Automated Synthesis of Privacy-Preserving Distributed Applications

Michael Backes  
Saarland University and MPI-SWS  
Germany  
backes@mpi-sws.org

Matteo Maffei  
Saarland University  
Germany  
maffei@cs.uni-saarland.de

Kim Pecina  
Saarland University  
Germany  
pecina@cs.uni-saarland.de

Abstract

We introduce a framework for the automated synthesis of security-sensitive distributed applications. The central idea is to provide the programmer with a high-level declarative language for specifying the system and the intended security properties, abstracting away from any cryptographic details. A compiler takes as input such high-level specifications and automatically produces the corresponding cryptographic implementations (i.e., cryptographic library, cryptographic protocols, and F# source code).

In this work, we focus on two important, and seemingly contradictory, security properties, namely, authorization and privacy. On the one hand, the access to sensitive resources should be granted only to authorized users; on the other hand, these users would like to share as little personal information as possible with third parties. These opposing goals make it challenging to enforce privacy-aware authorization policies in a distributed setting.

The high-level declarative language builds on Evidential DKAL, a logic for authorization policies of decentralized systems, which we extend to reason about privacy policies. Specifically, the traditional says modality from authorization logics is accompanied by existential quantification in order to express the secrecy of sensitive information. The cryptographic realization of privacy-aware authorization policies is obtained by a powerful combination of digital signatures and zero-knowledge proofs. This approach is general and can be seen as a privacy-enabling plugin for existing authorization languages and proof-carrying authorization architectures.

We proved that the implementations output by the compiler enforce the intended authorization policies and we conducted an experimental evaluation to demonstrate the feasibility of our approach.

1. Introduction

One of the central challenges in the development of distributed systems is the design of cryptographic protocols that meet the desired functional requirements and enforce the intended security properties. There is a common understanding that basic security properties such as secrecy and authentication can easily be enforced via encryption and digital signatures, respectively. Modern applications, however, exhibit more sophisticated and heterogeneous security requirements: for example, social networks, e-health systems, and reviewing systems must fulfill sophisticated access control, privacy, and anonymity constraints. Devising a cryptographic infrastructure for the enforcement of these properties is challenging and highly error-prone for security experts, and even prohibitive for regular programmers, which do not have the required background and expertise in cryptography. Currently, many popular applications rely on trusted third parties to collect and process sensitive information (e.g., conference reviewing systems like EasyChair or social networks like Facebook). The presence of trusted parties simplifies the system design but gives rise to a number of privacy concerns related to the deliberate or accidental disclosure of sensitive information. Other applications are fully decentralized but employ ad-hoc cryptographic protocols that are not always flawless and, due to their diversity, break any form of interoperability.

We believe that the design of security-sensitive distributed applications should be driven by rigorous, formally certified, and possibly automated, techniques, as opposed to best practices and informal guidelines. Specifically, developers should be given the possibility to specify the functional behavior of the system and the intended security properties using convenient, security-oriented, programming abstractions that conceal cryptographic details. A compiler should automatically turn user-provided, high-level system descriptions into executable cryptographic implementations. Ideally, these implementations should be open-ended and interoperable, i.e., it should be possible to extend the system with new functionalities and to share in-
formation among different, independently developed, applications.

This work introduces such a framework, focusing on two important, and seemingly contradictory, security properties, namely, authorization and privacy. Authorization is a key ingredient in virtually any security infrastructure. The fundamental idea is to let the resource provider define a security policy that constrains the operations on sensitive resources and to let a reference monitor filter access requests in a way that an operation is allowed only if the requester has sufficient permissions according to the security policy. For instance, let us consider a simple university management system in which, at the end of each semester, students are given a certificate of the form $\text{Uni says Stud}(id, program, grd)$, reporting their id, the program they are enrolled in, and their grades. Such a certificate is typically implemented as a digital signature of the form $\text{sig}(\text{Stud}(id, program, grd))_{\text{Uni}}$, issued by the university administration on (a bit string encoding of) the predicate $\text{Stud}(id, program, grd)$. The system under consideration is open-ended in that the student certificate may be employed in a number of different services: scholarship assignment, discounted museum entrances, access to university buildings (as in the Grey system [16] for device-enabled authorization), and so on. For instance, the policy for scholarship assignments may be of the form $\forall x, y, z. \text{Uni says Stud}(x, y, z) \land \text{average}(z) \geq X \Rightarrow \text{GetScholarship}(x)$, where $X$ is the minimum average grade that is required to get the scholarship. Systems and authorization policies of this form can be conveniently described in a variety of logic-based authorization languages, such as DCC [3, 2], Aura [42], PCML$_5$ [9], and Evidential DKAL [7].

The combination of authorization and privacy, however, is challenging, even more so in the context of open-ended applications, in which it is not known in advance how sensitive information is used by other applications (for instance, the university does not necessarily know all services that make usage of student credentials). Let us consider, for instance, the entrance to university buildings: students might not want their movements within the university to be tracked. Let us suppose that one of the authorization policies ruling the entrance to the computer science laboratory is of the form

$$\forall x, y. \text{Uni says Stud}(x, cs, y) \land x \text{ says Acc}(lab) \Rightarrow \text{OkAcc}(lab) \quad (1)$$

Ideally, students would like to prove to be enrolled in the computer science program, without disclosing their identity. From a logical point of view, we propose to capture privacy constraints via existential quantification, i.e., privacy-relevant values are hidden by existentially quantified variables. For instance, students can provide the following piece of information:

$$\exists x, y. \text{Uni says Stud}(x, cs, y) \land x \text{ says Acc}(lab) \quad (2)$$

This logical characterization of privacy and authorization is simple and elegant, but providing a faithful cryptographic evidence thereof turns out to be quite challenging: digital signatures do not offer any sort of privacy and standard cryptographic solutions like encryptions and MACs are not suitable for open-ended applications. This is the reason why existing authorization languages do not allow for such a usage of existential quantification and fall short of supporting privacy properties. We present a general and automated procedure to implement privacy-aware authorization policies by means of a powerful combination of digital signatures and zero-knowledge proofs. The idea is to use signatures to justify the validity of logical formulas, as previously shown, and zero-knowledge proofs of knowledge of such signatures to justify the validity of variants of these formulas – variants in which the sensitive arguments are existentially quantified. The unique properties of zero-knowledge proofs assure the verifier of the validity of these formulas, without revealing any sensitive data that the prover wishes to keep secret. For instance, the cryptographic realization of the formula (2) is a zero-knowledge proof of knowledge of two signatures of the form $\text{sig}(\text{Stud}(x, cs, y))_{\text{Uni}}$ and $\text{sig}(\text{Acc}(lab))_x$, for some student $x$ and grades $y$ that are not revealed to the verifier. This approach is flexible and, depending on what information is kept secret, can be used to express a variety of privacy properties, such as data secrecy and user anonymity. Furthermore, this approach is well suited for open-ended applications, since each party can prove any statement of which she knows a cryptographic evidence and, while doing so, autonomously hide any information considered sensitive for the specific application.

Our contributions. To summarize, this work presents:

- a generally applicable and efficient cryptographic implementation of privacy-aware authorization policies, which builds on automorphic signatures [5] and the Groth-Sahai zero-knowledge proof system [39] (cf. § 2);
- a high-level, declarative language for distributed systems, which extends Evidential DKAL [7] to deal with privacy properties (cf. § 3);

---

1 A zero-knowledge proof combines two seemingly contradictory properties. First, it is a proof of a statement that cannot be forged, i.e., 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 [38]. A non-interactive zero-knowledge proof is a zero-knowledge protocol consisting of one message sent by the prover to the verifier. A zero-knowledge proof of knowledge additionally ensures that the prover knows the witnesses to the given statement.
• a compiler that turns high-level descriptions into executable implementations, comprising cryptographic libraries, cryptographic protocols, and F# source code (cf. § 4);
• a correctness result, which ensures that the implementations output by the compiler enforce the authorization policies specified by the user (cf. § 5);
• two case studies, consisting of a distributed reviewing system and a distributed social network, which demonstrate the possibility to specify relatively complex decentralized systems in a simple and elegant manner, without requiring any cryptographic expertise on the part of the users (cf. § 6);
• and an experimental evaluation, which demonstrates the feasibility of our approach (cf. § 7).

The details of the cryptographic realization and further case studies are located in the appendix.

Related Work. The seminal works by Abadi et al. [43, 4] on access control in distributed systems paved the way for the development of a number of authorization logics and languages [31, 18, 42, 37, 28, 9]. In the literature it is well-known that logical formulas based on a says modality can be implemented in a distributed setting via digital signatures, but the problem of ensuring the privacy of data employed in authorization proofs has not been tackled thus far. A noticeable exception is AuraConf [54], a confidentiality extension of the Aura [42] programming language based on public-key encryption and a monadic constructor. In AuraConf, the programmer has to specify the intended recipient of each data and the compiler is in charge of encrypting such data with the appropriate public key. In our approach, the programmer does not need to know in advance all the intended usages and recipients of the digital signatures issued in the protocol run, which is crucial to deal with open-ended systems. In fact, principals can use received signatures to construct arbitrary authorization proofs, using zero-knowledge proofs to selectively hide sensitive data.

Digital signatures and zero-knowledge schemes proved to be salient tools for achieving fine-grained anonymity properties in a number of applications, such as trusted computing [25], digital credentials [19], trust protocols [46, 10], and social networks [12]. The relationship between privacy-preserving cryptographic constructions and authorization logics, however, has been investigated only partially and in specialized settings. For instance, Li et al. [44] developed a framework for Automated Trust Negotiation using anonymous credentials, which is tailored to RT, a family of Trust-management languages [45]. Friksen et al. used a combination of hidden credentials, homomorphic encryption, and oblivious transfer to enforce access control policies while keeping both policies and credentials secret [36].

Our framework is not tailored to a specific language and can be seen as a generally applicable privacy-preserving plugin for authorization logics: in Appendix A, we report on a privacy-oriented extension of the Proof Carrying Authorization framework [8, 17] and we envision the usage of our framework within several other authorization languages, such as Aura [42], PCML∗ [9], F∗ [52], and SecurePal [18]. Some preliminary ideas on privacy-aware proof-carrying authorization were anticipated in a position paper by Maffei and Pecina [47].

Backes et al. have recently presented G2C [13], a goal-driven specification language for distributed applications. This language supports secrecy, access control, and anonymity, which are enforced by means of broadcast encryption and group signatures. Similarly to our approach, G2C conceals cryptographic details and lets a compiler generate the cryptographic implementation. G2C, however, does not support open-ended applications (i.e., it is not possible to extend the system in order to provide new functionalities without generating the whole protocol from scratch). Furthermore, the G2C compiler yields cryptographic protocols as opposed to executable implementations and it does not provide security by construction guarantees.

The proof of correctness for our compiler builds on the type theory for zero-knowledge proofs by Backes et al. [11] and, in particular, on their compiler from zero-knowledge statements to symbolic cryptographic libraries. These libraries model the ideal behavior of cryptographic schemes using standard language constructs and are thus suitable for verification but cannot be used for deployment. Our compiler, instead, converts a DKAL derivation of the intended protocol run into an executable implementation, which includes concrete, executable cryptographic libraries. Devising an efficient, yet expressive and flexible cryptographic realization of privacy-aware authorization proofs is one of the challenges faced in this work.

Recently, Meiklejohn et al. [48] and Almeida et al. [6] have independently presented two compilers for zero-knowledge proofs, which take as input a specification of the cryptographic statement to be proved. The cryptographic realization of such specifications is based on Σ-protocols [29]. Our compiler builds on the Groth-Sahai zero-knowledge proof scheme [39], which is based on pairing-based cryptography and is, in general, more efficient and more expressive. Furthermore, our compiler provides a higher level of abstraction, since it takes as input logical formulas as opposed to cryptographic statements.

2. Privacy-aware Evidential Authorization

This section gives an intuitive overview of our framework (§ 2.1), establishes the binding between authorization formulas and cryptographic messages (§ 2.2), de-
Table 1: Cryptographic evidence of authorization formulas.

<table>
<thead>
<tr>
<th>$ap$</th>
<th>:=</th>
<th>$\text{ver}_\text{sig}(u_s, u_A, F) \mid E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>:=</td>
<td>$ap \mid S_1 \land S_2 \mid S_1 \lor S_2 \mid \exists x. S$</td>
</tr>
</tbody>
</table>

(atomic predicates)

(formulas)

$$[M] = \begin{cases} \text{vk}_A \text{ says } F & \text{if } \text{ver}_\text{sig}(M, \text{vk}_A, F) \\ [S]_{\text{vk}} & \text{if } \text{ver}_\text{vk}(M, S) \end{cases}$$

$$[S]_{\text{vk}} = \begin{cases} u_k \text{ says } F & \text{if } S = \text{ver}_\text{sig}(u_s, u_k, F) \\ [S_1]_{\text{vk}} \land [S_2]_{\text{vk}} & \text{if } S = S_1 \land S_2 \\ [S_1]_{\text{vk}} \lor [S_2]_{\text{vk}} & \text{if } S = S_1 \lor S_2 \\ \exists x. [S']_{\text{vk}} & \text{if } S = \exists x. S' \end{cases}$$

writes the class of statements that can be proved in zero-knowledge (§2.3), and characterizes which of them provide meaningful security guarantees (§2.4).

2.1. Overview

Let us consider the example discussed in §1. The student’s goal is to provide evidence of the validity of the predicate $\text{OkAcc}(lab)$. The validity of this predicate is ruled by the authorization policy (1) and the minimal amount of information the student has to reveal is captured by formula (2). The key insight is that the student does not need to reveal her identity nor her grades, since they do not occur in the predicate $\text{OkAcc}(lab)$ and can thus be existentially quantified. The student has two signatures at her disposal, namely, $\text{sig}(\text{Stud}(\text{vk}_\text{id}, cs, \text{grd}))_{\text{uni}}$ and $\text{sig}(\text{Acc}(lab))_{\text{id}}$. Our idea is to let the student create a proof of the following statement:

$$\exists x_1, x_2, x_{id}, x_{grd}, \text{ver}_\text{sig}(x_1, \text{vk}_{\text{uni}}, \text{Stud}(x_{id}, cs, x_{grd})) \land \text{ver}_\text{sig}(x_2, x_{id}, \text{Acc}(lab)) \tag{3}$$

This statement says that there exist two signatures $x_1$ and $x_2$, a verification key $x_{id}$ (which constitutes the student’s id), and grades $x_{grd}$ such that (i) $x_1$ is a signature on the predicate $\text{Stud}(x_{id}, cs, x_{grd})$ that can be successfully verified with the university administration’s verification key $\text{vk}_{\text{uni}}$ and (ii) $x_2$ is a signature on the predicate $\text{Acc}(lab)$ that can be successfully verified with the student’s verification key $x_{id}$. Since we use zero-knowledge proofs of knowledge, the above statement actually implies that the prover knows the signatures and the verification key. Upon reception and verification of this proof, the verifier can safely derive the logical formula (2), which in turn allows $\text{OkAcc}(lab)$ to be derived as previously described.

2.2. Mapping Cryptographic Messages to Logical Formulas

Here and throughout this paper, we let $M$ range over cryptographic messages (digital signatures and zero-knowledge proofs), $a, b, m$ over names (i.e., bit strings), $x, y, z$ over variables, $u$ over names and variables, $F$ over authorization formulas, and $E$ over quadratic equations in $\mathbb{Z}_n$ [39], which are used to express arithmetic properties (e.g., average$(z) \geq X$ from §1). The predicate $\text{ver}_\text{sig}(\text{sig}, \text{vk}, m)$ denotes the successful verification of signature $\text{sig}$ on message $m$ with verification key $\text{vk}$, and the predicate $\text{ver}_\text{zk}(\text{ZK}, S)$ denotes the successful verification of the zero-knowledge proof $\text{ZK}$ for statement $S$.

Predicates of the form $\text{ver}_\text{sig}(u_s, u_k, F)$ and quadratic equations in $\mathbb{Z}_n$ form the class of atomic predicates, which are ranged over by $ap$ (cf. Table 1). Zero-knowledge statements, ranged over by $S$, are built on atomic predicates using conjunction, disjunction, and existential quantification.

The function $\llbracket \cdot \rrbracket : M \mapsto F$ establishes the logical interpretation of cryptographic messages. The logical interpretation of a signature on (the bit-string encoding of) the predicate $F$, verifiable with key $\text{vk}_A$, is $\text{vk}_A \text{ says } F$. The logical interpretation of a zero-knowledge proof is similarly defined by induction on the structure of the statement.

2.3. Construction of Zero-Knowledge Proofs

The deduction system in Table 2 characterizes what kind of statements a principal can prove in zero-knowledge starting from a database $\Gamma$ of signatures and zero-knowledge proofs at her disposal. Intuitively, these statements may regard properties of digital signatures, as formalized by the judgment $\Gamma \vdash_S S$, or be obtained by combining the statements of existing zero-knowledge proofs, as formalized by the judgment $\Gamma \vdash_{\text{ZK}} S$.

The statements proved by the judgment $\Gamma \vdash_S S$ are obtained by combining statements of the form $\text{ver}_\text{sig}(s, \text{vk}, F)$ (I-S-VER) in conjunctive (I-S-\&-) and disjunctive (I-S-\lor-) form. Notice that logical conjunctions reveal more information than each of the two individual conjuncts, while logical disjunctions reveal less information in that it is not possible to determine which of the two disjuncts holds true.

The statements proved by the judgment $\Gamma \vdash_{\text{ZK}} S$ build on the aforementioned statements (I-ZK-S) and on the

---

2. Our framework is independent of the underlying authorization logic: we just assume the presence of conjunction, disjunction, and existential quantification operators.
statements of the zero-knowledge proofs in $\Gamma$ (I-ZK-VER). Such statements can be refined by existential quantification (I-ZK-$\exists$) and conjunction elimination (E-ZK-$\land\!$-1 and E-ZK-$\land\!$-2) to hide information. It is also possible to combine the statements of two zero-knowledge proofs in conjunctive form (I-ZK-$\land\!$).

Notice that logical disjunctions are introduced by judgment $\Gamma \vdash S$ and not by judgment $\Gamma \vdash ZK S$, since we are not aware of efficient cryptographic constructions that allow the prover to create a zero-knowledge proof of $S \lor S'$ from a zero-knowledge proof of $S$.

**Example 1.** Let us consider again the example from § 2.1. We let $M_1 = \text{sig}((\text{Stud}(\text{vk}_{id}, cs, \text{grd}))_{\text{Uni}})$, $M_2 = \text{sig}((\text{Acc}(\text{lab}))_{id})$, $V_1 = \text{ver}_{\text{sig}}(M_1, \text{vk}_{\text{Uni}}, \text{Stud}(\text{vk}_{id}, cs, \text{grd}))$, $V_2 = \text{ver}_{\text{sig}}(M_2, \text{vk}_{id}, \text{Acc}(\text{lab}))$, and $\Gamma = M_1, M_2$. The creation of the zero-knowledge proof of statement (3) is specified in Table 3.

**Example 2.** It is worth to mention that I-ZK-$\exists$ may also be used to hide equality relations among secret values: for instance, one can derive $\exists x, y$. A says $p(x, y)$ from $\exists x. A \models p(x, x)$.

### 2.4. Validity of Zero-Knowledge Statements

It is interesting to observe that not all zero-knowledge statements are meaningful. For instance, suppose that a principal receives a zero-knowledge proof of the following statement:

$$\exists y, y_1. \text{ver}_{\text{sig}}(y, y_1, \text{Stud}(\text{vk}_{id}, cs, \text{grd}))$$  (4)

We would be tempted to let this principal entail $\exists y_1. y_1 \models \text{Stud}(\text{vk}_{id}, cs, \text{grd})$. This zero-knowledge proof, however, does not reveal the identity of the person issuing the signature, nor is there any evidence that this person is a principal of the system. In fact, this zero-knowledge proof might have been constructed by an attacker, using a fresh key-pair and, therefore, the formula $\exists y_1. y_1 \models \text{Stud}(\text{vk}_{id}, cs, \text{grd})$ is not necessarily entailed by the formulas proved by the principals of the system. Notice that we assume that the principals of the system are honest, i.e., they issue signatures to witness the validity of the corresponding logical predicates. We cannot, of course, assume the same for the attacker.

We stipulate that principals only sign verification keys that belong to principals of the system (as opposed to attacker’s keys). We call these keys trustworthy. Checking whether a verification key that occurs in a zero-knowledge proof is trustworthy is subtle. The idea is that a key is considered trustworthy if either it is revealed by the proof and known to belong to a principal of the system, or, recursively, it is endorsed by a trustworthy key. For instance, the statement (4) does not guarantee that the existentially quantified verification key $y_1$ is trustworthy. Conversely, the verification key $x_{id}$ that is existentially quantified in the statement (3) is signed by $\text{Uni}$ and, therefore, is trustworthy. Hence, this statement justifies formula (2).

In other words, a statement is well-formed if it ensures that all verification keys are trustworthy. Despite the simplicity of this intuition, the formal definition has to take into account a number of complications, including the presence of logical disjunctions in the statement. For instance, the statement $\exists y, y_1. \text{ver}_{\text{sig}}(y, y_1, F) \lor \text{ver}_{\text{sig}}(y, \text{vk}_{\text{Uni}}, F)$ is not well-formed, since we do not know which of the two disjuncts holds true. The idea is to transform a statement in conjunctive form and then to check that all keys in each sequence of conjunctions are registered. We formalize the notion of trustworthiness for keys below. Here and throughout this paper, we write $\ell$ to denote the sequence $u_1, \ldots, u_{\ell}$.

---

<table>
<thead>
<tr>
<th>I-ZK-$\land!$</th>
<th>$\Gamma \vdash ZK S$</th>
<th>$\exists x, y. S, S'$</th>
<th>$\exists x, S, S'$</th>
<th>$\exists x, S, S'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-ZK-$\land!$</td>
<td>$\Gamma \vdash ZK S$</td>
<td>$\exists x. S, S'$</td>
<td>$\exists x. S, S'$</td>
<td>$\exists x. S, S'$</td>
</tr>
</tbody>
</table>

---

3We assume a public-key infrastructure that binds public keys to their owner in a publicly-verifiable manner. Such a PKI may be centralized (e.g., Verisign) or decentralized (e.g., Webs of Trust). In the example from § 2.1, the university serves as a (centralized) PKI.
### Table 3 Deduction of the zero-knowledge proof for Example 1.

| $T_{ZK}$ := | $M_1 \in \Gamma \quad V_1$ | $1-S-VER$ | $\Gamma \vdash_S V_1 \quad 1-ZK-S$ | $M_2 \in \Gamma \quad V_2$ | $1-S-VER$ | $\Gamma \vdash_S V_2 \quad 1-ZK-S$ |
| | | | | | | |
| $\Gamma \vdash_{ZK} V_1 \land V_2$ | $1-ZK-\land$ |
| $\Gamma \vdash_{ZK} \exists x_1, x_2, x_{id}, x_{grd}, \text{ver}_\text{sig}(x_1, vk_{/m_1}, \text{Stud}(x_{id}, c_8, x_{grd})) \land \text{ver}_\text{sig}(x_2, x_{id}, \text{Acc}(lab))$ | $1-ZK-\exists (x4)$ |

### Table 4 Verification rules.

| $\text{VER-Sig}$ | $M \in \Gamma \quad \text{ver}_\text{sig}(M, vk_A, F)$ |
| | $\Gamma \vdash vk_A \text{ says } F$ |
| $\text{VER-ZK}$ | $M \in \Gamma \quad \text{ver}_k(M, S) \quad S \text{ well-formed}$ |
| | $\Gamma \vdash S_{\text{st}}$ |

for some $\ell$.

**Definition 1** (Trustworthiness of keys). A key $u$ is trustworthy in a monomial $M = \bigwedge_{i=1}^n ap_i$ iff one of the following conditions holds:

- $u = vk$ is registered
- there exists $ap_j = \text{ver}_\text{sig}(u_s, uk_k, F)$ such that $u$ is a variable occurring free in $F$ and $uk_k$ is trustworthy in $M$

**Definition 2** (Disjunctive form). We say a statement $S$ is in disjunctive form iff $S = \exists x. \bigvee_{i=1}^m M_i$, where $M_i = \bigwedge_{j=1}^n ap_j$.

It is clear that each statement can be rewritten in disjunctive form. In the following, we assume a disjunctive normal form for each statement $S$, written as $\text{dnf}(S)$.

**Definition 3** (Well-formedness of statements). A monomial $M = \bigwedge_{i=1}^m ap_i$ is well-formed iff for every $ap_i = \text{ver}_\text{sig}(u_s, uk_k, F)$, $uk_k$ is trustworthy in $M$.

A statement $S$ such that $\text{dnf}(S) = \exists x. \bigvee_{i=1}^m M_i$ is well-formed if each $M_i$ is well-formed.

We are now ready to characterize the logical formulas that are justified by the digital signatures and zero-knowledge proofs in $\Gamma$, as formalized in Table 4. The rules are self-explanatory: we just point out that the statements of zero-knowledge proofs are required to be well-formed.

---

3. **Specification Language: Privacy-aware Evidential DKAL**

The Distributed Knowledge Authorization Language (DKAL) [40, 41] is a logic-based language for modeling and analyzing decentralized policies. A distinctive feature of DKAL is the possibility to explicitly describe the exchange of information among principals. Recently, Blass et al. introduced Evidential DKAL [7], an extension of DKAL in which the formulas exchanged by principals are justified by digital signatures, which allows for more expressive logical derivations. In this section, we extend Evidential DKAL with existential quantification, in order to express privacy constraints, and with zero-knowledge proofs, in order to justify logical formulas in which sensitive values are existentially quantified.

3.1. **Overview of Evidential DKAL**

Authorization languages such as PCA or SecPAL [18] often rely on fragments of first-order or higher-order logic to describe and enforce authorization policies. The logic underlying DKAL, called infon logic, is fundamentally different: instead of dealing with the validity of statements, this logic focuses on the notion of information. A statement represents a piece of information, as opposed to a truth value, that a specific principal has obtained, and access control is decided by deriving certain information, as opposed to proving a formula valid.

Table 5 reports some fundamental rules of Evidential DKAL. Here and throughout this paper, we let $\Gamma$ denote a set of pieces of information, i.e., logical formulas and cryptographic messages. $\text{ENSUE}$ says that if $A$ knows $\Gamma$, then $A$ knows also the information derivable from $\Gamma$. The initial knowledge of principal $A$ is given in terms of knowledge assertions of the form $A : F$, which can be seen as axioms in the system. Rule $P-A$ says that given the knowledge assertion $A : F$, $A$ knows $F$. For easing the presentation, we deviate from the original presentation [7], which does not precisely specify how principals acquire the knowledge of signatures. We introduce rule $P-S$ to express that given the knowledge assertion $A : F$, $A$ can produce a

---

4The disjunctive normal form can be obtained, for instance, by lexicographical order.
Table 5 Selection of rules from Evidential DKAL.

<table>
<thead>
<tr>
<th>ENSUE</th>
<th>P-A</th>
<th>P-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$ knows $\Gamma$</td>
<td>$\Gamma \vdash F$</td>
<td>$A : \nu_k A$ says $F$</td>
</tr>
<tr>
<td>$M \vdash \nu_k^\theta M$</td>
<td>$A : F$</td>
<td>$\text{ver}_{\mu}(M, \nu_k A, F)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COMM-J</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if $F_B$ then $B$ sends $F$ to $p$</td>
<td>if $F_A$ then $A$ receives $F'$ from $q$</td>
<td>$B$ knows $F_B \eta$ $B$ knows $M$ $A$ knows $F_A \theta$ $p \eta = A$ $q \theta = B$ $[M] = F$ $F' \theta = F$</td>
</tr>
<tr>
<td>$A$ knows $F$</td>
<td>$A$ knows $F$</td>
<td>$A$ knows $M$</td>
</tr>
</tbody>
</table>

The communication rule COMM-J describes the exchange of cryptographic messages that justify (i.e., provide evidence of) a certain statement. This rule is used to synchronize a communication assertion of the form if $F_B$ then $B$ sends $F$ to $p$ with a communication assertion of the form if $F_A$ then $A$ receives $F'$ from $q$. In order to fire this rule, $B$ must know $F_B \eta$ and $A$ must know $F_A \theta$, for some substitutions $\eta$ and $\theta$ mapping variables to messages in the knowledge of $A$ and $B$, respectively. Furthermore, conditional guards may be omitted, obtaining communication assertions of the form $A$ sends $F$ to $B$ and $A$ receives $F$ from $B$. In order to fire COMM-J, we additionally require the sender and the receiver to coincide on both sides, i.e., $p \eta = A$ and $q \theta = B$. Finally, $B$ must know a cryptographic evidence $M$ for the statement $F$ sent to $A$ and the statement $F'$ expected from $A$ has to be unifiable with $F$ by substitution $\theta$. Once COMM-J is fired, $A$ knows the cryptographic evidence $M$ of $F$.

Example 3. Let us consider the example from § 2.1. The generation of student certificates is modeled by knowledge assertions of the form

$$Uni : \nu_k Uni \text{ says } Stud(\nu_k id, cs, grd)$$

(5)

One can derive $Uni$ knows $Stud(\nu_k id, cs, grd)$ by P-A and $Uni$ knows $M$ by P-S, where $M$ is a signature issued by $Uni$ on $Stud(\nu_k id, cs, grd)$.

The intended communication protocols is modeled by the DKAL derivation below (for the sake of readability, we omit self-generated signature $M$ on $F$.

Example 4. The issue of student certificates can be specified via the following assertions:

$$Uni \text{ sends } \nu_k Uni \text{ says } Stud(\nu_k id, cs, grd) \text{ to } id$$

(6)

$$id \text{ receives } \nu_k Uni \text{ says } Stud(\nu_k id, cs, y) \text{ from } Uni$$

(7)

We finally derive $id$ knows $\nu_k Uni$ says $Stud(\nu_k id, cs, grd)$ as follows:

Table 7 Core Rules of Privacy-aware Evidential DKAL.

<table>
<thead>
<tr>
<th>P-ZK</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \text{ knows } \Gamma$</td>
<td>$\Gamma \vdash S$</td>
<td>$A : \nu_k S$ says $F$</td>
</tr>
<tr>
<td>$\text{ver}_{\mu}(M, S)$</td>
<td></td>
<td>$\text{ver}_{\mu}(M, S)$</td>
</tr>
<tr>
<td>$A$ knows $M$</td>
<td>$A$ knows $M$</td>
<td>$A$ knows $M$</td>
</tr>
</tbody>
</table>

some trivial hypotheses and use the abbreviations defined in Example 1):

$$T_C := \frac{(6) \quad (7) \quad V_1}{\nu_k Uni \text{ knows } M_1 \quad P-S \quad \text{COMM-J}}$$

3.2. Privacy-Aware Evidential DKAL

Evidential DKAL does not feature any mechanism to enforce the privacy of sensitive information. We solve this problem by integrating our zero-knowledge deduction system and by refining the syntax of statements in order to support existential quantification. Further, for each principal $A$, we stipulate the existence of a set of anonymous identities that cannot be associated with $A$ (written as $A^k$, where $k$ is an index). These identities can be sent to other parties (within statements) and be used later on as receiver identifiers in COMM-J. Anonymous identifiers are cryptographically implemented by fresh public keys and can be realized at the network layer via rendezvous-points [32].

Formally, we extend evidential DKAL with the rules from Table 2, Table 4, Table 6, and Table 7. The PP-K rules identify principals with their fresh identities. P-ZK bridges between DKAL and the zero-knowledge deduction system,
bringing the zero-knowledge proofs derived by $A$ into $A$’s knowledge. COMM-A is a variant of COMM-J that is introduced for modeling anonymous communication: In this variant, the identity of the sender is replaced by the special symbol ‘?’ and is not known to the receiver.

**Example 5.** We describe the anonymous entrance to the laboratory by means of the following assertions, where $AS$ denotes the access control system:

\[
\text{id} : \text{vk}_\text{id} \text{ says } \text{Acc(}\text{lab}\text{)} \quad (8) \quad ? \text{ sends (2) to } AS \quad (9)
\]

$AS$ receives (2) from ? \quad (10)

The communication protocol is modeled by the derivation tree displayed in Table 8.

The student creates the zero-knowledge proof of statement (3) using the signature $M_1$ received from the university and the signature $M_2$ from the knowledge assertion (8) as witnesses. Using ENSUE, we can eventually derive $AS$ knows $\exists x, y. \text{Uni says Stud(x, cs, y)} \land x \text{ says Acc(}\text{lab}\text{)}$.

## 4. Compiler

We developed a compiler $C[\cdot] : T_{dkal} \rightarrow C_F$ that takes as input the Privacy-aware Evidential DKAL derivation capturing the intended system behavior and produces the $F#$ executable code for each of the principals in the system. In this section, we overview the algorithm and state the soundness results.

Intuitively, the compiler builds on a translation function $T : R_{dkal} \rightarrow C_F$ from DKAL rules to $F#$ code. Each rule of the zero-knowledge deduction system (cf. Table 2) is translated into a sequence of calls to functions of the zero-knowledge library. Communication rules (namely, COMM-J and COMM-A) are translated by extending the current code of the sender and the receiver with the output and input of the cryptographic message, respectively. In order to protect the secrecy of exchanged information, the communication is always encrypted with the public key of the receiver. The code of the receiver is further extended to verify the signature or the zero-knowledge proof received from the network.

The compiler produces the code of each principal by scanning the DKAL derivation tree down (i.e., from the outermost hypotheses until the thesis) and, for each rule $R$, by appending the code $T(R)$ to the current code for the corresponding principal.

**Example 6.** Let us describe the compiler in more detail by illustrating the code produced by translating the $T_A$ derivation tree from Example 5 (cf. Table 9). The compiler generates three functions, one for each role (i.e., university, student, and access control system). These functions are meant to be integrated in the code of the respective application and are consequently parameterized by a number of values, including the cryptographic keys of the running principal, the ones of the intended communication partners, network addresses, and so on. On the right-hand side of each line of code, we indicate the rule application (i.e., the rule name and the derivation tree) that has been processed by the compiler. For the moment, we invite the reader to ignore the annotations between square brackets, which play a role in the formalization of the correctness result (cf. § 5) but do not have any computational significance.

The $T_A$ tree is scanned top-down. The first rule that is processed is the application of P-S in $T_C$, which introduces the signature issued by the university on the predicate $\text{Stud(vk}_\text{id}, cs, grd)$ into the derivation tree: this rule is translated into a call to the signature creation function in the university’s code. The next rule is COMM-J from $T_C$, which describes the transmission of the student’s certificate: the signature is first encrypted with the recipient’s encryption key and then sent to the address of the recipient, who receives the message, decrypts the ciphertext, and finally verifies the signature. The application of P-S in $T_A$ leads to a call to the signature creation function in the student’s code. The rules in $T_{ZK}$ model the creation of the zero-knowledge proof and are translated into a sequence of calls to the corresponding functions in the zero-knowledge library. Finally, the application of COMM-A in $T_A$ models the transmission of the zero-knowledge proof and is translated similarly to COMM-J.

## 5. Formal Verification

We verify the correctness of our compiler using the security type system for $F#$ developed by Bengtson et al. [20] and recently extended by Backes et al. in order to support zero-knowledge proofs [11]. The semantics of $F#$ is formalized using RCF, a concurrent lambda calculus that has successfully been used to encode and verify the security of $F#$ protocol implementations [21]. Authorization policies are expressed in code by means of annotations: *assumptions* introduce new hypotheses (i.e., formulas that are assumed to hold) and *assertions*\(^6\) declare formulas that are expected to logically follow from the previously introduced hypotheses.

**Definition 4** (Safety [20]). A program $P$ is safe if and only if, in all executions of $P$, all assertions are entailed by the current assumptions.

In general, we are interested in the safety of programs that are executed in parallel with the attacker. The attacker

\(^6\)RCF assertions are not to be confused with DKAL assertions.
Table 8 Derivation tree modeling the communication protocol from Example 5

\[
T_A := \frac{\text{id knows } M_Z}{\text{organizes } M_Z} \quad \frac{\text{id knows } M_1, M_2}{\text{organizes } M_1, M_2} \quad \frac{\text{id knows } M_2}{\text{organizes } M_2}
\]

\[
\begin{align*}
T_C &: \frac{\text{V}_2}{\text{organizes } M_2} \\
\text{MSG-}U &\quad \frac{\text{id knows } M_Z, (3)}{\text{organizes } M_Z} \\
T_Z &\quad \frac{\text{id knows } M_Z}{\text{organizes } M_Z} \\
\end{align*}
\]

Table 9 Source code for the running example

\[
\begin{array}{l}
\text{Uni}(sk_{Uni}, ek_{id}, vk_{Uni}, vk_{id}, ad, cs, grd) \triangleq \\
\quad [\text{assume } F_{Uni}] \\
\quad \text{let } s = \text{sig } sk_{Uni} \\
\quad \text{let } msg = \text{enc } ek_{id} \text{ in } \text{pickle } \text{Stud}(vk_{id}, cs, \text{grd}) \text{ in } (\text{P-S}) \\
\quad \text{let } c = \text{connect } ad \text{ in } \text{send } c \text{ msg } (\text{COMM-J}) \\
\text{AccSys}(\text{dk}, vk_{Uni}, vk_{id}, ad, cs, lab) \triangleq \\
\quad [\text{assume } F_{id}] \\
\quad \text{let } s = \text{sig } sk_{id} \text{ in } \text{pickle } \text{Acc}(lab) \text{ in } (\text{P}) \\
\quad \text{let } msg = \text{enc } ek_{id} \text{ in } \text{V}_{sk} \text{ msg in } (\text{P-S}) \\
\quad \text{let } x = \text{unpickle}(\text{ver}(3) x_{sk}) \text{ in } \text{assert } (2) ; (\text{COMM-A}) \\
\text{where } F_{Uni} = \text{vk}_{Uni} \text{ says Stud}(vk_{id}, cs, \text{grd}) \\
\quad F_{id} = \text{vk}_{id} \text{ says AccSys}(vk_{id}, cs, lab) \triangleq \\
\quad \text{let } c = \text{listen } ad \text{ in } \text{let } msg = \text{recv } c \text{ in } \text{match } x \text{ with } \text{Stud}(vk_{id}, cs, \text{grd}) \Rightarrow \\
\quad [\text{assert } F_{Uni}] ; (\text{P}) \\
\quad \text{let } s' = \text{sig } sk_{id} \text{ in } \text{match } x \text{ with } \text{Stud}(vk_{id}, cs, \text{grd}) \Rightarrow \\
\quad [\text{assert } F_{id}] ; (\text{P-S}) \\
\text{Stud}(sk_{id}, dk_{id}, ek_{AS}, vk_{Uni}, vk_{id}, ad, ad_{AS}, cs, lab) \triangleq \\
\quad \text{let } c = \text{listen } ad \text{ in } \text{let } msg = \text{recv } c \text{ in } \text{match } x \text{ with } \text{Stud}(vk_{id}, cs, \text{grd}) \Rightarrow \\
\quad [\text{assert } F_{Uni}] ; (\text{P}) \\
\quad \text{let } s' = \text{sig } sk_{id} \text{ in } \text{match } x \text{ with } \text{Stud}(vk_{id}, cs, \text{grd}) \Rightarrow \\
\quad [\text{assert } F_{id}] ; (\text{P-S}) \\
\text{where } \text{Stud}(sk_{id}, dk_{id}, ek_{AS}, vk_{Uni}, vk_{id}, ad, ad_{AS}, cs, lab) \triangleq \\
\quad \text{let } c = \text{listen } ad \text{ in } \text{let } msg = \text{recv } c \text{ in } \text{match } x \text{ with } \text{Stud}(vk_{id}, cs, \text{grd}) \Rightarrow \\
\quad [\text{assert } F_{Uni}] ; (\text{P}) \\
\quad \text{let } s' = \text{sig } sk_{id} \text{ in } \text{match } x \text{ with } \text{Stud}(vk_{id}, cs, \text{grd}) \Rightarrow \\
\quad [\text{assert } F_{id}] ; (\text{P-S})
\end{array}
\]

is modeled as some arbitrary (untyped) expression that has access to the functions exported by the program. The idea is to let the attacker create arbitrary parallel instances of the protocol roles and to let him send and receive messages on the network channels. If the attacker cannot break the safety property, then the program is robustly safe. In the following, we write \( P Q \) to denote the application of program \( P \) to program \( Q \) (i.e., \( P \) can access the functions exported by \( Q \)).

**Definition 5** (Formal threat model [20]). A program \( A \) is an attacker if and only if \( A \) contains no occurrence of assert and each type annotation within \( A \) is unit.

A program \( P \) is robustly safe if and only if the application \( A P \) is safe for all attackers \( A \).

The compiler automatically generates assumptions and assertions that capture the logical formulas that are introduced and derived in the DKAL derivation, respectively. In particular, the translation of a knowledge assertion of the form \( A : F \) (rules P-A and P-S) introduces an assumption of the form assert \( F \) into \( A \)'s code. The translation of any other rule with a thesis of the form \( A \) knows \( F \) (e.g., EN-SUE) or \( A \) knows \( M \) (e.g., COMM-J) introduces an assertion of the form assert \( F \) in \( A \)'s code, where \( F \) is the logical interpretation of \( M \) (i.e., \([M] = F\)), thereby ensuring that the formula \( F \) is indeed derivable at run-time.

**Example 7.** Let us consider the annotations in Table 9. The P-S rule in \( T_C \) introduces the formula \( \text{vk}_{Uni} \) says \( \text{Stud}(\text{vk}_{id}, cs, \text{grd}) \) in the DKAL derivation and therefore the compiler inserts assume \( \text{vk}_{Uni} \) says \( \text{Stud}(\text{vk}_{id}, cs, \text{grd}) \) into the university's code. The COMM-J rule in \( T_C \) models the transmission of the information \( \text{vk}_{Uni} \) says \( \text{Stud}(\text{vk}_{id}, cs, \text{grd}) \) from the university to the student. Consequently, the compiler inserts assert \( \text{vk}_{Uni} \) says \( \text{Stud}(\text{vk}_{id}, cs, \text{grd}) \) into the student's code after the signature verification, which ensures that \( \text{vk}_{Uni} \) says \( \text{Stud}(\text{vk}_{id}, cs, \text{grd}) \) is indeed derivable from the current assumptions at run-time.

For verification purposes, \( F \# \) programs are linked to a symbolic cryptographic library, comprising functions
for public-key encryption, digital signatures, and zero-knowledge proofs. This library models the ideal behavior of cryptographic primitives using standard language constructs. We programmed the symbolic zero-knowledge library making usage of the tool developed by Backes et al. [11], which takes as input a zero-knowledge statement and produces methods for the construction and verification of the corresponding zero-knowledge proof. For protocols based on digital signatures and public-key encryption, safety carries over to programs linked to concrete cryptographic libraries [14, 35]. Preliminary results for the computational soundness of symbolic abstractions of zero-knowledge proofs have been proved in [15]. The soundness of the type system ensures that well-typed programs are safe.

**Theorem 1 (Safety by typing [20]).** If $\emptyset \vdash P : U$, then $P$ is safe.

For verifying the security of the F# code that is generated by our compiler, it is in principle enough to run the type-checker. This *compilation validation* approach has the advantage of smoothly supporting optimizations of the result of the compilation and of the compiler itself. Nevertheless, we additionally prove that all programs output by the compiler are well-typed. This *security by construction* approach has the advantage of making the type-checking of the result of the compilation unnecessary, unless the code is modified, and gives stronger guarantees about the correctness of the compiler.

**Theorem 2 (Soundness of the compilation).** For all logical derivations $T \in T_{dkal}$, there exists a type $U$ such that $\emptyset \vdash C[T] : U$.

Our main theorem states that well-typed programs that use the cryptographic library and the functions produced by the compiler are safe. In the following, we write $I_C[T]$ to denote the typed interface of the library produced by the compiler and $P \cdot C[T]$ to denote the program obtained by linking $P$ to such a library.

**Theorem 3 (Robust safety of the compilation).** If $I_C[T] \vdash P : U$, then $P \cdot C[T]$ is robustly safe.

6. Case studies

In this section, we utilize our framework to specify a distributed reviewing system and a distributed social network. Given these logical specifications, the compiler automatically produces the corresponding cryptographic implementations. The goal is to demonstrate the possibility to specify relatively complex decentralized systems in a simple and elegant manner, without requiring any cryptographic expertise on the part of the users.

### 6.1. Distributed Reviewing System

Current reviewing systems (e.g., Easychair and EDAS) are designed around a trusted party that serves as custodian of a huge amount of data about the submission and reviewing behavior of thousands of researchers, aggregated across multiple conferences. The deliberate or accidental disclosure of such information is a recognized privacy problem [50]. In this section, we design a decentralized reviewing system that offers strong privacy guarantees.

The first functionality that a reviewing system should offer is paper assignment. This functionality is realized by the following protocol (for the sake of readability, we depict the protocol and omit the corresponding knowledge and communication assertions):

\[
\begin{align*}
\text{Chair} & \xrightarrow{\text{COMM-J}} \text{Chair says} \text{RevAssign(id, paper)} \rightarrow \text{id} \\
\end{align*}
\]

The authorization policy for reviews is as follows:

\[
\begin{align*}
\forall x_{id}, y_{fp}, z_{rev}.

\text{Chair says} \text{RevAssign}(x_{id}, y_{fp}) & \land x_{id} \text{ says} \text{Rev}(y_{fp}, z_{rev}) \Rightarrow \text{Rev}(y_{fp}, z_{rev})
\end{align*}
\]

Ideally, reviewers should upload the minimal amount of information required to show that the review was submitted by an authorized reviewer. This can be achieved as follows:

\[
\begin{align*}
\text{Chair} & \xleftarrow{\text{COMM-A}} \text{Chair says} \text{RevAssign}(x_{id}, paper) \rightarrow \text{id} \\
\land x_{id} \text{ says} \text{Rev}(paper, rev)
\end{align*}
\]

The information transmitted to the PC chair does not reveal the identity of the reviewer, which may be desirable, for instance, if the paper’s author is a colleague of the PC chair or the reviewer does not want to reveal her identity to the whole PC. Appendix B of this paper reports the complete formalization and describes additional features, such as the management of rebuttals.

### 6.2. Distributed Social Network

The users of social networks have to face a surprisingly vast range of privacy issues. Well-understood problems, such as the centralized management and sharing of personal information, are accompanied by novel threats: for example, the Italian police is reported to have stipulated agreements with Facebook to get unfettered access to user profiles [34] and, in other countries, people that used social networks to organize protest activities were subject of repercussions, censorship, and coercion [33, 49, 1, 55, 27]. We used our framework to design a distributed social network that provides access control and, at the same time, user anonymity. This social network is close in spirit to the one that was recently developed by Backes et al. [12]. The cryptographic implementation that we obtain by compilation,
however, is substantially different, being based on pairing-based zero-knowledge protocols as opposed to traditional \( \Sigma \)-protocols [29], arguably simpler, and open-ended. Users may establish social relations as follows:

\[
\begin{align*}
A & \leftrightarrow B \text{ says } \text{Friend}(A) & \text{COMM-J} & B \\
\text{COMM-J} & A \text{ says } \text{Friend}(B) & \rightarrow
\end{align*}
\]

The predicate \( B \text{ says } \text{Friend}(A) \) represents a friendship request from \( B \) to \( A \) (social relations are unidirectional); the predicate \( A \text{ says } \text{Friend}(B) \) represents the corresponding friendship confirmation. \( B \) can use this information to engage in a number of activities. Suppose, for instance, that the access to \( A \)'s wall is limited to friends: \( B \) can anonymously post messages on \( A \)'s wall by existentially quantifying his identity, as shown below:

\[
\begin{align*}
A & \leftrightarrow \exists x. \ A \text{ says } \text{Friend}(x) & \wedge x \text{ says } \text{Wallpost}(w) & \text{COMM-A} & B
\end{align*}
\]

We additionally provide a method for downloading resources anonymously. Realizing this functionality is challenging since \( B \) does not want to reveal her identity to \( A \), who has to know, however, to whom to send the response. We use anonymous identifiers to solve this problem (cf. § 3.2). Suppose \( B \) is interested in a picture that can only be seen by \( A \)'s friends. Similarly to the previous protocol, \( B \) can prove to be a friend without revealing his identity and, in addition, give \( A \) a fresh anonymous identifier \( B^1 \), which she can use as intended recipient in the following communication assertion, as shown below:

\[
\begin{align*}
A & \leftrightarrow \exists x. \ A \text{ says } \text{Friend}(x) & \wedge x \text{ says } \text{getResource}(B^1, \text{pic-id}) & \text{COMM-A} & B
\end{align*}
\]

Appendix C gives a formal account of anonymous identifiers and illustrates other interesting features of the social network, such as friend-of-a-friend relationships and pseudonyms.

7. Implementation and Experimental Evaluation

We conducted an experimental evaluation to demonstrate the feasibility of our approach. This section overviews the cryptographic setup (§ 7.1) and discusses the experimental results (§ 7.2).

7.1. Cryptographic Setup

Devising a cryptographic realization for our logical framework turned out to be quite challenging. The cryptographic scheme has to be efficient and, at the same time, flexible enough to support the different usages of existential quantification, namely, the hiding of predicate arguments, principal identities, and equality relations among secret values (cf. Example 2).

\( \Sigma \)-protocols constitute a particularly efficient and widely deployed class of zero-knowledge protocols. Existing solutions, however, are not flexible enough to implement arbitrary existential quantification: for instance, existentially quantifying verification keys that are both signed and used to verify signatures (e.g., the key \( \text{vk}_{id} \) in statement (3)) is impractically slow [10].

A general solution to this problem was discovered only recently by Abe et al. [5], who introduced the notion of automorphic signatures. The distinctive feature of this signature scheme is that verification keys lie in the message space. Since messages and signatures consist of elements of a bilinear group and verification is done by evaluating a set of pairing-product equations, automorphic signatures make a perfect counterpart to the powerful zero-knowledge proof system by Groth and Sahai [39], which supports a large class of statements over bilinear groups. Our cryptographic implementation builds on a combination of these two cryptographic schemes. An other advantage of the Groth-Sahai scheme over \( \Sigma \)-protocols is the possibility (i) to re-randomize proofs without knowing their witnesses [19] and (ii) to existentially quantify information in existing proofs. We exploit the first property to hide the equality relations among secret values, while the second property is crucial for open-ended applications. We implemented the cryptographic library in Java and we relied on the jPBC library [30] for the computation of mathematical operations. A detailed description of our cryptographic implementation is reported in Appendix D.

7.2. Experimental evaluation

We conducted our experimental evaluation on a standard notebook with a 2.5 GHz dual-core processor\(^7\) and 8 GB of main memory. We measured the time required to create and verify various proofs and studied how these are influenced by the length of the security parameter, the size of the statement, and the number of existentially quantified values. In our experiments, the elliptic curves are such that the key length equals twice the security parameter [53], e.g., we use 160 bit keys to obtain a security parameter of 80 bits.

As illustrated in Figure 1, time and size grow linearly in the size of the statement, although increasing the number of conjuncts (and disjuncts) is more expensive than increasing the number of predicate arguments. The reason lies buried within the automorphic signature scheme for vectors of messages. Intuitively, setting up such a vector is computationally more expensive than filling it with elements [5].

\(^7\)The full model description is “Intel Core i5-2520M”
An exact analysis shows that adding one argument to a predicate costs roughly 5 seconds of proof generation time for a security parameter of 80 bits, while a zero-knowledge proof of a statement composed of one predicate with four arguments is computationally as expensive as a zero-knowledge proof for a statement composed of two predicates with one argument each.

The graphs in Figure 2, Figure 3, Figure 4, and Figure 5 (cf. § 7) depict the results obtained for some of the proofs illustrated in this paper for various security parameters. Time and size grow linearly in the length of the security parameter. The results for a security parameter of 80 bits vary from 31 seconds for anonymously posting messages on a friend’s wall up to 41 seconds for anonymously accessing a university lab. The reason for the small time and size differences is that the zero-knowledge statements are structurally very similar: they all comprise two signature verifications and the number of predicate arguments is only marginally different.

As previously discussed, existential quantification is very expensive in $\Sigma$-protocols [10] and the protocols themselves change depending on which and how many values are existentially quantified. The Groth-Sahai proof system combined with our encoding of predicates, instead, allows are existentially quantified. The Groth-Sahai proof system is computationally as expensive as a zero-knowledge proof for a statement composed of two predicates with one argument each.

The graphs in Figure 2, Figure 3, Figure 4, and Figure 5 (cf. § 7) depict the results obtained for some of the proofs illustrated in this paper for various security parameters. Time and size grow linearly in the length of the security parameter. The results for a security parameter of 80 bits vary from 31 seconds for anonymously posting messages on a friend’s wall up to 41 seconds for anonymously accessing a university lab. The reason for the small time and size differences is that the zero-knowledge statements are structurally very similar: they all comprise two signature verifications and the number of predicate arguments is only marginally different.

As previously discussed, existential quantification is very expensive in $\Sigma$-protocols [10] and the protocols themselves change depending on which and how many values are existentially quantified. The Groth-Sahai proof system combined with our encoding of predicates, instead, allows for a very efficient and arbitrary existential quantification. As a matter of fact, existential quantification comes at no costs at all in our implementation, as illustrated in Figure 6 (cf. § 7). Actually, the more information is existentially quantified, the shorter the zero-knowledge proof is. This is explained by the fact that we implement existential quantification by deleting certain information (specifically, the opening information for the corresponding commitment) from the proof and the performed computations are always the same.

We remark that our proof-of-concept implementation is not optimized in any way. In particular, we do not yet exploit readily available optimizations such as the multi-core architecture of today’s processors and batch verification techniques [23]. As the computations are largely independent, multi-core architectures yields a great performance gain. Batch verification techniques significantly speed up the verification process; the verification performance gains can be well above 50% [23]. We are working on the integration of such optimizations in our implementation.

### 8. Conclusion and Future Work

Ensuring the privacy of sensitive data is crucial for the widespread deployment of authorization infrastructures. In this paper, we show how to enforce privacy-aware evidential authorization using a powerful and efficient combination of digital signatures and zero-knowledge proofs. We developed a high-level declarative language that lets the user conveniently specify the system and the desired security properties, and a compiler that automatically produces executable cryptographic implementations. Although we studied the theoretical properties of our framework in the context of Evidential DKAL, our cryptographic construction is language-independent and generally applicable: in Appendix A, we report on a privacy-oriented extension of the Proof Carrying Authorization framework [8, 17] and we envision the usage of our framework in several other authorization languages, such as Aura [42], PCML$_5$ [9], F$^*$ [52], and SecPal [18].

We are currently implementing our framework on top of JXTA and JGroups, two open-source development platforms for distributed systems. These platforms provide high-level communication primitives that conceal the network layer and allow the designer to focus on the functional behavior of the system; our framework provides security-oriented data abstractions that hide the cryptographic layer as well.

We have formally proved that the implementations produced by the compiler enforce the intended authorization policies. Local privacy properties expressed via existential quantification are directly guaranteed by the zero-knowledge property of the employed proofs. Global privacy properties, such as strong secrecy and anonymity, are harder to enforce by construction, since they are not closed by composition and depend on the system as a whole. They can, however, be verified directly on the protocols produced by the compiler using off-the-shelf cryptographic protocol verifiers (e.g., ProVerif [22]). As a future work, it would be interesting to develop techniques to quantitatively measure global privacy properties (e.g., in terms of information flow, k-anonymity, etc.).

Finally, we intend to extend our framework in a number of directions. For example, we would like to develop primitives to share and process distributed data structures, yet preserving the privacy of sensitive information; this could be achieved by a combination of homomorphic encryptions and secure multiparty computations. It would also be interesting to offer support for other security properties, such as linear authorization policies [24] and trust properties [26].

### Acknowledgments

This work was partially supported by the initiative for excellence and the Emmy Noether program of the German federal government.

### References


[38] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. JACM, 38(3):690–728, 1991.


Figure 1. Comparison of the experimental evaluation of zero-knowledge proofs for predicates with various argument lengths against the conjunction of zero-knowledge proofs for predicates with only one argument. The security parameter is fixed to 80 bits.

Figure 2. Experimental evaluation on the zero-knowledge proof corresponding to formula (3) (a computer science student requesting access to a lab) for various security parameters.

Figure 3. Experimental evaluation on the zero-knowledge proof sent by the reviewer in the example in § 6.1 for various security parameters.
Figure 4. Experimental evaluation on the zero-knowledge proof for posting a message on a friend’s wall from § 6.2 for various security parameters.

Figure 5. Experimental evaluation on the zero-knowledge proof for retrieving a resource from a friend from § 6.2 for various security parameters.

Figure 6. Experimental evaluation on the computational and spacial impact of existential quantification for a predicate with 5 arguments and fixed security parameter of 80 bits.
A. Privacy-aware Proof-carrying Authorization

Our framework is an ideal plugin for proof carrying authorization (PCA), one of the most popular approaches for the enforcement of access control policies. PCA is successfully deployed in a number of different settings, including Web applications [17], mobile devices [16], and file systems [37, 28]. In a nutshell, the idea is to formalize a policy as a set of logical rules and to let the requester construct a formal proof showing that she has permissions to access the desired resource according to the provider’s policy. This policy may, of course, depend on logical formulas that are assumed by other principals in the system and whose validity is justified by digital signatures. The proof sent by the requester eases the task of the reference monitor on the provider’s side, which should otherwise explore the whole security policy and contact the other principals in the system to make sure that the requester is indeed granted the required permissions.

PCA builds on a standard higher-order logic with a says modality and it is natively equipped with the rule VER-SIG from Table 4 (which is called SIGNED in the original presentation [8]). We can easily extend PCA to support privacy-aware authorization policies by accompanying digital signatures with zero-knowledge proofs. Specifically, we extend PCA with the rule VER-ZK from Table 4. The requester can construct the proof to access the desired resource according to the rules from Table 2.

B. Distributed Reviewing System

In § 6.1, we describe an anonymous peer-to-peer reviewing system. We augment this system and take rebuttals into account. We show two extensions, one suitable for ordinary anonymous reviews and one for double-blind reviews.

B.1. Distributed Reviewing System with Rebuttals

Extending the distributed reviewing system with a rebuttal phase comprises giving the authors the possibility to respond to a review and, in turn, equipping the reviewers with the possibility to take the received rebuttal into account and to send a revised review to the authors. For the sake of readability, we let the authors communicate directly with the reviewer; an actual implementation would build on a cloud service or it would let the PC chair act as a proxy.

We modify the reviewer messages sent to the authors and include a fresh anonymous identity \( id^1 \). This identity will be used by the authors to send their rebuttal to their reviewer.

After receiving the reviews of their paper, the authors have a set of anonymous identities. These belong to all the paper reviewers. The rebuttals are eventually sent to the reviewers. Here, we show one exemplary message.

Finally, every reviewer writes her final report on each paper and sends it the PC chair as before.

Let us now briefly describe the DKAL derivation of this reviewing system, starting with the authors. For the sake of readability, we do not show the prepended predicates from the PC chair in the communication assertions. Firstly, we have to introduce all the necessary information into their knowledge, i.e., the paper itself and the rebuttal. Secondly, we need to specify the communication patterns. As they are very similar to each other, we only list the communication assertions for receiving a first review and for sending back the rebuttal. We need to put restrictions on these communication assertions to derive a sensible protocol. More specifically, the authors must have received a review prior to sending a rebuttal. We indicate this prerequisite and set the premise of the corresponding communication assertion to the predicate \( \exists y^1 \ y^1 \text{ says Rev}(paper, x_{rev}, y^1) \). Table 10 displays these rules.

The reviewers send a review and receive the corresponding rebuttal. Therefore, the only knowledge assertion for the reviewer introduces the review and the communication assertions consist of the assertions responsible for sending of the review, and the reception of the rebuttal. We set the premise of the communication assertions to true to indicate that they can always be applied in a communication rule. The selected set of assertions are shown in Table 11.

---

8The difference between \( y^1 \) and \( y_{id} \) becomes clear if we add the predicate from the PC chair: the PC chair signs the identity \( id \) but the reviewers send a fresh anonymous identity to the authors.
We now describe in detail the communication rule that is responsible for the exchange of the rebuttal from the authors to the reviewer.

**COMM-J**

\[
\begin{align*}
\text{if } \exists y_{id}. \ x_{rev} \ \text{says} \ Rev(paper, x_{rev}, y_{id}) \ \text{then} \ \text{Authors sends Authors says Rebuttal(paper, rebuttal) to } y_{id} \\
\text{if true then } id^1 \ \text{receives Authors says Rebuttal(paper, y_{rebuttal}) from Authors} \\
\text{Authors knows } \exists y_{id}. \ y_{id} \ \text{says} \ Rev(paper, x_{rev}, y_{id}) \eta \quad \text{Authors knows } M_{Rebuttal} \quad \text{id knows true } \theta \\
[y_{rebuttal} = \text{Rebuttal(paper, rebuttal)}] \quad \text{Authors says Rebuttal(paper, y_{rebuttal}) } \theta = \text{Authors says Rebuttal(paper, rebuttal)} \\
\end{align*}
\]

The communication assertions of the reviewers contain only the variable \(y_{rebuttal}\) to be instantiated \(\theta\) with the rebuttal. The substitution yields the ground predicate \(\text{Rebuttal(paper, rebuttal)}\) and the communication can proceed from the reviewers’ perspective.

On the sender side, all variables, \(x_{rev}\) and \(y_{id}\), are located in the precondition. We designed the system such that the rebuttal for paper can only be sent if a review for that paper was previously received, i.e., the predicate \(\exists y_{id}. \ y_{id} \ \text{says} \ Rev(paper, x_{rev}, y_{id})\) holds true for some review \(x_{rev}\) and some reviewer \(y_{id}\). In our concrete example, the variable \(x_{rev}\) is instantiated with the review \(rev\) previously received from the reviewer with the fresh anonymous identity \(id^1\), which instantiates the variable \(y_{id}\).

After the communication rule has successfully fired, the reviewer is in possession of the cryptographic evidence for the rebuttal message from the authors. In a concrete protocol run, the reviewer has obtained a digital signature on the (bit string encoding of) the rebuttal predicate.

**B.2. Double-blind Distributed Reviewing System with Rebuttals**

We now give a full description of a double-blind distributed reviewing system. The core idea is that only the program chair knows the binding between authors and papers, and between papers and reviewers; in our system, the program chair epitomizes an optimistic trusted third party that serves as a public-key infrastructure and only interferes in the protocol in case of a dispute.

Since the authors must remain incognito to the reviewers due to the double-blind reviewing process, the paper submission process as well as the rebuttal phase have to be re-designed. The idea is for the authors to send along with the paper registration a fresh anonymous identity. The program chair will distribute this identity along with the paper assignment to the reviewers who, in turn, use it to communicate with the authors.

There is, however, one more caveat that we need to address in our protocol design, namely, the authors must have a means to authenticate with the reviewer. More precisely, in the double-blind reviewing process, a reviewer does not know any authors’ identity. Yet, we want to design our protocols in a way that the authors authenticate the rebuttal with their reviewer, i.e., the authors must prove authorship of the paper in question to a reviewer. Therefore, during paper registration, the program chair sends the predicate \(Paper(\text{Authors, paper})\) to the authors. The resulting protocol up to the rebuttal phase looks as follows:

\[
\text{Authors} \quad \text{COMM-J} \quad \text{Authors says Subm(Authors, paper, Authors}) \rightarrow \text{Chair} \\
\quad \text{COMM-J} \quad \text{Chair says Paper(Authors, paper)} \\
\text{Chair} \quad \text{COMM-J} \quad \text{Chair says RevAssign(id, paper, Authors}) \rightarrow \text{id}
\]

Surprisingly, these changes already suffice to obtain double-blindness for our system. The reviewer is now in possession of an encryption key that gives her a confidential yet anonymous channel to the authors of a paper. Conversely, the authors have a signature from the program chair to authenticate their rebuttal with the respective reviewer; the integrity between the two parties is enforced by the zero-knowledge proofs, which draw their credibility from the signatures issued by the program chair. The rest of the protocol is borrowed from the non-blind distributed reviewing system and slightly modified to account
for the postulated anonymity of the authors.

Authors ← 3x_{id}. Chair says RevAssign(x_{id}, paper, Authors^{1}) ∧ x_{id} says Rev(paper, rev^{1})  

COMM-A id

COMM-J 3x_{Authors}. Chair says Paper(x_{Authors}, paper) ∧ x_{Authors} says Rebuttal(paper, rebuttal)

← 3x_{id}. Chair says RevAssign(x_{id}, paper, Authors^{1}) ∧ x_{id} says Rev(paper, rev')  

COMM-J

C. Distributed Social Network

Nowadays, social networks typically only treat first-order or direct friends, i.e., one establishes social relations only with direct friends. Access, however, is usually also granted to second-order friends, i.e., friends of friends. In our system, the white-listing principle ensures that only people that are given explicit access can obtain resources. While this gives us nice privacy properties, it excludes a more flexible access mechanism such as access to friends-of-friends that may be desirable in certain cases.

We extend our distributed social network to allow for friends of friends to access user resources while abiding by our white-listing policy that social relatives still need to be given explicit access. We devise a protocol for users to establish friend-of-friends relations without revealing their identities.\(^9\) More precisely, if A is a friend of C and B is a friend of A, then B can establish a second-level social relation with C without revealing his identity to C or to A. Additionally, the intermediate contact A also remains anonymous to C.

The core idea is to (i) let B anonymously announce his intention to be a friend of a friend along with a fresh anonymous identity, i.e., key pair. Of course, this gives us no incentive to accept B as a friend of a friend. Therefore, (ii) B anonymously contacts A and asks her to prove to us that we are indeed a friend of a friend. If (iii) A relays this message to us, we can be assured that the initial request indeed originates from a friend of a friend and (iv) grant access. This informal description results in the following protocol:

\[ C \leftarrow 3x_{B}. \text{A says RequestFoF}(C, B^{1}) \quad \text{COMM-A} \quad B \ (i) \]

\[ A \leftarrow 3x_{B}. \text{A says Friend}(x_{B}) \land x_{B} \text{ says Relay(requestFoF, C, B^{1})} \quad \text{COMM-A} \quad B \ (ii) \]

\[ C \leftarrow 3x_{A}, x_{B}. \text{C says RequestFoF}(C, x_{id}) \land x_{A} \text{ says Friend}(x_{B}) \land x_{B} \text{ says Relay(requestFoF, C, B^{1})} \quad \text{COMM-A} \quad A \ (iii) \]

\[ C \quad \text{COMM-J} \quad C \text{ says Friend-of-a-Friend}(B^{1}) \quad \rightarrow \quad B \ (iv) \]

Let us now consider the corresponding communication assertions and, in particular, the respective premises. In our example, we consider that B has not particular premises for the rules (i) and (ii) and we set them both to true, i.e., B can always fire these communications. Rule (iii), i.e., the relaying of B’s request to C, requires that B has already fired communication (ii). Finally, the premise for communication (iv) is the most interesting: this logical formula states that there must have been an initial request, a relay request, and it must enforce that the identity (in our example B^{1}) is the same for both requests. These demands yield the following formula:

\[ \exists x_{A}, x_{B}. \]
\[ x_{B} \text{ says RequestFoF}(C, x_{id}) \]
\[ \land \ C \text{ says Friend}(x_{A}) \land x_{A} \text{ says Friend}(x_{B}) \land x_{B} \text{ says Relay(requestFoF, C, x_{id})} \]

Now that we acquired a friend-of-a-friend relationship, we can grant access to our resources and we augment the access control policy by adding the following formula where the conclusion Authorized(pic, x_{id}) serves as precondition for communication assertions:

\[ \forall x_{id}. \ C \text{ says Friend-of-a-Friend}(x_{id}) \land x_{id} \text{ says getResource(pic)} \implies \text{Authorized}(pic, x_{id}) \]

\(^9\)Users must reveal their identity during the establishment of direct social relations (cf. § 6.2).
**Pseudonyms.** Until now, we mixed principal identities and fresh identities without any explicit separation; both represent principals and therefore deserve to be treated equally. In fact, they are even implemented in the same way, i.e., by means of an encryption key pair and a signature key pair. On the other hand, there is a tremendous difference in their semantics. More precisely, identifiers such as $A$ and $B$ are registered in a PKI and seeing either of the keys is sufficient to identify their owner. The fresh identities, however, are not registered in a PKI. In fact, their sole purpose is to allow for anonymous communication. As such, we use fresh anonymous identities as pseudonyms for users, similarly to Backes et al. [12].

For instance, $B$ contacts $A$ and sends the predicate $B$ says $\text{Friend}(A, B^1)$ that additionally contains a fresh anonymous identifier to establish a social relation. $A$ includes this identifier inside her signature and responds with the predicate $A$ says $\text{Friend}(B, B^1)$. Our zero-knowledge deduction system gives $B$ the opportunity to use that signature as before by always hiding the anonymous identifier. Alternatively, $B$ can hide his plain identifier and selectively reveal his pseudonym.

**D. Cryptographic Realization**

We now introduce the cryptographic primitives that are necessary to assemble the aforementioned zero-knowledge proofs ($\S$ D.1) and we describe how each zero-knowledge deduction rule from Table 2 can be implemented ($\S$ D.2).

**D.1. Cryptographic Primitives**

**Bilinear map.** The usage of bilinear maps makes the zero-knowledge scheme used in our framework particularly efficient and expressive. A bilinear map $e$ maps elements from $\mathbb{G}_1 \times \mathbb{G}_2$ into the target group $\mathbb{G}_T$ and is bilinear. More precisely, the following equation holds for all values $\mathbb{G}, \mathcal{H}, x,$ and $y$:

$$e(x \cdot \mathbb{G}, y \cdot \mathcal{H}) = e(\mathbb{G}, y \cdot \mathcal{H})^x = e(x \cdot \mathbb{G}, \mathcal{H})^y = e(\mathbb{G}, \mathcal{H})^{xy}$$

We use the convention that calligraphic uppercase letters denote elliptic curve elements, while lowercase letters denote elements in a finite group $\mathbb{Z}_n$ for some $n$. All cryptographically relevant bilinear maps operate on elliptic curves.

**Digital signature scheme.** We use the digital signature scheme that was recently proposed by Abe et al. [5]. The distinguishing characteristic of this signature scheme is that verification keys reside in the message space, i.e., it is possible to sign verification keys without relying on auxiliary encodings (e.g., hashing). This property is particularly important to combine digital signatures with zero-knowledge proofs, since the aforementioned encodings are hard to deal with in zero-knowledge and made previous proof constructions highly inefficient.

The message space is the set of all Diffie-Hellman tuples $\mathcal{DH} = \{(i \cdot \mathbb{G}, i \cdot \mathcal{H}) \mid i \in \mathbb{Z}_q\}$ where the curve elements $\mathbb{G}$ and $\mathcal{H}$ are globally fixed generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, and $q$ is a globally fixed large prime. A signature key $sk = s$ consists of a secret exponent $s$ and a verification key $vk = (\mathbb{X}, \mathbb{Y}) = (s \cdot \mathbb{G}, s \cdot \mathcal{H})$. We write $x \in_R S$ to say that $x$ is chosen uniformly at random from the set $S$.

On input of a message $(\mathcal{M}, \mathcal{N}) = (m \cdot \mathbb{G}, m \cdot \mathcal{H})$ and the public parameters $pp = (\mathcal{F}, \mathcal{K}, \mathcal{T})$, the signer chooses $c, r \in_R \mathbb{Z}_q$ and the signature consists of the tuple $\text{sig} = (A, C, D, R, S)$, where

$$A := \frac{1}{x + c} \cdot (\mathcal{K} + r \cdot \mathcal{T} + \mathcal{M}) \quad C := c \cdot \mathcal{F} \quad D := c \cdot \mathcal{H} \quad R := r \cdot \mathbb{G} \quad S := r \cdot \mathcal{H}$$

The verifier has to check the following equalities in order to verify the signature $(A, C, D, R, S)$ on message $(\mathcal{M}, \mathcal{N})$:

$$e(A, \mathcal{Y} + D) = e(\mathcal{K} + \mathcal{M}, \mathcal{H}) \cdot e(\mathcal{T}, S) \quad e(C, \mathcal{H}) = e(\mathcal{F}, D) \quad e(R, \mathcal{H}) = e(\mathbb{G}, \mathcal{S})$$

When analyzing this signature scheme, one notices that the message always appears on the left-hand side of the pairing (i.e., $\mathbb{G}_1$) whereas the verification key always appears on the right-hand side of the pairing (i.e., $\mathbb{G}_2$). An important insight here is that a verification key actually consists of two group elements, $\mathbb{X}$ from $\mathbb{G}_1$ and $\mathbb{Y}$ from $\mathbb{G}_2$. This makes it possible to sign verification keys. A complication arises, however, if a verification key is both signed and used to verify a signature, since we have to show that $\mathbb{X}$ and $\mathbb{Y}$ are somehow connected. The idea is to prove that the following equation holds:

$$e(\mathbb{X}, \mathcal{H}) = e(\mathbb{G}, \mathcal{Y})$$

This equation connects the message part $\mathbb{X}$ of the verification key with the key part $\mathbb{Y}$. The validity of this statement can easily be checked using the bilinearity of $e$. 

20
This digital signature scheme is existentially unforgeable against chosen-message attacks, which is the standard notion of security for digital signature schemes. We remark that this signature scheme can be extended to vectors of messages with only a small constant overhead per additional element. We exploit this property to sign logical formulas, by encoding logical predicates and their arguments into vectors of messages.

Example. Let us consider the example from § 2.1. The PC chair encodes the predicate \( \text{RevAssign}(id, \text{paper}) \) as \( \vec{m} = (\text{RevAssign}, id, \text{paper}) \) and embeds \( \vec{m} \) into the message space of the signature scheme by computing \( (\mathcal{M}, \mathcal{N}) = (\vec{m} \cdot \mathcal{G}, \vec{m} \cdot \mathcal{H}) \). The PC chair can then compute the signature as previously described. □

This signature scheme allows for very efficient zero-knowledge proofs when used in combination with the Groth-Sahai proof scheme, as described below.

Commitments. Intuitively, a commitment is the digital equivalent of a message inside a sealed envelope that lies on top of a table: the message creator cannot change the content of the message and principals around the table cannot look inside the envelope until it is opened.

Commitments are an essential building block for zero-knowledge proofs. Intuitively, a zero-knowledge proof scheme shows properties and relations of committed values without opening the commitments: for instance, given three commitments \( c_1, c_2, \) and \( c_3 \), it is possible to prove that the multiplication of the values committed to in \( c_1 \) and \( c_2 \) yields the value committed to in \( c_3 \). The prover can open some of the commitments to selectively reveal information to the verifier.

Groth-Sahai zero-knowledge proof system. The Groth-Sahai zero-knowledge proof system [39] is a very flexible and general scheme that is based on the previously described cryptographic primitives. It captures so-called multi-scalar multiplication equations in both \( \mathbb{G}_1 \) and \( \mathbb{G}_2 \), pairing product equations, and quadratic equations in \( \mathbb{G}_T \). The supported kinds of equations are depicted in Table 12: \( a, b, t, A, B, T, \) and \( \gamma \) denote constants that are known to the verifier, while \( x, y, \mathcal{X}, \) and \( \mathcal{Y} \) denote values for which the verifier knows the commitments but not the opening information. Groth-Sahai proofs are proofs of knowledge, i.e., intuitively, they show that the prover is in possession of the witnesses used in the proof.

In order to create a proof for a signature \( (A, C, D, R, S) \) on \( (\mathcal{M}, \mathcal{N}) \), the prover has to show that the verification equation holds as well as all intermediate steps used to derive that equation. More precisely, she has to show the pairing product equations dictated by the signature verification equations, which are shown below. Here and throughout the paper, we write \( C_v \) to denote the commitment to \( v \) and \( [C] \) to denote the value committed in \( C \).

\[
e(\{C_A\}, \{C_y\}) \cdot e(\{C_A\}, \{C_D\}) = e(\{C_Z\}, \{C_H\}) \cdot e(\{C_M\}, \{C_H\}) \cdot e(\{C_T\}, \{C_S\})
\]

In order to reveal some of the committed values, the prover has simply to forward the opening information to the verifier. Conversely, the prover can hide values inside their commitments by not forwarding the opening information. This gives the prover a simple, yet flexible, mechanism to fine-tune the level of privacy.

The resulting proofs are non-interactive zero-knowledge proofs of knowledge. In particular, they consist only of a single message \( (\vec{\pi}, \vec{C}, \vec{O}) \), comprising the proofs \( \vec{\pi} \) of the individual equations, the commitments \( \vec{C} \), and the opening information \( \vec{O} \) for the values revealed by the proof.

We finally remark that Groth-Sahai proofs can be randomized, i.e., given a proof \( \vec{\pi} \), we can transform \( \vec{\pi} \) into a proof \( \vec{\pi}' \) that proves exactly the same statement but the proof itself and the commitments therein look different. This is crucial in applications aiming at providing unlinkability properties: for instance, a principal can receive a proof and then transform it into a new proof in which further information is existentially quantified, in a way that the two proofs are not linkable. It is even possible to randomize even only a selected subset of the commitments. In particular, it is possible to randomize two commitments on the same value for which the opening information are available such that the two resulting commitments coincide. More precisely, given \( C_v \) and \( C_v' \), it is possible to randomize them and turn them into the same commitment \( C_v'' \), as described by Bolenkij et al. [19]. The randomization of single commitments is important to faithfully implement existential quantification, as described in § D.2.

Pseudonyms. In the spirit of Pseudo-Trust [46], we use a one-way function to generate pseudonyms. More precisely, we use scalar multiplication in \( \mathbb{G}_2 \) to generate pseudonyms, i.e., a principal randomly chooses her secret \( s \) and computes her pseudonym \( p \) as \( p = s \cdot \mathcal{H} \), which constitutes our one-way function. Note that a pseudonym corresponds to the second part of a verification key and \( s \) corresponds to the signing key.

When a user wants to use her pseudonym to authenticate a formula \( F \), she uses her secret value \( s \) to sign \( F \) and her pseudonym acts as verification key for the signature on \( F \).
D.2. Proof Construction

We have described all the cryptographic primitives used as basic building blocks of our zero-knowledge proofs. We now describe how we can implement each of the zero-knowledge deduction rules from Table 2 using the aforementioned primitives.

I-S-∧ and I-ZK-∧. Intuitively, the proof for the logical conjunction of two statements is easily obtained by concatenating the two individual proofs. In other words, if we are given two proofs \((\pi_1, \tilde{C}_1, \tilde{O}_1)\) and \((\pi_2, \tilde{C}_2, \tilde{O}_2)\), the proof \((\pi_1 \cdot \pi_2, \tilde{C}_1 \cdot \tilde{C}_2, \tilde{O}_1 \cdot \tilde{O}_2)\) shows the logical conjunction of the two individual proofs. Here and throughout the paper, we write \(X \cdot Y\) to denote the concatenation of \(X\) and \(Y\).

E-ZK-∧. This rules constitute the inverse of I-ZK-∧ and, indeed, the cryptographic realization reflects this intuition: given a proof \((\pi_1 : \pi_2, \tilde{C}_1 : \tilde{C}_2, \tilde{O}_1 : \tilde{O}_2)\) of a logical conjunction of two statements, the proof of the first conjunct is \((\pi_1, \tilde{C}_1, \tilde{O}_1)\).

I-S-∨. We encode a logical disjunction as follows. We assume that all our formulas \(F\) are stated in disjunctive normal form: 
\[
F = \bigvee_{i=1}^n F_i
\]
where \(x_i \cdot F_i\) denotes \(\bigwedge_j x_i \cdot f_{i,j} = t_{i,j}\). Each \(f_{i,j} = t_{i,j}\) corresponds to an equation provable using the Groth-Sahai proof scheme (cf. Table 12), e.g., \(f_{i,1}\) is a pairing product equation and \(f_{i,2}\) is a multi-scalar equation. We introduce fresh variables \(x_i\) and manipulate the equations as follows:

\[
F_\ell := \bigwedge_{i=1}^n x_i F_i
\]

where \(x_i \cdot F_i\) denotes \(\bigwedge_j x_i \cdot f_{i,j} = t_{i,j}\).\(^{10}\) We choose the \(x_i\) in the following way. Let \(\ell\) be such that the \(\ell\)-th equation holds. We set \(x_i = 1\) and \(x_j = 0\) for \(j \neq i\).

For each fresh variable \(x_i\), we now introduce the following quadratic equations in \(\mathbb{Z}_n\):

\[
x_i - x_i^2 = 0
\]

Finally, we add the equation

\[
\sum_i x_i = 1
\]

The result is a proof of the formula \(F\). We refer to Appendix I for a detailed discussion.

The logical disjunction is true if at least one formula inside the disjunction is true. This is reflected by proving the sum over all variables \(x_i\) is equal to 1. If one wants to prove that at least \(\ell\) equations of the disjunction are true, one shows that \(\sum_i x_i = \ell\). A similar construction can be used to generate zero-knowledge proofs of logical conjunctions where rules I-ZK-∧ and E-ZK-∧ cannot be applied; we detail this technique in Appendix H.

I-ZK-∃. In principle, if we are given a zero-knowledge proof \((\pi, \tilde{C}, \tilde{O} : o)\) and we want to existentially quantify the value with the opening information \(o\), we have just to drop \(o\), thereby obtaining \((\pi, \tilde{C}, \tilde{O})\). Special care has to be taken, however, if we have to existentially quantify a value that is used in different statements and is associated with commitments that have different opening information: before dropping that information, we have first to randomize the commitments to make them equal, in a way that the verifier can still deduce that the secret value occurring in the different statements is the same.

E. Generation of F# Code

E.1. Cryptographic Library

We present the F# library with methods to create, mold, and verify zero-knowledge proofs. Each method is indexed with an abstract zero-knowledge statement, i.e., a statement in which existentially quantified values (resp. public values) are replaced by placeholders \(\alpha_i\) (resp. \(\beta_j\)). We let \(S\) range over abstract statements and write \([S]\) to denote the abstract statement obtained from \(S\) by replacing each existentially quantified variable with a distinct \(\alpha_i\) and each public value with a distinct \(\beta_j\) (the statement is read from left to right, the index of \(\alpha\)-placeholders is increased as soon as a new variable is found, while

---

\(^{10}\)Here, we implicitly assume that all equations are suitable for zero-knowledge proofs (cf. Groth and Sahai [39], Section 8).
the index of $\beta$-placeholders is increased as soon as a public value, not necessarily new, is found), and finally dropping the existential quantifier. A function indexed with $[S]$ can deal with any concrete statement $S$ obtained by replacing placeholders with actual values.

F\# library

Zero-knowledge proof creation:

\[
\text{create}_S \ x_1 \ldots x_i \ y_1 \ldots y_j
\]

The zero-knowledge proof creation function $\text{create}_S$ takes as input the secret witnesses $x_1, \ldots, x_i$ and the public values $y_1, \ldots, y_j$, creates a zero-knowledge proof for the statement $S := \exists \alpha_1, \ldots, \alpha_i. S\{x_k/\alpha_k\}\{y_\ell/\beta_\ell\}$. The return value is a zero-knowledge proof that can successfully be verified if and only if the proven statement $S$ is true.\(^{11}\)

Zero-knowledge proof verification:

\[
\text{verify}_S \ z \ y_1 \ldots y_j
\]

The zero-knowledge proof verification takes as input a zero-knowledge proof $z$ and the public values $y_1$ to $y_j$. If the proven statement $\exists \alpha_1, \ldots, \alpha_i. S\{y_\ell/\beta_\ell\}$ holds true, the verification returns a tuple consisting of the public values. Should the proven statement not pass the verification, an exception is thrown.

Zero-knowledge proof molding:

- \[
\text{create}_{\text{EZKAnd}}(S, S') \ z
\]

This function takes as input a zero-knowledge proof for a statement $S$ and returns a zero-knowledge proof for the statements $S' := S \land S''$ or $S = S' \land S''$ for some $S''$. This zero-knowledge proof can be used as if it had been created with the zero-knowledge creation function.

- \[
\text{create}_{\text{IZKExists}}(S, S') \ z
\]

This function takes as input a zero-knowledge proof for a statement $S$ and returns a zero-knowledge proof for the statements $S' := S\{\alpha_i+1/\beta_\ell\}$, i.e., the statement derived from $S$ by existentially quantifying the $\ell$-th public value. Here, we assume that the $\ell$-th public value becomes the $i + 1$-th secret witness. This zero-knowledge proof can be used as if it had been created with the zero-knowledge creation function.

- \[
\text{create}_{\text{IZKAnd}}(S_1, S_2) \ z_1 z_2
\]

This function takes as input a zero-knowledge proof of statement $S_1$, a zero-knowledge proof of statement $S_2$, and returns a zero-knowledge proof of $S_1 \land S_2$. This zero-knowledge proof can be used as if it had been created with the zero-knowledge creation function.

Zero-knowledge proof public values:

\[
\text{pub}_S \ z
\]

This function takes as input a zero-knowledge proof for statement $S$ and returns a tuple containing the public values of that proof.

E.2. DKAL to F\# Compiler

We have all the cryptographic library functions needed to transform a DKAL derivation into F\# code. As all DKAL rules are localized by means of the knows-modality, it is sufficient to describe how the code for every single deduction rule is translated. The code of each principal $A$ is simply obtained by scanning the DKAL derivation top-down (i.e., from the outermost hypotheses until the thesis) and, for each rule associated with $A$, by appending the corresponding code to the existing code for $A$. While generating F\# code, we identify principals with their pseudonyms, i.e., we generate code for $A$ even though the deduction yields knowledge for principal $A^k$.

\(^{11}\)Although it may seem unintuitive, it is easily possible to create an “invalid” zero-knowledge proof, for instance, by using random values $(X_1, X_2, X_3, X_4, X_5)$ in place of an actual signature $(A, C, D, R, S)$ for message $(M, N)$ (cf. § D.2). This proof looks valid but it will (with overwhelming probability) not pass the zero-knowledge verification.
The code output by the compiler contains assumptions and assertions. Even though they do not have any computational significance, we add them to state the intended safety property and to prove the correctness of the compiler.

One of the central challenges in formalizing the compiler is the mapping between DKAL bit strings into F\# variables. This is problematic, since DKAL does not come with a notion of variable substitution, which has thus to be hard-coded into the compiler. Intuitively, we would like to enforce the following convention in the compiler for the translation of DKAL bit strings \( b \) into F\# values: if \( b \) is an a-priori known verification key, then we use the variable bound to that key (we assume that such variables exist, i.e., verification keys have been pre-distributed). Should \( b \) not be an a priori known verification key but rather a value obtained from a previous signature verification or a zero-knowledge proof verification, we use the variable bound to \( b \) (which exists by construction). If neither of the previous cases applies, then we use directly the bit string \( b \) (which we assume to be a priori known in accordance with DKAL). We specify this intuition using the substitution \( \phi \).

This substitution \( \phi \) takes as input an identity \( A \) and a bit string \( b \) and returns the variable \( x_b \). This variable \( x_b \) is bound in \( A \)'s code as described above. Thus, the compilation takes as input a substitution \( \phi \) and a DKAL rule and returns a substitution \( \phi' \) and F\# code. We write \( \phi_A \) to denote the substitution for \( A \)'s code, i.e., \( \phi_A(b) := (A, b) \phi \), and we write \( \phi_A[b \mapsto x_b] \) to denote the update specific to \( A \)'s code, i.e., \( \phi_A[b \mapsto x_b] := \phi((A, b) \mapsto x_b) \).

For all principals \( A \), we initially define \( \phi_A \) as follows:

\[
\phi_A(b) = \begin{cases} 
  x_{vk} & \text{if } b = vk \\
  x_{sk_A} & \text{if } b = sk_A \\
  b & \text{otherwise}
\end{cases}
\]

In other words, we map each verification key to the corresponding variable and the other bit strings to themselves. The compiler updates \( \phi \) whenever bit strings different from verification keys are received from the network, read from signatures, or conveyed by zero-knowledge proofs.

During the run of a protocol, principals generate a plethora of zero-knowledge proofs, all proving different statements. We use the substitution \( \phi^\varpi \) to keep track and associate a statement and a principal to a cryptographic evidence for that statement. This correlation comes in handy and, in fact, is crucial since our zero-knowledge rules do not mention cryptographic evidence but exactly this evidence is utterly necessary for actual cryptographic operations, i.e., the compiler. Initially, \( \phi^\varpi \) is the empty substitution. We define the abbreviations \( \phi_A^\varpi \) and \( \phi_B^\varpi \) as expected.

### Compiler algorithm to translate DKAL rules to F# code

<table>
<thead>
<tr>
<th>PP-K-1</th>
<th>PP-K-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A^k ) knows ( F )</td>
<td>( A ) knows ( F )</td>
</tr>
<tr>
<td>( A ) knows ( F )</td>
<td>( A^k ) knows ( F )</td>
</tr>
</tbody>
</table>

We identify principal \( A \) with her anonymous identity \( A^k \). The PP-K rules allow us to switch principals in the knows modality; they do not require any F# code as they only indicate that a different public-key pair is used when encrypting/decrypting messages to be sent on/received from the network.

The ENSUE rule and the P-ZK rule constitute the copula between the \( \Gamma \vdash \) judgment and the \( \Gamma \vdash_{zk} \) judgment, respectively, and the knows modality and does not require any code.

<table>
<thead>
<tr>
<th>P-A</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A : F )</td>
</tr>
</tbody>
</table>

Knowledge assertions in DKAL are nodes that introduce infons into the knowledge of a principal. The F# equivalent is an
This knowledge assertion also introduces the formula \( F = p(M_1 \phi_A, \ldots, M_\ell \phi_A) \) into the knowledge of principal \( A \). This knowledge is, however, witnessed by the digital signature \( M \) on \( F \). The \( \ell + 1 \)-th component “\((\)" is an artifact of our verification technique: this unit value conveys the logical formula \( F \) (cf. Appendix F.2) and is computationally irrelevant.

The successful signature verification with \( I \)'s verification key (denoted by \( v_k \)) ensures that the logical formula \( F = p(M_1, \ldots, M_\ell) \) holds true for the tuple \( x'_M \) where \( x'_M \) is a fresh variable. The subsequent tuple splitting supplies the individual components for later use (in particular, this split allows us to use the verification keys contained within the tuple).
This code does not manipulate any cryptographic evidence but it checks that the formula \( S_1 \lor S_2 \) holds true.

All native DKAL infon logic deduction rules

This code applies to all native DKAL infon logic deduction rules where \( F_r \) corresponds to the deduced formula. The assertion ensures that the respective formula indeed holds true.

\[
\text{assert}(F_r \phi_A)
\]

\[
\text{let } x_M = \text{create}_{[S]}() (M_1 \phi_A) \ldots (M_j \phi_A) \text{ in}
\]

\[
\Gamma \vdash_S S \quad \phi := \phi_A[M \mapsto x_M] \quad \phi^\phi := \phi^\phi_A[S]
\]

From now on, we use the bit string \( M \) whenever we need to refer to this zero-knowledge proofs and we use the \( F^\# \) variable \( x_M \) to address this zero-knowledge proof. \( M_1, \ldots, M_j \) denote the public values of the proof (in this case corresponding to all names occurring in the statement as there are no witnesses); these are known by construction.

\[
\text{let } x_p = \text{pub}_{[S]}(M \phi_A) \text{ in}
\]

\[
\text{let } (x_{p_1}, \ldots, x_{p_k}) = x_p \text{ in}
\]

\[
\text{let } x_M = \text{verify}_{[S]}(M \phi_A) (M_1 \phi_A) (M_2 \phi_A) \ldots (M_j \phi_A) \text{ in}
\]

\[
\text{assert}([S] \phi_A)
\]

\[
\text{let } (x_{M_1}, \ldots, x_{M_{j+1}}) = x'_{M} \text{ in}
\]

\[
\phi := \phi_A[M_1 \mapsto x_{M_1}] \cdots [M_j \mapsto x_{M_j}] \quad \phi^\phi := \phi^\phi_A[S \mapsto M]
\]

We first extract the public values (there might be values we do not know yet but they are already bound in \( \phi_A \) due to its initialization). The well-formedness condition and the substitution \( \phi_A \) ensure that we use all (a priori) known verification keys. Appendix F.2 clarifies the necessity of the well-formedness condition. We split the result \( x'_M \) of the verification to make its individual components available.

\[
\text{let } x_{M'} = \text{create}_{\text{ZKexists}([S], [\exists x. S])} (M \phi_A) \text{ in}
\]

\[
\phi := \phi_A[M' \mapsto x_{M'}] \quad \phi^\phi := \phi^\phi_A[\exists x. S \mapsto M']
\]

where \( M = (S\{u/x\})^{\phi_A}. \) From now on, we let \( M' \) denote the zero-knowledge proof for the statement \( \exists x. S. \)

\[
\text{let } x_{M_1} = \text{create}_{\text{ZKand}([S],[S_1])} (M \phi_A) \text{ in}
\]

\[
\phi := \phi_A[M_1 \mapsto x_{M_1}] \quad \phi^\phi := \phi^\phi_A[\exists x. S_1 \mapsto M_1]
\]

\[
\text{let } x_{M_2} = \text{create}_{\text{ZKand}([S],[S_2])} (M \phi_A) \text{ in}
\]

\[
\phi := \phi_A[M_2 \mapsto x_{M_2}] \quad \phi^\phi := \phi^\phi_A[\exists x. S_2 \mapsto M_2]
\]

where \( M = (\exists x. S_1 \land S_2)^{\phi_A}. \) \( M_1 \) and \( M_2 \) denote the freshly created zero-knowledge proofs of the statements \( S_1 \) and \( S_2 \) respectively.
make its components accessible. If she expects a signature or a zero-knowledge proof, the respective verification functions is applied. Finally, the tuple is split into the components involved. First, she does the “reverse”, i.e., she reads the ciphertext from the network and decrypts it. Depending on whether she knows a signature or a zero-knowledge proof, she does the respective verification function.

The sender encrypts the message and sends the obtained ciphertext out onto the network. The receiver’s task is slightly more involved. First, she does the “reverse”, i.e., she reads the ciphertext from the network and decrypts it. Depending on whether she expects a signature or a zero-knowledge proof, the respective verification functions is applied. Finally, the tuple is split into the components accessible.

The compiler itself works on a top-down manner. Rule applications in a DKAL derivation without further rule applications “on top” (i.e., axioms in that derivation) can be translated and pruned (informally, pruning a rule applications means removing the hypotheses of that rule as well as the rule bar; we leave the conclusion as new hypothesis). Axioms in a DKAL derivation, i.e., knowledge assertions can always be compiled. Code will be appended to the already existing code of a principal; the initial code of all principals is empty.

Compiler from a DKAL derivation $\mathcal{D}$ to $\mathcal{F}_\#$

Init:
$C_I = ()$ for all principals $I$

Algorithm:
1. let $m$ be a rule application in $\mathcal{D}$ without further rule applications on top (i.e., $m$ is either an axiom or all rules on top of $m$ have already been pruned away)
2. generate code for $m$ and append to code $C_I$ of the respective principal(s) $I$
3. prune the rule application \( m \) from \( D \)

The algorithm terminates when \( D \) contains no rule applications anymore and only consists of original root conclusion. The compiler appends ( ) to the code of all principals.

---

The compiler generates sequential code. This code is not unique as it depends on the order in which rule applications are chosen in step 1.

**F. Proofs**

This section is devoted to proving our main theorem, namely, that our compiler generates code that type-checks. We start by introducing the primitives we use to symbolically encode zero-knowledge, describe how we model zero-knowledge, and develop all the theory that finally ends in proving that the compiler produces well-typed code.

**F.1. Symbolic Cryptographic library**

The F7 type-checker proves safety properties of F# code linked to a symbolic cryptographic library. Our cryptographic library builds on sealing, an extremely versatile technique that can be used to faithfully model a number of cryptographic primitives, such as symmetric encryption, asymmetric cryptographic, and, most notably, zero-knowledge proofs.

**Sealing [20].** A seal comprises two functions: (i) a sealing function that takes as input a message, stores this message in a secret list along with a fresh value, and returns this fresh value and (ii) an unsealing function that takes as input a fresh value, scans the secret list in search for the associated message, and returns this message. The clue is that it is not possible to extract the sealed value given only the fresh value; this extraction indispensably requires the unsealing function. In a sense, a sealing mechanism corresponds to a symmetric encryption scheme, where sealing corresponds to encryption and unsealing to decryption. The sealed value corresponds to the encrypted message and the fresh value to the ciphertext. Similarly, we can use sealing to model public-key encryption (resp. digital signatures), making the sealing function (resp. unsealing function) public.

**Types of library functions for digital signatures and encryptions.** In the symbolic cryptographic libraries released with the F5 and F7 type checkers, the sealing function is given the polymorphic type \( \alpha \rightarrow \text{unit} \) and the unsealing function is given type \( \text{unit} \rightarrow \alpha \), where \( \alpha \) is the type of the sealed value and \( \text{unit} \) is the type of the fresh value. Thus, we have \( \text{EncKey}(\alpha) := \alpha \rightarrow \text{unit} \) and \( \text{DecKey}(\alpha) = (\alpha \rightarrow \text{unit} \ast (\text{unit} \rightarrow \alpha) \). Encryption and decryption functions are typed as follows:

\[
\begin{align*}
\text{encrypt} & : \alpha \rightarrow \text{EncKey}(\alpha) \rightarrow \text{unit} \\
\text{decrypt} & : \text{unit} \rightarrow \text{DecKey}(\alpha) \rightarrow \alpha
\end{align*}
\]

We adopt this typing discipline for asymmetric encryption, instantiating \( \alpha \) with \( \text{unit} \) (both signatures and zero-knowledge proofs can be upcast to \( \text{unit} \)). We deviate from traditional approaches, however, as far as the typing discipline for digital signatures is concerned. In particular, we have to make the type signing and verification keys recursive as, in our protocols, verification keys are used in place of identifiers and may be signed. The types for keys and cryptographic functions are
reported below:

\[
\begin{align*}
\mathcal{T}_{\mathbf{vk}} & := (\prod_{i=1}^{n} p_k \text{ of } x_1 : T_1 \cdots \cdots x_{\ell_k} : T_{\ell_k} \{ y \text{ says } p_k(x_1, \ldots, x_{\ell_k}) \}) \\
\mathcal{U} & := (\prod_{i=1}^{n} p_k \text{ of } x_1 : T_1 \cdots \cdots x_{\ell_k} : T_{\ell_k} \{ \exists x, y. \text{ sk} = (x, y) \land y \text{ says } p_k(x_1, \ldots, x_{\ell_k}) \}) \\
& \text{ where } T_i \in \{ \text{unit}, \alpha \} \\
\mathcal{V}_{\mathbf{vk}} & := y : \text{unit} \ast \mathcal{T}_{\mathbf{vk}} \\
\text{VerKey} & := \mu \alpha. \text{unit} \rightarrow \mathcal{T}_{\mathbf{vk}} \text{ unfold } \rightarrow y : \text{unit} \ast \mathcal{T}_{\mathbf{vk}} \{ \text{VerKey}/\alpha \} \\
\text{SigKey} & := (y : \text{unit} \ast \mathcal{T}_{\mathbf{vk}} \{ \text{VerKey}/\alpha \} \rightarrow \text{unit}) \ast \text{VerKey} \\
\mathcal{U} & := \mathcal{U} \{ \text{VerKey}/\alpha \} \\
\text{sig} & : \text{sk} : \text{SigKey} \rightarrow \mathcal{U} \rightarrow \text{unit} \\
\text{check} & : y : \text{VerKey} \rightarrow \text{unit} \rightarrow \mathcal{T}_{\mathbf{vk}} \{ \text{VerKey}/\alpha \}
\end{align*}
\]

with the following implementation:

\[
\begin{align*}
\text{check } \text{vk } s & = \\
\text{sig } \text{sk } m & = \\
\text{let } (x, y) & = \text{sk} \text{ in } \\
x (y, m) & = \\
\text{let } (v, m') & = \text{vk } s \text{ in } \\
\text{if } v & = \text{vk} \text{ then } \\
m' & = \\
\text{else } \\
\text{failwith(verification failed)}
\end{align*}
\]

Logical predicates of the form \( A \text{ says } p(M_1, \ldots, M_n) \) are encoded by signatures on tuples of the form \((p, M_1, \ldots, M_n)\), where \( M_i \) is either a unit value (e.g., a review, a picture id, or a wall post) or a verification key. The refinement type \( \mathcal{T}_{\mathbf{vk}} \{ \text{VerKey}/\alpha \} \) characterizes such tuples. Technically, this type is a tagged union of dependent tuples, with refinements expressing the predicate that is enforced to hold whenever such tuples are signed.

The signing function takes as input a payload of type \( \mathcal{U} \) and a signing key for such a payload,\(^{12}\) and it returns a signature of type unit. The signature verification function check takes as input the signature, the verification key, and the signed message, and it returns the signed message with the original (stronger) type \( \mathcal{T}_{\mathbf{vk}} \{ \text{VerKey}/\alpha \} \).

**Modeling zero-knowledge proofs using a sealing mechanism** [11]. A zero-knowledge proof consists of several parts: the secret witnesses, the public components, and the matched components. The secret witnesses are exactly the values hidden by the proof. Logically, they are existentially quantified in the proven formula. The public values are all the values that are revealed by the proof. Intuitively, the public values comprise all the names that occur in a zero-knowledge statement except for the secret witnesses. Matched components are a subset of the public values and are known beforehand to the verifier. It is possible to extract public values from a zero-knowledge proof. We anticipate here that matched components play a crucial role when type-checking a zero-knowledge proof as they will be the only values with a type different from unit. In our Modeling, the public values and the matched components coincide, i.e., the prover has to know all matched values beforehand.\(^{13}\)

Intuitively, zero-knowledge proofs enjoy three fundamental properties: they can be verified if and only if the statement holds true (completeness and soundness) and they do not reveal any information about the secret witnesses (zero-knowledge). Additionally, we require that zero-knowledge proofs can only be created if the prover is in possession of the witnesses to the proven statement (proofs of knowledge).

We express these four properties using the sealing mechanism described above. When creating a zero-knowledge proof, we require as input all the witnesses and all the public values and, therefore, we model proofs of knowledge. All these values (of type unit) are packed away in a seal. Hence the create\(_S\) function is given type \( x_1 : \text{unit} \rightarrow \cdots \rightarrow x_i : \text{unit} \rightarrow y_1 : \text{unit} \rightarrow \cdots \rightarrow y_j : \text{unit} \rightarrow \text{unit} \) (cf. § E.1).

\(^{12}\)The importance of type \( \mathcal{U} \) and its connection to VerKey becomes clear in the proof.

\(^{13}\)This is equivalent to differentiating between public values and matched components. If there were values the verifier did not know before, she could extract them from the zero-knowledge proof and use them as matched components (of type unit) in the proof verification.
The verification has access to the sealed value, i.e., secret witnesses and public values, and checks whether the alleged statement holds true. We mention that if two verification keys are pattern-matched in a zero-knowledge statement, then the corresponding identifiers are pattern-matched as well. This encoding is consistent with the soundness property. At the same time, if the statement to be proven holds true, then the verification function succeeds, which guarantees completeness. Since the secret witnesses never leave the verification function and it is not possible to extract them from a zero-knowledge proof, this encoding also faithfully models the zero-knowledge property. The verify function has type \( z_{zk} : \text{unit} \rightarrow y_1 : T_1 \rightarrow \ldots \rightarrow y_j \rightarrow T_j \rightarrow T_1 * \cdots * T_j + : \{[[S]]\} \), where the \( T_i \)'s are the expected types of the public components.

Finally, the functions to mold zero-knowledge proofs are given access to the (secret) seals of the two associated zero-knowledge proofs. This makes the (symbolic) implementation of \( \text{create}_E \text{ZKAnd}(S, S') \), \( \text{create}_E \text{ZKExists}(S, S') \), \( \text{create}_E \text{IZKAnd}(S_1, S_2) \) straightforward. For instance, the implementation of \( \text{create}_E \text{ZKAnd}(S_1, S_2) \) takes as inputs two proofs, recovers the witnesses by unsealing, and concludes by calling the create function \( create \). We then need to show that all \( \alpha \)'s, or the final refinement type that carries the intended identities are existentially quantified as well.

The type of \( \text{create}_E \text{ZKAnd}(S, S') \) and \( \text{create}_E \text{ZKExists}(S, S') \) is \( \text{unit} \rightarrow \text{unit} \), while the type of \( \text{create}_E \text{IZKAnd}(S_1, S_2) \) is \( \text{unit} \rightarrow \text{unit} \).

One may wonder how we generate the functions since there are potentially infinitely many of these (e.g., one may take a proof for statement \( S \) and generate a proof for \( S \land \cdots \land S \)). However, every protocol only deals with a finite number of zero-knowledge proofs and, in particular, every DKAL derivation is finite. As a consequence, we only need to generate finitely many of these functions, i.e., exactly those that are required by the DKAL derivation.

### F.2. Well-typedness of Generated RCF Code

We will now develop all necessary theory required to prove our main theorem, namely, that the code we generate is well-typed. We start by showing that signature creation and verification are well-typed.

We recall

\[
\mathcal{T}_{vk} := (\langle x_1 : T_1 * \cdots * x_{\ell_k} : T_{\ell_k} \rangle \overline{\{ y \text{ says } p_k(x_1, \ldots, x_{\ell_k}) \}})
\]

\[
\mathcal{U} := (\langle x_1 : T_1 * \cdots * x_{\ell_k} : T_{\ell_k} \rangle \overline{\{ \exists x, y. \text{ sk} = (x, y) \land y \text{ says } p_k(x_1, \ldots, x_{\ell_k}) \}})
\]

where \( T_i \in \{ \text{unit}, \alpha \} \)

\[
\mathcal{T}_{sk} := y : \text{unit} * \mathcal{T}_{vk}
\]

\[
\text{VerKey} := \mu \alpha. \text{unit} \rightarrow \mathcal{T}_{vk} \overset{\text{unfold}}{=} \text{unit} \rightarrow y : \text{unit} * \mathcal{T}_{sk}(\text{VerKey}/\alpha)
\]

\[
\text{SigKey} := (y : \text{unit} * \mathcal{T}_{sk}(\text{VerKey}/\alpha) \rightarrow \text{unit}) * \text{VerKey}
\]

\[
\mathcal{U} := \mathcal{U} * \text{VerKey}/\alpha
\]

\[
\text{sig} : \text{sk} : \text{SigKey} \rightarrow \mathcal{U} \rightarrow \text{unit}
\]

\[
\text{sig sk m} = \text{let } (x, y) = \text{sk in} \ x (y, m)
\]

**Lemma 1 (VerKey :: pub).** If \( E \vdash \alpha \), then \( E \vdash \text{VerKey :: pub} \).

**Proof.** The public key is an iso-recursive type; \textsc{Kind Rec} allows us to assume that \( \alpha :: \text{pub} \) to show the rest of the derivation, i.e., we must show that \( E, \alpha :: \text{pub} \vdash \text{unit} \rightarrow \mathcal{T}_{vk} :: \text{pub} \). Here and for the rest of the proof, we assume that all names are freshly chosen (which can be easily achieved by \( \alpha \)-renaming). \textsc{Kind Fun} requires us to show that unit is tainted and that \( T_{vk} \) is public. By \textsc{Kind Unit}, we immediately get that \( \text{unit :: tnt} \). For the rest, we use \textsc{Kind Sum} and we need to show that all individual tuple elements are public. The individual components are either unit, \( \alpha \)'s, or the final refinement type that carries the intended formula.

We can immediately conclude that all unit are public and that the \( \alpha \)'s are public as the initial application of \textsc{Kind Rec} added the necessary binding to \( E \). The resulting typing environment contains all the binding for the tuple elements. For the refinement type, we apply rule \textsc{Kind Refine Public} and we are left to show that the refinement type is well-formed (as before, the unit value carrying the formula is public by \textsc{Kind Unit}).
The well-formedness can be shown by applying \( \text{TypE} \). As \( E \) is well-formed by assumption and we only added entries to \( E \) with fresh names, we do not destroy the well-formedness condition and we are left to show that all free names and all free variables of the refinement type \( T_{\text{sk}}^o \) are already bound in the current \( E \). Since all the free names and free variables of the refinement type occur in the tuple, this condition is also fulfilled and we can conclude that \( E \vdash \text{VerKey} :: \text{pub} \). □

**Encoding of conditionals**

We encode conditionals as follows:

\[
\begin{align*}
\text{if } M = N \text{ then } & A \\
\text{else } & B : V \\
\end{align*}
\]

\[ \text{let } x = (M = N) \text{ in } \]

\[ \text{match } x \text{ with } \text{inr}() \rightarrow A \text{ else } B \]

---

**Lemma 2** (\( \text{sig well-typed} \)). *If all types occurring in \( \text{sig} \) are defined as in § 3.4.1 and \( E \vdash \circ, \text{sk}, m \notin \text{Dom}(E) \), then \( E \vdash \text{sig} : (\text{sk} : \text{SigKey}) \rightarrow \text{U} \rightarrow \text{unit} \).*

*Proof.* We apply \( \text{ValFun} \) to introduce the arguments into the typing environment; the argument names do not occur in \( E \) by assumption and the extended environment remains well-formed. Since \( \text{sk} \) is of type \( \text{SigKey} \), the tuple split succeeds (we assume that \( x \) and \( y \) are freshly chosen names and do not coincide with any element from the domain of \( E \), a condition easily achievable by \( \alpha \)-renaming) and it remains to show that \( E, x : \text{unit} * T_{\text{sk}}^o \{\text{VerKey}/\alpha\} \rightarrow \text{unit}, y : \text{VerKey}, \{\text{sk} = (x, y)\} \vdash x (y, m) : \text{unit} \).

First, we notice that by Lemma 1 and \( \text{SubPubTNT} \), we deduce that \( \text{VerKey} < : \text{unit} \) and by \( \text{ExpSubSum} \), we can use the verification key, denoted by \( y \) as the first component of the tuple.

Second, we note that \( m \) has type \( \text{U} \) but we need \( m \) of type \( T_{\text{sk}} \) to match the type of the seal. Luckily, we know inside our typing environment that \( \text{sk} = (x, y) \) and by existential elimination \( \text{FOL} \), \( \text{E} \), \( \text{I} \), \( \text{ELIM} \) and logical conjunction elimination \( \text{AND} \) and \( \text{ELIM} \) applied to \( \text{U} \), we can deduce that \( m : T_{\text{sk}}^o \) where the \( y \) in \( T_{\text{sk}}^o \) is indeed replaced by the verification key \( y \), the second component from the tuple split of the signing key.

Finally, we can use \( \text{ExpApp} \) and obtain that the returned value is of type unit.

\[ \] □

In the proof for the well-typedness of the signature creation function, we have seen that, even though we expect a type \( \text{U} \) different from \( T_{\text{sk}} \), we can logically transform \( \text{U} \) into \( T_{\text{sk}} \).\(^\text{14}\) However, one would not expect to produce value of type \( \text{U} \) when signing; one should just have to assume or derive the formulas necessary to create \( T_{\text{sk}} \). And indeed, we derive \( \text{U} \) from just \( T_{\text{sk}}^o \) formally in Lemma 5.

\[
\text{check} : (y : \text{VerKey}) \rightarrow \text{unit} \rightarrow T_{\text{sk}}^o \{\text{VerKey}/\alpha\}
\]

\[
\begin{align*}
\text{check} & = \\
\text{let } (vk', m') = & \text{vk } s \text{ in } \\
\text{if } vk' = & \text{vk then } \text{m}' \\
\text{else } & \text{failwith(verification failed)} \\
\end{align*}
\]

**Lemma 3** (check well-typed). *If all types occurring in \( \text{check} \) are defined as in § 3.4.1 and \( E \vdash \circ, \text{s}, \text{vk}, m \notin \text{Dom}(E) \), then \( E \vdash \text{check} : \text{unit} \rightarrow (y : \text{SigKey}) \rightarrow \text{unit} \rightarrow T_{\text{sk}}^o \{\text{VerKey}/\alpha\} \).*

*Proof.* We apply \( \text{ValFun} \) to introduce the arguments into the typing environment; the argument names do not occur in \( E \) by assumption and the extended environment remains well-formed. Since \( \text{vk} \) is of type \( \text{VerKey} \) and \( s \) is of type \( \text{unit} \), we can apply \( \text{ExpApp} \) and obtain a tuple. This tuple is, in turn, immediately split (in this proof, we assume that binders such as \( \text{vk}' \) and \( m' \) are freshly chosen names and do not coincide with any element from the domain of \( E \), a condition easily achievable by \( \alpha \)-renaming) and we obtain the modified environment \( E \) that binds \( \text{vk}' \) : unit and \( m' : T_{\text{sk}}^o \{\text{VerKey}/\alpha\}\{\text{vk}'/y\} \). In the following, we do not consider the alternative branches as they raise exceptions that type-check for any type.

\(^{14}\text{Since the difference is only in the logical refinement.}\)
The \( \text{let} \) expression binds \( x \) to \( \text{vk} = \text{vk}' \). Rule \( \text{Exp Eq} \) gives \( x \) the type \( \text{bool} \), which in \( F7 \) is defined as a refined sum type \( \text{unit} + \text{unit} \), true is defined as \( \text{inr}() \) and false as \( \text{inl}() \). Thus, we can apply \( \text{Exp Match} \) and we need to type-check the remainder process under the extended typing environment \( E \) that contains the formula \( \{ x = \text{true} \} \). Using the refinement of \( \text{bool} \), we can deduce inside our logic that \( \text{vk} = \text{vk}' \) and continue type-checking. We have a similar process for the next conditional; we end up with an environment that contains the formula \( \{ m' = m \} \).

Let us take a look at the types now. Due to the unsealing and the tuple split, we know that \( m' : \mathcal{T}_{\text{vk}}^o(\text{VerKey}/\alpha)\{\text{vk}'/y\} \). Since \( \text{vk} = \text{vk}' \), we can deduce that \( m' : \mathcal{T}_{\text{vk}}^o(\text{VerKey}/\alpha)\{\text{vk}/y\} \), i.e., we can replace the unsealed value \( \text{vk}' \) with the input verification key \( \text{vk} \). We return \( m' \) and the claim follows.

Type-checking heavily relies on the order in which the type-checking is conducted. For instance, a signature verification will only succeed with a meaningful result if the verification key can be given a matching type; this may in turn depend on other signature verifications. Therefore, even well-formed monomials may not be sufficient to guarantee that our generated code type-checks. We mitigate this by slightly modifying Definition 1 of trustworthy keys that takes the order in which keys appear into account.

**Definition 6 (Trustworthiness of keys).** A key \( u \) is trustworthy in a monomial \( M = \bigwedge_{i=1}^{m} a_p \) iff one of the following conditions holds:

- \( u = \text{vk} \) is registered
- there exists \( a_{p_j} = \text{ver}_{\text{sig}}(u_s, u_k, F) \) such that \( u \) is a variable occurring free in \( F \) and \( u_k \) is trustworthy in \( M_{ij} \), where \( M_{ij} = \bigwedge_{i=1}^{j} a_{p_i} \)

Since \( u \) must be trustworthy in \( M_{ij} \) and we have already generated and, in particular, type-checked the code that establishes the trustworthiness of \( u \). Definition 3 remains unchanged but uses the revised definition of trustworthy keys.

First, we start by proving a well-known fact in computer science that justify our assumption that formulas are given in disjunctive form.

**Lemma 4 (Representation lemma).** For every quantifier-free Boolean formula \( f \), there is a formula \( g \) in disjunctive form such that \( f \Leftrightarrow g \).

**Proof.** The proof is by induction: whenever there is a conjunction on top of a disjunction, apply the distributivity law \((x \lor y) \land z \Leftrightarrow (x \land z) \lor (y \land z)\) or \(z \land (x \lor y) \Leftrightarrow (z \land x) \lor (z \land y)\). Possible negations will be pushed to the level of atomic predicates using De Morgan’s laws \( \neg(x \land y) \Leftrightarrow \neg x \lor \neg y \) and \( \neg(x \lor y) \Leftrightarrow \neg x \land \neg y \).

**Definition 7 (Appropriate typing environment).** We call a typing environment \( E \) appropriate if and only if for all principals \( I \) occurring in the protocol all of the following conditions hold:

- \( E \vdash \phi \)
- \( E \vdash \text{check} : \text{VerKey} \rightarrow \text{unit} \rightarrow \alpha \)
- \( E \vdash \text{dec} : \text{DecKey} \rightarrow \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{enc} : \text{EncKey} \rightarrow \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{failwith} : \text{unit} \rightarrow \alpha \)
- \( E \vdash \text{send} : \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{recv} : \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{ek}_I : \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{dk}_I : \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{vk}_I : \text{VerKey} \)
- \( E \vdash \text{sk}_I : \text{SigKey} \)
Lemma 6. Let \( \{sk_f = (sk'_f, vk_f)\} \in \text{Dom}(E) \)
We say a typing environment \( E \) is appropriate with respect to sealing if an only if \( E \) is appropriate and
\[ E \vdash \text{seal}_S : (\text{unit} \rightarrow \text{unit}) \ast (\text{unit} \rightarrow \text{unit}) \text{ for all abstract statements } S \text{ occurring.} \]
We first show that an appropriate typing environment allows us to transform \( T_{sk}^\alpha \) into \( \mathcal{U} \) as defined in § F.1.

Lemma 5. If \( E \) is appropriate and \( E \vdash m : T_{sk}^\alpha \{\text{VerKey}/\alpha\}\{vk_f/y\}, \) then \( E \vdash m : \mathcal{U}\{sk_f/sk\} \)
Proof. Let \( E \) be appropriate and let \( E \vdash m : T_{sk}^\alpha \{\text{VerKey}/\alpha\}\{vk_f/y\}. \) As \( E \) is appropriate, \( E \vdash sk_f = (sk'_f, vk_f). \) Let \( vk_f \) says \( F \) be the refinement of \( m. \) With this refinement and the logic properties about conjunction, we obtain
\[ \text{sk}_f = (sk'_f, vk_f) \land \text{vk}_f \text{ says } F. \]
We apply the logical rule FOL EXIST INTRO and obtain the refinement required by \( \mathcal{U} \):
\[ \exists x, \ y. \ \text{sk}_f = (x, y) \land y \text{ says } F. \]

The premise that \( E \vdash sk_f = (sk'_f, vk_f) \) is naturally obtained. After generating a signing key, its owner must split the key to retrieve the verification key and distribute it to other hosts (e.g., via a certifying PKI). This tuple splitting of the secret key sk automatically introduces the formula \( sk = (sk'_f, \text{vk}_f) \) into the current typing environment.

At the point when that principal needs to create a message of type \( \mathcal{U} \), she assumes the formula necessary to give that message the type \( T_{sk}^\alpha \{\text{VerKey}/\alpha\} \) where \( y \) is instantiated by the actual verification key and Lemma 5 yields that the corresponding message is also of type \( \mathcal{U}. \)

From now on, we use the following two conventions that \( \Pi\overline{x} : \overline{T}.U \) and \( \Sigma\overline{x} : \overline{T}.U \) denote \( \Pi x_1 : T_1, \Pi x_2 : T_2, \ldots, \Pi x_\ell : T_\ell, U \) and \( \Sigma x_1 : T_1, \Sigma x_2 : T_2, \ldots, \Sigma x_\ell : T_\ell, U, \) respectively.

Lemma 6. Let \( C = \text{fun } y_1 : T_1 \rightarrow \cdots \rightarrow y_\ell : T_\ell \rightarrow C' \) and \( E \) be such that \( \text{fn}(C) \subseteq \text{Dom}(E) \) and \( \text{bn}(C) \cap \text{Dom}(E) = \emptyset. \) If \( E, y_1 : T_1, \ldots, y_\ell : T_\ell \vdash C' : T \) for some type \( T, \) then \( E \vdash C : \Pi y_1 : T_1, \ldots, y_\ell : T_\ell, T). \)

Proof. The only rule that can be applied to code of the form \( x : T \rightarrow .R \text{ is VAL FUN}. \) If this rule is applied \( \ell \) times to \( C, \) the desired result follows.

Lemma 7 (Appropriate typing environment extension). Let \( E \) be appropriate and let \( E' \) be such that \( E' \vdash \diamond \) and \( \text{Dom}(E) \cap \text{Dom}(E') = \emptyset. \) Then \( E, E' \) is appropriate.

Proof. As \( \text{Dom}(E) \cap \text{Dom}(E') = \emptyset \) and \( E' \vdash \diamond, \) we have that \( E, E' \vdash \diamond \) using ENV ENTRY and induction on the length of \( E'. \) Since \( E \) is appropriate, by weakening, \( E, E' \) will also prove necessary requirements.

Corollary 1 (Appropriate w.r.t. to sealing typing environment extension). Let \( E \) be appropriate w.r.t. sealing for a set of seal occurring in a protocol and let \( E' \) be such that \( E' \vdash \diamond \) and \( \text{Dom}(E) \cap \text{Dom}(E') = \emptyset. \) Then \( E, E' \) is appropriate w.r.t. sealing.

From now on, we use the following shortcuts:
\[ \{M = N\} := x : \{y : \text{bool} \mid (y = \text{inr}()) \land M = N\} \lor (y = \text{inl}()) \land M \neq N\}, \_ : \{x = \text{inr}()\} \]
\[ \_ : \text{unit} \mid F\} = \_ : \{F\} \]
where \( x \) is a fresh variable occurring nowhere else. This shortcut is used in the following typing rule for conditionals, which, as proved below, is derivable in F7.

Rule EXP IF

\[
\begin{array}{c}
\text{EXP IF} \\
E \vdash \diamond E \vdash M : T E \vdash N : U E, x : \{M = N\} \vdash A : V E \vdash B : V x \text{ fresh} \\
E \vdash \text{if } M = N \text{ then } A \text{ else } B : V
\end{array}
\]

\[\text{As } T_{sk} \text{ is a union type, there are several possibilities.}\]
Lemma 8. Rule EXP If is derivable in F7.

Proof. We show that the hypothesis of EXP If are strong enough to imply the premises deriving from type-checking the desugared code for the If statement. We start by type-checking the desugared version of the code under a typing environment $E$.

\[
\begin{align*}
\text{let } x &= (M = N) \\
\text{match } x \text{ with true } &\rightarrow A \text{ else } B \} =: C
\end{align*}
\]

where $x$ is a fresh name that occurs nowhere else.

In the following, we let $W := \{ y : \text{bool} \mid x = \text{true} \land M = N \lor x = \text{false} \land M \neq N \}$.

We type-check $C$ under the typing environment $E' := E, x : W$.

We now show that the hypotheses of EXP If imply the hypothesis of derivation for the desugared if-statement, i.e., we show that (1)-(5) imply (b)-(e).

(a) $E \vdash \varnothing$

(b) $E \vdash M : T$

(c) $E \vdash N : U$

(d) $E, \{ M = N \} \vdash A : V$

(e) $E \vdash B : V$

(1) $E \vdash M : T$

(2) $E \vdash N : U$

(3) $x \notin \text{fv}(M, N, V)$

(4) $E, x : W \vdash x : \text{bool}$

(5) $E, x : W, (\{ \text{inr}(y) = x \} \vdash A : V$

(6) $E, x : W, \{ \forall y. \text{inr}(y) \neq x \} \vdash B : V$

(1) and (2):

Immediately by (b) and (c).

(3):

Follow from our convention that the variable $x$ in the desugared version is fresh and occurs nowhere else.

(4):

Follow since $E, x : W \vdash W \llhd \text{bool}$ by SUB REFINE LEFT and SUB REFL.

(5):

This case is the most involved in the proof.
Proof step:

This is proven as hypothesis of rule Exp If.

The shape of the current typing environment is almost the one from (d). We apply weakening (Lemma 6 [20]) thrice to add the entries \(\{\text{inr}() = x\}, () : \text{unit}, \) and \(x : W\) in that order from our current typing environment. The order is important to maintain the well-formedness condition required by the lemma.

First, we note that forms\((E_1) \vdash M = N\), i.e., the formula contained in \(x : W\) combined with the formula \(\text{inr}() = x\) (i.e., \(x = \text{true}\)), yields the desired formula \(M = N\). We apply strengthening (Lemma 4 [20]) which allows us to drop \(\{M = N\}\) from our typing environment.

This is the required hypothesis (5) of the desugared version of the if statement.

(6):

We apply weakening twice and obtain hypothesis (e).

We have proven that the premises of rule Exp If imply the premises required to hold by the desugared version of an if-statement. This concludes our proof.

Lemma 9. Let \(E\) be appropriate and let \(C\) be a cascade of \(\ell\) if statements that only test equality between two variables and the cascade ends with \(C'\), i.e., \(C\) is of the form

\[
C := \begin{cases} 
  \text{if } M_1 = N_1 \text{ then} \\
  \quad \text{if } M_2 = N_2 \text{ then} \\
  \quad \ldots \\
  \quad \text{if } M_\ell = N_\ell \text{ then} \\
  \quad \quad C' \\
  \quad \text{else} \\
  \quad \quad \text{failwith(\(\ell\)-th component does not match)} \\
  \quad \ldots \\
  \quad \text{else} \\
  \quad \quad \text{failwith(Second component does not match)} \\
  \quad \text{else} \\
  \quad \quad \text{failwith(First component does not match)} \\
\end{cases}
\]

If \(E, \vdash \{M_1 = N_1\}, \ldots, \{M_\ell = N_\ell\} \vdash C' : V\) for some type \(V\) and for all \(i\), \(E \vdash M_i : T_i\) and \(E \vdash N_i : U_i\) for some \(T_i\) and \(U_i\), then \(E \vdash C : V\).

Proof. The proof is by induction on the number of applications of the rule Exp If. As \(E\) is appropriate, we have that \(E \vdash \Box\). The type \(V\) from Exp If is instantiated by \(V\) from the lemma. The else branch always type-checks because the exception that is raised is of the (polymorphic) type \(\alpha\) and, thus, in particular of type \(V\).

Matching of tuples

We use the following syntactic sugar:

\[
\text{match } M \text{ with } h(x_1, \ldots, x_m) \rightarrow A \text{ else } B
\]
Zero-knowledge verification function for a statement $S$

1. $\text{verify}_S = \text{fun } z : \text{unit } \to$
2. \quad $\text{fun } y_1 : T_1 \to \ldots \text{ fun } y_j : T_j \to$
3. \quad let $z' = (\text{snd seal}_S) z$ in
4. \quad let $(x'_1, \ldots, x'_i, y'_1, \ldots, y'_j) = z'$ in
5. \quad if $y'_1 = y_1$ then
6. \quad \quad \ldots
7. \quad \quad if $y'_j = y_j$ then
8. \quad \quad \quad checking$(S, M)$
9. \quad \quad else
10. \quad \quad \quad failwith(The $j$-th argument does not match)
11. \quad \quad \ldots
12. \quad else
13. \quad \quad failwith(The first argument does not match)

\[
\begin{align*}
\text{checking} \ (S, \{ap_1 \land \cdots \land ap_i, M_2, \ldots, M_n\}) : \\
&\text{match check } \text{vk}_1 \text{ sig}_1 \text{ with } \text{p}_{k_1}(v^1_1, \ldots, v^m_1) \\
&\text{match check } \text{vk}_2 \text{ sig}_2 \text{ with } \text{p}_{k_2}(v^1_2, \ldots, v^m_2) \\
&\quad \ldots \\
&\text{match check } \text{vk}_\ell \text{ sig}_\ell \text{ with } \text{p}_{k_\ell}(v^1_\ell, \ldots, v^m_\ell) \\
&\quad (y_1 : T_1 \land \cdots \land y_j : T_j, \{S\}) \\
&\quad \text{else} \\
&\quad \quad \text{checking}(S, \{M_2, \ldots, M_n\}) \\
&\quad \ldots \\
&\text{else} \\
&\quad \quad \text{checking}(S, \{M_2, \ldots, M_n\}) \\
&\text{else} \\
&\quad \text{checking}(S, \{M_2, \ldots, M_n\})
\end{align*}
\]

where the $k$-th atomic predicate is tagged with $p_k$.

\[
\text{checking}(S, \{\}) : \\
\quad \text{failwith("Verification failed")}
\]

**Theorem 4** (Typability of zero-knowledge verification code). Let $E$ be an appropriate typing environment with respect to sealing and let $\text{verify}_S$ be the generated code for the well-formed statement $S$. Then $E \vdash \text{verify}_S : \text{unit } \to T_1 \to \cdots \to T_j \to (y_1 : T_1, \ldots, y_j : T_j, \{\text{unit } \mid [S]\})$ where the $T_k$’s denote the expected types of the public components, i.e., VerKey if $y_k$ is expected to be a verification key or unit otherwise.

**Proof.** Let $E$ be appropriate with respect to sealing. Thus, $E \vdash \infty$. We assume that all names in $E$ are different from the names in the generated code; a condition that can easily be achieved by $\alpha$-renaming. In the following, we let $L_k$ denote the body of the verification procedure, beginning with line $k$, and we let $T_S$ denote $(y_1 : T_1, \ldots, y_j : T_j, \{\text{unit } \mid [S]\})$. 36
Lemma 10
Proof.
We will split the proof in two parts and start with the sender side.

type-checks the receiver code and the empty monomial as base case. still holds as we generate code “from left to right” of a monomial (cf. Definition 6) and we just argued that we can deduce typing environment \(j\) obtained thus far back to the original typing environment \(j\) by Lemma 6.

As \(E\) and thus \(E_1\) is appropriate by Corollary 1 (we assumed that all names occurring in the verification function are different from those occurring in \(E\)) \(E_1 \vdash L_3 \vdash T_2\) where \(E_1 = E, v_1 : T_1, \ldots, v_j : T_j\) by Lemma 6.

We use KIND PAIR and SUB PUB TNT to derive that unit \(<: unit \ast \cdots \ast unit; we can apply EXP SPLIT to arrive at the typing environment \(E_2\) where \(E_2 = E_1, x_1 : unit, \ldots, x_k : unit, y_1 : unit, \ldots, y_j : unit, \{ (x_1, \ldots, x_i, y_1, \ldots, y_j) = \{ ')'.

The cascade of statements, i.e., the factually matching of the matched components of the proof, requires us to prove that \(E_3 \vdash L_7 : T_3\) by Lemma 9, where \(E_3 = E_2, \{ y_1 = y_1 \}, \ldots, \{ y_j = y_j \}\).

We prove the signature verification part by induction on the number of atomic predicates in a monomial. At this point, the condition that the statement is well-formed and, accordingly, all monomials in that statement are well-formed becomes utterly important. Without this condition, we could not assume that the verification keys are given the type VerKey, and we would not be able to derive appropriate types for the values covered by the verified signature.

\[
\text{match check } vk \text{ sig with } p_k(v_1, \ldots, v_m) \rightarrow C \text{ else failwith(Verification Error)}
\]

where \(C\) denotes the continuation code, i.e., either the return statement or a further signature verification. The variables \(v_K\) are fresh variables proven equal to matched components and secret witnesses by match in the logic (except for \(v_m\) as it is a nameless variable that carries the logical formula). This important step has the potential to give secret witnesses stronger types (i.e., the type VerKey). Thus, in the subsequent code, whenever we need to use a secret witness, we use a \(v_k\) that has been proven equal to that witness.

The verification key \(vk\) has the proper type VerKey since all monomials are well-formed (cf. Definition 3), i.e., \(E_4 \vdash vk : \text{VerKey}\). Note that \(vk\) is either a matched component \(y_k\) for some \(k\) or it is a variable that has been proven equal to a secret witness by a previous signature verification; the verification key thus has the strong type VerKey. We can apply EXP APPL two times and the signature verification type-checks with type \(T_{jk}\{\text{VerKey}/\alpha}\).

After the matching, we are required to prove that \(E_4 := E_3, v : (U_1 \ast \cdots \ast U_m,), : p_k v = \text{check } vk \text{ sig}, v_1 : U_1, \ldots, v_m : U_m, : \{ (v_1, \ldots, v_m) = v \} \vdash C\). Here, the \(U_i\) denote the type given by \(T_{jk}\{\text{VerKey}/\alpha}\) for the branch \(p_k\).

We stress that the logical formula entailed by the signature verification is “stored” in \(v_m\).

We just showed that a signature verification type-checks under the assumption that all keys are trustworthy. This invariant still holds as we generate code “from left to right” of a monomial (cf. Definition 6) and we just argued that we can deduce the strong type for the components. To show that our generated code type-checks, we us the above proof as induction step and the empty monomial as base case.

To complete this part of the proof, we need to prove type-checks. \(S\) is given in disjunctive normal form, i.e., \(S := \exists x. \bigvee M_k\) where each \(M_k := ap_1 \land \cdots \land ap_t\). We proved that the individual formulas corresponding to the atomic predicates \(ap_j\), are logically entailed by the signature verification. Together, all of these touples imply \(M_k\) which, due to the disjunction, implies \(S\). If the verification of a signature fails, we can use weakening to shrink the typing environment obtained thus far back to the original typing environment \(E\) and proceed to the next monomial.

\[
\text{Lemma 10 (Typability of communication code). If } E_A \text{ and } E_B \text{ are appropriate, } E_A \vdash F_A, \text{ and } E_B \vdash F_B, \text{ then } E_A \text{ type-checks the receiver code and } E_B \text{ type-checks the sender code.}
\]

Proof. We will split the proof in two parts and start with the sender side.

- By assumption, \(E_B \vdash F_B\) and the assertion type-checks. The value \(x_M\) is either a signature of type unit or a zero-knowledge proof of type unit. As \(E_B\) is appropriate, we also know that the encryption key for \(A\) has the correct type and we can apply rule EXP APPL two times to obtain that \(E \vdash \text{encrypt } ek_A x_M : \text{unit}\) and EXP LET requires us to show that \(E_2 \vdash \text{send } x_{msg} : \text{unit}\) where \(E_2 = E_B, x_{msg} : \text{unit}\). Consequently, we can apply rule EXP APPL and we are finished with the sender part of the proof.

- As in the above case, \(E_A \vdash F_A\). As \(E_A\) is well-formed, we can use EXP APPL to rec\(v\) and EXP LET requires us to show that the continuation code type-checks under the environment \(E_2 = E_A, x_{msg} : \text{unit}\).

We can immediately apply rule EXP APPL twice; we proved \(x_{msg}\) of the correct type above and as \(E_2\) is appropriate, we know that \(d_k\) is of the correct type, i.e., \(E_2 \vdash d_k : \text{unit} \rightarrow \text{unit}\). Again, EXP LET requires us to show that \(E_3 = E_2, x_M : \text{unit} \vdash \text{VERIFY}(x_M)\). There are two ways that \(\text{VERIFY}(x_M)\) is generated:

37
Let \( T \) be a DKAL derivation and let \( C \) be the code generated for principal \( I \) from \( T \) by the algorithm from Appendix E.2. If \( E \) is appropriate, \( fofo(C) \subseteq \text{Dom}(E) \), and \( bo(C) \cap \text{Dom}(E) = \emptyset \), then \( E \vdash C : \text{unit} \).

**Proof.** The proof is by induction on the structure of the DKAL derivation \( T \). Given a DKAL rule, the premises of a rule require knowledge of certain cryptographic evidence \( M \). Since we start our induction with base cases, the knowledge is based either on given digital signatures or given zero-knowledge proofs.

We do not consider bound variables in the premise; consistent \( \alpha \)-renaming solves any coincidence with elements from the domain of the typing environment.

We start with the base cases, i.e., rules I-S-VER and I-ZK-VER. For the sake of presentation, we assume that we are compiling code for \( A \) rather than some unnamed principal.

**P-A:** This rule introduces infs into DKAL. This translates into the assumption that \( F \) holds.

**P-S:** Similar to the P-A knowledge assertion, this rule introduces the formula \( F \) into DKAL, however, with the corresponding cryptographic evidence. We assume formula \( F \) and create the cryptographic evidence. It is clear that \( F \) holds after the execution and since we tagged the tuple encoding formula \( F \) accordingly, the cryptographic evidence is well-formed.

We can apply Exp Appl and Exp Let extends the typing environment as expected.

We have just proven the induction hypothesis. We proceed to the induction step where we assume that the hypotheses to examined rule hold. More precisely, we assume that the current typing environment is appropriate with respect to sealing and that the all logical formulas occurring in the already processed DKAL derivation are entailed by the code generated so far.

**VER-SIG, VER-ZK:** By hypothesis, \( A \) knows the cryptographic evidence \( M \circ A \). Signatures are of type unit, as \( E \) is appropriate, \( E \vdash vkI : \text{VerKey} \); we can apply rule Exp Appl two times and derive that \( x_M' : \mathcal{T}_{vkI}\{\text{VerKey}/\alpha\} \). In particular, we know that the formula (taken from the DKAL rule) \( \text{ver}_{\text{sig}}(M, vkI, F) \) holds in \( A \)'s environment and the components \( x_M \) of \( x_M' \) now carry the strong type as stated in the verification key type.

The typing environment is still appropriate as we only added fresh variables to it.

**I-ZK-VER** Although logically different, the code is the same as for the rule VER-SIG.

**PP-K** We generate the code for the principal who performs some operations. Hence, it does not matter whether \( A \) or \( A^k \) is in possession of cryptographic evidence as the program in both cases is the same. We do not require any code for the PP-K rules.

**ENSUE, P-ZK** The ENSUE rule is the copula between the \( \Gamma \vdash - \)judgment and the knows modality and does not require any code.

**I-S-\( \land \), I-S-\( \lor \):** By induction hypothesis, the rule premises hold, i.e., \( \Gamma \vdash_S S_1 \) and (or) \( \Gamma \vdash_S S_2 \). Logically, it is clear that then the logical conjunction and the logical disjunction hold. This is solely a logical matter and does not require any F# code.

The typing environment is still appropriate as it did not change.
I-ZK-S: By hypothesis, A is in possession of the correct cryptographic evidence. Thus, we can call create_S on the values \( M_I \phi_A \). The code for create_S type-checks by as all arguments are unit or public (by Lemma 1); the interesting part is the according zero-knowledge proof verification that will yield strong types for its return value.

The typing environment is still appropriate as we only added fresh variables to it.

I-ZK-∃: By hypothesis, A possesses the cryptographic evidence \( x_M \) for statement \( S \). As for I-ZK-S, all types are occurring are public and consequently sub-type of unit.

The typing environment is still appropriate as we only added fresh variables to it.

E-ZK-∀: As our induction hypothesis states, A knows the cryptographic evidence for the logical conjunction of \( S_1 \land S_2 \). As with I-ZK-S and I-ZK-∃, all arguments are public and the function type-checks.

The typing environment is still appropriate as we only added fresh variables to it.

I-ZK-∀: The variables corresponding to the zero-knowledge proof stored in variables \( x_{S_1} \) and \( x_{S_2} \) are known to A by the hypothesis of the DKAL rule. The messages are all public and the function can type-check.

The typing environment is still appropriate as we only added fresh variables to it.

COMM-J: In this case, we assume that A is the receiver and B is the sender (as suggested by the communication rule).

A’s side: By induction hypothesis, the logical formula \( F_A \) holds in A’s typing environment and, as \( E \) is appropriate, we satisfy the preconditions for Lemma 10; we can be assured that the communication code type-checks. In particular, after the communication, the variable \( x_M' \) denotes a tuple for which the sent formula holds.

B’s side: By induction hypothesis, the logical formula \( F_B \) holds in B’s typing environment and, as \( E \) is appropriate, we satisfy the precondition for Lemma 10; we can be assured that the communication code type-checks.

After the code has executed, the typing environment is still appropriate as only fresh variables are added to it.

The compiler is defined to append () to the code of all principals; all code if of type unit.

Finally, it remains to show that we can create an appropriate typing environment. We use the following code:

\[
\text{Init}:
\begin{align*}
\text{import crypto-library} \\
\forall I : \\
\text{let} \ d_k : \text{DecKey} = \text{mkdk} () \text{ in} \\
\text{let} \ e_k : \text{EncKey} = \text{mkek} d_k \text{ in} \\
\text{let} \ s_k : \text{SigKey} = \text{mkseal} () \text{ in} \\
\text{let} \ (s'_k, v_k) = s_k \text{ in} \\
\forall S : \\
\text{let} \ S : (\text{unit seal}) = \text{mkseal} () \text{ in}
\end{align*}
\]

Lemma 11. Let \( C := \text{Init}; C' \) such that \( \text{fnfv}(C) \subseteq \{d_k, e_k, s_k, v_k \mid I \in \{I_1, \ldots, I_m\} \} \) and \( bn(C) \cap \{d_k, e_k, s_k, v_k \mid I \in \{I_1, \ldots, I_m\}\} = \emptyset \). If \( E \) is appropriate and \( E \vdash C' : T \), then \( \emptyset \vdash C : T \).

Proof. We do not consider bound variables here as consistent \( \alpha \)-renaming avoids any equality with elements from the domain of the typing environment.

Consecutively applying rule EXP LET yields a typing environment \( E \) that is appropriate. Since by assumption, the free names and the free variables of the continuation process \( C' \) do not coincide with the values bound in \( E \) (i.e., all keys), by assumption, \( E \vdash C' \).

The above lemma also shows how one can easily generate an appropriate typing environment. For actual applications, the encryption keys and the verification keys of individual principals should be send publicly on the network or be handed to a PKI. Since both encryption keys and verification keys are sub-type of unit, there are no type-checking problems.

Proof of Theorem 2. Corollary of Lemma 11 and Theorem 5 where \( U \) is instantiated as unit.

Proof of Theorem 3. Follows from Theorem 2 and Theorem 2 in [20].
G. Proofs

\[ T_{vk} := \sum_{k=1}^{n} p_k(x_1 : T_1 \ast \cdots \ast x_{\ell_k} : T_{\ell_k} \ast \{ y \text{ says } p_k(x_1, \ldots, x_{\ell_k}) \}) \]

\[ U := \sum_{k=1}^{n} p_k(x_1 : T_1 \ast \cdots \ast x_{\ell_k} : T_{\ell_k} \ast \{ \exists x, y. \ sk = (x, y) \land y \text{ says } p_k(x_1, \ldots, x_{\ell_k}) \}) \]

where \( T_i \in \{ \text{unit}, \alpha \} \)

\[ T'_{vk} := \sum_{k=1}^{n} p_k(x_1 : T_1 \ast \cdots \ast x_{\ell_k} : T_{\ell_k}) \]

where \( T_i \in \{ \text{unit}, \alpha \} \)

H. Malleable vs. Non-Malleable Zero-Knowledge Proofs

On malleable zero-knowledge proofs. So far, all zero-knowledge proofs with the exception of logical disjunction are malleable, i.e., one can disassemble (E-ZK-\( \land \)) and reassemble (I-ZK-\( \land \)) zero-knowledge proofs. Intuitively, one should not be able to disassemble a logical disjunction: disassembling \( F = F_1 \lor F_2 \) would result in valid proofs for \( F_1 \) and for \( F_2 \), respectively, and thus, \( F \) would have proven the logical conjunction \( F_1 \land F_2 \) in the first place.

Often non-malleability is a desirable feature, for instance, in online banking where the proof regarding valid credentials should not be separable from the transaction details. In the next paragraph, we describe how we can prevent the sort of malleability described above for logical conjunctions.

On non-malleable zero-knowledge proofs. As the Groth-Sahai proof scheme can be randomized [19], i.e., given a proof \( \Pi \), we can construct a different proof \( \Pi' \) that proves the same statement but is not copied, non-malleability as originally introduced by Sahai [51] cannot be achieved. Nonetheless, we can construct our proofs such that the rules to disassemble (e.g., E-ZK-\( \land \)) and assemble proofs (e.g., I-ZK-\( \land \)) cannot be applied anymore. In the following, we call such proofs non-malleable.

We can logically characterize the properties of non-malleable conjunctive statements by extending (i) the syntax of statements (cf. Table 1) with the new statement \( S_1 \land_n S_2 \) and (ii) the zero-knowledge deduction system (cf. Table 2) with the expected introduction rules.

I. Arithmetization, Logical Disjunction, and Non-Malleability

First, we describe the arithmetization for the logical disjunction and we show that it is well-formed. Then we show how to apply this technique to create a non-malleable zero-knowledge proof for the logical conjunction.
I.1. Proof of Logical Disjunction.

We assume that all our formulas $F$ are stated in disjunctive normal form, i.e.,

$$F := \bigvee_{i=1}^{n} F_i$$

where

$$F_i := \bigwedge_{j} f_{i,j} = t_{i,j}$$

The $f_{i,j} = t_{i,j}$ correspond to equations provable using the Groth-Sahai proof scheme (cf. Table 12), e.g., $f_{i,1} = t_{i,1}$ is a pairing product equation and $f_{i,2} = t_{i,2}$ is a multi-scalar equation. We introduce fresh variables $x_i$ and manipulate the equations as follows:

$$F' := \bigwedge_{i=1}^{n} x_i F_i$$

where

$$x_i F_i := \bigwedge_{j} x_i f_{i,j} = t_{i,j}$$

We detail this multiplication of equations with our fresh variables below in a dedicated paragraph.

For each fresh variable $x_i$, we now introduce the following quadratic equations in $\mathbb{Z}_n$:

$$x_i \cdot (1 - x_i) = x_i - x_i^2 = 0$$

More precisely, this means that either $x_i = 0$ or $x_i = 1$. By now, we have shown that a modified equation is holds (i.e., we could set $x_i = 1$) or that it does not hold (i.e., we must set $x_i = 0$ to vacuously make the equation true). What we need to show is that at least one of the equations was indeed true: we add the equation

$$\sum_{i} x_i = 1$$

This equation does not only show that at least one equation was true, it also makes the proof non-malleable. The reason is that all $x_i$ are hidden by the proof, i.e., the opening information are not known. On the other hand, the variables are mingled into the equations and cannot be removed. Thus if one wants to extend the set of equations, one would have to also create a fresh variable $x_s$, add the equation $x_s - x_s^2 = 0$, and add an equation of the form $x_s f_s = t_s$ (otherwise, the form deviates from what the receiver of the proof expects). These steps can be done by an attacker. However, this attacker would also have to create the equation

$$x_s + \sum_{i} x_i = 1$$

This is not possible as the $x_i$ and, in particular, the opening information of the correspond commitments is unknown to the attacker. Consequently, the result is a non-malleable proof of the formula $F$.

We choose the $x_i$ in the following way. There is at least one equation that holds as, otherwise, the formula is not satisfiable and due to the soundness properties of the Groth-Sahai scheme not provable. Without loss of generality, let the $i$-th equation hold. We set $x_i = 1$ and $x_j = 0$ for $j \neq i$.

**Multiplication of a Groth-Sahai equations with a variable.** The Groth-Sahai proof scheme can only deal with two unknown variables in the same monomial. For instance, the multi-scalar multiplication equations (cf. Table 12) can only deal with a variable of the respective group and one variable scalar. The intuitive reason is that the verification uses the bilinear map, which can only map two arguments into the target group. Consequently, when multiplying an equation with a variable, it may happen that we create terms that must deal with more than two variables.

We solve this problem by introducing new equations that combine two unknowns $x$ and $y$ into a third variable $z$. This equation shows that $z$ equals the combination of the other two. Thus, we can substitute with $z$ whenever we obtained too
many variables in a term. More precisely, we modify the equations in the following way.

\[
\begin{align*}
\text{New equation:} & \quad xA - A' = 0 \\
\text{New equation:} & \quad xB - B' = 0 \\
\text{New equation:} & \quad xY - Y' = 0 \\
\text{New equation:} & \quad xy - y' = 0 \\
\text{New equation:} & \quad bxX = 0 \\
\text{New equation:} & \quad \gamma y'z = 0 \\
\end{align*}
\]

The number of new formulas depends not on the number of equations but on the length, i.e., the number of terms of an equation. In particular, we get an amount of new equations linear in the sum of the lengths of all equations.

**Proof of the non-malleable logical conjunction.** Proving the logical conjunction well-formed is straight-forward now. The multiplication of all equations with fresh variables is the same as for the logical disjunction. The additional equations all state that the variables chosen are indeed 1, i.e., for all fresh variables \( x_i \), we add the equation \( x_i = 1 \). The non-malleability is achieved by adding the equation

\[
\sum_i x_i = n
\]

where \( n \) is the total number of conjunctive equations.
Table 10 Excerpt of the author rules in the distributed reviewing system.

<table>
<thead>
<tr>
<th>K-A</th>
<th>Authors: vk_Authors says Subm(Authors, paper) ver_sig(M_Subm, vk_Authors, Subm(Authors, paper))</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-A</td>
<td>Authors: vk_Authors says Rebuttal(paper, rebuttal) ver_sig(M_Rebuttal, vk_Authors, Rebuttal(paper, rebuttal))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C-A</th>
<th>Authors: if true then Authors receives $y_i'$ says Rev(paper, $x_{rev}$, $y_i$) from ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-A</td>
<td>Authors: if Rev(paper, $x_{rev}$, $y_i$) then Authors sends Authors says Rebuttal(paper, rebuttal) to $y_i$</td>
</tr>
</tbody>
</table>

Table 11 Excerpt of the reviewer rules in the distributed reviewing system.

<table>
<thead>
<tr>
<th>K-A</th>
<th>id : vk_id says Rev(paper, rev, id$^1$) ver_sig(M_Rev, vk_id, Rev(paper, rev, id$^1$))</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-A</td>
<td>id : if true then id$^1$ sends Rev(paper, rev, id$^1$) to Authors</td>
</tr>
<tr>
<td>C-A</td>
<td>id : if true then id$^1$ receives Authors says Rebuttal(paper, y$^1$_Rebuttal) from Authors</td>
</tr>
</tbody>
</table>

Table 12 Provable equations using the Groth-Sahai proof scheme.

**Pairing product equations:**

$$
\prod_{i=1}^{n} e(A_i, Y_i) \cdot \prod_{i=1}^{m} e(X_i, B_i) \cdot \prod_{i=1}^{m} \prod_{j=1}^{n} e(X_i, Y_j)^{\gamma_{ij}} = t_T
$$

**Multi-scalar multiplication equations in $G_1$:**

$$
\sum_{i=1}^{n} y_i A_i + \sum_{i=1}^{m} b_i X_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \gamma_{ij} y_j X_i = T_2
$$

**Multi-scalar multiplication equations in $G_2$:**

$$
\sum_{i=1}^{n} a_i Y_i + \sum_{i=1}^{m} x_i B_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \gamma_{ij} x_i Y_j = T_2
$$

**Quadratic equations in $Z_n$:**

$$
\sum_{i=1}^{n} a_i y_i + \sum_{i=1}^{m} x_i b_i + \sum_{i=1}^{m} \sum_{j=1}^{n} \gamma_{ij} x_i y_j = t
$$