Type-checking Zero-knowledge

Michael Backes
Saarland University and
MPI-SWS
backes@cs.uni-sb.de

Cătălin Hriţcu
Saarland University
Saarbrücken, Germany
hritcu@cs.uni-sb.de

Matteo Maffei
Saarland University
Saarbrücken, Germany
maffei@cs.uni-sb.de

ABSTRACT

This paper presents the first type system for statically analyzing security protocols that are based on zero-knowledge proofs. We show how certain 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 to real-world protocols by verifying the authenticity and secrecy properties of the Direct Anonymous Attestation (DAA) protocol. The analysis of DAA takes less than three seconds.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol Verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*

General Terms

Security, Theory, Verification

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 multitude of attacks on existing cryptographic protocols reported lately (e.g., [38, 16, 26, 19]) demonstrates the need for formalizing the intended security properties and developing automated techniques for automatically verifying these properties. 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 [2]. Type systems are particularly salient tools to statically and automatically enforce authorization policies on abstract protocol specifications [27, 28] and on concrete protocol implementations [14]. Type systems require little human effort and provide security proofs for an unbounded number of protocol executions. Furthermore, the analysis is modular, compositional, and usually guaranteed to terminate.

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 until recently [12] they could not cope with the most prominent and innovative modern cryptographic primitive: zero-knowledge proofs [29].

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 for their deployment in modern cryptographic applications. In fact, many anonymity protocols [17, 37] and electronic voting protocols [35, 6, 21] heavily rely on zero-knowledge proofs. These zero-knowledge proofs provide security properties 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 [37], 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 [17].

Statically analyzing protocols that use zero-knowledge proofs is conceptually and technically challenging. While the existing techniques for type-checking cryptographic protocols

usually rely on the type of keys for typing cryptographic messages, these techniques do not directly apply to zero-knowledge proofs, since in general zero-knowledge proofs 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 the safety properties guaranteed by zero-knowledge proofs can be formulated in terms of authorization policies and statically enforced by a type system. Our approach extends the state-of-the-art type system for authorization policies proposed by Fournet et al. [28]. Zero-knowledge proofs are given dependent types where the messages kept secret by the proof are existentially quantified in the logic. The fundamental idea is to express zero-knowledge statements as logical formulas and to define the type of zero-knowledge proofs using these formulas. The user still has the possibility to extend such types with additional logical formulas describing protocol-dependent security properties.

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 [39] to discharge proof obligations. The analysis is modular and compositional, and provides security proofs for an unbounded number of protocol executions.

We exemplify the applicability of our technique to real-world protocols by verifying the Direct Anonymous Attestation protocol (DAA) [17]. 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, 3], type systems were successfully used to analyze a wide range of security properties of cryptographic protocols, ranging from authenticity properties [31, 33, 34, 18, 8], to security despite compromised participants [32, 18, 28, 22], to authorization policies [27, 28, 14]. As mentioned before, type-checking is efficient, modular, compositional, and usually guaranteed to terminate. None of the existing type systems is, however, capable of dealing with zero-knowledge proofs.

To the best of our knowledge, ProVerif [15, 5] is the only automatic tool that has been applied to the analysis of protocols that use zero-knowledge proofs [12, 9, 24]. 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 [14], 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 [23, 9]), but is restricted to cryptographic primitives that can be expressed as convergent or linear equational theories [5]. 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 that use zero-knowledge proofs. Section 4 presents our type system for zero-knowledge. Section 5 discusses the implementation of our type-checker and the experimental evaluation of our technique. In Section 6 we apply our type system to analyze the Direct Anonymous Attestation (DAA) protocol. Section 7 concludes and provides directions for future work. The two appendices explain some of the technical details of our type system and list most of the typing rules. 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 [10].

2. ILLUSTRATIVE EXAMPLE

This section introduces the types of zero-knowledge proofs and 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 [37]. 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 m_s. This pseudonym replaces the actual identity of the party in the protocol. An arbitrary trust-management system like EigenTrust [36] or XenoTrust [25] can be used to certify the trust level of each pseudonym. Whenever a party (*prover*) wants to send a message m_p to another party (*verifier*), she has to bind m_p to her own pseudonym $\mathsf{hash}(m_s)$ and, in order to avoid impersonation, she has to prove the knowledge of m_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 m_s and additionally provides the non-malleability of m_p, i.e., changing m_p requires the adversary to redo the proof and thus to know m_s. The goal of this protocol is allowing the verifier to associate the trust level of the prover to m_p.

2.2 Zero-knowledge Proofs and Authorization Policies

Following [12], we represent our zero-knowledge proof by the following applied pi-calculus term: $\mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(m_s; \mathsf{hash}(m_s), m_p)$. In our setting, 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 (m_s in the example), while the other m arguments $M_1, \ldots, M_m$ form the *public component* and are revealed to the verifier ($\mathsf{hash}(m_s)$ and m_p in the example). The *statement* S of

the zero-knowledge proof is 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:

$$\mathsf{assume}\ \mathsf{Authenticate}(m_p, m_s) \qquad \mathsf{assume}\ \mathsf{Trust}(\mathit{pseudo}, k)$$
$$P \xrightarrow{\mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(m_s;\mathsf{hash}(m_s),m_p)} V$$
$$\mathsf{assert}\ \mathsf{Associate}(m_p, k)$$

Before generating the zero-knowledge proof to authenticate m_p using the knowledge of the secret m_s, the prover P assumes the logical predicate $\mathsf{Authenticate}(m_p, m_s)$. Before receiving the zero-knowledge proof, the verifier V knows that the trust level associated to the pseudonym *pseudo* is k and assumes the predicate $\mathsf{Trust}(\mathit{pseudo}, k)$. 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 m_s such that $\mathit{pseudo} = \mathsf{hash}(m_s)$ and allows the verifier to assert that the formula $\mathsf{Associate}(m_p, k)$ holds. The authorization policy can be expressed as follows:

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

Since this is the only assumption where the Associate predicate occurs, the verifier is allowed to give m_p trust level k only if the prover wants to authenticate m_p and knows some m_s for which the trust level of the pseudonym $\mathsf{hash}(m_s)$ is k. Everything that is not explicitly allowed by the authorization policy is prohibited.

2.3 The Type of Zero-knowledge Proofs

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

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

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 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 $\exists x.\ y_1 = \mathsf{hash}(x) \wedge \mathsf{Authenticate}(y_2, x)$ says that y_1 is the hash of some secret x such that $\mathsf{Authenticate}(y_2, x)$ has been assumed by the prover. Note that 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 $\mathsf{Authenticate}(y_2, x)$ is enforced by our type system.

The verifier can thus logically derive the following formula:

$$\exists m_s.\ \mathit{pseudo} = \mathsf{hash}(m_s) \ \wedge \ \mathsf{Authenticate}(m_p, m_s) \ \wedge \ \mathsf{Trust}(\mathit{pseudo}, k).$$

By a standard logical property that allows replacing equals by equals, the authorization policy allows the verifier to derive $\mathsf{Associate}(m_p, k)$.

3. CALCULUS

We consider a variant of the applied pi-calculus with constructors and destructors similar to the one in [4], and we extend it with zero-knowledge proofs. Following [28, 14], 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. The set of constructors includes pk that yields the public encryption key corresponding to a decryption key; enc for public-key encryption; vk that yields the verification key corresponding to a signing key; sign for digital signatures; and hash for hashes. The constant true represents the respective Boolean value, and has its canonical meaning in the authorization logic.

The set of *terms* (ranged over by K, L, M and N) is the free algebra built from names (a, b, c, m, n, and k), variables (x, y, z, v, and w), tuples ($\langle M_1, \ldots, M_n\rangle$), and constructors applied to other terms ($f(M_1, \ldots, M_n)$). We let u range over both 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 specified by the reduction 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 (which we denote as $g(M_1, \ldots, M_n) \Downarrow N$) or it can fail (denoted as $g(M_1, \ldots, M_n) \not\Downarrow$). The dec destructor decrypts an encrypted message given the corresponding decryption key ($\mathsf{dec}(\mathsf{enc}(M, \mathsf{pk}(K)), K) \Downarrow M$). 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 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 $\vee(\mathsf{true}, M) \Downarrow \mathsf{true}$).

3.3 Representing Zero-knowledge Proofs

Constructing Zero-knowledge Proofs. In a very similar way to what is proposed in [12], a non-interactive zero-knowledge proof of a statement S is represented as a term of the form $\mathsf{zk}_{n,m,S}(N_1, \ldots, N_n; M_1, \ldots, M_m)$, where $N_1, \ldots, N_n$ and $M_1, \ldots, M_m$ are two sequences of terms. The proof keeps the terms in $N_1, \ldots, N_n$ secret, while the terms $M_1, \ldots, M_m$ are revealed. 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. Since destructors cannot occur inside terms, 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 tuples, constructors and destructors (except for the public_m and $\mathsf{ver}_{n,m,l,S}$ destructors introduced below) 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 by writing the latter as $g^\sharp$. The statement S used in a term $\mathsf{zk}_{n,m,S}(N_1, \ldots, N_n; M_1, \ldots, M_m)$ is called an (n, m)-*statement*. It does not contain names or variables, and uses the placeholders α_i and β_j, with $i \in [1, n]$ and $j \in [1, m]$, to refer to the secret terms N_i and public terms M_j. For instance, the zero-knowledge term

$$\mathsf{zk}_{1,2,\mathsf{eq}^\sharp(\beta_1,\mathsf{dec}^\sharp(\mathsf{enc}(\beta_1,\beta_2),\alpha_1))}(\ k\ ;\ m, \mathsf{pk}(k)\ )$$

proves the knowledge of the decryption key k corresponding to the public encryption key $\mathsf{pk}(k)$. More precisely, the statement reads: "There exists a secret key k such that the decryption of the ciphertext $\mathsf{enc}(m, \mathsf{pk}(k))$ with this key yields m". As mentioned before, m and $\mathsf{pk}(k)$ are revealed by the proof while k is kept secret.

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

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

The evaluation relation $S \Downarrow_\sharp M$ is defined in terms of the reduction rules for the other destructors[1]. For example $g^\sharp(S_1, \ldots, S_n) \Downarrow_\sharp M$ if $\forall i \in [1, n].\ S_i \Downarrow_\sharp M_i$ and $g(M_1, \ldots, M_n) \Downarrow M$. The $\Downarrow_\sharp$ relation can in fact be seen as the extension of the $\Downarrow$ relation to statements. For instance, in the protocol from Section 2 we have

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

since $(\beta_1 = \mathsf{hash}(\alpha_1))\{m_s/\alpha_1\}\{\mathsf{hash}(m_s)/\beta_1\} \equiv (\mathsf{hash}(m_s) = \mathsf{hash}(m_s)) \Downarrow_\sharp \mathsf{true}$. Notice that in [12] the operational semantics of zero-knowledge proof verification is defined by an infinite equational theory, which needs to be compiled into a convergent rewriting system in order to be suitable to ProVerif. Since we use a type-system to analyze security protocols, in this paper we can take a simpler approach and formalize the verification of zero-knowledge proofs by means of a destructor like ver, whose semantics is not directly expressible in ProVerif.

The destructor public_m yields the public component of a zero-knowledge proof $(\mathsf{public}_m(\mathsf{zk}_{n,m,S}(\widetilde{N}, \widetilde{M})) \Downarrow \langle\widetilde{M}\rangle)$. Note that the private component is not revealed by any destructor, which intuitively guarantees the zero-knowledge property of the proofs.

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

3.4 Processes

Processes are essentially the same as in [28]. 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 $\mathsf{in}(M, x).P$ executed in parallel; $\mathsf{new}\ a : T.P$ generates a fresh name a of type T and then behaves as P; $P \mid Q$ 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 this succeeds and produces the term N $(g(\widetilde{M}) \Downarrow N)$ then the process behaves as $P\{N/x\}$, otherwise $(g(\widetilde{M}) \not\Downarrow)$ it behaves as Q; the process $\mathsf{let}\ \langle x_1, \ldots, x_n\rangle = M\ \mathsf{in}\ P$ splits the tuple M into its components.

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

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

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

Assertions specify logical formulas that are supposed to be entailed at run-time by the currently active assumptions. For instance, in the protocol from Section 2 the verifier asserts that it can associate trust level k to the message m_p ($\mathsf{assert}\ \mathsf{Associate}(m_p, k)$). In principle it might be possible to implement such assertions as (distributed!) dynamic checks. As in [28], 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, respectively. Furthermore, we add axioms in the logic that correspond to the semantics of destructors (e.g., for decryption we add the formula $\forall m, k.\ \mathsf{dec}^\sharp(\mathsf{enc}(m, \mathsf{pk}(k)), k) = m$ as an axiom). 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 [39] to discharge the proof obligations generated by our type system.

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 say that ϕ is closed if it does not have any free variables. We write $\phi\{\phi'/x\}$ for the outcome of the capture-avoiding substitution of ϕ' for each free occurrence of x in ϕ.

A 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 $\mathcal{E} = \mathsf{new}\ \widetilde{a} : \widetilde{T}.([\,] \mid P)$ 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 sometimes 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:

$$
\begin{aligned}
P := {} & \mathsf{new}\ m_p : \mathsf{Un}.(\mathsf{assume}\ \mathsf{Authenticate}(m_p, m_s) \mid \\
& \quad \mathsf{out}(c, \mathsf{zk}_{1,2,\beta_1=\mathsf{hash}(\alpha_1)}(m_s; \mathsf{hash}(m_s), m_p))) \\
V := {} & \mathsf{assume}\ \mathsf{TrustLevel}(pseudo, k) \mid \mathsf{in}(c, w). \\
& \quad \mathsf{let}\ x = \mathsf{ver}_{1,2,1,\beta_1=\mathsf{hash}(\alpha_1)}(w, pseudo)\ \mathsf{then} \\
& \quad \mathsf{let}\ \langle y_p \rangle = x\ \mathsf{then} \\
& \quad \mathsf{assert}\ \mathsf{Associate}(y_p, k) \\
PseudoTrust := {} & \mathsf{new}\ m_s : \mathsf{Private}. \\
& \quad P \mid let\ pseudo := \mathsf{hash}(m_s)\ in\ V \mid \mathsf{assume}\ Policy
\end{aligned}
$$

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 entailed by the active assumptions in every protocol execution.

Definition 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 2. (Opponent) A closed process is an *opponent* if it does not contain any assert and if the only type occurring therein is Un.

Definition 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 [28]. This basic type system is described in Sections 4.1 and 4.2. The most important novelty of our type system is the ability to reason about zero-knowledge proofs and Section 4.3 presents this in detail. Section 4.4 explains how our type system works on a fragment of the protocol for anonymous trust. Section 4.5 summarizes the security guarantees offered by the type system. Finally, the two appendices explain some technical details of our type system and list most of the typing rules.

4.1 Basic Types

Our type system has the following types: Private is the type of messages that are not revealed to the adversary; Un is the type of messages possibly known to the adversary; $\mathsf{Ch}(T)$ is the type of channels carrying messages of type T. As in [28, 14], tuples are given refinement types of the form $\langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}$. The formula C can depend on the variables $x_1, \ldots, x_n$. For example, in the protocol of Section 2, the verification of the zero-knowledge proof yields the term $\langle m_p \rangle$ 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 private 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. In all these cases the type T is usually a refinement type conveying a logical formula. For instance, $\mathsf{SigKey}(\langle x : \mathsf{Private} \rangle \{\mathsf{Ok}(x)\})$ is the type of keys that can be used to sign private messages M for which we statically know that $\mathsf{Ok}(M)$ holds.

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. We denote the formulas in a typing environment Γ by $forms(\Gamma)$. Intuitively, these formulas constitute a safe approximation of the formulas assumed at run-time.

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 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 is *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 is *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 $\mathsf{Private}$ from Section 2 is neither public nor tainted.

Typing Terms. The judgment $\Gamma \vdash M : T$ checks that message M has type T. The type of variables and names is simply looked up in the typing environment. A tuple $\langle M_1, \ldots, M_n \rangle$ has the refinement type $\langle x_1 : T_1, \ldots, x_n : T_n \rangle \{C\}$ if each M_i has type T_i and if additionally the formulas in the typing environment entail $C\{\widetilde{M}/\widetilde{x}\}$. The other cases are as one would expect (see Table 4 in Appendix A). For instance, if the message M has type T and the key K has type $\mathsf{SigKey}(T)$ then the signature $\mathsf{sign}(M, K)$ has type $\mathsf{Signed}(T)$.

Typing Processes. The typing judgment $\Gamma \vdash P$ checks whether the process P is well-typed. As we will show in Section 4.5, this guarantees that P is secure against an arbitrary adversary. The rules for type-checking processes are listed in Table 5 in Appendix A. 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 can be added to the typing environment in which Q is typed, and the top-level assumptions in Q can be added to the environment in which P is typed. This ensures that assumptions have global scope.

The process $\mathsf{assert}\ C$ is well-typed in a typing environment Γ only if $\mathit{forms}(\Gamma) \models C$. Intuitively, this guarantees the safety property of well-typed processes, since $\mathit{forms}(\Gamma)$ represents a safe approximation of the formulas assumed at run-time.

Type-checking the process $\mathsf{let}\ x = g(M_1, \ldots, M_n)\ \mathsf{then}\ P$ differs significantly from [28]. 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. 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 $\Gamma, x : T, x = \mathsf{check}^\sharp(M, K)$. This treatment of destructors is simpler and more elegant than the one in [28], and appears to be similar to the typing rules for splitting and matching from [14].

When type-checking a process that splits a tuple $\mathsf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathsf{in}\ P$, we need to ensure that M has a refinement type $\langle y_1 : T_1, \ldots, y_n : T_n \rangle \{C\}$. Then the continuation process P is checked in an environment extended with the bindings $x_1 : T_1, \ldots, x_n : T_n$ and two logical formulas. First, we assume the formula $\langle x_1, \ldots, x_n \rangle = M$, which is helpful if the same tuple is split again in P. More important, we assume that the formula in the refinement type holds, after appropriate variable replacement, i.e., the environment is extended with $C\{\widetilde{x}/\widetilde{y}\}$ when checking P. This is sound since the type system guarantees that when creating the tuple $\langle \widetilde{M} \rangle$ the formula $C\{\widetilde{M}/\widetilde{y}\}$ is entailed.

Example. As an example consider the final part of the verifier process in Section 3.7. The variable x holds a tuple with one component. The process splits this tuple and assigns the component to a variable y_p, and then asserts that y_p satisfies the predicate $\mathsf{Associate}(y_p, k)$. The process $\mathsf{let}\ \langle y_p \rangle = x\ \mathsf{in}\ \mathsf{assert}\ \mathsf{Associate}(y_p, k)$ is type-checked in the environment Γ_1 that contains:

$$(\forall m_p, k, m_s.(\mathsf{Authenticate}(m_p, m_s) \wedge \mathsf{TrustLevel}(\mathsf{hash}(m_s), k)) \Rightarrow \mathsf{Associate}(m_p, k)), \mathsf{TrustLevel}(\mathit{pseudo}, k),\ x : \langle w : \mathsf{Un} \rangle \{\exists v.\ \mathit{pseudo} = \mathsf{hash}(v) \wedge \mathsf{Authenticate}(w, v)\}.$$

After the tuple is split, the environment becomes:

$$\Gamma_2 = \Gamma_1, y_p : \mathsf{Un}, \langle y_p \rangle = x, \exists v.\ \mathit{pseudo} = \mathsf{hash}(v) \wedge\ \mathsf{Authenticate}(y_p, v).$$

In order to type-check the $\mathsf{assert}\ \mathsf{Associate}(y_p, k)$, we need to ensure that $\mathit{forms}(\Gamma_2) \models \mathsf{Associate}(y_p, k)$, which holds indeed in the authorization logic.

4.3 Type-checking Zero-knowledge

The main novelty of our type system is the treatment of zero-knowledge proofs.

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 tuple type listing the types of the arguments in the public component. The logical formula associated to this 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 $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ is entailed by the formulas in the typing environment.

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 in general on any key. This poses a problem since type-checking the verification of zero-knowledge proofs should propagate logical formulas in the typing environment of the verifier, and it is not clear what formulas to consider. For instance, when type-checking a process $\mathsf{let}\ \langle \widetilde{y} \rangle = \mathsf{ver}_{n,m,0,S}(z)\ \mathsf{then}\ P$, 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[2] (see Section 3.3). Such a formula, however, does not suffice to type-check most examples we have tried, since it does not mention any logical predicate.

In order to solve this problem, we allow the user to provide type annotations for each statement used in the process. For each (n, m)-statement S, this annotation is modeled by binding a distinguished free variable $s_{n,m,S}$ to a type of the form $\mathsf{Stm}(\langle y_1 : T_1, \ldots, y_m : T_m\rangle\{\exists x_1, \ldots, x_n.C\})$ in the initial typing environment. Additionally, the environment contains an implicit binding $s^{un}_{n,m,S} : \mathsf{Stm}(\mathsf{Un})$ used to type-check proofs of S generated by the adversary.

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

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

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 $C\{\widetilde{N}/\widetilde{x}\}\{\widetilde{M}/\widetilde{y}\}$ logically follows from the formulas in the typing environment of the prover. This way we ensure that the 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 Γ such that $\Gamma(s_{n,m,S}) = \mathsf{Stm}(T)$, for some $T = \langle y_1 : T_1, \ldots, y_m : T_m\rangle\{\exists x_1, \ldots, x_n.C\}$ annotated by the user, and a type T' such that $\Gamma \vdash N : \mathsf{ZKProof}_{n,m,S}(T')$. Zero-knowledge proofs are typically received from channels controlled by the attacker and in this case $T' = \langle y_1 : \mathsf{Un}, \ldots, y_m : \mathsf{Un}\rangle\{\mathsf{true}\}$, with $\mathsf{ZKProof}_{n,m,S}(T')$ being 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. Moreover, we need to check that N can be safely given the stronger type $\mathsf{ZKProof}_{n,m,S}(T)$.

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 is 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 $\mathsf{ZKProof}_{n,m,S}(T)$, since zero-knowledge proofs constructed by honest participants are type-checked against the type specified by the user. The same principle has been applied in [3] for verifying the secrecy of nonce handshakes.

This intuitive reasoning is formalized by the predicate $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'}$ (see Table 7 in Appendix B). If this predicate holds, then the zero-knowledge proof N of type $\mathsf{ZKProof}_{n,m,S}(T')$ is guaranteed to have the stronger type $\mathsf{ZKProof}_{n,m,S}(T)$ and we can safely give the last $m - l$ arguments of the proof type $\langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m\rangle\{C\{M_i/y_i\}_{i\in[1,l]}\}$. With this setup in place, the typing rule for the ver destructor is defined as follows:

$$\frac{\begin{array}{c}\Gamma \vdash N : \mathsf{ZKProof}_{n,m,S}(T') \qquad \Gamma(s_{n,m,S}) = \mathsf{Stm}(T), \\ \text{where } T = \langle y_1 : T_1, \ldots, y_m : T_m\rangle\{\exists x_1, \ldots, x_n.\ C\} \\ \forall i \in [1, l].\ \Gamma \vdash M_i : T_i \qquad \langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'} \text{ holds} \\ \Gamma, x : \langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m\rangle\{\exists \widetilde{x}.\ C\{M_i/y_i\}_{i\in[1,l]}\} \vdash P \\ \Gamma \vdash Q\end{array}}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(N, M_1, \ldots, M_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

4.4 Type-checking the Protocol for Anonymous Trust

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

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

The verifier receives the zero-knowledge proof from the untrusted[3] channel c. The zero-knowledge proof is thus bound to a variable w 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 continuation process. Type-checking the destructor application $\mathsf{ver}_{1,2,1,\beta_1=\mathsf{hash}(\alpha_1)}(w, \mathit{pseudo})$ can however rely on the fact that $\Gamma \vdash \mathit{pseudo} : \mathsf{Hash}(\mathsf{Private})$, and therefore pseudo is the hash of some message m_s of type $\mathsf{Private}$, which is neither public nor tainted. The statement guarantees that the prover knows m_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 $\langle y : \mathsf{Un}\rangle\{\exists v.\ \mathit{pseudo} = \mathsf{hash}(v) \wedge \mathsf{Authenticate}(y, v)\}$. This allows type-checking the continuation process as discussed at the end of Section 4.2.

4.5 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 $\{u_1, \ldots, u_n\} = \mathit{free}(P)$.

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

[3]We consider all free names to have type Un.

THEOREM 1 (ROBUST SAFETY). *Let* $\Gamma = s_{n_1,m_1,S_1} : \mathsf{Stm}(T_1), \ldots, s_{n_k,m_k,S_k} : \mathsf{Stm}(T_k)$. *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 [10].

5. IMPLEMENTATION

We implemented an automatic type-checker for the type system presented in this paper. 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 [39] to discharge the proof obligations. Internally, SPASS uses superposition for equational reasoning [7]. Our tool is written in Objective Caml, comprises approximately 3000 lines of code, and is available at [11].

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 to deal with the facts 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 (see Section 6) 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 30 non-trivial 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) [17]. 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. More precisely, 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.

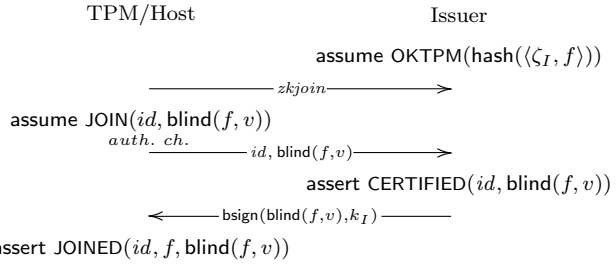
Every TPM has a unique id as well as a key-pair $(k_{id}, \mathsf{pk}(k_{id}))$ called *endorsement key* (EK). The issuer is assumed to know the public component $\mathsf{pk}(k_{id})$ of each EK. The protocol further assumes the existence a publicly known string bsn_I called the basename of the issuer. Every TPM has a secret seed $daaseed$ that allows it to derive secret values $f := \mathsf{hash}_{\mathsf{Private}}\big(\langle \mathsf{hash}_{\mathsf{Private}}\big(\langle daaseed, \mathsf{hash}(lpk_I)\rangle\big), cnt, n_0\rangle\big)$, where lpk_I is the long-term public key of the issuer, 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 [20]. 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 as 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 $\mathsf{Hash}_{\mathsf{Private}}(T)$.

Table 1 reports the process for the DAA system. For the sake of readability we use $\mathsf{let}\ \langle x_1, \ldots, x_n\rangle = g(\widetilde{M})\ \mathsf{then}\ P\ \mathsf{else}\ Q$ to denote the process $\mathsf{let}\ z = g(\widetilde{M})\ \mathsf{then}\ \mathsf{let}\ \langle x_1, \ldots, x_n\rangle = z\ \mathsf{in}\ P\ \mathsf{else}\ Q$, where z is a fresh variable. We also use $\langle T_1, \ldots, T_n\rangle$ to denote $\langle T_1, \ldots, T_n\rangle\{\mathsf{true}\}$.

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:



where $zkjoin = \mathsf{zk}_{2,4,S_{\mathsf{join}}}(f, v; id, \mathsf{blind}(f, v), \mathsf{hash}(\langle \zeta_I, f\rangle), \zeta_I)$. 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 $\mathsf{hash}\big(\langle \zeta_I, f\rangle\big)$ 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 $\mathsf{hash}\big(\langle \zeta_I, f\rangle\big)$. The statement of this zero-knowledge proof is modeled as follows:

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

Table 1 Our model of DAA

$$T_f := \mathsf{Hash}_{\mathsf{Private}}(\langle x : \mathsf{Hash}_{\mathsf{Private}}(\langle x_1 : \mathsf{Private}, x_2 : \mathsf{Un}\rangle), y : \mathsf{Un}, z : \mathsf{Un}\rangle)$$

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

$$s_{2,4,S_{\mathsf{join}}} : \langle y_{id} : \mathsf{Un}, y_U : \mathsf{Blind}(T_f), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}\rangle\ \{\exists x_1, x_2.(y_U = \mathsf{blind}(x_1, x_2)\ \wedge\ y_N = \mathsf{hash}(\langle y_\zeta, x_1\rangle))\}$$

$$s_{2,4,S_{\mathsf{sign}}} : \langle y_{vk} : \mathsf{BlindVerKey}(T_{k_I}), y_N : \mathsf{Un}, y_\zeta : \mathsf{Un}, y_m : \mathsf{Un}\rangle \{\exists x_f, x_c, x_v, x_{id}.\ y_N = \mathsf{hash}(\langle y_\zeta, x_f\rangle) \wedge \mathsf{CERTIFIED}(x_{id}, \mathsf{blind}(x_f, x_v)) \wedge\ \mathsf{SIGNED}(x_f, y_m)\}$$

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

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

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

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

$$\begin{aligned}
&\mathsf{issuer} :=\\
&\quad \mathsf{assume}\ \mathsf{OKTPM}(N_I) \mid\\
&\quad !\mathsf{in}(\mathit{pub}, \mathit{zkjoin}).\\
&\quad \mathsf{in}(\mathit{authch}, z).\\
&\quad \mathsf{let}\ \langle y_{id}, y_U\rangle = z\ \mathsf{in}\\
&\quad \mathsf{let}\ \langle\rangle = \mathsf{ver}_{2,4,4,S_{\mathsf{join}}}(\mathit{zkjoin}, y_{id}, y_U, N_I, \zeta_I)\ \mathsf{then}\\
&\quad (\mathsf{assert}\ \mathsf{CERTIFIED}(y_{id}, y_U) \mid\\
&\quad \mathsf{out}(\mathit{pub}, \mathsf{bsign}(\langle y_U\rangle, k_I)))\\
&\\
&\mathsf{verifier} :=\\
&\quad !\mathsf{in}(\mathit{pub}, \mathit{zksign}).\\
&\quad \mathsf{let}\ \langle x_N, x_\zeta, x_m\rangle = \mathsf{ver}_{2,4,1,S_{\mathsf{sign}}}(\mathit{zksign}, \mathsf{bvk}(k_I))\ \mathsf{then}\\
&\quad \mathsf{assert}\ \mathsf{AUTHENTICATED}(x_m)
\end{aligned}$$

The DAA protocol assumes an authentic channel between the TPM and the issuer in order to authenticate the blinded f-value, and the authors suggest a challenge-response handshake based on the TPM endorsement key as a possible implementation [17]. Type-checking a challenge-response handshake would require us to introduce challenge-response nonce types similarly to [30, 31, 34]. 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 $s_{2,4,S_{\mathsf{join}}}$ is $\mathsf{Stm}(T_{\mathsf{join}})$, where T_{join} is

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

and the type of the f-value is $T_f := \mathsf{Hash}_{\mathsf{Private}}(\langle x : \mathsf{Hash}_{\mathsf{Private}}(\langle x_1 : \mathsf{Private}, x_2 : \mathsf{Un}\rangle), y : \mathsf{Un}, z : \mathsf{Un}\rangle)$. The formula in T_{join} simply gives 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 $\langle\!\langle\, S\,\rangle\!\rangle_{2,4,4,S_{\mathsf{join}},T_{\mathsf{join}},\mathsf{Un}}$ holds on the verifier's side and the logical formula is inserted into the typing environment. 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

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

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:

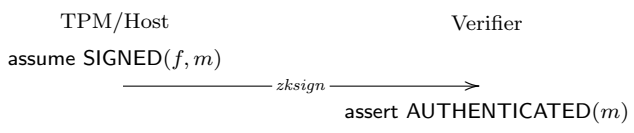
$$\begin{aligned}
&\forall id, f, v_1, v_2.(\mathsf{JOIN}(id, \mathsf{blind}(f, v_1)) \wedge \mathsf{OKTPM}(\mathsf{hash}(\langle v_2, f\rangle))\\
&\qquad \Rightarrow \mathsf{CERTIFIED}(id, \mathsf{blind}(f, v_1))) \wedge\\
&(\exists id'.\mathsf{CERTIFIED}(id', \mathsf{blind}(f, v_1)) \Rightarrow \mathsf{JOINED}(id,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

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 $\mathsf{OKTPM}(\mathsf{hash}(\langle v_2, f\rangle))$). 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:



if $zksign = \mathsf{zk}_{2,4,S_{\mathsf{sign}}}(f, \mathsf{sign}(f, k_I); \mathsf{bvk}(k_I), \mathsf{hash}(\langle\zeta, f\rangle), \zeta, m)$ and $S_{\mathsf{sign}} := (\mathsf{bcheck}(\alpha_2, \beta_1) = \alpha_1 \wedge \mathsf{hash}(\langle\beta_3, \alpha_1\rangle) = \beta_2)$. 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[4]. This hash is used in the rogue tagging procedure mentioned above.

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

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

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 ($y_N = \mathsf{hash}(\langle y_\zeta, x_f\rangle)$), 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 the type of the verification key, while the equality $y_N = \mathsf{hash}(\langle y_\zeta, x_f\rangle)$ 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 $\langle\!\langle\, S \,\rangle\!\rangle_{2,4,4,S_{\mathsf{sign}},T_{\mathsf{sign}},\mathsf{Un}}$ holds, and the logical formula is inserted into the typing environment. 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 includes message m (assumption $\mathsf{SIGNED}(f, m)$). Note that the id of the TPM is existentially quantified, since it is not known to the verifier.

Our type-checker can prove in less than three seconds that $s_{2,4,S_{\mathsf{join}}} : \mathsf{Stm}(T_{\mathsf{join}}), s_{2,4,S_{\mathsf{sign}}} : \mathsf{Stm}(T_{\mathsf{sign}}) \vdash_{\mathsf{Un}} \mathsf{daa}$. By Theorem 1, this guarantees that process daa is robustly safe.

7. CONCLUSIONS

This paper shows how certain security properties of zero-knowledge proofs can be characterized as authorization policies and statically enforced by a novel type system. The zero-knowledge proofs are given dependent types where the messages kept secret by the proof are existentially quantified in the logic. We developed a type-checker for this type system and use 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 is 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 implement 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 proposed by Fournet et al. [28], which our approach is grounded on, has been recently applied to the analysis of protocol implementations written in F# [14]. 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 the extended version of this paper [10] we explore 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 verify the correct behavior of remote parties and to safely derive authorization decisions. A formal elaboration of these ideas is an interesting direction for future research.

Finally, while there is no restriction on the shape of the statements that can be proved in our abstract setting, finding a sound and efficient cryptographic implementation for these symbolic zero-knowledge proofs is far from trivial. In [13] the authors identify the properties a concrete zero-knowledge proof system should satisfy in order to soundly implement a simpler abstraction of zero-knowledge, and mention two existing cryptographic constructions that satisfy these properties. The efficiency and expressivity of these constructions have, however, not been thoroughly studied and constitute interesting topics for further investigation.

Acknowledgments. The prototype implementation of our type system was developed by Stefan Lorenz and Kim R. Pecina. We thank Cédric Fournet, Andrew D. Gordon, and the anonymous reviewers for their valuable comments. We

[4]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.

also express our gratitude to Catherine Howell for her help on technical writing.

8. REFERENCES

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

[2] M. Abadi. Logic in access control. In *Proc. 18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 228–233. IEEE Computer Society Press, 2003.

[3] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Proc. 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 2001.

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

[5] 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.

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

[7] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[8] 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.

[9] M. Backes, C. Hriţcu, and M. Maffei. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 195–209. IEEE Computer Society Press, 2008.

[10] 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.

[11] 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.

[12] 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.

[13] M. Backes and D. Unruh. Computational soundness of symbolic zero-knowledge proofs against active attackers. In *Proc. 21th IEEE Symposium on Computer Security Foundations (CSF)*, pages 255–269. IEEE Computer Society Press, 2008.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] A. D. Gordon and A. Jeffrey. Authenticity by typing

for security protocols. *Journal of Computer Security*, 4(11):451–521, 2003.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

APPENDIX

A. TYPING RULES

Tables 2 and 3 are devoted to the types of some of the constructors and destructors. In Table 4 we define the term typing judgment. Table 5 lists the rules for typing processes, which are defined using the auxiliary environment extraction relation (Table 6) and the statement verification predicate later described in Appendix B.

B. STATEMENT VERIFICATION

As discussed in Section 4.3, the typing rule for the verification of a zero-knowledge proof of the form $\mathsf{zk}_{n,m,S}(N_1,\ldots,N_n;M_1,\ldots,M_m)$ relies on the predicate $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,U}$ in case the verification succeeds, where Γ is the current typing environment of the verifier, l is the number of public messages pattern-matched in the verification destructor, $\mathsf{Stm}(T)$ is the type specified by the user for the (n,m)-statement S, and $\mathsf{ZKProof}_{n,m,S}(U)$ is the type assigned to the zero-knowledge proof before the verification. The predicate $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,U}$ holds only

Table 2 Typing Constructors $f : (T_1,\ldots,T_n) \mapsto U$

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

Table 3 Typing Destructors $g : (T_1,\ldots,T_n) \mapsto U$

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

Table 4 Typing Terms $\Gamma \vdash M : T$

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

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

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

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

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

Table 5 Typing Processes $\Gamma \vdash P$

$$\textsc{Proc-Out}\quad \frac{\Gamma \vdash M : \mathsf{Ch}(T) \qquad \Gamma \vdash N : T \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{out}(M, N).P}$$

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

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

$$\textsc{Proc-Par}\quad \frac{P \rightsquigarrow \Gamma_P \qquad \Gamma, \Gamma_P \vdash Q \qquad Q \rightsquigarrow \Gamma_Q \qquad \Gamma, \Gamma_Q \vdash P}{\Gamma \vdash P \mid Q}$$

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

$$\textsc{Proc-Ver}\quad \frac{\Gamma \vdash N : \mathsf{ZKProof}_{n,m,S}(T') \qquad \Gamma(s_{n,m,S}) = \mathsf{Stm}(T), \text{ where } T = \langle y_1 : T_1, \ldots, y_m : T_m \rangle \{\exists x_1, \ldots, x_n.\ C\} \qquad \forall i \in [1, l].\ \Gamma \vdash M_i : T_i \qquad \langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,T'} \text{ holds} \qquad \Gamma, x : \langle y_{l+1} : T_{l+1}, \ldots, y_m : T_m \rangle \{\exists \widetilde{x}.\ C\{M_i/y_i\}_{i \in [1,l]}\} \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ver}_{n,m,l,S}(N, M_1, \ldots, M_l)\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

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

$$\textsc{Proc-Split}\quad \frac{\Gamma \vdash M : \langle y_1 : T_1, \ldots, y_n : T_n \rangle \{C\} \qquad \Gamma, x_1 : T_1, \ldots, x_n : T_n, \langle x_1, \ldots, x_n \rangle = M, C\{\widetilde{x}/\widetilde{y}\} \vdash P}{\Gamma \vdash \mathsf{let}\ \langle x_1, \ldots, x_n \rangle = M\ \mathsf{in}\ P}$$

$$\textsc{Proc-Assume}\quad \frac{\Gamma, C \vdash \diamond}{\Gamma \vdash \mathsf{assume}\ C} \qquad \textsc{Proc-Assert}\quad \frac{\Gamma \vdash \diamond \qquad \mathit{forms}(\Gamma) \models C}{\Gamma \vdash \mathsf{assert}\ C}$$

Table 6 Environment Extraction $P \rightsquigarrow \Gamma$

$$\textsc{Extr-New}\quad \frac{P \rightsquigarrow \Gamma_P}{\mathsf{new}\ a : T.P \rightsquigarrow a : T, \Gamma_P} \qquad \textsc{Extr-Par}\quad \frac{P \rightsquigarrow \Gamma_P \qquad Q \rightsquigarrow \Gamma_Q}{P \mid Q \rightsquigarrow \Gamma_P, \Gamma_Q}$$

$$\textsc{Extr-Assume}\quad \mathsf{assume}\ C \rightsquigarrow C \qquad \textsc{Extr-Empty}\quad P \rightsquigarrow \emptyset$$

if the zero-knowledge proof is actually of the stronger type $\mathsf{ZKProof}_{n,m,S}(T)$ (we have that $\mathsf{ZKProof}_{n,m,S}(T) <: \mathsf{ZKProof}_{n,m,S}(U)$). Without loss of generality we assume that $T = \langle y_1 : T_1, \ldots, y_m : T_m \rangle \{\exists x_1, \ldots, x_n.C\}$ and $U = \langle y_1 : U_1, \ldots, y_m : U_m \rangle \{\mathsf{true}\}$. The formalization is reported in Table 7.

The predicate $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,U}$ relies on the auxiliary function $[|S\{\widetilde{x}/\widetilde{\alpha}\}\{\widetilde{y}/\widetilde{\beta}\}|]^{y_{l+1}:T_{l+1},\ldots,y_m:T_m;C}_{\Gamma'}$ (which we will often abbreviate as $[|S|]_{\Gamma'}$), where Γ' extends Γ with type bindings for the fresh variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, corresponding to the arguments in the private and public component of the zero-knowledge proof, respectively. The variables $x_1, \ldots, x_n$ are given type $\top$ (top) in Γ', since no type information is available for them. The variables $y_1, \ldots, y_l$ are given types $T_1, \ldots, T_l$, since the first l messages in the public component are pattern-matched in the verification destructor with values having types $T_1, \ldots, T_l$. Finally, the variables $y_{l+1}, \ldots, y_m$ are given types $U_{l+1}, \ldots, U_m$, as specified in the weaker type $\mathsf{ZKProof}_{n,m,S}(U)$ assigned to the zero-knowledge proof before verification. The function $[|S|]_{\Gamma'}$ is described in detail in Section B.1. Intuitively, it returns a typing environment Γ'' that refines the type bindings in Γ' so that if $\Gamma'' \vdash x_i : T'_i$, then $\Gamma \vdash N_i : T'_i$, and similarly if $\Gamma'' \vdash y_j : T_j$, then $\Gamma \vdash M_j : T_j$. The function exploits the assumption that verification succeeds, so by the operational semantics $S\{\widetilde{N}/\widetilde{\alpha}\}\{\widetilde{M}/\widetilde{\beta}\}$ holds true. This is used to partially infer the types of the messages in the private and public components of the zero-knowledge proof. Please note that the messages in the private component are not known to the verifier.

With this setup in place, we say that the predicate $\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,T,U}$ holds if $[|S\{\widetilde{x}/\widetilde{\alpha}\}\{\widetilde{y}/\widetilde{\beta}\}|]^{y_{l+1}:T_{l+1},\ldots,y_m:T_m;C}_{\Gamma'}$ returns a typing environment Γ'' such that the formulas in Γ'' entail C and the variables $y_{l+1}, \ldots, y_m$, corresponding to the public messages that are returned by the verification destructor, have types $T_{l+1}, \ldots, T_m$ in Γ''. In other words, Γ'' proves that the zero-knowledge proof has the stronger type $\mathsf{ZKProof}_{n,m,S}(\langle y_1 : T_1, \ldots, y_m : T_m \rangle \{\exists x_1, \ldots, x_n.C\})$ specified by the user.

B.1 Inferring the type of terms in zero-knowledge proofs

The function $[|S|]_{\Gamma'}$ is defined by induction on the structure of the statement S. If the statement is a conjunction of the form $S_1 \wedge^\sharp S_2$, then we know that both S_1 and S_2 hold true and we can thus combine together the typing information and the formulas obtained from each of the statements. Formally, the function yields the typing environment composed of the intersection of the type bindings and the conjunction of the formulas in the typing environments $[|S_1|]_{[|S_2|]_{\Gamma'}}$ and $[|S_2|]_{[|S_1|]_{\Gamma'}}$. The intersection of two type bindings $x : T$ and $x : U$ is defined only if T and U are comparable and it returns $x : T$ if $T <: U$, and $x : U$ otherwise; this definition is extended to typing environments as expected. Note that the information obtained from a first processing of S_2 is used when processing S_1 and vice versa. This symmetrical formulation ensures that the order of the conjuncts in a statement is not relevant.

In case the statement is a disjunction $S_1 \vee^\sharp S_2$, then we do not know which statement between S_1 and S_2 holds true, and we can thus only infer typing information and formulas that are guaranteed by both the statements. Therefore the

Table 7 Statement Verification

$$\langle\!\langle\, S \,\rangle\!\rangle_{\Gamma,n,m,l,\langle y_1:T_1,\ldots,y_m:T_m\rangle\{\exists x_1,\ldots,x_n.C\},\langle y_1:U_1,\ldots,y_m:U_m\rangle\{\mathsf{true}\}} \text{ holds}$$
$$\text{if } [|S\{\widetilde{x}/\widetilde{\alpha}\}\{\widetilde{y}/\widetilde{\beta}\}|]_{\Gamma'}^{y_{l+1}:T_{l+1},\ldots,y_m:T_m;C} = \Gamma'',\ \mathit{forms}(\Gamma'') \models C \text{ and } \forall j \in [l+1,m].\ \Gamma'' \vdash y_j : T_j$$
$$\text{where } \Gamma' = \Gamma, \widetilde{x} : \widetilde{\top}, y_1 : T_1, \ldots, y_l : T_l, y_{l+1} : U_{l+1}, \ldots, y_m : U_m \text{ and } \Gamma' \vdash \diamond$$

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

$$[|S_1 \wedge^\sharp S_2|]_{\Gamma'} = \mathit{binds}([[S_1]]_{[|S_2|]_{\Gamma'}}) \sqcap \mathit{binds}([[S_2]]_{[|S_1|]_{\Gamma'}}), (\textstyle\bigwedge_{C \in \mathit{forms}([[S_1]]_{[|S_2|]_{\Gamma'}}) \cup \mathit{forms}([[S_2]]_{[|S_1|]_{\Gamma'}})} C)$$

$$[|S_1 \vee^\sharp S_2|]_{\Gamma'} = \mathit{binds}([[S_1]]_{\Gamma'}) \sqcup \mathit{binds}([[S_2]]_{\Gamma'}), ((\textstyle\bigwedge_{C \in \mathit{forms}([[S_1]]_{\Gamma'})} C) \vee (\bigwedge_{C \in \mathit{forms}([[S_2]]_{\Gamma'})} C))$$

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

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

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

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

$$[|\langle v_{M_1}, \ldots, v_{M_k}\rangle = v_N|]_{\Gamma'} = \Gamma'[v_{M_1} : U_1, \ldots, v_{M_k} : U_k], C^*\{\widetilde{v_M}/\widetilde{z}\}, \langle v_{M_1}, \ldots, v_{M_k}\rangle = v_N \text{ if } \Gamma'(v_N) = \langle \widetilde{z} : \widetilde{U}\rangle\{C^*\}$$

$$[|S|]_{\Gamma'} = \Gamma', S \text{ otherwise}$$

Definitions: For all Γ_1 and Γ_2 such that $\mathit{dom}(\Gamma_1) = \mathit{dom}(\Gamma_2)$ (same variables and names in the same order) and $\mathit{forms}(\Gamma_1) = \mathit{forms}(\Gamma_2) = \emptyset$, we define $\Gamma_1 \sqcap \Gamma_2$ and $\Gamma_1 \sqcup \Gamma_2$ as follows:

$$\emptyset \sqcap \emptyset = \emptyset \qquad (\Gamma_1, u : T) \sqcup (\Gamma_2, u : U) = (\Gamma_1 \sqcup \Gamma_2), u : T \sqcup_{(\Gamma_1 \sqcup \Gamma_2)} U$$
$$\emptyset \sqcup \emptyset = \emptyset \qquad (\Gamma_1, u : T) \sqcap (\Gamma_2, u : U) = (\Gamma_1 \sqcap \Gamma_2), u : T \sqcap_{(\Gamma_1 \sqcap \Gamma_2)} U$$

$$T \sqcap_\Gamma U = \begin{cases} T & \text{if } \Gamma \vdash T <: U \\ U & \text{otherwise if } \Gamma \vdash U <: T \end{cases} \qquad T \sqcup_\Gamma U = \begin{cases} U & \text{if } \Gamma \vdash T <: U \\ T & \text{otherwise if } \Gamma \vdash U <: T \end{cases}$$

function yields the typing environment composed of the union of the type bindings and the disjunction of the formulas in the typing environments $[|S_1|]_{\Gamma'}$ and $[|S_2|]_{\Gamma'}$.

In the following, we let v range over $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ and let σ denote $\{\widetilde{N}/\widetilde{\alpha}\}\{\widetilde{M}/\widetilde{\beta}\}$, i.e., the substitution that replaces each place-holder in the statement S by the corresponding argument of the proof. Suppose that the argument of the function $[|S|]_{\Gamma'}$ is a statement of the form $\mathsf{check}^\sharp(v_M, v_K) = v_N$. If the key v_K has type $\mathsf{VerKey}(T^*)$ in Γ' and T^* is untainted but public, then the function returns $\Gamma'[v_N : T^*], \mathsf{check}^\sharp(v_M, v_K) = v_N$. Since T^* is not tainted, the signing key is not known to the attacker. Because $\mathsf{check}(v_M\sigma, v_K\sigma) = v_N\sigma$ holds, the type system guarantees that $v_N\sigma$ has type T^* in Γ'. We can thus safely strengthen the type binding for v_N and include the formula $\mathsf{check}^\sharp(v_M, v_K) = v_N$ in the typing environment. Note that we are checking whether type T^* is *unconditionally untainted*, i.e., $\Gamma', \mathsf{false} \not\vdash T^* :: \mathsf{tnt}$. This implies that $\Gamma' \not\vdash T^* :: \mathsf{tnt}$ but our stronger formulation additionally ensures that the statement verification predicate is closed under environment extension, so that the standard weakening lemma holds.

If the key v_K has type $\mathsf{VerKey}(T^*)$ in Γ' and T^* is neither tainted *nor public*, then the function returns $\Gamma[v_N : T^*, y_{l+1} : T_{l+1}, \ldots, y_m : T_m], C, \mathsf{check}^\sharp(v_M, v_K) = v_N$. Because $\mathsf{check}(v_M\sigma, v_K\sigma) = v_N\sigma$ holds, the type system guarantees that the message $v_N\sigma$ is neither tainted nor public. Intuitively, since $v_N\sigma$ occurs in the zero-knowledge proof (either in the private or in the public component) and the attacker knows only public messages, the prover must be a well-typed participant. This guarantees that the type of the zero-knowledge proof is $\mathsf{ZKProof}_{n,m,S}(T)$ and we can thus safely refine the type binding for $y_{l+1}, \ldots, y_m$ and include the formulas $C, \mathsf{check}^\sharp(v_M, v_K) = v_N$ in the typing environment. Similar reasoning applies to decryptions and hashes. Other cases can be added to this function in order to deal with additional cryptographic primitives. For further details, we refer the interested reader to the long version [10].