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 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 behaviour 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. Both the protocol transformation and the verification are fully automated.

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 strengthen security protocols so that they withstand attacks even in the presence of compromised participants. The notion of "security despite compromise" [13] 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 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[1] [14]. Zero-knowledge proofs go beyond the traditional understanding of cryptography that only ensures secrecy and authenticity of a communication. This primitive's unique

[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.

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 [9], 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 [8]. 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 [19,4].

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 [3] 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.

[2] Please note that 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.

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 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. [13]. 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. [10], 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 [7]. While the older transformation was proven correct, the more recent one relies on a type-checker [6] for verifying that each of the generated cryptographic implementations is secure.

The main difference with respect to [10,7] 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 [7], the verification requires the user to provide “a brief, hand-crafted argument to complete the proof” of the generated protocols and sometimes also to manually strengthen the generated types. On the other hand, the transformation proposed in [7] returns executable protocol implementations, while in this paper we focus on designing stronger protocol specifications.

Katz and Yung [15], and later Cortier et al. [11] 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 [5]. Datta et al. [12] propose a methodology for modular development of protocols where security properties are added to a protocol through generic transformations. Abadi et al. [1] give a compiler for protocols using secure channels into implementations that use cryptography. None of these transformations considers security despite compromised participants.

Translation validation, as introduced by Pnueli et al. [18], is an accepted technique to detect compiler bugs and prevent 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 [16,20], to model checking and theorem proving [18]. In the current paper we use a type-checker to validate the results of our translation.

Fournet et al. [13] 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 [3] 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 refinement [6], union and intersection types [17].

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 5 concludes and provides directions for future work. Due to space constraints, we defer many 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 [2] and that of the type-checker [3] 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. [13]. 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 [2] we discuss how our transformation handles symmetric encryption and apply these ideas to the original protocol by Fournet et al. [13].

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

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 [13,3], 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 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 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 $\mathsf{Compromised}(v)$. For simplicity 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:

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

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

Even when the user is compromised the protocol is still robustly safe since the only way for the attacker to interact with the honest proxy is to follow the protocol and, by impersonating the user u, 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 also safe 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, 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 the user sending any query, 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 authorization policy (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 [13], 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 any protocol so that such attacks are prevented by enforcing robust safety even in the presence of compromised participants.

2.3 Symbolic Representation of Zero-knowledge

Before illustrating our approach, we briefly recap the technique introduced in [4] to symbolically represent zero-knowledge proofs. Zero-knowledge proofs are expressed as terms of the form $\mathsf{zk}_S(\widetilde{M}; \widetilde{N})$[3]. The statement S of the zero-knowledge

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

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 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 statement $S\{\widetilde{M}/\widetilde{\alpha}\}\{\widetilde{N}/\widetilde{\beta}\}$, i.e., the statement obtained after the instantiation of the place-holders, 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.

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 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:

$$\begin{array}{ccc} \text{User} & \text{Proxy} & \text{Store} \\ & \xrightarrow{\quad ZK_1 \quad} & \\ & & \xrightarrow{\quad ZK_1, ZK_2 \quad} \end{array}$$

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

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

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

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

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 ($\mathsf{enc}\big((\alpha_1,\alpha_2),\beta_2\big) = \beta_4$). 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 are placed in the private component since they were encrypted in the original protocol and could be secrets[4]. Furthermore,

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

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). 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 ($\beta_7 = \mathsf{enc}((\beta_8, \alpha_1), \beta_2)$). We remark that 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.

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 [2].

3.1 High-level Overview of the Transformation

Our transformation relies on the information obtained from several static analyses of the original protocol. One such analysis is *secrecy analysis*, which for each output message returns the secret values and the public values occurring in that message. For instance, restricted names are regarded as private together with 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.

We also use a *data-dependency analysis* that for each output performed by a participant computes the previous inputs performed by this participant on which the output (directly or indirectly) depends, 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. Additionally, our transformation assumes that each input is linked to the corresponding output. This information is used to determine what zero-knowledge proofs have to be forwarded together with the zero-knowledge proof replacing the output message.

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 zero-knowledge proofs that have to be forwarded, i.e., the zero-knowledge proofs received in the inputs on which the output depends. 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 consists of a *zero-knowledge verification generation*, which after each input introduces additional checks that verify all the received zero-knowledge proofs. 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. 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:

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

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

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

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.

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

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

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

Finally, the terms in the public component are 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. 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:

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

The secrecy analysis returns

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

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.

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

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:

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

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 in the original protocol is replaced by the zero-knowledge proof ZK_2 together with the zero-knowledge proof received from the user, on which the values in ZK_2 depend. For more detail on the transformation algorithm we refer the interested reader to the long version of this paper [2].

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 [3] 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). We also add union and intersection types to [3] (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[5]. As in [13,6] 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. 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 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 $\mathsf{Pair}(x_u : \mathsf{Un}, \{x_q : \mathsf{Un} \mid \mathsf{Request}(x_u, x_q) \wedge \mathsf{Registered}(x_u)\})$, 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).

[5] For non-dependent pair types we omit the unused variable and write $\mathsf{Pair}(T, U)$.

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.

Our type system has two special types $\top$ (top) and $\bot$ (bottom)[6]. 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[7] 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 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, 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)[8].

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.

[6] Type $\bot$ is encoded as $\{x : \mathsf{Un} \mid \mathsf{false}\}$.
[7] We call types T and U equivalent by subtyping if both $T <: U$ and $U <: T$.
[8] We assume a classical authorization logic that fulfills the law of excluded middle.

On the one hand, since type Un is always public and tainted and since true is valid in any environment, we have $\mathsf{pub}(\mathsf{Un}) = \mathsf{tnt}(\mathsf{Un}) = \mathsf{true}$. 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[9], 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 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 $\mathsf{pub}(\mathsf{SigKey}(T)) = \mathsf{tnt}(T)$.

4.4 Union and Intersection Types

We extend the type system from [3] with union and intersection types [17]. 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.

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,

[9] 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.

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. [13], 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 principals are uncompromised.

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 $\neg\mathsf{Compromised}(p)$ 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 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 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 if the proxy is compromised.

4.6 Type-checking Zero-knowledge

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 2.4 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:

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

where $C_1 = \mathsf{enc}((x_1,x_2),y_2) = y_4 \wedge \mathsf{Red}(\mathsf{check}(y_3,y_1),y_4 \wedge \mathsf{Request}(u,x_2))$. 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 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 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.

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, 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 $\mathsf{enc}((x_1,x_2),y_2) = y_4 \wedge \mathsf{check}(y_3,y_1) \rightsquigarrow y_4$ 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_3 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 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_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 $T_1^* \vee \{\!|\, \mathsf{tnt}(T_1^*) \,|\!\}$. 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 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 in the strengthened protocol the authentication decision of the store is indeed justified by the authorization policy, even if the proxy is compromised.

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 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.

We plan to work on the efficiency of the transformation. For instance, we can apply techniques similar to the ones proposed in [10,7] to reduce the number of zero-knowledge proofs forwarded by each principal. Our approach is specifically suited to reason about optimizations since we use a type-checker to check whether or not the transformed process is robustly safe despite compromised participants.

Acknowledgments. We thank Cédric Fournet and Andrew D. Gordon for the useful discussions. Cătălin Hriţcu is supported by a fellowship from Microsoft Research and the International Max Planck Research School for Computer Science.

References

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

2. 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.
3. 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. Long version and implementation available at: http://www.infsec.cs.uni-sb.de/projects/zk-typechecker/.
4. 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.
5. 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.
6. 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.
7. 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/.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. In *Advances in Cryptology: CRYPTO '03*, pages 110–125. Springer-Verlag, 2003.
16. G. C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.
17. B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.
18. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer-Verlag, 1998.
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.
20. J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.