Type-checking Zero-knowledge

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

Abstract

This paper presents the first type system for statically analyzing security protocols that are based on zero-knowledge proofs. We show how several properties offered by zero-knowledge proofs can be characterized in terms of authorization policies and statically enforced by a type system. The analysis is modular and compositional, and provides security proofs for an unbounded number of protocol executions. We develop a new type-checker that conducts the analysis in a fully automated manner. We exemplify the applicability of our technique and of our type-checker to real-world protocols by verifying the authenticity properties of the Direct Anonymous Attestation (DAA) protocol. The analysis of DAA takes less than three seconds.

1 Introduction

The design of cryptographic protocols is notoriously difficult and error-prone, and manual security proofs for such protocols are difficult to do. The need for automated techniques for protocol verification as well as expressive instruments to formulate the intended security properties are documented by the multitude of attacks on existing cryptographic protocols reported lately (e.g., [34, 12, 22, 15]). Logic-based authorization policies constitute a well-established and expressive framework for describing a wide range of security properties of cryptographic protocols, varying from authenticity properties to access control policies. Type systems are particularly salient tools to statically and automatically enforce authorization policies on abstract protocol specifications [23, 24] and on concrete protocol implementations [10]. Static verification facilitates protocol design and reduces the overhead of run-time checks. Furthermore, type systems require little human effort, and provide security proofs for an unbounded number of protocol executions. The analysis is modular, compositional and in general guaranteed to terminate.

One of the central challenges in the verification of authorization policies for modern applications is the expressiveness of the analysis and its ability to statically characterize the security properties guaranteed by complex cryptographic operations. For instance, current analysis techniques support traditional cryptographic primitives such as encryption and digital signatures, but they cannot cope with the most prominent and innovative modern cryptographic primitive: zero-knowledge proofs [25].

A zero-knowledge proof combines two seemingly contradictory properties. First, it constitutes 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. Early general-purpose zero-knowledge proofs were primarily designed for showing the existence of such proofs for the class of statements under consideration. These proofs were very inefficient and consequently of only limited use in practical applications. The recent advent of efficient zero-knowledge proofs for special classes of statements is rapidly changing this scenario. The unique security features that zero-knowledge proofs offer combined with the possibility to efficiently implement some of these proofs non-interactively have paved the way to modern cryptographic applications. In fact, several modern protocols such as anonymity protocols (e.g., Direct Anonymous Attestation (DAA) [13]), electronic voting protocols (e.g., [31, 4, 17]), and trust protocols (e.g., Pseudo Trust [33]), heavily rely on zero-knowledge proofs.

Statically analyzing zero-knowledge proofs is conceptually and technically challenging. Zero-knowledge proofs provide security guarantees that go far beyond the traditional and well-understood

secrecy and authenticity properties. For instance, zero-knowledge proofs may guarantee authentication yet preserve the anonymity of protocol participants, as in the Pseudo Trust protocol [33], or they may prove the reception of a certificate from a trusted server without revealing the actual content, as in the Direct Anonymous Attestation (DAA) protocol [13]. Current techniques for type-checking cryptographic protocols do not apply to zero-knowledge proofs, since such techniques usually rely on the type of keys for typing messages; zero-knowledge proofs, however, do not depend on any key infrastructure.

1.1 Our Contributions

This paper presents the first type system for statically analyzing the security of protocols based on non-interactive zero-knowledge proofs. We show how several properties guaranteed by zero-knowledge proofs can be characterized in terms of authorization policies and statically enforced by a type system. The analysis is modular and compositional, and provides security proofs for an unbounded number of protocol executions. Our approach is grounded on a state-of-the-art type system for authorization policies proposed by Fournet et al. [24]. This technique has been recently applied to the analysis of protocol implementations written in F# [10].

We develop a new type-checker that automates the analysis. The tool verifies that protocol specifications are well-typed and relies on the first-order logic automated theorem prover SPASS [35] to discharge proof obligations.

We exemplify the applicability of our technique to real-world protocols by verifying the Direct Anonymous Attestation protocol (DAA) [13]. We formalize the authenticity properties of this protocol in terms of authorization policies and we apply our type system to statically verify them. Our type-checker analyzed this sophisticated protocol in less than three seconds. This promising result indicates that our static analysis technique has the potential to scale up to industrial-size protocols.

1.2 Related Work

Dating back to the seminal work by Abadi on secrecy by typing [1], type systems were successfully used to analyze a wide range of security properties of cryptographic protocols, ranging from authenticity properties [27, 29, 30, 14, 5], to security despite compromised participants [28, 14, 24, 18], to authorization policies [23, 24, 10]. As mentioned before, type systems are efficient, usually enjoy guaranteed termination, and provide modularity and compositionality. None of the existing type systems, however, is capable of dealing with zero-knowledge proofs.

To the best of our knowledge, ProVerif [11, 3] is the only automatic tool that has been applied to the analysis of protocols based on zero-knowledge proofs [9, 6, 20]. This tool is based on Horn-clause resolution, and can analyze trace-based security properties as well as behavioral properties. The analysis with ProVerif is not compositional, and often has unpredictable termination behaviour, with seemingly harmless code changes leading to divergence. Also, as argued in [10], type systems scale better to large protocols and more efficiently analyze protocol implementations. In terms of expressiveness, ProVerif can deal with behavioral properties that are generally out of scope for current type systems (e.g., privacy and coercion-resistance in electronic-voting protocols [19, 6]), but is restricted to cryptographic primitives that can be expressed as a confluent rewriting system. Our analysis does not pose any constraint on the semantics of cryptographic primitives and, as opposed to ProVerif, can deal with authorization policies using arbitrary logical structure (e.g., arbitrarily nested quantifiers).

1.3 Outline

Section 2 illustrates our approach on a simple protocol for anonymous trust. Section 3 describes the process calculus we use to model security protocols. Section 4 presents our type system for zero-knowledge. Section 5 discusses the implementation of our type-checker in Objective Caml and the experimental evaluation of our technique. In Section 6 we apply our type system to analyze the Direct Anonymous Attestation (DAA) protocol. In Section 7 we explain how to use zero-knowledge proofs in the design of systems that guarantee security despite partial compromise. Section 8 concludes and provides directions

for future work. Due to space constraints, we defer all proofs and some of the technical details of our type system to an extended version of the paper [7].

2 Illustrative Example

This section introduces the types of zero-knowledge proofs and, in general, highlights the fundamental ideas of our type system, which will be elaborated in more detail in the following sections. As a running example, we consider a simple protocol for anonymous trust that is inspired by the Pseudo Trust protocol proposed by Lu et al. in [33]. The goal of this protocol is to allow parties to exchange data proving each other's trust level while preserving anonymity. These two seemingly conflicting requirements are met by an authentication scheme based on zero-knowledge proofs. We show how to characterize the notion of anonymous trust in terms of authorization policies and how to statically verify them by our type system.

2.1 A Protocol for Anonymous Trust

Each party has a public pseudonym, which is the hash of a secret. This pseudonym replaces the actual party's identity in the communication protocol. An arbitrary trust-management system like EigenTrust [32] or XenoTrust [21] can be used to certify the trust level of each pseudonym. Whenever a party (*prover*) wants to send a message msg to another party (*verifier*), she has to bind msg to her own pseudonym $\mathsf{hash}(s)$ and, in order to avoid impersonation, she has to prove the knowledge of s without revealing it. This authentication scheme is realized by a non-interactive zero-knowledge proof that is sent from the prover to the verifier. The zero-knowledge proof guarantees that the prover knows s and additionally provides the non-malleability of msg, i.e., changing msg requires the adversary to redo the proof and thus to know s. The goal of this protocol is allowing the verifier to associate the trust level of the prover to msg.

2.2 Zero-knowledge Proofs and Authorization Policies

Following [9], we represent our zero-knowledge proof by the following applied pi-calculus term: $\mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(\ s\ ;\ \mathsf{hash}(s),\ msg\ )$. A zero-knowledge proof $\mathsf{zk}_{n,m,S}(N_1,\ldots,N_n; M_1,\ldots,M_m)$ has $n+m$ arguments. The first n arguments $N_1,\ldots,N_n$ form the *private component* of the proof and are kept secret (s in the example), while the other m arguments $M_1,\ldots,M_m$ form the *public component* and are revealed to the verifier ($\mathsf{hash}(s)$ and msg in the example). The *statement* S of the zero-knowledge proof is expressed as a Boolean formula over the placeholders α_i and β_j (with $i \in [1,n]$ and $j \in [1,m]$), which stand for the argument N_i in the private component and the argument M_j in the public component, respectively. The verification of a zero-knowledge proof succeeds if and only if the statement obtained by replacing the place-holders by the corresponding private and public arguments holds true.

In order to express the security property guaranteed by this protocol as an authorization policy, we decorate the security-related protocol events as follows:

$$\mathsf{assume}\ \mathsf{Authenticate}(msg,s) \qquad\qquad \mathsf{assume}\ \mathsf{Trust}(pseudo,k)$$

$$P \xrightarrow{\qquad\qquad \mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(s;\mathsf{hash}(s),msg) \qquad\qquad} V$$

$$\mathsf{assert}\ \mathsf{Associate}(msg,k)$$

Before generating the zero-knowledge proof to authenticate msg using the knowledge of the secret s, the prover P assumes $\mathsf{Authenticate}(msg,s)$. Before receiving the zero-knowledge proof, the verifier V knows that the trust level associated to the pseudonym *pseudo* is k and assumes $\mathsf{Trust}(pseudo,k)$. As discussed before, this information can be obtained by means of an external trust-management system. The verifier receives and verifies the zero-knowledge proof, checking that the first argument in the public component is *pseudo*. This guarantees that the prover knows the secret s such that $pseudo = \mathsf{hash}(s)$ and allows the verifier to assert $\mathsf{Associate}(msg,k)$. The authorization policy can be expressed as follows:

$$Policy := \forall msg,k,s.(\mathsf{Authenticate}(msg,s)\ \wedge\ \mathsf{Trust}(\mathsf{hash}(s),k))\ \Rightarrow \mathsf{Associate}(msg,k)$$

This policy allows the verifier to give msg trust level k (assertion $\mathsf{Associate}(msg, k)$) only if the prover wants to authenticate msg (assumption $\mathsf{Authenticate}(msg, s)$) and knows some s for which the trust level of the pseudonym $\mathsf{hash}(s)$ is k (assumption $\mathsf{Trust}(\mathsf{hash}(s), k)$).

2.3 The Type of Zero-knowledge Proofs

To illustrate our technique, let us consider the type associated to the zero-knowledge proof $\mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(\ s\ ;\ \mathsf{hash}(s),\ msg\ )$:

$$\mathsf{ZKProof}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(\langle y_1 : \mathsf{Hash}(\mathsf{Private}), y_2 : \mathsf{Un}\rangle\{\exists x.\ y_1 = \mathsf{hash}(x)\ \wedge\ \mathsf{Authenticate}(x, y_2))\}$$

This dependent type indicates that the public component is composed of two messages. The first message y_1 is of type $\mathsf{Hash}(\mathsf{Private})$, i.e., it is the public hash of a secret message. The type Private describes messages that are not known to the adversary. The second message y_2 is of type Un (untrusted), i.e., it may come from and be sent to the adversary. The logical formula $\exists x.\ y_1 = \mathsf{hash}(x)\ \wedge\ \mathsf{Authenticate}(x, y_2)$ says that y_1 is the hash of some secret x such that $\mathsf{Authenticate}(x, y_2)$ has been assumed by the prover. This formula contains an equality constraint on the structure of messages as well as a logical predicate.

After the verification of the zero-knowledge proof, the verifier can safely assume that the formula holds true. The constraint $\exists x.\ y_1 = \mathsf{hash}(x)$ is guaranteed by the semantics of the zero-knowledge proof, while the assumption $\mathsf{Authenticate}(x, y_2)$ is enforced by our type system. The verifier can thus logically derive $\exists s.\ \mathsf{Authenticate}(msg, s) \wedge \mathsf{Trust}(\mathit{pseudo}, k) \wedge \mathit{pseudo} = \mathsf{hash}(s)$. By a standard logical property which allows replacing equals by equals, the authorization policy allows the verifier to derive $\mathsf{Associate}(msg, k)$.

3 Calculus

We consider a variant of the applied pi-calculus with constructors and destructors, similar to the one in [2], and we extend it with zero-knowledge proofs. Following [24, 10], the calculus also includes special operators to assume and assert logical formulas. This section overviews the syntax and semantics of the calculus.

3.1 Constructors and Terms

Constructors are function symbols that are used to build terms, and are ranged over by f. The set of constructors includes pair for constructing pairs; 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 constant true represents the respective Boolean value, and has its canonical meaning in the authorization logic. The nullary constructor ok is used by the type system to convey logical formulas.

The set of *terms* is the free algebra built from names, variables, and constructors applied to other terms. We let u range over names and variables.

3.2 Destructors

Destructors are partial functions that processes can apply to terms, and are ranged over by g. The semantics of destructors is ruled by the evaluation relation $\Downarrow$: given the terms $M_1, \ldots, M_n$ as arguments, the destructor g can either succeed and provide a term N as a result ($g(M_1, \ldots, M_n) \Downarrow N$) or it can fail ($g(M_1, \ldots, M_n) \not\Downarrow$). The fst destructor extracts the first component of a pair ($\mathsf{fst}(\mathsf{pair}(M, N)) \Downarrow M$), while snd extracts the second one ($\mathsf{snd}(\mathsf{pair}(M, N)) \Downarrow N$). The application of eq succeeds, yielding the constant true, if the two arguments are syntactically the same ($\mathsf{eq}(M, M) \Downarrow \mathsf{true}$). The destructors $\wedge$ and $\vee$ model conjunctions and disjunctions, respectively ($\wedge(\mathsf{true}, \mathsf{true}) \Downarrow \mathsf{true}$, $\vee(M, \mathsf{true}) \Downarrow \mathsf{true}$, and $\vee(\mathsf{true}, M) \Downarrow \mathsf{true}$). The destructor dec decrypts an encrypted message given the corresponding decryption key ($\mathsf{dec}(\mathsf{enc}(M, \mathsf{pk}(K)), K) \Downarrow M$). The check destructor checks a signed message using a verification key, and if this succeeds returns the message without the signature ($\mathsf{check}(\mathsf{sign}(M, K), \mathsf{vk}(K)) \Downarrow M$). The

exercise destructor simply returns its argument ($\mathsf{exercise}(M) \Downarrow M$), and is applied to ok tokens to justify logical assumptions in the typing environment (for more details we refer to Section 4.2).

3.3 Representing Zero-knowledge Proofs

Constructing for Zero-knowledge Proofs. In a very similar way to what is proposed in [9], 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, and S denotes a logical statement over these terms. The proof keeps the terms in $N_1, \ldots, N_n$ secret, while the terms $M_1, \ldots, M_m$ are revealed. For clarity we will use semicolons to separate the secret terms from the public ones, and we will often write $\widetilde{N}$ instead of $N_1, \ldots, N_n$ if n is clear from the context.

Statements. In order to express a wide class of zero-knowledge proofs, comprising for instance proofs of signature verifications and decryptions, we need to use destructors inside logical formulas. However, destructors cannot occur inside terms so we need to define a larger class of objects, called *statements*, that also contains destructors. The set of statements (ranged over by S) is the free algebra built from names, variables, the placeholders α_i and β_j, as well as constructors and destructors applied to other statements. It is easy to see that all terms are also statements. For clarity we distinguish an actual destructor g from its counterpart used within statements, we denote the latter by $g^\sharp$. The statement S used by 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 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 which are matched against the first l arguments in the public component of the proof. If the proof is valid $\mathsf{ver}_{n,m,l,S}$ 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 evaluates to true. The ver destructor is formalized as follows:[1]

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

The evaluation relation $S \Downarrow_\sharp M$ is defined in terms of the reduction rules for the other destructors:[2][3]

$$\textsc{Ev-Term}\quad M \Downarrow_\sharp M \qquad \textsc{Ev-Constr}\quad \frac{\forall i \in [1, n].\ S_i \Downarrow_\sharp M_i}{f(S_1, \ldots, S_n) \Downarrow_\sharp f(M_1, \ldots, M_n)} \qquad \textsc{Ev-Destr}\quad \frac{\forall i \in [1, n].\ S_i \Downarrow_\sharp M_i \qquad g(M_1, \ldots, M_n) \Downarrow M}{g^\sharp(S_1, \ldots, S_n) \Downarrow_\sharp M}$$

For instance, in the protocol from Section 2 we have that

$$\mathsf{ver}_{1,2,1,\beta_1=\mathsf{hash}(\alpha_1)}(\mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(s; \mathsf{hash}(s), msg), pseudo) \Downarrow \langle msg \rangle$$

since $(\beta_1 = \mathsf{hash}(\alpha_1))\{s/\alpha_1\}\{\mathsf{hash}(s)/\beta_1\} \equiv \mathsf{hash}(s) = \mathsf{hash}(s) \Downarrow_\sharp \mathsf{true}$. The destructor public_m yields the public component of a zero-knowledge proof ($\mathsf{public}_m(\mathsf{zk}_{n,m,S}(\widetilde{N}, \widetilde{M})) \Downarrow (\widetilde{M})$). Note that the private component is not revealed by any destructor.

[1] The notation $\langle M_{l+1}, \ldots, M_m \rangle$ represents a tuple having dependent type and is further discussed Section 4.1.

[2] This definition is not circular since the ver destructor cannot appear inside statements.

[3] The $\Downarrow_\sharp$ relation can in fact be seen as the extension of the $\Downarrow$ relation to statements.

3.4 Processes

Processes are essentially the same as in [24]. The process $\mathsf{out}(M, N).P$ outputs message N on channel M and then behaves as P; the process $\mathsf{in}(M, x).P$ receives a message N from channel M and then behaves as $P\{N/x\}$; the process $!\mathsf{in}(M, x).P$ behaves as an unbounded number of copies of $\mathsf{in}(M, x).P$ executing in parallel; $\mathsf{new}\ a : T.P$ generates a fresh name a of type T and then behaves as P (the type annotation does not have any computational significance); $P \mid Q$ behaves as P executed in parallel with Q; $\mathbf{0}$ is a process that does nothing; $\mathsf{let}\ x = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$ applies the destructor g to the terms $\widetilde{M}$ and if the destructor application succeeds behaves as P otherwise behaves as Q.

The processes $\mathsf{assume}\ C$ and $\mathsf{assert}\ C$, where C is a logical formula, are used to express authorization policies inside the processes, and do not have any computational significance. Assumptions are used to mark security-related events in processes, such as the intention to authenticate message msg by the party knowing the secret value s ($\mathsf{assume}\ \mathsf{Authenticate}(msg, s)$), but also authorization policies such as:

$$\mathsf{assume}\ \forall msg, k, s.(\mathsf{Authenticate}(msg, s)\ \wedge\ \mathsf{Trust}(\mathsf{hash}(s), k))\ \Rightarrow \mathsf{Associate}(msg, k)$$

The scope of assumptions is global, i.e., once an assumption becomes active it affects all processes that run in parallel.

Intuitively, assertions dynamically test that certain conditions are fulfilled with respect to the assumptions currently active, for example in the protocol from Section 2 the verifier asserts that it can associate trust level k to the message msg ($\mathsf{assert}\ \mathsf{Associate}(msg, k)$). In principle it might be possible to implement such assertions as (distributed!) dynamic checks. As in [24], we take a totally different approach here. Our type system guarantees statically that in well-typed protocols all asserted formulas are valid at runtime, even in the presence of an arbitrary adversary.

3.5 Authorization logic

Our calculus and type system are largely independent of the exact choice of authorization logic. The logic is required to fulfill some standard properties, such as monotonicity, closure under substitution and allowing the replacement of equals by equals. Additionally, statements are not only used by zero-knowledge terms, but they also have a close connection with the formulas in our authorization logic. For this reason, we require that all statements are also formulas in the logic, and we assume that $\mathsf{eq}^\sharp$ corresponds to equality in the authorization logic (simply denoted by "="), while $\wedge^\sharp$ and $\vee^\sharp$ correspond to conjunction and disjunction in the logic. Furthermore, we add axioms in the logic that correspond to the semantics of the other destructors, i.e., for every rule $g(M_1, \ldots, M_n) \Downarrow N$ we add an axiom $g^\sharp(M_1, \ldots, M_n) = N$ in the logic. This ensures that if a statement S evaluates to a term M ($S \Downarrow_\sharp M$) then $S = M$ holds in the logic ($\models S = M$). Under this assumption, from the semantics of the ver destructor we can immediately infer that if $\mathsf{ver}_{n,m,l,S}(\mathsf{zk}_{n,m,S}(\widetilde{N}, M_1, \ldots, M_l, \ldots, M_m), M_1, \ldots, M_l) \Downarrow \langle M_{l+1}, \ldots, M_m\rangle$ then $\models S\{\widetilde{N}/\widetilde{\alpha}\}\{\widetilde{M}/\widetilde{\beta}\}$, which captures the soundness of our construction for zero-knowledge.

In our implementation we consider first-order logic with equality as the authorization logic and we use the automated theorem prover SPASS [35] to discharge proof obligations.

3.6 Notations and Conventions

Throughout the paper, we identify any phrase ϕ of syntax up to consistent renaming of bound names and variables. We let $fn(\phi)$ denote the set of free names in ϕ, $fv(\phi)$ the set of free variables, and $free(\phi)$ the set of free names and variables. We write $\phi\{\phi'/x\}$ for the outcome of the capture-avoiding substitution of ϕ' for each free occurrence of x in ϕ. We say that ϕ is closed if it does not have any free variables.

A context is a process with a hole where other processes can be plugged in. An evaluation context $\mathcal{E}$ is a context of the form $\mathcal{E} = \mathsf{new}\ \widetilde{a} : \widetilde{T}.([\,]|P)$ for some process P. We use $\mathsf{new}\ \widetilde{a} : \widetilde{T}$ to denote a sequence $\mathsf{new}\ a_1 : T_1 \ldots \mathsf{new}\ a_k : T_k$ of typed name restrictions and, for the sake of readability, we use $let\ x := M\ in\ P$ to denote $P\{M/x\}$.

3.7 Modeling the Protocol for Anonymous Trust

With this setup in place we can formally model the protocol for anonymous trust from Section 2 as follows:

$P := \mathsf{new}\ msg : \mathsf{Un}.(\mathsf{assume}\ \mathsf{Authenticate}(msg, s) \mid \mathsf{out}(c, \mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(s; \mathsf{hash}(s), msg)))$

$V := \mathsf{assume}\ \mathsf{TrustLevel}(pseudo, k) \mid \mathsf{in}(c, w).\ \mathsf{let}\ x = \mathsf{ver}_{1,2,1,\beta_1=\mathsf{hash}(\alpha_1)}(w, pseudo)\ \mathsf{then}$
$\mathsf{let}\ y = \mathsf{fst}(x)\ \mathsf{then}\ \mathsf{let}\ z = \mathsf{snd}(x)\ \mathsf{then}\ \mathsf{let}\ v = \mathsf{exercise}(z)\ \mathsf{then}\ \mathsf{assert}\ \mathsf{Associate}(y, k)$

PseudoTrust $:= \mathsf{new}\ s : \mathsf{Private}.\ P \mid let\ pseudo := \mathsf{hash}(s)\ in\ V \mid \mathsf{assume}\ Policy$

3.8 Operational Semantics and Safety

As for the pi-calculus, the operational semantics of our calculus is defined in terms of *structural equivalence* ($\equiv$) and *internal reduction* ($\rightarrow$). Structural equivalence captures rearrangements of parallel compositions and restrictions. Internal reduction defines the semantics of communication and destructor application.

A process is safe if and only if all its assertions are satisfied in every protocol execution.

Definition 3.1 ***(Safety)*** *A closed process P is* safe *if and only if for every C and Q such that $P \rightarrow^* \mathsf{new}\ \widetilde{a} : \widetilde{T}.(\mathsf{assert}\ C \mid Q)$, there exists an evaluation context $\mathcal{E} = \mathsf{new}\ \widetilde{b} : \widetilde{U}.[\ ] \mid Q'$ such that $Q \equiv \mathcal{E}[\mathsf{assume}\ C_1 \mid \ldots \mid \mathsf{assume}\ C_n]$, $fn(C) \cap \widetilde{b} = \emptyset$, and we have that $\{C_1, \ldots, C_n\} \models C$.*

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

Definition 3.2 ***(Opponent)*** *A process is an* opponent *if it does not contain any* assert *and if the only type occurring therein is* Un.

Definition 3.3 ***(Robust Safety)*** *A closed process P is* robustly safe *if and only if $P \mid O$ is safe for every opponent O.*

4 Type System

The type system presented in this paper extends a recent type system for statically enforcing authorization policies in distributed systems [24].

4.1 Basic Types

Our type system has the following types: Private is the type of messages that are not revealed to the adversary, while Un is the type of messages possibly known to the adversary; $\mathsf{Ch}(T)$ is the type of channels carrying messages of type T; $\mathsf{Ok}(C)$ is the type of an ok term conveying the logical formula C; $\mathsf{Pair}(x : T, U)$ is the dependent type of a term $\mathsf{pair}(N, M)$, where N has type T and M has type $U\{N/x\}$.

Dependent pair types and ok-types are generally used together to form dependent tuple types[4] of the form $\mathsf{Pair}(x_1 : T_1, \ldots, \mathsf{Pair}(x_n : T_n, \mathsf{Ok}(C))\ldots)$, where T_1,...,T_n are not dependent on x_1,...,x_n, but the formula C can depend on all these variables. We will use $\langle x_1 : T_1, \ldots, x_n : T_n\rangle\{C\}$ to denote types of this form, and $\langle M_1, \ldots, M_n\rangle$ to denote a value of such a type, i.e., to denote $\mathsf{pair}(M_1, \ldots, \mathsf{pair}(M_n, \mathsf{ok})\ldots)$. For example, in the protocol of Section 2, the verification of the zero-knowledge proof yields the term $\langle msg\rangle$ of type

$$\langle y : \mathsf{Un}\rangle\{\exists x.\ pseudo = \mathsf{hash}(x)\ \wedge\ \mathsf{Authenticate}(y, x)\}.$$

In addition to these base types, we also consider types for the different cryptographic primitives. For digital signatures, $\mathsf{SigKey}(T)$ and $\mathsf{VerKey}(T)$ denote the types of the signing and verification keys for messages of type T, while $\mathsf{Signed}(T)$ is the type of signed messages of type T. 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 the public encryption keys and of the 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. The type $\mathsf{Hash}(T)$ denotes the type of a hashed message of type T.

[4] Such types are called refinement types in [10].

4.2 Typing Judgments

The type system relies on four typing judgments: well-formed environment ($\Gamma \vdash \diamond$), subtyping ($\Gamma \vdash T <: U$), term typing ($\Gamma \vdash M : T$), and process typing ($\Gamma \vdash P$), which are described in the following.

Well-formed Environment. The type system relies on a *typing environment*, which is a list containing name and variable bindings of the form $u : T$, together with formulas of the authorization logic. Intuitively, these formulas constitute a static lower bound for the formulas assumed at run-time (with respect to the logical entailment relation). A typing environment is *well-formed*, written $\Gamma \vdash \diamond$, if no name or variable is bound more than once, and if all free names and variables inside the types and formulas appearing in the environment are bound beforehand. All the other typing judgments check that the environment they use is well-formed.

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 to be useful. 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 subtyping relation 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 is *public* in this case. In addition, we allow messages of type Un that were 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 is *tainted*.

For example, in our type system the types $\mathsf{PubKey}(T)$ and $\mathsf{VerKey}(T)$ are 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, the $\mathsf{Private}$ type we used in Section 2 is neither public nor tainted.

Typing Terms. With this setup in place we can define the term typing judgment $\Gamma \vdash M : T$, which checks that message M has type T. The type of variables and names is simply looked up in the typing environment. A term $\mathsf{pair}(M_1, M_2)$ has type $\mathsf{Pair}(x : T_1, T_2)$ whenever M_1 has type T_1 and M_2 has type $T_2\{M_1/x\}$. An ok term has type $\mathsf{Ok}(C)$ in environment Γ if the formula C follows logically from the formulas in Γ, which we denote as $\mathit{forms}(\Gamma) \models C$. The other cases are as one would expect. For instance if the message M has type T and the key K has type $\mathsf{SigKey}(T)$ then we can derive that the signature $\mathsf{sign}(M, K)$ has type $\mathsf{Signed}(T)$.

Typing Processes. As we will show in Section 4.6, the typing judgment $\Gamma \vdash P$ guarantees that P is secure against an arbitrary adversary. The output process $\mathsf{out}(M, N).P$ is well-typed, if the term M has a channel type $\mathsf{Ch}(T)$, N is of type T and the process P is well-typed. For instance, this guarantees that the adversary can only receive messages of type Un at run-time, since it is initially given only channels of type $\mathsf{Ch}(\mathsf{Un})$. Similarly, the input process $\mathsf{in}(M, x).P$ is well-typed only if M has type $\mathsf{Ch}(T)$ and P is well-typed assuming x of type T. The process $\mathsf{new}\ n : T.P$ is well-typed if P is well-typed assuming n of type T. When type-checking a parallel composition $P \mid Q$ the top-level assumptions in P are added to the typing environment in which Q is typed, and the top-level assumptions in Q are added to the environment in which P is typed. This ensures that assumptions have global scope.

We have $\Gamma \vdash \mathsf{let}\ x = \mathsf{exercise}(M)\ \mathsf{then}\ P$ only if M has type $\mathsf{Ok}(C)$ and $\Gamma, C \vdash P$. Adding C to the typing environment of the continuation process is sound since the type system guarantees that the formula C is assumed at run-time for any ok term of type $\mathsf{Ok}(C)$. The process $\mathsf{assert}\ C$ is well-typed in a typing environment Γ, only if $\mathit{forms}(\Gamma) \models C$. This guarantees the safety of the process since, as mentioned before, the formulas in Γ represent a static lower bound of the formulas assumed at run-time.

Type-checking process $\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{then}\ P\ \mathsf{else}\ Q$ is more interesting and differs significantly from [24]. As usual, we need to check whether the arguments $M_1, \ldots, M_n$ have the types required by the destructor, and obtain a new type T for the result of the destructor application. The continuation process P is, however, type-checked in a typing environment extended not only with the binding $x : T$, but also with the logical formula "$x = g^\sharp(M_1, \ldots, M_n)$". This can be used for further reasoning in the logic,

and proves to be crucial for our treatment of the pair splitting destructors (see Section 4.3). For instance, when checking that $\Gamma \vdash \mathsf{let}\ x = \mathsf{check}(M, K)\ \mathsf{then}\ P$ we first need to ensure that M has type $\mathsf{Signed}(T)$ and K has type $\mathsf{VerKey}(T)$ for the same T. Then we can type-check the process P in the environment $\Gamma, x : T, x = \mathsf{check}^\sharp(M, K)$. For whatever it is worth, this treatment of destructors is simpler and more elegant than the one in [24]. The rules for typing processes are reported in Table 1 in the appendix.

4.3 Type-checking the Example

As an example consider the final part of the verifier process in Section 3.7. The variable x holds a pair containing the received message and an ok term. The process splits the pair, exercises the second component z, and asserts that the first component y satisfies the predicate $\mathsf{Associate}(y, k)$.

$$\ldots\ \mathsf{let}\ y = \mathsf{fst}(x)\ \mathsf{then}\ \mathsf{let}\ z = \mathsf{snd}(x)\ \mathsf{then}\ \mathsf{let}\ v = \mathsf{exercise}(z)\ \mathsf{then}\ \mathsf{assert}\ \mathsf{Associate}(y, k)$$

This process is type-checked in the environment:

$$\begin{array}{l}\Gamma_1 = \ldots (\forall msg, k, s.(\mathsf{Authenticate}(msg, s) \wedge \mathsf{TrustLevel}(\mathsf{hash}(s), k)) \Rightarrow \mathsf{Associate}(msg, k)),\\ \quad \mathsf{TrustLevel}(pseudo, k), x : \mathsf{Pair}(w : \mathsf{Un}, \mathsf{Ok}(\exists v.\ pseudo = \mathsf{hash}(v) \wedge \mathsf{Authenticate}(w, v))).\end{array}$$

After the pair is split the environment becomes:

$$\Gamma_2 = \Gamma_1, y : \mathsf{Un}, y = \mathsf{fst}^\sharp(x), z : \mathsf{Ok}(\exists v.\ pseudo = \mathsf{hash}(v) \wedge\ \mathsf{Authenticate}(\mathsf{fst}^\sharp(x), v)), z = \mathsf{snd}^\sharp(x).$$

First, note that we added the formulas "$y = \mathsf{fst}^\sharp(x)$" and "$z = \mathsf{snd}^\sharp(x)$" corresponding to the destructors we encountered. Second, when we split the second component of the pair, the type assigned to z is the initial ok-type where we replaced the variable w by $\mathsf{fst}^\sharp(x)$, as it is usual for dependent pair types. Finally, when reaching the assert the environment is

$$\Gamma_3 = \Gamma_2, \exists v.\ pseudo = \mathsf{hash}(v)\ \wedge\ \mathsf{Authenticate}(\mathsf{fst}^\sharp(x), v),$$

where the new formula was added when checking the exercise destructor. In order to type-check the assert we need to ensure that $forms(\Gamma_3) \models \mathsf{Associate}(y, k)$, where $forms(\Gamma_3)$ is

$$\begin{array}{l}\{(\forall msg, k, s.(\mathsf{Authenticate}(msg, s)\ \wedge\ \mathsf{TrustLevel}(\mathsf{hash}(s), k))\ \Rightarrow \mathsf{Associate}(msg, k)),\\ \mathsf{TrustLevel}(pseudo, k), y = \mathsf{fst}^\sharp(x), z = \mathsf{snd}^\sharp(x), \exists v.\ pseudo = \mathsf{hash}(v)\ \wedge\ \mathsf{Authenticate}(\mathsf{fst}^\sharp(x), v)\}.\end{array}$$

This holds in the authorization logic, but would not hold if "$y = \mathsf{fst}^\sharp(x)$" was not added when checking $\mathsf{let}\ y = \mathsf{fst}(x)$.

4.4 Type-checking Zero-knowledge

The Zero-knowledge Type. We give zero-knowledge proofs of the form $\mathsf{zk}_{n,m,S}(\widetilde{N}; \widetilde{M})$ type $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\{\exists x_1, \ldots, x_n.C\})$. This type contains a dependent tuple type listing the types of the arguments in the public component, together with an Ok type. The logical formula associated to the Ok type is 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 $C\{\widetilde{N}/\widetilde{x}\}$ is entailed by the formulas in the typing environment. Note that the existentially quantified variables are replaced by the corresponding arguments in the private component.

Subtyping. The zero-knowledge type $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\{C\})$ is public only if the types $T_1, \ldots, T_m$ of all the public arguments of the proof are also public. This is necessary since the adversary can extract the public component of $\mathsf{zk}_{n,m,S}(\widetilde{N}; \widetilde{M})$ using the public_m destructor. For the same reason, the zero-knowledge type $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\{C\})$ is tainted only if the types of all public arguments of the proof are tainted. Finally, the type $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\{C\})$ is a subtype of $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T'_1, \ldots, y_m : T'_m\rangle\{C'\})$ in Γ, if T_i is a subtype of T'_i for all i, and if additionally $forms(\Gamma) \cup \{C\} \models C'$.

Type Annotations. Typing all the other cryptographic primitives we consider relies on the type of some key, which the user has to annotate explicitly. Zero-knowledge proofs, however, do not depend on any key in general. This poses a problem since type-checking the verification of zero-knowledge proofs should allow propagating logic formulas in the typing environment of the verifier, and it is not clear what formulas to consider. For instance, when type-checking a process $\mathsf{let}\ \langle \widetilde{y} \rangle = \mathsf{ver}_{n,m,0,S}(z)\ \mathsf{then}\ P$[5], we can safely assume that the formula $\exists \widetilde{x}.\ S\{\widetilde{x}/\widetilde{\alpha}\}\{\widetilde{y}/\widetilde{\beta}\}$ holds for the continuation process P. This is in fact guaranteed by the operational semantics of the ver destructor (see Section 3.2). Such a formula however does not suffice to type-check most examples we have tried, since it does not mention any logical predicates.

In order to solve this problem we require the user to provide type annotations for each statement used in the process. For each (n, m)-statement S this annotation is modeled as a free variable in the initial typing environment having the distinguished name $f^{code}_{n,m,S}$, and bound to a type of the form $\mathsf{Stm}(\langle y_1 : T_1, \ldots, y_m : T_m \rangle \{\exists x_1, \ldots, x_n.C\})$. Additionally, the environment contains an implicit binding $f^{env}_{n,m,S} : \mathsf{Un}$ used to type-check proofs of S generated by the adversary. We let $f_{n,m,S}$ range over $f^{code}_{n,m,S}$ and $f^{env}_{n,m,S}$.

Typing Zero-knowledge Proofs. With this setup in place, we can finally formalize the typing rule for zero-knowledge proofs:

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

Note that we require that $\Gamma \vdash \langle M_1, \ldots, M_m \rangle : \langle y_1 : T_1, \ldots, y_m : T_m \rangle \{C\{\widetilde{N}/\widetilde{x}\}\}$, which not only makes sure that the public arguments have the required types, but it also effectively checks that $C\{\widetilde{N}/\widetilde{x}\}$ logically follows from the formulas in the typing environment. In this way we ensure that honest participants can only generate proofs for statements that are associated to formulas already entailed by the environment.

Typing Zero-Knowledge Verification. Suppose that we are given the process $\mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(N, M_1, \ldots, M_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q$, a typing environment $\Gamma, f^{code}_{n,m,S} : T$, where the type $T = \mathsf{Stm}(\langle y_1 : T_1, \ldots, y_m : T_m \rangle \{\exists x_1, \ldots, x_n.C\})$ is annotated by the user, and a type T' such that $\Gamma \vdash N : T'$. Zero-knowledge proofs are typically received from channels controlled by the attacker and in this case $T' = \mathsf{ZKProof}_{n,m,S}(\langle y_1 : \mathsf{Un}, \ldots, y_m : \mathsf{Un} \rangle \{\mathsf{true}\})$, which is equivalent to Un by subtyping. In order to type-check this process, we first check that the terms $M_1, \ldots, M_l$ have type $T_1, \ldots, T_l$, since these arguments are matched against the first l terms in the public component of the verified proof.

The main idea for obtaining stronger guarantees than those given by the semantics of the ver destructor is to use the types of the matched public arguments to derive the type of the other arguments of the proof, even the private ones. For instance, if a matched public argument is a hash of type $\mathsf{Hash}(U)$ and the statement proves the knowledge of the value inside, then we can derive that this value has type U. Similarly, if a matched public argument is a key of type $\mathsf{VerKey}(U)$ and the statement proves the verification of a signature using this key, then the message that was signed has type U. This kind of reasoning can be exploited to infer both type information and logical formulas. Furthermore, if we can statically verify that at least one of the arguments of the proof is neither public nor tainted, then we know that the zero-knowledge proof has been generated by a honest participant (since the adversary knows only terms that are public or tainted). This immediately implies that the proof has the stronger type T, since zero-knowledge proofs constructed by honest participants are given the type specified by the user.

The verification of the type of the zero-knowledge proof is formalized by the predicate $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'}$ (see Table 2 in the appendix). If this predicate holds, then the zero-knowledge proof N is guaranteed to have the stronger type T and we can safely give the last $m - l$ arguments of the proof type $\langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m \rangle \{C\{\widetilde{M_l}/\widetilde{y_l}\}\}$. The typing rule for the $\mathsf{ver}_{n,m,l,S}$ destructor is formalized as follows:

$$\frac{\begin{array}{c}\Gamma \vdash f_{n,m,S} : \mathsf{Stm}(\langle y_1 : T_1, \ldots, y_m : T_m \rangle \{C\}) = T \\ \Gamma \vdash N : \mathsf{ZKProof}_{n,m,S}(\langle y_1 : U_1, \ldots, y_m : U_m \rangle \{\mathsf{true}\}) = T' \qquad \forall i \in [1,l].\ \Gamma \vdash M_i : T_i \\ \langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'}\ \mathsf{holds} \qquad \Gamma, x : \langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m \rangle \{C\{\widetilde{M_l}/\widetilde{y_l}\}\} \vdash P \qquad \Gamma \vdash Q\end{array}}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(N, M_1, \ldots, M_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

[5]Notice that we did not match any of the public arguments of the proof.

4.5 Type-checking the Example (Continued)

We explain how to type-check the prover and the verifier from Section 3.7, in the following initial typing environment: $\Gamma_0 = f_{1,2,\beta_1=\mathsf{hash}(\alpha_1)} : \mathsf{Stm}(T_{zk}), s : \mathsf{Private}, c : \mathsf{Un}, k : \mathsf{Un}$, where $T_{zk} = \langle y_1 : \mathsf{Hash}(\mathsf{Private}), y_2 : \mathsf{Un}\rangle\{\exists x.\ y_1 = \mathsf{hash}(x) \wedge \mathsf{Authenticate}(x, y_2)\}$.

For the prover, we need to check that the term $\mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(s; \mathsf{hash}(s), msg)$ has type $\mathsf{ZKProof}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(T_{zk})$, in the extended environment $\Gamma_1 = \Gamma_0, msg : \mathsf{Un}, \mathsf{Authenticate}(msg, s)$. For this we need to show that $\Gamma_1 \vdash \mathsf{hash}(s) : \mathsf{Hash}(\mathsf{Private})$, $\Gamma_1 \vdash msg : \mathsf{Un}$ and $forms(\Gamma_1) \models \exists x.\ \mathsf{hash}(s) = \mathsf{hash}(x) \wedge \mathsf{Authenticate}(x, msg)$, which holds by instantiating x to s.

The verifier receives the zero-knowledge proof from the untrusted channel c. The zero-knowledge proof is thus bound to a variable x of type Un and, by subtyping, also of type $\mathsf{ZKProof}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(\langle y_1 : \mathsf{Un}, y_2 : \mathsf{Un}\rangle\{\mathsf{true}\})$. Note that this type is not strong enough to type-check the rest of the verifier's code. Type-checking the destructor application $\mathsf{ver}_{1,2,1,\beta_1=\mathsf{hash}(\alpha_1)}(x, pseudo)$ can however rely on the fact that $\Gamma \vdash pseudo : \mathsf{Hash}(\mathsf{Private})$, and therefore $pseudo$ is the hash of some message s of type $\mathsf{Private}$, which is neither public nor tainted. The statement guarantees that the prover knows s and this is enough to ensure that the zero-knowledge proof is generated by a honest participant. Therefore $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'}$ holds, and the type system gives the result of the destructor application type $\langle y : \mathsf{Un}\rangle\{\exists v.\ pseudo = \mathsf{hash}(v) \wedge \mathsf{Authenticate}(y, v)\}$. This allows for type-checking the remainder of the verifier's code, as discussed in Section 4.3.

4.6 Security Guarantees

Our type system statically guarantees that in well-typed processes all asserted formulas are valid at runtime (safety), even in the presence of an arbitrary adversary (robust safety). We use $\Gamma \vdash_{\mathsf{Un}} P$ to denote $\Gamma, u_1 : \mathsf{Un}, \ldots, u_n : \mathsf{Un} \vdash P$, where $\{u_1, \ldots, u_n\} = \mathit{free}(P)$.

Theorem 4.1 (Robust Safety) *Let $\Gamma = \{f^{code}_{n_1,m_1,S_1} : \mathsf{Stm}(T_1), \ldots, f^{code}_{n_k,m_k,S_k} : \mathsf{Stm}(T_k)\}$, where the type annotation for each (n, m)-statement S is unique, i.e., $\forall i, j \in [1, k].(n_i, m_i, S_i) \neq (n_j, m_j, S_k)$. For every closed process P, if $\Gamma \vdash_{\mathsf{Un}} P$ then P is robustly safe.*

Due to space constraints, the proof of this theorem and of all the necessary lemmas are given in the extended version of the paper [7].

5 Implementation

We have implemented an automatic type-checker for the type system presented in this paper. The tool is written in Objective Caml and comprises approximately 3000 lines of code. The type-checking phase is guaranteed to terminate, and generates proof obligations that are discharged independently, leading to a modular and robust analysis. We use first-order logic with equality as the authorization logic and we employ the automated theorem prover SPASS [35] to discharge the proof obligations. Our implementation is available at [8].

The implementation uses an algorithmic version of our type system. Devising a variant of the type system that is suitable for an implementation required us to eliminate the subsumption rule and deal with the fact that the constructors and destructors are in fact polymorphic and that the instantiation of the type variables needs to be made automatically. The latter problem is not trivial since our type system features subtyping. On the other hand, asking the user to annotate every constructor and destructor application would have been unacceptable from a usability perspective.

We tested our tool on the DAA protocol and several simpler examples including the anonymous trust management protocol given in Section 2. The analysis of DAA terminated in less than three seconds and discharged 39 proof obligations, while for the simpler examples the time needed was less than half a second. These promising results show that our static analysis technique has the potential to scale up to industrial-size protocols.

6 Case Study: Direct Anonymous Attestation Protocol

To exemplify the applicability of our type system to real-world protocols, we modeled and analyzed the authenticity properties of the Direct Anonymous Attestation protocol (DAA) [13]. DAA constitutes a cryptographic protocol that enables the remote authentication of a hardware module called the Trusted Platform Module (TPM), while preserving the anonymity of the user owning the module. Such TPMs are now included in many end-user notebooks.

The goal of the DAA protocol is to enable the TPM to sign arbitrary messages and to send them to an entity called the verifier in such a way that the verifier will only learn that a valid TPM signed that message, but without revealing the TPM's identity. The DAA protocol relies heavily on zero-knowledge proofs to achieve this kind of anonymous authentication.

The DAA protocol is composed of two sub-protocols: the *join protocol* and the *DAA-signing protocol*. The join protocol allows a TPM to obtain a certificate from an entity called the issuer. The DAA-signing protocol enables a TPM to authenticate a message and to prove the verifier to own a valid certificate without revealing the TPM's identity. The protocol ensures that even the issuer cannot link the TPM to its subsequently produced DAA-signatures.

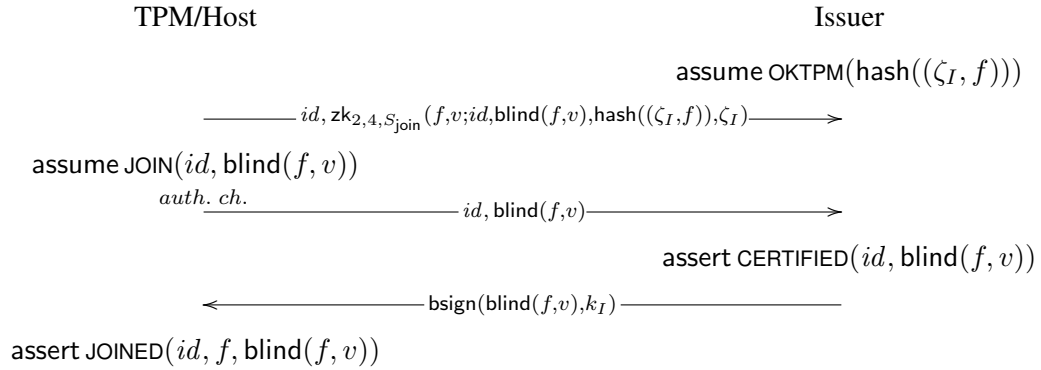
Every TPM has a unique id as well as a key-pair $(k_{id}, \mathsf{pk}(k_{id}))$ called *endorsement key* (EK). The issuer is assumed to know the public component $\mathsf{pk}(k_{id})$ of each EK. The protocol further assumes the existence a publicly known string bsn_I called the basename of the issuer. Every TPM has a secret seed $daaseed$ that allows it to derive secret values $f := \mathsf{hash}\big(\ (\mathsf{hash}\big(\ (daaseed, \mathsf{pk}(k_{id}))\ \big), cnt, n_0)\ \big)$, where $\mathsf{pk}(k_{id})$ is the public-key of the TPM, cnt is a counter, and n_0 is the integer 0. Each such f-value represents a virtual identity with respect to which the TPM can execute the join and the DAA-signing protocol.

In order to prevent the issuer from learning f-values, DAA relies on blind signatures [16]. The idea is that the TPM sends the disguised (or blinded) f-value $\mathsf{blind}(f, r)$, where r is a random blinding factor, to the issuer, which then produces the blind signature $\mathsf{bsign}(\mathsf{blind}(f, r), k_I)$. The TPM can later unblind the signature obtaining a regular signature $\mathsf{sign}(f, k_I)$ of the f-value which can be publicly verified in the manner of a regular digital signature. The unblinding of blind signatures is ruled by the unblind destructor, while the verification of the unblinded signature is denoted by the bcheck destructor. The type $\mathsf{Blind}(T)$ describes blinded messages of type T, $\mathsf{BlindSigKey}(T)$ and $\mathsf{BlindVerKey}(T)$ describe signing and verification keys for blind signatures of messages of type T, $\mathsf{BlindSigned}(T)$ describes blind signatures of messages of type T, and $\mathsf{Blinder}(T)$ describes a blinding factor for messages of type T. DAA additionally relies on secret hashes $\mathsf{hash}_{\mathsf{Private}}(M)$, which are given type $\mathsf{Hash}_{\mathsf{Private}}(T)$.

The process specification for DAA is reported in Table 3 in the appendix.

6.1 Join protocol

In the join protocol, the TPM can receive a certificate for one of its f-values f from the issuer. The join protocol has the following overall shape:



The TPM sends to the issuer the blinded f-value $\mathsf{blind}(f, v)$, for some random blinding factor v. The TPM is also required to send the hash value $\mathsf{hash}\big((\zeta_I, f)\big)$ along with its request where ζ_I is a value derived from the issuer's basename bsn_I. This message is used in a rogue-tagging procedure allowing the issuer

to recognize corrupted TPMs. All these messages are transmitted together with a zero-knowledge proof, which guarantees that the f-value f is hashed together with ζ_I in $\mathsf{hash}\big((\zeta_I, f)\big)$. The statement of this zero-knowledge proof is modeled as follows:

$$S_{\mathsf{join}} \;:=\; (\mathsf{blind}(\alpha_1, \alpha_2) = \beta_2 \;\wedge\; \mathsf{hash}\big((\beta_4, \alpha_1)\big) = \beta_3)$$

The DAA protocol assumes an authentic channel between the TPM and the issuer to authenticate the blinded f-value, and the authors suggest a challenge-response handshake as a possible implementation [13]. Type-checking a challenge-response handshake would require us to introduce challenge-response nonce types similarly to [26, 27, 30]. For the sake of simplicity, we abstract away from the actual cryptographic implementation of such an authentic channel, and we let the TPM send its own identifier together with the blinded f-value over a private channel shared with the issuer. Note that the blinded f-value is still known to the attacker, since it occurs in the public component of the zero-knowledge proof, which is sent over an untrusted channel. Finally, the issuer sends to the TPM the blind signature $\mathsf{bsign}(\mathsf{blind}(f, v), k_I)$.

Type-checking the Join Protocol. The type specified by the user for $f^{code}_{2,4,S_{\mathsf{join}}}$ is $\mathsf{Stm}(T_{\mathsf{join}})$, where T_{join} is

$$\langle y_{id} : \mathsf{Un}, y_U : \mathsf{Blind}(T_f), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}\rangle\{\exists x_1, x_2.(y_U = \mathsf{blind}(x_1, x_2) \;\wedge\; y_N = \mathsf{hash}\big((y_\zeta, x_1)\big))\}$$

and the type of the f-value is $T_f := \mathsf{Hash}_{\mathsf{Private}}(\,(x : \mathsf{Hash}_{\mathsf{Private}}((x_1 : \mathsf{Private}, x_2 : \mathsf{Un})), y : \mathsf{Un}, z : \mathsf{Un})\,)$. The formula in this type gives just a logical characterization of the structure of the messages sent by the TPM to the issuer, which is directly guaranteed by the statement S_{join} of the zero-knowledge proof. Therefore $\langle\!\langle\, S \,\rangle\!\rangle_{\,2,4,4,S_{\mathsf{join}},T_{\mathsf{join}},T_{\mathsf{Un}}}$ holds on the verifier's side and the logical formula is inserted into the typing environment (T_{Un} is the untrusted type $\mathsf{ZKProof}_{2,4,S_{\mathsf{join}}}(\langle y_1 : \mathsf{Un}, y_2 : \mathsf{Un}, y_3 : \mathsf{Un}, y_4 : \mathsf{Un}\rangle\{\mathsf{true}\})$ of zero-knowledge proofs received from untrusted channels). The type of the authentic channel is $\mathsf{Ch}(\langle y_{id} : \mathsf{Un}, y_U : \mathsf{Blind}(T_f)\rangle\{\mathsf{JOIN}(y_{id}, y_U)\})$. The type system guarantees that the TPM assumes $\mathsf{JOIN}(id, U)$ before sending id and the blinded f-value U on such a channel. Finally, the type of the issuer's signing key is

$$\mathsf{BlindSigKey}(\langle y_U : \mathsf{Blind}(T_f)\rangle\{\exists id.\mathsf{CERTIFIED}(id, y_U)\})$$

The type system guarantees that whenever the issuer releases a certificate for message M, M is a blinded secret and there exists id such that $\mathsf{CERTIFIED}(id, M)$ is entailed by the formulas in the typing environment. The authorization policy for the join protocol is as follows:

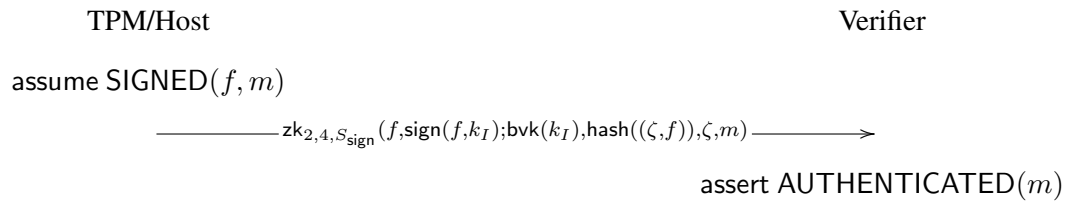
$$\begin{aligned}\forall id, f, v_1, v_2.(\mathsf{JOIN}(id, \mathsf{blind}(f, v_1)) \wedge \mathsf{OKTPM}(\mathsf{hash}\big((v_2, f)\big)) \Rightarrow \mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))) \wedge \\ (\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1)) \Rightarrow \mathsf{JOINED}(id, f, \mathsf{blind}(f, v_1)))\end{aligned}$$

This policy allows the issuer to release a blind signature for TPM id (assertion $\mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))$) only if the TPM id has started the join protocol to authenticate $\mathsf{blind}(f, v_1)$ (assumption $\mathsf{JOIN}(id, \mathsf{blind}(f, v_1))$) and the f-value f is associated to a valid TPM (assumption $\mathsf{OKTPM}(\mathsf{hash}\big((v_2, f)\big))$). Additionally, the policy guarantees that whenever a TPM id successfully completes the join protocol (assertion $\mathsf{JOINED}(id, f, \mathsf{blind}(f, v_1))$), the issuer has certified $\mathsf{blind}(f, v_1)$ (assertion $\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1))$).

6.2 DAA-signing protocol

After successfully executing the join protocol, the TPM has a valid certificate for its f-value f signed by the issuer. Since only valid TPMs should be able to DAA-sign a message m, the TPM has to convince a verifier that it possesses a valid certificate. Of course, the TPM cannot directly send it to the verifier, since this would reveal f. Instead, the TPM produces $zksign$, a zero-knowledge proof that it knows a valid certificate. If the TPM would, however, just send $(zksign, m)$ to the verifier, the protocol would be subject to a trivial message substitution attack. Message m is instead combined with the proof so that one can only replace m by redoing the proof (and this again can only be done by knowing a valid certificate). The overall shape of

the DAA-signing protocol is hence as follows:



with $S_{\mathsf{sign}} := (\mathsf{bcheck}(\alpha_2, \beta_1) = \alpha_1 \wedge \mathsf{hash}\big((\beta_3, \alpha_1)\big) = \beta_2)$. The zero-knowledge proof guarantees that the secret f-value f is signed by the issuer and that such a value is hashed together with a fresh value ζ[6]. This hash is used in the rogue tagging procedure mentioned above.

Type-checking the DAA-signing Protocol. The type specified by the user for $f^{code}_{2,4,S_{\mathsf{sign}}}$ is $\mathsf{Stm}(T_{\mathsf{sign}})$, where

$$\begin{aligned} T_{\mathsf{sign}} := \; & \langle y_{vk} : \mathsf{BlindVerKey}(T_{k_I}), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}, y_m : \mathsf{Un}\rangle \\ & \{\exists x_f, x_c, x_v, x_{id}.\; y_N = \mathsf{hash}\big((y_\zeta, x_f)\big) \wedge \mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v)) \wedge \mathsf{SIGNED}(x_f, y_m)\} \end{aligned}$$

This type guarantees that the f-value of the TPM has been certified by the issuer (assertion $\mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v))$, captures the constraint on the hash inherited from the statement of the zero-knowledge proof ($y_N = \mathsf{hash}\big((y_\zeta, x_f)\big)$), and states that the user has signed message m (assumption $\mathsf{SIGNED}(x_f, y_m)$). On the verifier's side, the assertion $\mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v))$ is guaranteed to hold by the verification of the certificate proved by zero-knowledge and by type of the verification key, while the equality $y_N = \mathsf{hash}\big((y_\zeta, x_f)\big)$ is enforced by the semantics of the ver destructor. Furthermore the type of the verification key guarantees that the f-value is of type T_f. Since values of this type are neither public nor tainted, the proof is generated by a honest TPM, and thus $\langle\!\langle\, S \,\rangle\!\rangle_{\,2,4,4,S_{\mathsf{sign}},T_{\mathsf{sign}},T_{\mathsf{Un}}}$ holds, and the logical formula is inserted into the typing environment (T_{Un} is the usual untrusted type for zero-knowledge proofs received from untrusted channels). The authorization policy for the DAA-signing protocol is:

$$\forall f, v, m.(\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v)) \wedge \mathsf{SIGNED}(f, m)) \Rightarrow \mathsf{AUTHENTICATED}(m)$$

This policy allows the verifier to authenticate message m (assertion $\mathsf{AUTHENTICATED}(m)$) only if the sender proves the knowledge of some certified f-value f associated to some TPM id (assertion $\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v))$) and the zero-knowledge proof of knowledge of f includes message m (assumption $\mathsf{SIGNED}(f, m)$). Note that the TPM id is existentially quantified, since it is not known to the verifier.

Our type-checker can prove in less than three seconds that $f^{code}_{2,4,S_{\mathsf{join}}} : \mathsf{Stm}(T_{\mathsf{join}}), f^{code}_{2,4,S_{\mathsf{sign}}} : \mathsf{Stm}(T_{\mathsf{sign}}) \vdash_{\mathsf{Un}} \mathsf{daa}$. By Theorem 4.1, this guarantees that process daa is robustly safe.

7 Security Despite Compromise

In this section we discuss about the security of partially compromised systems, in which some of the participants are under the control of the adversary. We show that zero-knowledge proofs can be used to prove correct behavior to remote parties, which can safely derive authorization decisions based on these proofs.

The DAA protocol guarantees the intended authenticity property even if the TPM is corrupted. Intuitively, this property is guaranteed by the zero-knowledge proof used in the DAA-signing protocol: the TPM can generate this proof only if the f-value has been previously certified by the issuer. One might wonder whether the authorization policy for the DAA-signing protocol holds if we type-check the TPM code as an opponent, i.e., by removing all assumptions and assertions and giving all messages, including $daaseed$ and the f-value f, type Un.

[6] In the pseudonymous variant of the DAA-signing protocol ζ is derived in a deterministic fashion from the basename bsn_V of the verifier. Our analysis can be easily adapted to this variant.

The authorization policy and the type of the zero-knowledge proof for the DAA-signing protocol are given in Section 6.2. The first part of the policy ($\exists id'.\mathsf{CERTIFIED}(id', blind(f, v))$) is guaranteed by the zero-knowledge proof itself. No matter whether the TPM is corrupted or not, the proof guarantees that the TPM knows a signature of type $\mathsf{Signed}(\langle x : T_f\rangle\{\exists y, id.\mathsf{CERTIFIED}(id, \mathsf{blind}(x, y))\})$. The type of the verification key alone allows the verifier to check the type of the signature even if this signature is in the private component of the zero-knowledge proof.

The second part of authorization policy (i.e., $\mathsf{SIGNED}(f, m)$) holds only if the TPM is not corrupted. If the TPM is corrupted, then the adversary can generate a valid zero-knowledge proof without assuming $\mathsf{SIGNED}(f, m)$. As expected, this variant of DAA would not type-check because the statement verification would no longer allow the verifier to derive $\mathsf{SIGNED}(f, m)$, since f, the content of the signature, would have type Un instead of T_f. This scenario should, however, not be regarded as an attack since the adversary is just following the protocol. This suggests that, in order to prove that DAA is secure against TPM compromise, we could actually weaken the authorization policy as follows:

$$\forall f, v, m.(\exists id'.\mathsf{CERTIFIED}(id', blind(f, v)) \ \wedge \ \mathsf{VERIFIED}(f, cert; \mathsf{bvk}(k_I), N, \zeta, m)) \\ \Rightarrow \mathsf{AUTHENTICATED}(m)$$

Whenever the verification of a zero-knowledge proof of the form $\mathsf{zk}_{n,m,S}(M_1, \ldots, M_n; N_1, \ldots, N_m)$ succeeds we introduce a new assumption $\exists x_1, \ldots, x_n.\mathsf{VERIFIED}(x_1, \ldots, x_n, N_1, \ldots, N_m)$ that is automatically assumed in the process and justified in the typing environment. This can be achieved by changing the operational semantics of the ver destructor and refining the corresponding typing rule. The special assumption VERIFIED binds the private component and the public component of a successfully verified zero-knowledge proof. In the example above, $\mathsf{VERIFIED}(f, \ldots, m)$ binds the secret f-value f and the message m in the zero-knowledge proof of the DAA-signing protocol. After this modification, DAA can be successfully verified, even if the TPM is type-checked as an opponent. The same holds for the simple protocol introduced in Section 2, if the prover is type-checked as an opponent.

In general, we notice that some formulas are guaranteed to hold by the verification of a zero-knowledge proof whether the prover is corrupted or not. These formulas are the ones obtained from the validity of cryptographic operations proved by zero-knowledge (e.g., the assumption $\exists id'.\mathsf{CERTIFIED}(id', blind(f, v))$ obtained from the verification of the issuer' signature in DAA), and the special formula $\exists x_1, \ldots, x_n.\mathsf{VERIFIED}(x_1, \ldots, x_n, N_1, \ldots, N_n)$. We argue that this may be used to systematically enforce authenticity properties on partially compromised systems.

8 Conclusions

This paper shows how several security properties of zero-knowledge proofs can be characterized as authorization policies and statically enforced by a novel type system. We developed a tool to automate the type-checking procedure and rely on an automated theorem-prover to discharge proof obligations. We applied our technique to verify the authenticity properties of the Direct Anonymous Attestation (DAA) protocol.

The analysis technique proves to be very efficient. Furthermore, the combination of types and authorization logics constitutes a truly expressive framework to model and analyze a variety of trace-based security properties. Zero-knowledge proofs perfectly fit into this framework, offering the possibility to realize fine-grained authorization policies that rely on the existential quantification in the logic. This is particularly well suited for protocols for privacy and anonymity.

The type system for authorization policies proposed by Fournet et al. [24], on which our approach is grounded, has been recently applied to the analysis of protocol implementations written in F# [10]. We are confident that our technique could be incorporated into such a framework in order to verify implementations of protocols based on zero-knowledge proofs.

In Section 7 we give some ideas about the usage of zero-knowledge proofs in the design of systems that guarantee security despite partial compromise. We believe that zero-knowledge proofs are the natural candidate for strengthening protocol specifications against compromised participants, since they can be used to prove the correct behavior of remote parties and to safely derive authorization decisions. We consider a formal elaboration of these ideas an interesting direction for future research.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

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

[3] M. Abadi, B. Blanchet, and C. Fournet. Automated verification of selected equivalences for security protocols. In *Proc. 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340. IEEE Computer Society Press, 2005.

[4] A. Acquisti. Receipt-free homomorphic elections and write-in ballots. Cryptology ePrint Archive, Report 2004/105, 2004. `http://eprint.iacr.org/`.

[5] M. Backes, A. Cortesi, R. Focardi, and M. Maffei. A calculus of challenges and responses. In *Proc. 5th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 101–116. ACM Press, 2007.

[6] M. Backes, C. Hriţcu, and M. Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. To appear in 21th IEEE Symposium on Computer Security Foundations (CSF), 2008.

[7] M. Backes, C. Hriţcu, and M. Maffei. Type-checking zero-knowledge. Long version available at `http://www.infsec.cs.uni-sb.de/~hritcu/publications/zk-types-full.pdf`, 2008.

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

[9] M. Backes, M. Maffei, and D. Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. To appear in IEEE Symposium on Security and Privacy, 2008.

[10] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations, 2008. To appear in 21th IEEE Symposium on Computer Security Foundations (CSF).

[11] B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 82–96. IEEE Computer Society Press, 2001.

[12] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS. In *Advances in Cryptology: CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1998.

[13] 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.

[14] M. Bugliesi, R. Focardi, and M. Maffei. Dynamic types for authentication. *Journal of Computer Security*, 15(6):563–617, 2007.

[15] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. Formal analysis of Kerberos 5. *Theoretical Computer Science*, 367(1):57–87, 2006.

[16] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: CRYPTO'82*, pages 199–203, 1983.

[17] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. To appear in IEEE Security and Privacy, 2008.

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

[19] S. Delaune, S. Kremer, and M. Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 28–42. IEEE Computer Society Press, 2006.

[20] S. Delaune, M. Ryan, and B. Smyth. Automatic verification of privacy properties in the applied pi calculus. To appear in 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM'08), 2008.

[21] B. Dragovic, E. Kotsovinos, S. Hand, and P. R. Pietzuch. Xenotrust: Event-based distributed trust management. In *Proc. 14th International Workshop on Database and Expert Systems Applications (DEXA'03)*, pages 410–414. IEEE Computer Society Press, 2003.

[22] D. Fisher. Millions of .Net Passport accounts put at risk. *eWeek*, May 2003. (Flaw detected by Muhammad Faisal Rauf Danka).

[23] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. In *Proc. 14th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, pages 141–156. Springer-Verlag, 2005.

[24] 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.

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

[26] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *Journal of Computer Security*, 4(11):451–521, 2003.

[27] A. D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3):435–484, 2004.

[28] A. D. Gordon and A. Jeffrey. Secrecy despite compromise: Types, cryptography, and the pi-calculus. In *Proc. 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653, pages 186–201. Springer-Verlag, 2005.

[29] C. Haack and A. Jeffrey. Timed spi-calculus with types for secrecy and authenticity. In *Proc. 16th International Conference on Concurrency Theory (CONCUR)*, volume 3653, pages 202–216. Springer-Verlag, 2005.

[30] C. Haack and A. Jeffrey. Pattern-matching spi-calculus. *Information and Computation*, 204(8):1195–1263, 2006.

[31] A. Juels, D. Catalano, and M. Jakobsson. Coercion-resistant electronic elections. In *Proc. 4nd ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 61–70. ACM Press, 2005.

[32] S. D. Kamvar, M. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *In Proc. 12th International World Wide Web Conference (WWW)*, pages 640–651. ACM Press, 2003.

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

[34] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 29–40, 1996.

[35] C. Weidenbach, R. Schmidt, T. Hillenbrand, R. Rusev, and D. Topic. System description: SPASS version 3.0. In *Automated Deduction – CADE-21 : 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 514–520. Springer, 2007.

A Appendix

Table 1 Typing Processes $\Gamma \vdash P$

PROC-OUT
$$\frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(M, N).P}$$

PROC-(REPL)-IN
$$\frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma, x : T \vdash P}{\Gamma \vdash [!]\mathsf{in}(M, x).P}$$

PROC-NEW
$$\frac{T \in \{\mathsf{Un}, \mathsf{Ch}(U), \mathsf{SigKey}(U), \mathsf{PrivKey}(U), \mathsf{Private}\} \qquad \Gamma, a : T \vdash P}{\Gamma \vdash \mathsf{new}\ a : T.P}$$

PROC-STOP
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

PROC-PAR
$$\frac{P \rightsquigarrow \Gamma_P \qquad \Gamma, \Gamma_P \vdash Q \qquad Q \rightsquigarrow \Gamma_Q \qquad \Gamma, \Gamma_Q \vdash P}{\Gamma \vdash P \mid Q}$$

PROC-DES
$$\frac{\begin{array}{c} g : (T_1, \ldots, T_n) \mapsto T \qquad \forall i \in [1, n].\ \Gamma \vdash M_i : T_i \\ \Gamma, x : T, \mathsf{eq}^\sharp(x, g^\sharp(M_1, \ldots, M_n)) \vdash P \qquad \Gamma \vdash Q \qquad g \notin \{\mathsf{snd}, \mathsf{exercise}, \mathsf{ver}\} \end{array}}{\Gamma \vdash \mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

PROC-SECOND
$$\frac{\Gamma \vdash M : \mathsf{Pair}(y : T_1, T_2) \qquad \Gamma, x : T_2\{\mathsf{fst}^\sharp(M)/y\}, \mathsf{eq}^\sharp(x, \mathsf{snd}^\sharp(M)) \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathsf{let}\ x = \mathsf{snd}(M)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

PROC-EXERCISE
$$\frac{\Gamma \vdash M : \mathsf{Ok}(C) \qquad \Gamma, C, x : \mathsf{Ok}(C) \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathsf{let}\ x = \mathsf{exercise}(M)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

PROC-VER
$$\frac{\begin{array}{c} \Gamma \vdash f_{n,m,S} : \mathsf{Stm}(\langle y_1 : T_1, \ldots, y_m : T_m \rangle\{C\}) = T \\ \Gamma \vdash M : \mathsf{ZKProof}_{n,m,S}(\langle y_1 : U_1, \ldots, y_m : U_m \rangle\{\mathsf{true}\}) = T' \qquad \forall i \in [1, l].\ \Gamma \vdash N_i : T_i \\ \langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'}\ \mathsf{holds} \qquad \Gamma, x : \langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m \rangle\{C\{\widetilde{N_l}/\widetilde{y_l}\}\} \vdash P \qquad \Gamma \vdash Q \end{array}}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(M, N_1, \ldots, N_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

PROC-ASSUME
$$\frac{\Gamma, C \vdash \diamond}{\Gamma \vdash \mathsf{assume}\ C}$$

PROC-ASSERT
$$\frac{\Gamma \vdash \diamond \qquad \mathit{forms}(\Gamma) \models C}{\Gamma \vdash \mathsf{assert}\ C}$$

Table 2 Statement Verification

$\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,\langle y_1:T_1,\ldots,y_m:T_m\rangle\{\exists x_1,\ldots,x_n.C\},\langle y_1:U_1,\ldots,y_m:U_m\rangle\{\mathsf{true}\}}$ holds

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

where $\Gamma' = \widetilde{x} : \widetilde{\mathsf{Un}}, y_1 : T_1, \ldots, y_l : T_l, y_{l+1} : U_{l+1}, \ldots, y_m : U_m$ and $dom(\Gamma) \cap (\widetilde{x} \cup \widetilde{y}) = \emptyset$

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

$[|S_1 \wedge^\sharp S_2|]_\Gamma \;=\; \Gamma_2$ if $[|S_1|]_\Gamma = \Gamma_1$ and $[|S_2|]_\Gamma = \Gamma_2$

$[|S_1 \vee^\sharp S_2|]_\Gamma \;=\; \Gamma_1 \cap \Gamma_2$ if $[|S_1|]_\Gamma = \Gamma_1$ and $[|S_2|]_\Gamma = \Gamma_2$

$[|\mathsf{check}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N|]_\Gamma \;=\; \Gamma[v_N : T^*], \mathsf{check}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N$
if $\Gamma(v_K) = \mathsf{VerKey}(T^*)$ and $\Gamma \not\vdash T^* :: \mathsf{tnt}$ and $\Gamma \vdash T^* :: \mathsf{pub}$

$[|\mathsf{check}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N|]_\Gamma \;=\; \Gamma[v_N : T^*, y_{l+1} : T_{l+1}, \ldots, y_m : T_m], C, \mathsf{check}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N$
if $\Gamma(v_K) = \mathsf{VerKey}(T^*)$ and $\Gamma \not\vdash T^* :: \mathsf{tnt}$ and $\Gamma \not\vdash T^* :: \mathsf{pub}$

$[|\mathsf{dec}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N|]_\Gamma \;=\; \Gamma[v_N : T^*, y_{l+1} : T_{l+1}, \ldots, y_m : T_m], C, \mathsf{dec}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N$
if $\Gamma(v_M) = \mathsf{PubEnc}(T^*)$ and $\Gamma \not\vdash T^* :: \mathsf{tnt}$ and $\Gamma \not\vdash T^* :: \mathsf{pub}$

$[|\mathsf{bcheck}(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N|]_{\Gamma,C} \;=\; \Gamma[v_N : \langle x : T\rangle\{\exists y.C'\{\mathsf{blind}(x, y)/z\}], \mathsf{bcheck}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N$
if $\Gamma(v_K) = \mathsf{BlindVerKey}(\langle z : \mathsf{Blind}(T)\rangle\{C'\})$ and $\Gamma \not\vdash T : \mathsf{tnt} \;\wedge\; \Gamma \vdash T : \mathsf{pub}$

$[|\mathsf{bcheck}(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N|]_{\Gamma,C} \;=\; \Gamma[v_N : \langle x : T\rangle\{\exists y.C'\{\mathsf{blind}(x, y)/z\}, y_{l+1} : T_{l+1}, \ldots, T_m : T_m],$
$C, \mathsf{bcheck}^\sharp(v_M, v_K)\ \mathsf{eq}^\sharp\ v_N$
if $\Gamma(v_K) = \mathsf{BlindVerKey}(\langle z : \mathsf{Blind}(T)\rangle\{C'\})$ and $\Gamma \not\vdash T : \mathsf{tnt} \;\wedge\; \Gamma \not\vdash T : \mathsf{pub}$

$[|v_M\ \mathsf{eq}^\sharp\ \mathsf{fst}^\sharp(v_N)|]_\Gamma \;=\; \Gamma[v_M : T], \mathsf{fst}^\sharp(v_N)\ \mathsf{eq}^\sharp\ v_M$ if $\Gamma(v_N) = \mathsf{Pair}(x : T, U)$

$[|v_M\ \mathsf{eq}^\sharp\ \mathsf{snd}^\sharp(v_N)|]_\Gamma \;=\; \Gamma[v_M : U\{\mathsf{fst}^\sharp(v_M)/x\}], v_M\ \mathsf{eq}^\sharp\ \mathsf{snd}^\sharp(v_N)$, if $\Gamma(v_N) = \mathsf{Pair}(x : T, U)$

$[|v_N\ \mathsf{eq}^\sharp\ \mathsf{exercise}^\sharp(v_N)|]_\Gamma \;=\; \Gamma, C^*, v_N\ \mathsf{eq}^\sharp\ \mathsf{exercise}^\sharp(v_N)$ if $\Gamma(v_N) = \mathsf{Ok}(C^*)$

$[|v_M\ \mathsf{eq}^\sharp\ \mathsf{hash}(v_N)|]_\Gamma \;=\; \Gamma[v_N : T^*], C, v_M\ \mathsf{eq}^\sharp\ \mathsf{hash}(v_N)$
if $\Gamma(v_M) = \mathsf{Hash}(T^*)$ and ($\Gamma \vdash T^* :: \mathsf{tnt}$ or $\Gamma \vdash T^* : \mathsf{pub}$)

$[|v_M\ \mathsf{eq}^\sharp\ \mathsf{hash}(v_N)|]_\Gamma \;=\; \Gamma[v_N : T^*, y_{l+1} : T_{l+1}, \ldots, y_m : T_m], C, v_M\ \mathsf{eq}^\sharp\ \mathsf{hash}(v_N)$
if $\Gamma(v_M) = \mathsf{Hash}(T^*)$ and $\Gamma \not\vdash T^* :: \mathsf{tnt}$ and $\Gamma \not\vdash T^* : \mathsf{pub}$

$[|S|]_\Gamma \;=\; \Gamma, S$ otherwise

Table 3 DAA system

T_{k_I} := $\mathsf{BlindSigKey}(\langle y_U : \mathsf{Blind}(T_f)\rangle\{\exists id.\mathsf{CERTIFIED}(id, y_U)\})$

$\mathsf{P}_{\mathsf{join}}$:= $\forall id, f, v_1, v_2.(\mathsf{JOIN}(id, \mathsf{blind}(f, v_1)) \wedge \mathsf{OKTPM}(\mathsf{hash}\big((v_2, f)\big)) \Rightarrow \mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))) \wedge$
$(\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1)) \Rightarrow \mathsf{JOINED}(id, f, \mathsf{blind}(f, v_1)))$

$\mathsf{P}_{\mathsf{sign}}$:= $\forall f, v, m.(\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v)) \wedge \mathsf{SIGNED}(f, m)) \Rightarrow \mathsf{AUTHENTICATED}(m)$

daa := $let\ S_{\mathsf{join}} := (\mathsf{blind}(\alpha_1, \alpha_2) = \beta_2 \ \wedge\ \mathsf{hash}\big((\beta_4, \alpha_1)\big) = \beta_3)\ in$
$let\ S_{\mathsf{sign}} := (\mathsf{bcheck}(\alpha_2, \beta_1) = \alpha_1 \ \wedge\ \mathsf{hash}\big((\beta_3, \alpha_1)\big) = \beta_2)\ in$
$let\ \zeta_I := \mathsf{hash}\big((n_1, bsn_I)\big)\ in$
$\mathsf{new}\ k_I : \mathsf{BlindSigKey}(T_{k_I}).\ \mathsf{new}\ daaseed : \mathsf{Private}.$
$let\ f := \mathsf{hash}_{\mathsf{Private}}\big(\ (\mathsf{hash}_{\mathsf{Private}}\big(\ (daaseed, \mathsf{pk}(k_{id}))\ \big), cnt, n_0)\ \big)\ in$
$let\ N_I := \mathsf{hash}(\zeta_I, f)\ in$
$\mathsf{new}\ authch : \mathsf{Ch}(\langle y_{id} : \mathsf{Un}, y_U : \mathsf{Blind}(T_f)\rangle\{\mathsf{JOIN}(y_{id}, y_U)\}).$
$(\mathsf{tpm} \mid \mathsf{issuer} \mid \mathsf{verifier} \mid \mathsf{assume}\ \mathsf{P}_{\mathsf{join}} \mid \mathsf{assume}\ \mathsf{P}_{\mathsf{sign}})$

$\mathsf{tpm} :=$
$\mathsf{new}\ v : \mathsf{Blinder}(T_f).$
$let\ U := \mathsf{blind}(f, v)\ in$
$(\mathsf{assume}\ \mathsf{JOIN}(id, U) \mid$
$let\ zkjoin := \mathsf{zk}_{2,4,S_{\mathsf{join}}}(f, v; id, U, N_I, \zeta_I)\ in$
$\mathsf{out}(pub, zkjoin).\mathsf{out}(authch,\ \langle id, U\rangle\ ).$
$\mathsf{in}(pub, x).$
$let\ cert = \mathsf{unblind}(x, v, \mathsf{bvk}(k_I))\ in$
$let\ \langle x_f\rangle = \mathsf{bcheck}(cert, \mathsf{bvk}(k_I))\ in$
$let\ x' = \mathsf{eq}(x_f, f)\ in$
$(\mathsf{assert}\ \mathsf{JOINED}(id, f, U) \mid$
$\mathsf{new}\ m : \mathsf{Un}.\ \mathsf{new}\ \zeta : \mathsf{Un}.$
$let\ N := \mathsf{hash}\big((\zeta, f)\big)\ in$
$(\mathsf{assume}\ \mathsf{SIGNED}(f, m) \mid$
$let\ zksign := \mathsf{zk}_{2,4,S_{\mathsf{sign}}}(f, cert; \mathsf{bvk}(k_I), N, \zeta, m)\ in$
$\mathsf{out}(pub, zksign))))$

$\mathsf{issuer} :=$
$\mathsf{assume}\ \mathsf{OKTPM}(N_I) \mid$
$!\mathsf{in}(pub, zkjoin).$
$\mathsf{in}(authch,\ \langle y_{id}, y_U\rangle\ ).$
$let\ \langle\rangle = \mathsf{ver}_{2,4,4,S_{\mathsf{join}}}(zkjoin, y_{id}, y_U, N_I, \zeta_I)\ in$
$(\mathsf{assert}\ \mathsf{CERTIFIED}(y_{id}, y_U) \mid$
$\mathsf{out}(pub, \mathsf{bsign}(\langle y_U\rangle, k_I)))$

$\mathsf{verifier} :=$
$!\mathsf{in}(pub, zksign).$
$let\ \langle x_N, x_\zeta, x_m\rangle = \mathsf{ver}_{2,4,1,S_{\mathsf{sign}}}(zksign, \mathsf{bvk}(k_I))\ in$
$\mathsf{assert}\ \mathsf{AUTHENTICATED}(x_m)$