Achieving Security Despite Compromise Using Zero-knowledge

Michael Backes[1,2], Martin Grochulla[1], Cătălin Hriţcu[1], Matteo Maffei[1]

[1]Saarland University, Saarbrücken, Germany [2]MPI-SWS

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 fulfill authorization policies despite participant compromise. The central idea is to automatically transform the original cryptographic protocols by adding zero-knowledge proofs. These proofs allow the participants to convince remote parties that they followed the protocol, without revealing any secret data. We use an enhanced type system for zero-knowledge to verify that the transformed protocols are secure despite compromise. Both the protocol transformation and the verification are fully automated.

This paper is intended for presentation only and should not be included in the formal proceedings.

1 Introduction

A central challenge in the design of security protocols for modern applications is the ability to automatically devise abstract security protocols that satisfy specifications of sophisticated 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 consider strengthening security protocols so that they withstand attacks even in the presence of participant compromise. 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 study of the specification of the protocol.

Strengthening security protocols so that they achieve security despite compromise naturally calls for incorporating the most prominent and innovative modern cryptographic primitive in their design: zero-knowledge proofs [16]. 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 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 [11], 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 [10]. 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 [19,6].

1.1 Our Contributions

We present a general technique for strengthening security protocols in order to fulfill authorization policies despite participant compromise, as well as an enhanced type system for verifying that the strengthened protocols are secure despite compromise.

The central idea is to automatically transform the original security protocols by suitably including non-interactive zero-knowledge proofs that allow a participant to convince other participants that she correctly followed the protocol, without revealing any of her secret data. Our approach is general and can strengthen any protocol based on public-key cryptography, digital signatures, hashes, and symmetric-key cryptography. Moreover, the transformation automatically derives proper type annotations for the strengthened protocol provided that the original protocol is augmented with type annotations. This frees protocol designers from inspecting the strengthened protocol to conduct a successful security analysis, but instead solely requires them to inspect the original, simpler protocol.

The type system extends our previous type system for zero-knowledge [5] 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 to express type information that is conditioned by a participant being uncompromised. 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.

Due to space constraints, we postpone the discussion on related work to Appendix A.

1.2 Outline

Section 2 uses an illustrative example to explain the intuition behind our transformation. Section 3 describes the transformation of this example in detail. Our enhanced type system for zero-knowledge is presented in Section 4. Section 5 concludes and provides directions for future work. Appendix B describes the syntax and semantics of the calculus we use to specify both the original and the transformed protocols. Finally, Appendix C lists the complete code of the example using asymmetric cryptography, before and after the transformation. Due to space constraints, we defer all the technical details of the transformation and of the enhanced type system for zero-knowledge to an extended version of the paper [2]. Both the implementation of the transformation [3] and that of the type-checker [5] 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.

User | Proxy | Store

assume $\mathsf{Request}(u, q)$

$\xrightarrow{\mathsf{sign}(\mathsf{enc}((q,p),\mathsf{pk}(k_{PE})),k_U)}$

assume $\mathsf{Registered}(u)$

$\xrightarrow{\mathsf{sign}(\mathsf{enc}((u,q),\mathsf{pk}(k_S)),k_{PS})}$

assert $\mathsf{Authenticate}(u, q)$

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 [14,5], 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 if u has indeed sent that query and u is registered in the system. This security property is formulated as the following authorization policy:

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

2.2 Security despite compromised participants

If all the participants follow the protocol, then the protocol above satisfies the authorization policy, i.e., it is robustly safe. The 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 p by (a) revealing all her secrets to the attacker; (b) introducing the assumption $\mathsf{Compromised}(p)$; (c) removing the code of p, since it is controlled by the attacker; and (d) introducing assumptions of the form $\mathsf{assume}\ \mathsf{Compromised}(p) \Rightarrow F$ for each assumption F in the code of p. For instance, we add the following assumption for the compromise of user u:

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

meaning that the attacker might contact the proxy to authenticate any query in the name of u. We can easily see that the authorization policy is satisfied since the only way for the attacker to interact with the honest participants is to follow the protocol and, by impersonating the user u, to authenticate a query with a valid password. This attack is, however, harmless since the attacker is just following the protocol. The authorization policy (1) is vacuously satisfied if the store is compromised, since no assertion has to be justified, and if both the proxy and the user are compromised, since the two hypotheses in the authorization policy (1) are always entailed. Therefore the only interesting case is when the proxy is compromised and the other participants are not. In this case, even after introducing the assumption

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

the authorization policy is not satisfied since the proxy might just send the second message without the user sending any query, which would lead to an $\mathsf{Authenticate}(u,q)$ assertion not preceded by a $\mathsf{Request}(u,q)$ assumption.

As suggested in [14], we could document the attack by weakening the authorization policy. This could be achieved by introducing a logical formula stating that the proxy *controls* the assumption $\mathsf{Request}(u,q)$, i.e., if the proxy is compromised, then all $\mathsf{Request}(u,q)$ assumptions are entailed in the logic.

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 the protocol so that the attack is prevented and the authorization policy is satisfied even in the presence of compromised participants.

2.3 Strengthening the protocol

Before illustrating our approach, we briefly recap the technique introduced in [6] to symbolically represent zero-knowledge proofs. Zero-knowledge proofs are expressed as terms of the form $\mathsf{zk}_S(\widetilde{M};\widetilde{N})$[1], where S is the statement, which is a Boolean formula built over cryptographic operations and place-holders α_i and β_j referring to the terms M_i and N_j, respectively. The terms M_i form the *private component* of the proof and they will never be revealed, while the terms N_i form the *public component* of the proof and they are revealed to the verifier. The verification of a zero-knowledge proof succeeds if and only if the

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

statement $S\{\widetilde{M}/\widetilde{\alpha}\}\{\widetilde{N}/\widetilde{\beta}\}$ obtained after the instantiation of the place-holders holds true. For instance, $\mathsf{zk}_{\mathsf{check}(\alpha_1,\beta_1)\to\alpha_2}(\mathsf{sign}(m,k),m;\mathsf{vk}(k))$ 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 B for more detail.

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 principal 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:

$$\begin{tikzcd}
\text{User} \arrow[r, "ZK_1"] & \text{Proxy} \arrow[r, "{ZK_1,ZK_2}"] & \text{Store}
\end{tikzcd}$$

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

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

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

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

The first zero-knowledge proof proves 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 ($\mathsf{check}(\beta_3,\beta_1)\to\beta_4$) and the result is the encryption of the query and the password with the proxy's encryption key ($\mathsf{enc}((\alpha_1,\alpha_2),\beta_2)=\beta_4$). Notice that we model proofs of knowledge, i.e., the user proves to know the secret query α_1 and the secret password α_2.

The public component contains only messages that were public in the original protocol. The query and the password occur in the private component since they were encrypted in the original protocol and they should be thus kept secret. Furthermore, notice that the statement of the zero-knowledge proof 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). In general, the statement of the generated zero-knowledge proof is computed as the conjunction of the single 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 ($\mathsf{check}(\beta_5,\beta_4)\to\beta_9$) of an encryption of two secret terms α_1 and α_2 ($\mathsf{dec}(\beta_9,\alpha_3)\to(\alpha_1,\alpha_2)$). The zero-knowledge proof additionally ensures that the message β_6 sent by the proxy is the signature ($\mathsf{check}(\beta_6,\beta_1)\to\beta_7$) of an encryption of the user's name and the query α_1 received from the user ($\beta_7 = \mathsf{enc}((\beta_8, \alpha_1), \beta_2)$). We remark that this zero-knowledge proof states that the query α_1 signed by the user is the same as the one signed by the proxy. Notice also that the proof does not reveal the secret password α_2 received from the user.

The resulting protocol does fulfill the authorization policy since the proxy, even if compromised, 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 that query from the user.

3 Transformation

This section presents our technique for strengthening cryptographic protocols with zero-knowledge proofs. The transformation is illustrated on the protocol from Section 2.

3.1 Overview of the Algorithm

The first step of the transformation consists of a *code inspection*, which for each output message scans the process and returns the list of secret values and the list of public values occurring in that message. For instance, restricted names are regarded as private as well as decryption keys and variables storing the result of a decryption. Free names, public keys, and verification keys are instead considered public. This information is used when generating zero-knowledge proofs to determine what terms should occur in the private component and what terms should instead occur in the public component.

The second step consists of a *dependency analysis*, which returns the list of inputs which each output depends on. For instance, in the protocol from Section 2, the message output by the proxy contains a query received from the user. Therefore the output performed by the proxy depends on the previous input. Additionally, by relying on syntactic annotations provided by the user, each input is linked to the corresponding output. For instance, in the protocol from Section 2, the input on the proxy's side is linked to the output on the user's side. This information is used to determine what zero-knowledge proofs have to be forwarded together with the zero-knowledge proof replacing the output message.

The third step consists of a *zero-knowledge proof generation*, which replaces each output message by the corresponding zero-knowledge proof and the zero-knowledge proofs that have to be forwarded, i.e., the zero-knowledge proofs received in the inputs which the output depends on. The fourth step consists of a *zero-knowledge verification generation*, which after each input introduces the code for the verification of the zero-knowledge proofs that are received in that input.

The last step is the *transformation of types*. We assume that the user provides typing annotations for the original protocol and our algorithm generates typing annotations for the protocol extended with zero-knowledge proofs. The type system is discussed in Section 4.

3.2 Transforming the example

The spi calculus code for the user in our example from Section 2 is as follows:

$$\mathsf{new}\ q.(\mathsf{assume}\ \mathsf{Request}(u, q) \mid \mathsf{out}(ch, \mathsf{sign}(\mathsf{enc}((q, p), \mathsf{pk}(k_{PE})), k_U)))$$

The names p, k_U, k_{PE} are bound by restrictions located in front of the parallel composition of the processes modelling the user, the proxy, and the store. The code inspection returns

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

Restricted names are regarded as secret, while public keys and verification keys are considered public. The zero-knowledge proof generation is defined by induction on the structure of the term output in the original protocol. In the following, we show how the zero-knowledge proof generation constructs 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, an empty private component, and an empty public component. Since the considered term is a signature, we introduce the statement $\mathsf{check}(\beta_1, \beta_2) \rightarrow \beta_3$, thus 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 actually 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 public, since it is sent on a public channel, and the same holds for the verification key and the ciphertext.

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

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 since they are secret, while the proxy's public key $\mathsf{pk}(k_{PE})$ is added to the public component.

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

Finally, the terms in the public component are rearranged 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, whose semantics requires the messages known to the verifier, which are checked for equality in the verification, to be in the first part of the public component. The result of this rearrangement is the zero-knowledge proof shown in Section 2, which replaces the message $\mathsf{sign}(\mathsf{enc}((q,p), \mathsf{pk}(k_{PE})), k_U)$ originally output by the user.

We now illustrate how the the zero-knowledge proof ZK_2 is generated. The spi calculus code for the proxy in our example is as follows:

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

The code inspection returns

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

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 says that the proxy has signed an encryption of the pair consisting of the name u and of a secret term.

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

As opposed to the term output by the user, the signature output by the proxy contains a variable x_2. The zero-knowledge proof describes how the variable x_2 has been obtained. This information is returned by the dependency analysis and we have that $\mathsf{check}(x, \mathsf{vk}(k_U)) \rightarrow x_1$ and $\mathsf{dec}(x_1, k_{PE}) \rightarrow (x_2, x_3)$. These equalities are translated into the following statement:

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

The statement guarantees that the query x_2 and the secret password are obtained by the decryption of a ciphertext ($\mathsf{dec}(\beta_6, \alpha_3)$) signed by the user ($\mathsf{check}(\beta_8, \beta_9) \rightarrow \beta_6$). 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 is obtained by rearranging the terms in the public component, as previously discussed. Finally, the signature output in the original protocol is replaced by the zero-knowledge proof ZK_2 and the zero-knowledge proof received from the user. Due to space constraints, we omit the presentation of the transformation for protocols based on symmetric-key cryptography and the generation of the code for the verification of the zero-knowledge proofs. We refer the interested reader to [2]. The code for the original protocol and the code obtained after the transformation is shown in Appendix C.

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. The current type system improves our previous type system for zero-knowledge [5] to also work in the case 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). We also add union and intersection types to [5] (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. Type Un (untrusted) describes messages possibly known to the adversary, while messages of type $\mathsf{Private}$ 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 channel where the attacker can read and write messages, modeling a public network like the Internet.

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. As in [14,8] 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. In the following we use $\{T \mid C\}$ to denote $\{x : T \mid C\}$ if x does not appear in C.

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. We remark that 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. 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 be used to sign private messages for which we know that the Good predicate holds.

Typing the Original Example (Uncompromised Setting) We are going to illustrate the type system on the original protocol from Section 2. Since the query q the user sends to the proxy is not sensitive we give it type Un. 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 $\mathsf{Pair}(x_q : \mathsf{Un}, \mathsf{Private})$. This type is, however, not strong enough to convince the proxy that the predicate $\mathsf{Request}(u, q)$ holds for the values it receives. For this we give (q, p) the stronger type $T_1 = \mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\})$. This type ensures not only that q and p are of the right type, but also that the predicate $\mathsf{Request}(u, q)$ holds.

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 $T_2 = \mathsf{Pair}(x_u : \mathsf{Un}, \{x_q : \mathsf{Un} \mid \mathsf{Request}(x_u,x_q) \wedge \mathsf{Registered}(x_u)\})$, 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 indeed justified by the policy, so if all participants are honest the original protocol is secure (robustly safe).

4.2 Kinding and Subtyping

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.

Our type system has two special types $\top$ (top) and $\bot$ (bottom). Type $\top$ is a supertype of any type, while $\bot$ is the empty type that is a subtype of all types.

A useful property of refinement types in our type system is that $\{x : T \mid C\}$ is equivalent by subtyping[2] 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 a always a subtype of T and the transitivity of subtyping, this also 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 (in this case $\{x : T \mid C\} <:> \bot <: \mathsf{Un}$). Conversely, type $\{x : T \mid C\}$ is tainted if T is tainted and additionally $\forall x.C$ holds. In the following,

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

we use $\{\!|\, C \,|\!\}$ to denote the refinement type $\{x : \top \mid C\}$, where x does not appear in C. Note that type $\{\!|\, C \,|\!\}$ is either equivalent to $\top$ (if C holds), or to $\bot$ (if $\neg C$ holds)[3].

4.3 Logical Characterization of Kinding

We capture the precise conditions that an 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 an environment if and only if type T is public in this environment. In a similar way tnt provides a logical characterization of taintedness. Since in the current type system we do not have any unconditionally secure types these two functions are total.

On the one hand, since type Un is always public and tainted and true is valid in any environment, we have $\mathsf{pub}(\mathsf{Un}) = \mathsf{tnt}(\mathsf{Un}) = \mathsf{true}$. On the other hand 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[4], so equivalent to Un. So we have that $\mathsf{pub}(\mathsf{Private}) = \mathsf{tnt}(\mathsf{Private}) = \mathsf{false}$.

As intuitively explained at the end of the last 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 $\mathsf{pub}(\mathsf{SigKey}(T)) = \mathsf{tnt}(T)$.

4.4 Union and Intersection Types

We extend the type system from [5] with union and intersection types [18]. A message has type $T \wedge U$ if and only if it has both type T and type U. Also, if a message has type T then it also has type $T \vee U$ for any U. 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. We can define a type $\mathsf{PrivateUnlessP}$ that is private if and only if p is uncompromised as $\{\mathsf{Private} \mid \neg\mathsf{Compromised}(p)\} \vee \{\mathsf{Un} \mid \mathsf{Compromised}(p)\}$ (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.

[3] We assume a classical authorization logic that fulfills the law of excluded middle.

[4] In an inconsistent environment all assertions are going to be 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.

4.5 Compromising Participants

As explained in Section 2.2 when a participant p is compromised all its secrets are revealed to the attacker and the predicate $\mathsf{Compromised}(p)$ 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 $\mathsf{PrivKey}(T_1)$[5] 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 $T_1' = \mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Un} \mid \mathsf{Request}(u, x_q)\})$. 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, containing types that are secure only under the condition that certain principals are uncompromised, for both the uncompromised and the compromised scenarios.

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 $T_1 = \mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\})$, is now changed to $T_1^* = \mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{PrivateUnlessP} \mid \mathsf{Request}(u, x_q)\})$. In the uncompromised setting $\neg\mathsf{Compromised}(p)$ 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 $\mathsf{Compromised}(p)$ is entailed, $\mathsf{PrivateUnlessP}$ is equivalent to Un and T_1^* is equivalent to T_1'. Using this type T_1^* 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. 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 $T_2^* = \{T_2 \mid \neg\mathsf{Compromised}(p)\} \vee \{\mathsf{Un} \mid \mathsf{Compromised}(p)\}$. 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 both 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^*. However, 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 despite the compromise of the proxy.

[5] Where T_1 denotes $\mathsf{Pair}(x_q : \mathsf{Un}, \{x_p : \mathsf{Private} \mid \mathsf{Request}(u, x_q)\})$

4.6 Type-checking Zero-Knowledge Proofs

As first done in [3], 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 2.3 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 $S_1 \triangleq \mathsf{enc}((\alpha_1,\alpha_2),\beta_2) = \beta_4 \wedge \mathsf{check}(\beta_3,\beta_1) \to \beta_4$, is given type:

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

where $C_1 = \mathsf{enc}((x_1,x_2),y_2) = y_4 \wedge \mathsf{check}(y_3,y_1) \to y_4 \wedge \mathsf{Request}(u,x_2)$. 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 only succeeds if the statement is valid.

However, very often conveying the statement alone does not suffice to type-check the examples we have tried, since it 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.

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 that is created by the user, assumed 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 $\mathsf{enc}((x_1,x_2),y_2) = y_4 \wedge \mathsf{check}(y_3,y_1) \to y_4$ holds, the type-checker tries to infer additional information.

Since y_1 has type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ and $\mathsf{check}(y_3,y_1) \to y_4$ holds, we infer that y_3 also has type $\mathsf{PubEnc}(T_1^*) \vee \{\!|\, \mathsf{tnt}(\mathsf{PubEnc}(T_1^*)) \,|\!\}$, i.e., y_3 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 then the type $\mathsf{VerKey}(\mathsf{PubEnc}(T_1^*))$ is equivalent to Un. However, intuitively this is not the case since the user is not compromised. So the new type inferred for y_3 is equivalent to $\mathsf{PubEnc}(T_1^*) \vee \bot$ and therefore to $\mathsf{PubEnc}(T_1^*)$. Since y_3 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_3 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 $T_1^* \vee \{\!|\, \mathsf{tnt}(T_1^*) \,|\!\}$. We have already showed that $\mathsf{tnt}(T_1^*) = \mathsf{false}$ 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 equalities between terms. These equalities are, however, critical for linking the different messages. Most importantly, they ensure that the query received in ZK_2 is the same as the one 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 the authentication decision of the store is indeed justified by the authorization policy.

5 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 zero-knowledge proofs that are used by each participant to convince remote parties that she has followed the protocol, without revealing any of her secrets. Furthermore zero-knowledge proofs are forwarded by participants 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. Both the implementation of the transformation [3] and the one of the type-checker [5] are available online.

We plan to work on the efficiency of our transformation. For instance, we can apply techniques similar to the ones proposed in [12,9] to reduce the number of messages forwarded by each principal.

In this work we use a type-checker to check whether or not the transformed process is robustly safe in the compromise scenario. Although we did not find counterexamples in our experiments, a formal proof would give us more confidence in the correctness of the transformation without relying on the type-checker. This would not only be useful in theory, but also in practice, in case the type-checker would fail to terminate when proving the strengthened (and therefore more complex) protocol secure.

We are currently working on the automated generation of F# protocol implementations from spi calculus processes. The goal is to automatically generate executable protocol implementations that are secure despite principal compromise. For the concrete implementation of zero-knowledge proofs, we intend to rely on a recently proposed library of efficient implementations of sigma-protocols [7], which can be made non-interactive with the Fiat-Shamir heuristic.

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. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise with zero-knowledge. Long version available at `http://www.infsec.cs.uni-sb.de/~hritcu/publications/zk-compromise-full.pdf`, 2009.
3. M. Backes, M. P. Grochulla, C. Hriţcu, and M. Maffei. Achieving security despite compromise with zero-knowledge. Implementation available at `http://www.infsec.cs.uni-sb.de/projects/zk-compromise/`, 2009.
4. M. Backes, C. Hriţcu, and M. Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 195–209. IEEE Computer Society Press, 2008.
5. M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. In *15th ACM Conference on Computer and Communications Security (CCS 2008)*, pages 357–370. ACM Press, 2008. Implementation available at: `http://www.infsec.cs.uni-sb.de/projects/zk-typechecker/`.
6. 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.
7. E. Bangerter, J. Camenisch, S. Krenn, A. Sadeghi, and T. Schneider. Automatic generation of sound zero-knowledge protocols. IACR Cryptology ePrint Archive: Report 2008/471, Nov. 2008. `http://eprint.iacr.org/`.
8. 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.
9. K. Bhargavan, R. Corin, P.-M. Dénielou, C. Fournet, and J. J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. Submitted for publication, July 2008. Available at: `http://www.msr-inria.inria.fr/projects/sec/sessions/`.
10. 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.
11. 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.
12. 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.
13. V. Cortier, B. Warinschi, and E. Zălinescu. Synthesizing secure protocols. In J. Biskup and J. Lopez, editors, *Proceedings of the 12th European Symposium On Research In Computer Security (ESORICS'07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 406–421, Dresden, Germany, September 2007. Springer.
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, 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.
16. 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`.
17. J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology: CRYPTO '03*, pages 110–125. Springer-Verlag, 2003.
18. B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.
19. 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*, number 4572 in Lecture Notes in Computer Science, pages 218–231. Springer-Verlag, 2007.

A Related Work

Security despite compromised participants was introduced by Fournet et al. [14]. The authors conjecture 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 principals and achieving security despite compromise.

The closest work to ours is by Corin et al. [12], 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 transformation does not consider secrecy (all messages are assumed to be public) or payload binding (the generated protocols are susceptible to message substitution attacks), but these limitations have recently been addressed by the authors in [9]. While the older transformation was proven correct, the more recent one relies on a type-checker [8] for verifying that each of the generated cryptographic implementations is secure. We also take the latter approach here, but plan to investigate the correctness of the compiler in the future.

The main difference with respect to [12,9] 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. In addition, assuming that the original protocol is provided with suitable typing annotations, in our approach both the transformation and the verification are fully automated. In [9], the verification requires the user to provide "a brief, hand-crafted argument to complete the proof" of the generated protocols and sometimes also to strengthen the generated types. On the other hand, the transformation proposed in [9] returns executable protocol implementations, while in this paper we focus on designing stronger protocol specifications, and leave the automated implementation of these strengthened protocols as future work.

Goldreich et al. transform secure multi-party computation protocols that are secure against honest but curious adversaries into protocols secure against compromised participants [15]. They also employ zero-knowledge proofs, but they work in the setting of computational cryptography, while we work in the symbolic one. Furthermore, their solution requires broadcast communication for the generated protocol, while we only need point-to-point communication. Katz and Yung [17], and later Véronique Cortier et al. [13] proposed transformations from protocols secure against passive, eavesdropping attackers to protocols secure against active ones.

B Spi calculus with Destructors

In this paper we consider a variant of the spi calculus with arbitrary constructors and destructors. The calculus is similar to the ones in [1,14], and was extended to zero-knowledge proofs in [5]. This section overviews the syntax and semantics of the calculus.

B.1 Constructors and Terms

Constructors are function symbols that are used to build terms. The set of constructors we consider in this paper[6] 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 1), 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 1 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 $(M_1, (M_2, \ldots, (M_n, \mathsf{ok})))$.

B.2 Destructors

Destructors are partial functions that processes can apply to terms, and are ranged over by g (Table 2). The semantics of destructors is specified by the reduction relation $\Downarrow$ (Table 3): 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.

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

Table 2 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 $l \in [1, m]$.

Table 3 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.

B.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 $\rightarrow$ predicate capturing the destructor reduction relation, as well as conjunctions and disjunctions of such basic statements. The syntax of *statements* is given in Table 4.

Table 4 Syntax of statements

$S, P ::=$	statements
$g(M_1, \ldots, M_n) \rightarrow N$	destructor reduction (g of arity n)
$M = N$	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 $\mathsf{zk}_{1,2,\mathsf{eq}^\sharp(\beta_1, \mathsf{dec}^\sharp(\mathsf{enc}(\beta_1,\beta_2),\alpha_1))}(\ k\ ;\ m, \mathsf{pk}(k)\ )$ 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.

Table 5 Semantics of statements $[\![S]\!] \in \{\mathsf{true}, \mathsf{false}\}$

$$[\![g(\widetilde{M}) \rightarrow N]\!] = \begin{cases} \mathsf{true} & \text{if } g \neq \mathsf{ver} \text{ and } g(\widetilde{M}) \Downarrow N \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$[\![M = N]\!] = \begin{cases} \mathsf{true} & \text{if } M = N \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\begin{aligned} [\![S_1 \wedge S_2]\!] &= [\![S_1]\!] \wedge [\![S_2]\!] \\ [\![S_1 \vee S_2]\!] &= [\![S_1]\!] \vee [\![S_2]\!] \\ [\![\mathsf{true}]\!] &= \mathsf{true} \\ [\![\mathsf{false}]\!] &= \mathsf{false} \end{aligned}$$

The semantics of statements is defined in Table 5. The semantics of the $\rightarrow$ predicate is defined in terms of the reduction relation for destructors, unless the destructor is ver in which case the statement is simply false[7].

B.4 Processes

Additional to the processes from [14] and [5], 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.

B.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,

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

Table 6 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 union types
	$\mathsf{assume}\ C$	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.

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. We assume an additional $\rightarrow$ relation directly corresponding to the reduction relation for destructors (with the exception of ver). 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). Finally, the logic is assumed to include some of the usual logical connectives with their canonical meaning: true, false, $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\exists$, and $\forall$.

Note that, unlike in the earlier versions of this type system [5], **we assume that the logic is classical** and not intuitionistic – the law of the excluded middle is an explicit prerequisite on the logic.

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

B.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 ($P \rightarrow Q$). *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 7.

Table 7 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 defines the semantics of process synchronizations and conditionals. It is the smallest relation on closed processes satisfying the rules in Table 8.

Table 8 Internal reduction $P \rightarrow Q$

(RED-I/O)	$\mathsf{out}(a, M).P \mid \mathsf{in}(a, x).Q \rightarrow P \mid Q\{M/x\}$
(RED-!I/O)	$\mathsf{out}(a, M).P \mid !\mathsf{in}(a, x).Q \rightarrow P \mid Q\{M/x\} \mid !\mathsf{in}(a, x).Q$
(RED-DESTR)	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow P\{N/x\}$, *if* $g(\widetilde{M}) \Downarrow N$
(RED-ELSE)	$\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow Q$, *if* $g(\widetilde{M}) \not\Downarrow$
(RED-SPLIT)	$\mathsf{let}\ (x, y) = (M, N)\ \mathsf{in}\ P \rightarrow P\{M/x\}\{N/y\}$
(RED-IF)	$\mathsf{if}\ M = M\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow P$
(RED-ELSE)	$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow Q$, *if* $M \neq N$
(RED-CASE)	$\mathsf{case}\ x = M\ \mathsf{in}\ P \rightarrow P\{M/x\}$
(RED-CTXT)	$\mathcal{E}[P] \rightarrow \mathcal{E}[Q]$, *if* $P \rightarrow Q$
(RED-EQ)	$P \rightarrow Q$, *if* $P \equiv P', P' \rightarrow Q'$, and $Q' \equiv Q$

C The Complete Example

C.1 The original protocol

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

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

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

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

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

C.2 The strengthened protocol

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

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

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

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

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