,(1)}\big) &= \big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ x\big)\\
\big(M_{1,1\,(2)},\ldots,M_{1,6\,(2)}\big) &= \big(\mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ x,\ y\big)\\
\big(M_{2,1\,(2)},\ldots,M_{2,3\,(2)}\big) &= \big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ x\big)\\
\sigma_{1\,(1)} &= \varepsilon\\
\sigma_{1\,(2)} &= \{\hat{y}_{1,5}/x\}\\
\sigma_{2\,(2)} &= \{\hat{y}_{1,5}/x\}\\
\tilde{L}_{1\,(1)} &= \big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE})\big)\\
\tilde{L}_{1\,(2)} &= \big(\mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ \hat{y}_{1,5}\big)\\
\tilde{L}_{2\,(2)} &= \big(\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ \hat{y}_{1,5}\big)
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{generate}_{\mathtt{ver}}(Prot, Q, 1) = \ldots \\
&\qquad\qquad \mathsf{let}\ \hat{x}_{1,V} = \mathsf{ver}_{2,4,2,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4} \\
&\qquad\qquad\qquad (\hat{x}_1, \mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}))\ \mathsf{then} \\
&\qquad\qquad \mathsf{let}\ \langle x, \hat{x}_{1,4}\rangle = \hat{x}_{1,V}\ \mathsf{in}\ Q \\
&\mathtt{generate}_{\mathtt{ver}}(Prot, Q, 2) = \ldots \\
&\qquad\qquad \mathsf{let}\ \hat{y}_{2,V} = \mathsf{ver}_{2,4,3,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4} \\
&\qquad\qquad\qquad (\hat{y}_2, \mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ \hat{y}_{1,5})\ \mathsf{then} \\
&\qquad\qquad \mathsf{let}\ \hat{y}_{1,V} = \mathsf{ver}_{3,9,5,\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9\wedge\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_3=\mathsf{pk}(\alpha_3)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\beta_7=\mathsf{enc}((\beta_8,\alpha_1),\beta_2)\wedge\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7 \\
&\qquad\qquad\qquad (\hat{y}_1, \mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ \hat{y}_{1,5})\ \mathsf{then} \\
&\qquad\qquad \mathsf{let}\ \langle \hat{y}_{2,4}\rangle = \hat{y}_{2,V}\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \langle y, \hat{y}_{1,7}, \hat{y}_{1,8}, \hat{y}_{1,9}\rangle = \hat{y}_{1,V}\ \mathsf{in}\ Q
\end{aligned}
$$

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{generate}_{\mathtt{ver}}\,(Prot, Q, 1) = \\
&\qquad\qquad \mathsf{let}\ \langle \hat{x}_1\rangle = \hat{\hat{x}}\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \langle \hat{x}_{1,1}, \ldots, \hat{x}_{1,4}\rangle = public_4(\hat{x}_1)\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \hat{x}_{1,V} = \mathsf{ver}_{2,4,2,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4} \\
&\qquad\qquad\qquad \big(\hat{x}_1, \mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE})\big)\ \mathsf{then} \\
&\qquad\qquad \mathsf{let}\ \langle x, \hat{x}_{1,4}\rangle = \hat{x}_{1,V}\ \mathsf{in}\ Q \\
&\mathtt{generate}_{\mathtt{ver}}\,(Prot, Q, 2) = \\
&\qquad\qquad \mathsf{let}\ \langle \hat{y}_1, \hat{y}_2\rangle = \hat{\hat{y}}\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \langle \hat{y}_{2,1}, \ldots, \hat{y}_{2,4}\rangle = public_4(\hat{y}_2)\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \langle \hat{y}_{1,1}, \ldots, \hat{y}_{1,9}\rangle = public_9(\hat{y}_1)\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \hat{y}_{2,V} = \mathsf{ver}_{2,4,3,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4} \\
&\qquad\qquad\qquad \big(\hat{y}_2, \mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ \hat{y}_{1,5}\big)\ \mathsf{then} \\
&\qquad\qquad \mathsf{let}\ \hat{y}_{1,V} = \mathsf{ver}_{3,9,5,\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9\wedge\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_3=\mathsf{pk}(\alpha_3)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\beta_7=\mathsf{enc}((\beta_8,\alpha_1),\beta_2)\wedge\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7 \\
&\qquad\qquad\qquad \big(\hat{y}_1, \mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ \hat{y}_{1,5}\big)\ \mathsf{then} \\
&\qquad\qquad \mathsf{let}\ \langle \hat{y}_{2,4}\rangle = \hat{y}_{2,V}\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \langle y, \hat{y}_{1,7}, \hat{y}_{1,8}, \hat{y}_{1,9}\rangle = \hat{y}_{1,V}\ \mathsf{in}\ Q
\end{aligned}
$$

Summing up all steps we get:

$$
\begin{aligned}
\texttt{generate}_{\texttt{ver}} &: \mathcal{P} \times \mathcal{P} \times \mathcal{L} \to \mathcal{P} \\
\texttt{generate}_{\texttt{ver}}\,(P, Q, k) &= \mathsf{let}\ \left\langle \hat{\hat{x}}_1, \ldots, \hat{\hat{x}}_n \right\rangle = \hat{\hat{x}}\ \mathsf{in} \\
&\qquad \mathsf{let}\ \hat{x}_2 = N_2\ \mathsf{in} \ldots \mathsf{let}\ \hat{x}_n = N_n\ \mathsf{in} \\
&\qquad \mathsf{let}\ \langle \hat{x}_{n,1}, \ldots, \hat{x}_{n,j_n} \rangle = \mathsf{public}_{j_n}(\hat{x}_n)\ \mathsf{then} \\
&\qquad \ldots \\
&\qquad \mathsf{let}\ \langle \hat{x}_{1,1}, \ldots, \hat{x}_{1,j_1} \rangle = \mathsf{public}_{j_1}(\hat{x}_1)\ \mathsf{then} \\
&\qquad \mathsf{let}\ \hat{x}_{n,V} = \mathsf{ver}_{i_n,j_n,l_n,S_n}\left(\hat{x}_n, \tilde{L}_n\right)\ \mathsf{then} \\
&\qquad \ldots \\
&\qquad \mathsf{let}\ \hat{x}_{2,V} = \mathsf{ver}_{i_2,j_2,l_2,S_2}\left(\hat{x}_2, \tilde{L}_2\right)\ \mathsf{then} \\
&\qquad \mathsf{let}\ \hat{x}_{1,V} = \mathsf{ver}_{i_1,j_1,l_1-1,S_1}\left(\hat{x}_1, \tilde{L}_1\right)\ \mathsf{then} \\
&\qquad \mathsf{let}\ \langle \hat{x}_{n,l_n+1}, \ldots \hat{x}_{n,j_n} \rangle = \hat{x}_{n,V}\ \mathsf{in} \\
&\qquad \ldots \\
&\qquad \mathsf{let}\ \left\langle \hat{x}_{2,l_2+1}, \ldots \hat{x}_{2,j_2} \right\rangle = \hat{x}_{2,V}\ \mathsf{in} \\
&\qquad \mathsf{let}\ \left\langle x, \hat{x}_{n,l_1+1}, \ldots \hat{x}_{n,j_1} \right\rangle = \hat{x}_{1,V}\ \mathsf{in}\ Q \\
&\quad \text{with } x := \texttt{variable}(P, k) \\
tc(\texttt{graph})\,(P, k) &= (k_1, \ldots, k_n) \\
N_p &= \begin{cases} \mathsf{dec}(\hat{\hat{x}}_p, \hat{k}_p), & \texttt{private\_channel}\,(P, k_p) \wedge p \neq 1 \\ \mathsf{id}(\hat{\hat{x}}_p), & \texttt{otherwise}, \end{cases} \qquad 1 \leqslant p \leqslant n \\
\texttt{generate}_{\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 generated process. The algorithm generates 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.

$$\texttt{transform} : \mathcal{P} \to \mathcal{P}$$
$$\texttt{transform}\,(P) = \mathsf{new}\ \tilde{k}\mathsf{out}(ch_k, \tilde{\mathsf{pk}(k)})\left(\texttt{transform}'\,(P,P)\right)$$

$$\texttt{transform}' : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$$
$$\texttt{transform}'\,(P, k : \mathsf{out}(u,M).Q) = \begin{cases} k : \mathsf{out}(u, M_k).\texttt{transform}'\,(P,Q)\,, \\ \qquad \text{if } ' \in \texttt{labels}_{\texttt{in}}(P) \cap \texttt{labels}_{\texttt{out}}(P) \\ k : \mathsf{out}(u,M).\texttt{transform}'\,(P,Q)\,, \\ \qquad \text{otherwise} \end{cases}$$
$$\texttt{transform}'\,(P, k : [!]\mathsf{in}(u,x).Q) = \begin{cases} k : [!]\mathsf{in}(u, \hat{\tilde{x}}_{in}).\texttt{generate}_{\texttt{ver}}\,(P, \texttt{transform}'\,(P,Q)\,, k)\,, \\ \qquad \text{if } k \in \texttt{labels}_{\texttt{in}}(P) \cap \texttt{labels}_{\texttt{out}}(P) \\ k : [!]\mathsf{in}(u,x).\texttt{transform}\,(Q)\,, \\ \qquad \text{otherwise} \end{cases}$$
$$\texttt{transform}'\,(P, \mathsf{new}\ nQ) = \mathsf{new}\ n\texttt{transform}'\,(P,Q)$$
$$\texttt{transform}'\left(P, Q|Q'\right) = \texttt{transform}'\,(P,Q)\,|\texttt{transform}'\left(P,Q'\right)$$
$$\texttt{transform}'\,(P,0) = 0$$
$$\texttt{transform}'\big(P, \mathsf{let}\ x = g(\tilde{M})\ \mathsf{then}\ Q\ \texttt{else}\ Q'\big) = \mathsf{let}\ x = g(\tilde{M})\ \mathsf{then}\ \texttt{transform}'\,(P,Q)\ \texttt{else}\ \texttt{transform}'\left(P,Q'\right)$$
$$\texttt{transform}'\,(P, \mathsf{let}\ \langle x_1,\ldots,x_n\rangle = M\ \mathsf{in}\ Q) = \mathsf{let}\ \langle x_1,\ldots,x_n\rangle = M\ \mathsf{in}\ \texttt{transform}'\,(P,Q)$$
$$\texttt{transform}'\,(P, \mathsf{assume}\ C) = \mathsf{assume}\ C$$
$$\texttt{transform}'\,(P, \mathsf{assert}\ C) = \mathsf{assert}\ C$$

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

$$\texttt{labels}_{\texttt{out}}\,(P) = \left(k_1,\ldots,k_n\right),$$
$$\tilde{k} = \left\{\hat{k}_i | \texttt{private\_channel}\,(P,k_i)\right\},$$
$$\tilde{\mathsf{pk}(k)} = \left\{\mathsf{pk}(k) | k \in \tilde{k}\right\},$$
$$ch_k \notin free(\texttt{transform}'(P,P))$$

C.2 Transforming 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 $\mathsf{zk}_{i,j,S}(n_1,\ldots,n_i;m_1,\ldots,m_j)$ 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 $\Gamma \vdash m_k : T'_k$.

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

According to the definition of $\mathtt{generate}_{\mathtt{zk}}$ 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:

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

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

$$
\begin{aligned}
&\mathsf{zk}_{2,4,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4}\\
&\qquad\big(q,p;\mathsf{vk}(k_U),\ \mathsf{pk}(k_{PE}),\ \mathsf{sign}\big(\mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big),k_U\big),\ \mathsf{enc}\big((q,p),\mathsf{pk}(k_{PE})\big)\big)
\end{aligned}
$$

we get

$$\begin{aligned}
&\mathsf{ZKProof}_{2,4,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4}\\
&\quad\Big(\langle y_1 : \mathsf{VerKey}(\mathsf{PubEnc}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\}))),\\
&\quad y_2 : \mathsf{PubKey}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\})),\\
&\quad y_3 : \mathsf{Signed}(\mathsf{PubEnc}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\}))),\\
&\quad y_4 : \mathsf{PubEnc}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\}))\rangle\\
&\quad \{\exists x_1, x_2 : y_4 = \mathsf{enc}((x_1, x_2), y_2) \wedge \mathsf{check}(y_3, y_1) \rightsquigarrow y_4 \wedge \mathsf{Request}(u, x_1)\}\Big)
\end{aligned}$$

and for

$$\begin{aligned}
&\mathsf{zk}_{3,9,\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9\wedge\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_3=\mathsf{pk}(\alpha_3)\wedge\beta_7=\mathsf{enc}((\beta_8,\alpha_1),\beta_2)\wedge\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7}\\
&\quad\big(x_2,\ x_3,\ k_{PE}; \mathsf{vk}(k_{PS}),\ \mathsf{pk}(k_S),\ \mathsf{pk}(k_{PE}),\ \mathsf{vk}(k_U),\ x,\\
&\quad \mathsf{sign}\big(\mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big), k_{PS}\big),\ \mathsf{enc}\big((u, x_2), \mathsf{pk}(k_S)\big),\ u,\ x_1\big)
\end{aligned}$$

we get

$$\begin{aligned}
&\mathsf{ZKProof}_{3,9,\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9\wedge\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_3=\mathsf{pk}(\alpha_3)\wedge\beta_7=\mathsf{enc}((\beta_8,\alpha_1),\beta_2)\wedge\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7}\\
&\quad\Big(\langle y_1 : \mathsf{VerKey}(\mathsf{PubEnc}(\langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\})),\\
&\quad y_2 : \mathsf{PubKey}(\langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\}),\\
&\quad y_3 : \mathsf{PubKey}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\})),\\
&\quad y_4 : \mathsf{VerKey}(\mathsf{PubEnc}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\}))),\\
&\quad y_5 : \mathsf{Signed}(\mathsf{PubEnc}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\}))),\\
&\quad y_6 : \mathsf{Signed}(\mathsf{PubEnc}(\langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\})),\\
&\quad y_7 : \mathsf{PubEnc}(\langle x_u : \mathsf{Un}, x_q : \mathsf{Un}\rangle\{\mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\}),\\
&\quad y_8 : \mathsf{Un},\\
&\quad y_9 : \mathsf{PubEnc}(\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\}))\rangle\\
&\quad \{\exists x_1, x_2, x_3 : \mathsf{check}(y_5, y_4) \rightsquigarrow y_9 \wedge \mathsf{dec}(y_9, x_3) \rightsquigarrow (x_1, x_2) \wedge y_3 = \mathsf{pk}(x_3)\\
&\quad \wedge y_7 = \mathsf{enc}((y_8, x_1), y_2) \wedge \mathsf{check}(y_6, y_1) \rightsquigarrow y_7 \wedge \mathsf{Registered}(y_8)\}\Big)
\end{aligned}$$

D The Complete Example

D.1 The Original Protocol

$$
\begin{aligned}
&\mathsf{new}\ k_U.\\
&\mathsf{new}\ k_{PE}.\\
&\mathsf{new}\ k_{PS}.\\
&\mathsf{new}\ k_S.\\
&\mathsf{new}\ p.\\
&\mathsf{out}(ch, \mathsf{pk}(k_{PE})).\\
&\mathsf{out}(ch, \mathsf{pk}(k_S)).\\
&\mathsf{out}(ch, \mathsf{vk}(k_U)).\\
&\mathsf{out}(ch, \mathsf{vk}(k_{PS})).\\
&(\mathsf{assume}\ \ \neg\mathsf{Compromised}(\mathsf{p})\\
&\ |\ \mathsf{assume}\ \mathsf{Compromised}(proxy) \Rightarrow \forall u.\mathsf{Registered}(u)\\
&\ |\ policy \mid U \mid P \mid S)
\end{aligned}
$$

$$
policy = \mathsf{assume}\ \ \forall u, q : \mathsf{Request}(u, q) \wedge \mathsf{Registered}(u) \Rightarrow \mathsf{Authenticate}(u, q)
$$

$$
\begin{aligned}
U = &\mathsf{new}\ q.\big(\mathsf{assume}\ \ \mathsf{Request}(u, q)\\
&|\mathsf{out}(ch, \mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
P = &(\mathsf{assume}\ \ \mathsf{Registered}(\mathsf{u})\\
&|\mathsf{in}(ch, x).\\
&\mathsf{let}\ x_1 = \mathsf{check}(x, \mathsf{vk}(k_U))\ \ \mathsf{then}\\
&\mathsf{let}\ x_2 = \mathsf{dec}\big(x_1, k_{PE}\big)\ \ \mathsf{then}\\
&\mathsf{let}\ (x_3, x_4) = x_2\ \mathsf{in}\\
&\mathsf{out}(ch, \mathsf{sign}\big(\mathsf{enc}\big((u, x_3), \mathsf{pk}(k_S)\big), k_{PS}\big)))
\end{aligned}
$$

$$
\begin{aligned}
S = &\mathsf{in}(ch, y).\\
&\mathsf{let}\ y_1 = \mathsf{check}(y, \mathsf{vk}(k_{PS}))\ \ \mathsf{then}\\
&\mathsf{let}\ y_2 = \mathsf{dec}\big(y_1, k_S\big)\ \ \mathsf{then}\\
&\mathsf{let}\ (y_3, y_4) = y_2\ \mathsf{in}\\
&\mathsf{assume}\ \ \mathsf{Authenticate}(y_3, y_4)
\end{aligned}
$$

D.2 The Strengthened Protocol

$$
\begin{aligned}
&\mathsf{new}\ k_U.\\
&\mathsf{new}\ k_{PE}.\\
&\mathsf{new}\ k_{PS}.\\
&\mathsf{new}\ k_S.\\
&\mathsf{new}\ p.\\
&\mathsf{out}(ch, \mathsf{pk}(k_{PE})).\\
&\mathsf{out}(ch, \mathsf{pk}(k_S)).\\
&\mathsf{out}(ch, \mathsf{vk}(k_U)).\\
&\mathsf{out}(ch, \mathsf{vk}(k_{PS})).\\
&(\mathsf{assume}\ \ \mathsf{Compromised}(\mathsf{p})\\
&\ |\ \mathsf{assume}\ \mathsf{Compromised}(proxy) \Rightarrow \forall u.\mathsf{Registered}(u)\\
&\ |\ policy \mid U \mid P \mid S)
\end{aligned}
$$

$$
policy = \mathsf{assume}\ \ \forall u, q : \mathsf{Request}(u, q) \wedge \mathsf{Registered}(u) \Rightarrow \mathsf{Authenticate}(u, q)
$$

$$
\begin{aligned}
U = {} & \mathsf{new}\ q.\big(\mathsf{assume}\ \ \mathsf{Request}(u, q)\\
& |\mathsf{out}(ch, \mathsf{zk}_{2,4,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4}\\
& \quad \big(q, p; \mathsf{vk}(k_U), \mathsf{pk}(k_{PE}), \mathsf{sign}\big(\mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big), k_U\big),\\
& \quad \mathsf{enc}\big((q, p), \mathsf{pk}(k_{PE})\big)\big)\big)\big)
\end{aligned}
$$

$$
\begin{aligned}
P = {} & (\mathsf{assume}\ \ \mathsf{Registered}(u)\\
& |\mathsf{in}(ch, \hat{x}).\\
& \mathsf{let}\ \hat{x}_{1,P} = \mathsf{public}_4(\hat{x})\ \ \mathsf{then}\\
& \mathsf{let}\ \big\langle \hat{x}_{1,1}, \hat{x}_{1,2}, \hat{x}_{1,3}, \hat{x}_{1,4} \big\rangle = \hat{x}_{1,P}\ \mathsf{in}\\
& \mathsf{let}\ \hat{x}_{1,V} = \mathsf{ver}_{2,4,2,\beta_4=\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)\wedge\mathsf{check}(\beta_3,\beta_1)\rightsquigarrow\beta_4}\\
& \quad \big(\hat{x}, \mathsf{vk}(k_U), \mathsf{pk}(k_{PE})\big)\ \ \mathsf{then}\\
& \mathsf{let}\ (x, \hat{x}_{1,4}) = \hat{x}_{1,V}\ \mathsf{in}\\
& \mathsf{let}\ x_1 = \mathsf{check}(x, \mathsf{vk}(k_U))\ \ \mathsf{then}\\
& \mathsf{let}\ x_2 = \mathsf{dec}\big(x_1, k_{PE}\big)\ \ \mathsf{then}\\
& \mathsf{let}\ (x_3, x_4) = x_2\ \mathsf{in}\\
& \mathsf{out}(ch, (\mathsf{zk}_{3,9,\mathsf{check}(\beta_5,\beta_4)\rightsquigarrow\beta_9\wedge\mathsf{dec}(\beta_9,\alpha_3)\rightsquigarrow(\alpha_1,\alpha_2)\wedge\beta_3=\mathsf{pk}(\alpha_3)}\\
& \qquad\qquad \wedge\beta_7=\mathsf{enc}((\beta_9,\alpha_1),\beta_2)\wedge\mathsf{check}(\beta_6,\beta_1)\rightsquigarrow\beta_7\\
& \quad \big(x_3, x_4, k_{PE}; \mathsf{vk}(k_{PS}), \mathsf{pk}(k_S), \mathsf{pk}(k_{PE}), \mathsf{vk}(k_U), x,\\
& \quad \mathsf{sign}\big(\mathsf{enc}\big((u, x_3), \mathsf{pk}(k_S)\big), k_{PS}\big), \mathsf{enc}\big((u, x_3), \mathsf{pk}(k_S)\big), u, x_1\big), \hat{x})))
\end{aligned}
$$

$$
\begin{aligned}
S = & \mathsf{in}(ch, \hat{\hat{y}}). \\
& \quad \mathsf{let}\ (\hat{y}_1, \hat{y}_2) = \hat{\hat{y}}\ \mathsf{in} \\
& \quad \mathsf{let}\ \hat{y}_{2,P} = \mathsf{public}_4(\hat{y}_2)\ \ \mathsf{then} \\
& \quad \mathsf{let}\ \left\langle \hat{y}_{2,1}, \hat{y}_{2,2}, \hat{y}_{2,3}, \hat{y}_{2,4} \right\rangle = \hat{y}_{2,P}\ \mathsf{in} \\
& \quad \mathsf{let}\ \hat{y}_{1,P} = \mathsf{public}_9(\hat{y}_1)\ \ \mathsf{then} \\
& \quad \mathsf{let}\ \left\langle \hat{y}_{1,1}, \hat{y}_{1,2}, \hat{y}_{1,3}, \hat{y}_{1,4}, \hat{y}_{1,5}, \hat{y}_{1,6}, \hat{y}_{1,7}, \hat{y}_{1,8}, \hat{y}_{1,9} \right\rangle = \hat{y}_{1,P}\ \mathsf{in} \\
& \quad \mathsf{let}\ \hat{y}_{2,V} = \mathsf{ver}_{2,4,3,\beta_4 = \mathsf{enc}((\alpha_1,\alpha_2),\beta_2) \wedge \mathsf{check}(\beta_3,\beta_1) \rightsquigarrow \beta_4} \\
& \qquad \left(\hat{y}_2, \mathsf{vk}(k_U), \mathsf{pk}(k_{PE}), \hat{y}_{1,5}\right)\ \ \mathsf{then} \\
& \quad \mathsf{let}\ \hat{y}_{1,V} = \mathsf{ver}_{3,9,5,\mathsf{check}(\beta_5,\beta_4) \rightsquigarrow \beta_9 \wedge \mathsf{dec}(\beta_9,\alpha_3) \rightsquigarrow (\alpha_1,\alpha_2) \wedge \beta_3 = \mathsf{pk}(\alpha_3)} \\
& \qquad\qquad\qquad\qquad \wedge \beta_7 = \mathsf{enc}((\beta_9,\alpha_1),\beta_2) \wedge \mathsf{check}(\beta_6,\beta_1) \rightsquigarrow \beta_7 \\
& \qquad \left(\hat{y}_1, \mathsf{vk}(k_{PS}), \mathsf{pk}(k_S), \mathsf{pk}(k_{PE}), \mathsf{vk}(k_U), \hat{y}_{1,5}\right)\ \ \mathsf{then} \\
& \quad \mathsf{let}\ \hat{y}_{2,4} = \hat{y}_{2,V}\ \mathsf{in} \\
& \quad \mathsf{let}\ \left\langle y, \hat{y}_{1,7}, \hat{y}_{1,8}, \hat{y}_{1,9} \right\rangle = \hat{y}_{1,V}\ \mathsf{in} \\
& \quad \mathsf{let}\ y_1 = \mathsf{check}(y, \mathsf{vk}(k_{PS}))\ \ \mathsf{then} \\
& \quad \mathsf{let}\ y_2 = \mathsf{dec}\left(y_1, k_S\right)\ \ \mathsf{then} \\
& \quad \mathsf{let}\ (y_3, y_4) = y_2\ \mathsf{in} \\
& \quad \mathsf{assume}\ \ \mathsf{Authenticate}(y_3, y_4)
\end{aligned}
$$

D.3 Spi2RCF: The Original Protocol Converted to RCF

typedef $PrivateUnlessP = \{sCompromised(p)\} \rightarrow \{sCompromised(p)\}$
typedef $t1 = q : \mathsf{Un} * (PrivateUnlessP * \{sRequest(u, q)\})$
typedef $t2 = (u\_1 : \mathsf{Un} * (q : \mathsf{Un} * \{((sRequest(u\_1, q) \wedge sRegistered(u\_1)) \vee sCompromised(p))\}))$
$(\nu ch \updownarrow \mathsf{Un})$
$(\nu u \updownarrow \mathsf{Un})$
$\mathsf{let}\ k\_US = mkSK\ \langle (k : ek\ \langle t1 \rangle *$
$\quad \{x : \mathsf{Un} \mid \exists Y, Z1, Z2.\ (encrypted(k, pair(Z1, pair(Z2, Y)), x) \wedge sRequest(u, Z1))\})\rangle\ \ ()\ \mathsf{in}$
$\mathsf{let}\ k\_PE = mkDK\ \langle t1 \rangle\ \ ()\ \mathsf{in}$
$\mathsf{let}\ k\_PS = mkSK\ \langle (k : ek\ \langle t2 \rangle *$
$\quad \{x : \mathsf{Un} \mid \exists Y, Z1, Z2.\ (encrypted(k, pair(Z1, pair(Z2, Y)), x)$
$\quad \wedge\ ((sRequest(Z1, Z2) \wedge sRegistered(Z1)) \vee sCompromised(p)))\})\rangle\ \ ()\ \mathsf{in}$
$\mathsf{let}\ k\_SE = mkDK\ \langle t2 \rangle\ \ ()\ \mathsf{in}$
$\mathsf{let}\ vk\_US = mkVK\ \ k\_US\ \mathsf{in}$
$\mathsf{let}\ pk\_PE = mkEK\ \ k\_PE\ \mathsf{in}$
$\mathsf{let}\ vk\_PS = mkVK\ \ k\_PS\ \mathsf{in}$
$\mathsf{let}\ pk\_SE = mkEK\ \ k\_SE\ \mathsf{in}$
$ch!pk\_PE; ch!pk\_SE; ch!vk\_US; ch!vk\_PS;$
(
$\quad \mathsf{assume}\ \forall Cu, Cq.\ ((sRequest(Cu, Cq) \wedge sRegistered(Cu)) \implies sAuthenticate(Cu, Cq))$
) ↱ (
$\quad$ (
$\quad\quad$ (
$\quad\quad\quad \mathsf{assume}\ \neg sCompromised(p)$
$\quad\quad$) ↱ (

```
        assume ∀Cu. (sCompromised(p) ⟹ sRegistered(Cu))
    )
) ↱ (
    (
        (νp ↕ {anon4 : Un | sCompromised(p)})
        let q = mkUn () in
        (
            assume sRequest(u, q)
        ) ↱ (
            let tosend = let temp2 = encrypt pk_PE in
            temp2 (q, (p, ())) in
            let tanf1 = let temp1 = sign k_US in
            temp1 (pk_PE, tosend) in
            ch!(tanf1, tosend)
        )
    ) ↱ (
        (
            (
                assume sRegistered(u)
            ) ↱ (
                let xp = ch? in
                let (x, km) = xp in let temp7 = check vk_US in
                let temp8 = temp7 x in
                let kmp = temp8 km in
                let (dk, x_1) = kmp in if dk = pk_PE then
                    let temp5 = decrypt k_PE in
                    let temp6 = temp5 x_1 in
                    let x_2 = temp6 in
                    let (x_3, x_4, dummy1) = x_2 in if dummy1 = () then
                        let tosendp = let temp4 = encrypt pk_SE in
                        temp4 (u, (x_3, ())) in
                        let tanf2 = let temp3 = sign k_PS in
                        temp3 (pk_SE, tosendp) in
                        ch!(tanf2, tosendp)
            )
        ) ↱ (
            let yp = ch? in
            let (y, kmp) = yp in let temp11 = check vk_PS in
            let temp12 = temp11 y in
            let kmpp = temp12 kmp in
            let (dk, y_1) = kmpp in if dk = pk_SE then
                let temp9 = decrypt k_SE in
                let temp10 = temp9 y_1 in
                let y_2 = temp10 in
                let (y_3, y_4, dummy4) = y_2 in if dummy4 = () then
                    assert sAuthenticate(y_3, y_4)
        )
    )
)
```

)