Security and Privacy by Declarative Design

Matteo Maffei  Kim Pecina  Manuel Reinert
Saarland University
Germany
{maffei,pecina,reinert}@cs.uni-saarland.de

Abstract—The privacy of users has rapidly become one of the most pervasive and stringent requirements in distributed computing. Designing and implementing privacy-preserving distributed systems, however, is challenging since these systems also have to fulfill seemingly conflicting security properties and system requirements: e.g., authorization and accountability require some form of user authentication and session management necessarily involves some form of user tracking.

In this work, we present a solution based on declarative design. The core component of our framework is a logic-based declarative API for data processing that exports methods to conveniently specify the system architecture and the intended security properties, and conceals the cryptographic realization.

Invisible to the programmer, the implementation of this API relies on a powerful combination of digital signatures, non-interactive zero-knowledge proofs of knowledge, pseudonyms, and reputation lists. We formally proved that the cryptographic implementation enforces the security properties expressed in the declarative specification.

The systems produced by our framework enjoy interoperability and open-endedness: they can easily be extended to offer new services and cryptographic data can be shared and processed by different services, without requiring any extra bootstrapping phase or interaction among parties.

We implemented the API in Java and conducted an experimental evaluation to demonstrate the practicality of our approach.

I. INTRODUCTION

Respecting the privacy of user data is rapidly becoming a crucial, and often compulsory, requirement for virtually any distributed infrastructure. Media are increasing awareness of privacy issues and the legal landscape appears to be shifting towards a model that promotes consumer control over personal data and imposes stringent requirements on the treatment of such data by third parties.

There is an increasing consensus that privacy and data protection should be embedded throughout the entire life cycle of technologies, from the early design stage to their deployment, as opposed to be achieved through ad-hoc add-ons. This concept, known as privacy by design [1], has recently been supported by the Federal Trade Commission [2] and the European Commission [3]. Making privacy a cornerstone in the design phase promises, among the other benefits, a smooth integration of privacy enhancing technologies in the overall system architecture, which avoids expensive retrofits or unnecessary tradeoffs between functionality and user privacy, flexibility in the choice of the privacy policies, and the possibility to give customers control over the usage and collection of their personal data.

At present, however, there is no standardized and generally applicable methodology for designing privacy-preserving distributed systems. This has led to a variety of solutions, whose heterogeneity hinders the interoperability of independently developed systems and whose ad-hoc and rigid nature makes system extensions and refinements prohibitive. A general-purpose design methodology would have the potential to dramatically change the landscape, promising better robustness, flexibility, and interoperability. Developing such a methodology is challenging for three fundamental reasons.

Security versus privacy. A generally applicable design methodology for privacy-preserving distributed systems requires the development of sophisticated and carefully designed cryptographic protocols to reconcile the privacy of users with other seemingly contradictory security requirements, such as authorization policies and accountability, or system functionalities, such as linkability of user actions (e.g., to implement pay-per-usage or access-only-once policies). How to make sure that the principal trying to access a sensitive resource is authorized if this principal is not willing to share any personal identifying information? How to link user actions without jeopardizing the privacy of users? How to hold misbehaving users accountable for their actions without compromising the privacy of honest users?

General applicability and efficiency. The cryptographic realization should guarantee all the aforementioned security guarantees without putting restrictions or assumptions on the structure of the system (e.g., the presence of a TTP) and without hampering the system performance (e.g., by requiring additional bootstrapping phases or interactions among parties). Furthermore, the cryptographic framework should allow for the extension of the system with new components (open-endedness) and the sharing of data among them (interoperability).

Sound and convenient development workflow. Finally, developing a generally applicable and efficient cryptographic infrastructure is not enough. Implementing distributed programs based on advanced cryptographic schemes is highly error-prone, as witnessed by the number of attacks on largely deployed cryptographic protocol implementations (see, e.g.,
controlled linkability, and accountability.

Intuitively, authorization policies are formalized by logical formulas, and authorization credentials as well as any other information exchanged by parties are expressed as proofs of validity of logical formulas (e.g., an access credential released by $B$ to $A$ is expressed as a proof of $B$ says Auth($A$)).

Proofs can be manipulated, e.g., by existentially quantifying sensitive arguments, thus capturing privacy requirements, and by combining them in conjunctive form with other proofs (e.g., after receiving the access credential, $A$ can produce a proof of $\exists r. B$ says Auth($x$) ∧ $x$ says Access($r$) to access resource $r$ anonymously).

We rely on service-specific pseudonyms (SSPs) to enable controlled linkability of user actions without revealing user identities. In a nutshell, SSPs are pseudonyms (i.e., they are bound to their owner and they protect her identity) with two service-oriented properties: uniqueness, i.e., each user has exactly one pseudonym for each service, and unlinkability, i.e., it is not possible to link the pseudonyms deployed by a user for different services. In other words, service-specific pseudonyms provide intra-service linkability and inter-service unlinkability of user actions. Revealing the pseudonym suffices to count and impose a limit on the number of access requests from the same user within a service, without necessarily revealing her identity and without tracking users across different services (e.g., $A$ can send $B$ a proof of $\exists x. B$ says Auth($x$) ∧ $x$ says Access($r$) ∧ SSP($x$, $s$, psd), where $s$ is the service and psd is $A$’s pseudonym for $s$).

Accountability is enforced by reputation lists, which are maintained by service providers and bind user pseudonyms to scores: using membership (or non-membership) proofs, users can prove whether or not a pseudonym belongs to a certain list and, if so, claim the corresponding score. For instance, assume that access requests are granted in service $s$ only to users who are listed in $L$ with a score of at least 5: $A$ can send $B$ a proof of $\exists x, y, z. B$ says Auth($x$) ∧ $x$ says Access($r$) ∧ SSP($x$, $s$, psd) ∧ SSP($x$, $s'$, $y$) ∧ ($y, z$) ∈ $L$ ∧ $z$ ≥ 5. SSP-based reputation lists constitute an effective way to ban misbehaving users from the system or to reward well-behaving ones without revealing their identities. We finally show how to extend our architecture with an identity escrow protocol, which allows an escrow agent to reveal the identity of misbehaving users.

**Cryptographic library.** Authorization credentials are realized by automorphic signatures [6] and existentially quantified logical formulas by Groth-Sahai zero-knowledge proofs [7]. We developed new protocols for service-specific pseudonyms and identity escrow that are compatible with Groth-Sahai zero-knowledge proofs.

Our cryptographic framework is generally applicable, since it does not make assumptions on the system architecture, does not involve any TTP (except for the optional one responsible for identity escrow), and does not require any extra bootstrapping phase or interaction among parties. Thanks to the malleability properties of the zero-knowledge scheme, the systems produced by our framework can be easily extended to offer new services (open-endedness) and independently developed systems can interoperate and share data with each other (interoperability). We are not aware of any existing cryptographic infrastructure that supports all the aforementioned security properties and system requirements.

**Soundness results.** We deploy a security type system for cryptographic implementations [8] to prove that the cryptographic realization enforces the authorization policies specified by the programmer even in an adversarial setting. Furthermore, we provide security proofs for the cryptographic schemes introduced in this paper, including proofs of anonymity for service-specific pseudonyms and for the identity escrow protocol.

**A. Overview of our Framework**

In this work, we present a new approach for the declarative specification and automated synthesis of privacy-preserving distributed systems.

**Declarative API.** The system is specified in a regular programming language, which we equip with a logic-based, declarative API for data processing. This API hides the cryptographic details and allows the programmer to conveniently specify the overall system architecture and the variety of security requirements such as authorization, privacy, controlled linkability, and accountability.

Outline. The remainder of this paper is organized as follows. **Section II** introduces the declarative API and **Section III** details the cryptographic realization thereof. **Section IV** proves the security of the cryptographic primitives.
introduced in this paper. Section VI formally establishes the connection between logical formulas and zero-knowledge proofs. Section VII overviews the implementation of our framework in Java and Section VIII presents the performance evaluation. Section IX concludes and gives directions for future research.

The long version of this paper is available online [9].

II. DECLARATIVE API

This section introduces our security-oriented, declarative API for the design of distributed systems. For easing the presentation and the formulation of the soundness results, we instantiate the API in ML. We remark, however, that the API is language-independent and can easily be implemented in any other programming language.

Inspired by prior work on information logics for distributed systems [10], [11], the programming abstraction we propose represents the information known to principals as logical formulas and the messages exchanged by parties as validity proofs for logical formulas. Our framework is independent of the choice of the logic: we just assume the presence of the “says” modality that binds logical formulas to principals and is a common ingredient of existing authorization logics, such as SecPal [12], Aura [13], and BL [14].

Table 1 illustrates the methods composing our API, along with the respective functional types. We describe these methods below, classifying them according to the security property they capture.

A. Authorization

Example 1. As a running example, we design a collaborative platform that combines two services. In the first service, a patient receives a certificate from the doctor attesting her visit and including additional information such as the date of the visit and the results of the examination. In the second service, the patient uses this information to evaluate her attending doctor on a rating platform such as Jameda [15] or Healthgrades [16]. We assume the following authorization policy for the rating platform RevSys that allows a patient to evaluate only her treating doctors:

\[
\forall Pat, Doc, subject, results, date, opinion. \\
Doc \text{ says } Visit(Pat, date, results) \land Pat \text{ says } Rating(opinion) \\
\implies Rated(Doc, opinion)
\]

Formulas are encoded as terms of the language (in ML, using data-type constructors): for the sake of readability, here and throughout the paper we use the standard logical notation. In the first service, the doctor Doc provides the patient Alice with a validity proof of his medical license, vouched for by the hospital Hosp, and an attestation of Alice’s visit, i.e., a proof for the formula Hosp says IsDoc(Doc, PI\text{Doc}) \land Doc says Visit(Alice, date, results). To express her opinion happy about the doctor, Alice submits a validity proof for the formula Doc Visit(Alice, date, results) \land Pat says Rating(happy).

Each user u has two identifiers: a private one of type uid that is used to refer to the principal executing a certain piece of code, and a public one of type uid\text{pub} that is used to refer to other principals. The function mkId takes as input a string, e.g., the name, and returns a pair of private and public identifiers.

The function mkSays y f takes the private identifier y of the user running the code, a formula f, and returns a validity proof for the predicate x says f, where x is the public identifier corresponding to y.

The API provides methods to manipulate proofs, which is crucial for the expressiveness of our framework. The function mk\wedge takes as input a proof of f_1 and a proof of f_2, and it returns a proof of f_1 \land f_2. This function, as well as the other API functions, raises an exception if the input is not of the expected form (in this case, a pair of validity proofs).

Conversely, the function split\wedge takes a proof of f_1 \land f_2, and returns a proof of f_1 and a proof of f_2. The function mk\vee takes as input a proof of f_1 and a formula of the form f_1 \lor f_2 or f_2 \lor f_1, and it returns a proof of the specified disjunctive statement. Due to our cryptographic implementation, the construction of a disjunction is only possible if the hide function has not been applied to the input proof yet, i.e., no argument is existentially quantified (see Section III). The function extractForm takes as input a proof and returns the corresponding formula. Finally, the function verify takes as input a proof and a formula, and it checks that the former is a proof of the latter.

Example 2. The code for the patient is shown below:

\[
\begin{align*}
&s.1 \text{ let } Pat \ xPat \ yPat \ xHosp \ xDoc \ xPI \text{Doc} \ xresults \ xdate \\
&s.2 \text{ let } xopinion \ xaddrPat \ xaddr\text{Doc} \ xaddr\text{results} = \\
&s.3 \text{ let } c = \text{ listen} xaddrPat; \text{ let } y = \text{ recv} c; \\
&s.4 \text{ if } \text{ verify} y \left( xHosp \text{ says } IsDoc(xDoc, xPI \text{Doc}) \land xDoc \text{ says } Visit(xPat, xdate, xresults) \right) \text{ then } \\
&s.5 \text{ let } (pf,IsVis\text{Pat}) = \text{ split} x, y; \\
&s.6 \text{ let } pf_x = \text{ mkSays yPat } Rating(xopinion); \\
&s.7 \text{ let } pf' = \text{ mk\wedge pf_vis}\text{Pat} pf_x; \\
&s.8 \text{ let } c' = \text{ connect} xaddr\text{results}; \text{ send pf } c'
\end{align*}
\]

The code is concise and self-explanatory: the patient receives a proof for the attestation of her visit from the doctor and constructs the rating proof. She combines the proof of her visit (obtained by splitting the proof received by the doctor apart) and the rating proof in conjunctive form. Finally, she sends the resulting proof to the rating platform. The communication functions such as listen are the standard communication primitives available in any language.

B. Privacy

The function hide allows for hiding sensitive arguments, which, as originally proposed by Maffei and Pecina [17], can be logically captured by existential quantification. This function takes as input a proof of f and a formula f'
obtained from $f$ by existentially quantifying some of the arguments, and it returns a proof of $f'$.

**Example 3.** The doctor certainly does not want the patient to know her personal identification number $PI_Doc$. Hence, she sends a proof in which this particular information is hidden. This changes the call to verify:

\[
\begin{align*}
\ldots \\
\text{if verify } y \quad \begin{cases} \\
\exists \exists PI_Doc. \\
X_{\text{Hosp}} \text{ says } ISDoc(X_{\text{Doc}}, W_{PI_Doc}) \\
X_{\text{Doc}} \text{ says } VISit(X_{\text{Pat}}, X_{\text{Date}}, X_{\text{List}})
\end{cases} \\
\ldots
\end{align*}
\]

Additionally, the patient might desire to submit her evaluation anonymously. She can do so by existentially quantifying her identity, the results, and the date, which is achieved by the following piece of code:

\[
\begin{align*}
\ldots \\
\text{let } pf' = \text{hide}(pf) \\
\ldots
\end{align*}
\]

where $pf$ is the proof produced in line [87] of the code shown in [Example 2]. This proof suffices to convince the rating platform of the patient’s evaluation for the doctor $Doc$ and, from a logical perspective, to entail the predicate Rating(opinion).

**C. Controlled Linkability**

The previous example suggests that hiding the identity of users may hinder the enforcement of meaningful authorization policies. For instance, in order to avoid biased results, we would like to make sure that patients cannot submit more than one evaluation. In general, there may be the need for the service provider to link the actions of the users, which should be achieved without making user actions linkable across different services. We rely on service-specific pseudonyms (SSPs) to achieve this goal: each user can create at most one valid SSP per service, which provides intra-service linkability, while her pseudonyms cannot be linked and tracked across different services, which provides inter-service unlinkability. Since SSPs hide the identity of their owner, they can be revealed; a simple comparison suffices to determine whether the user behind a given pseudonym is using a service for the first time or not. Notice that the service structure determines the degree of unlinkability offered to each user: increasing the number of services (e.g., by splitting a service) limits the tracking of user actions and provides stronger unlinkability guarantees.

The function $mkSSP$ takes as input the private user identifier $y$ and the service identifier $s$, and it returns a proof of the predicate $SSP(x, s, psd)$, which states that $psd$ is the pseudonym for the public identifier $x$ and the service $s$.

**Example 4.** We set the service structure so as to reflect doctor specializations. Assume that the doctor who visited the patient is an internist. Then the patient can extend the proof $pf$ produced in [Example 2] to accommodate both privacy and linkability requirements as follows:

\[
\begin{align*}
\ldots \\
\text{let } pf_{\text{ssp}} = mkSSP(y_{\text{Pat}}, x_{\text{Internist}}); \\
\text{let } s = \text{extractForm}(pf_{\text{ssp}}); \\
\text{match } s \text{ with SSP}(x_{\text{Pat}}, x_{\text{Internist}}, x_{\text{psd}}) \Rightarrow \\
\text{let } pf_{\wedge} = mk\wedge pf_{\text{ssp}}; \\
\text{let } pf' = \text{hide}(pf) \\
\ldots
\end{align*}
\]

Notice that the existential quantification binds all occurrences of the patient identifier, including the one in the SSP predicate. The rating platform can discard multiple evaluations by simply checking the pseudonyms conveyed by each proof.

**D. Accountability**

SSPs are designed to prevent the tracking of users across different services. In many applications, however, it is desirable to ban misbehaving users from the whole system or to reward well-behaving ones. We use reputation lists to achieve this kind of accountability requirements without disclosing user identities.

A reputation list binds SSPs to attributes. For the sake of simplicity, we assume that each reputation list refers to a specific service $s$ and contains pairs of the form $(psd, attr)$, where $psd$ is a pseudonym for service $S$ and $attr$ is an attribute. We could easily support lists referring to several
services and binding pseudonyms to several attributes, but this would significantly complicate the presentation without adding any theoretically interesting insight.

The function \( \text{mkLM} \) takes as input a pseudonym \( psd \), an attribute \( attr \), and a reputation list \( \ell \), and it returns a proof of \( (psd, attr) \in \ell \). The function \( \text{mkLN} \) takes as input a pseudonym \( psd \) and a reputation list \( \ell \), and it returns a proof of \( (psd, \_ \notin \ell) \), where \( \_ \) serves as wildcard.

The function \( \text{mkREL} \) takes as input a formula describing an arithmetic relation between attributes and returns the corresponding proof. We support arithmetic relations of the form \( b \, op \, b' \), with \( op \in \{ >, \geq, <, \leq, =, \neq \} \).

**Example 5.** We maintain a reputation list for each service (i.e., doctor specialization). This list contains the pseudonyms of the patients uploading offensive comments in the associated service. In order to prevent such patients from further participating in evaluation procedures, we require patients to prove that their pseudonyms have not been included in any of such lists. We show below how to extend the proof from [Example 4]. For simplicity, we focus on just one reputation list \( x_t \) for the service \( X_{Dentist} \), since the extension to multiple subjects is straightforward.

... \let pf''_{ssp} = \text{mkSSP} \ y_{Pat} \ x_{Dentist}; \let s' = \text{extractForm} \ pf''_{ssp}; \match s' \ with \ \text{SSP}(x_{Pat}, x_{Dentist}, x'_{psd}) \rightarrow \let pf''_{idr} = \text{mkLN} \ x'_{psd} \ x_t; \let pf''_{ES} = \text{mk}_{\lambda} \ (pf''_{ES}, pf''_{ssp}) \ \phi_{g}; \let pf'' = \text{hide} \ pf''_{ES} \lbrack \begin{array}{l}
  \exists w_{Pat}, w_{date}, w_{results}, w_{pad'}, w_r, x_{Doc} \ 
  \quad \text{says Visit}(w_{Pat}, w_{date}, w_{results}) \wedge \\
  \quad w_{Pat} \text{ says Rating}(x_{opinion}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Internist}, x_{pad}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Dentist}, w_{pad'}d) \\
  \quad (w_{pad'}, \_ \notin x_t) \end{array} \rbrack; \let pf'' = \text{hide} \ pf''_{ES} \lbrack \begin{array}{l}
  \exists w_{Pat}, w_{results}, w_{pad'}, w_r, x_{Doc} \ 
  \quad \text{says Visit}(w_{Pat}, w_{date}, w_{results}) \wedge \\
  \quad w_{Pat} \text{ says Rating}(x_{opinion}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Internist}, x_{pad}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Dentist}, w_{pad'}d) \\
  \quad (w_{pad'}, \_ \notin x_t) \end{array} \rbrack\rbrack.

The patient pseudonym for \( x_{Dentist} \) is existentially quantified, which makes patient evaluations unlinkable across different doctor specializations.

Finally, we remark that a user can in principle obtain multiple pseudonyms for a service if she registers several user identifiers with the corresponding provider. Notice, however, that the registration phase is not anonymous (see [Example 1]) and the service provider has to willingly register users multiple times.

**E. Identity Escrow**

In some scenarios, it is desirable to have a mechanism to reveal the identity of misbehaving users, e.g., if the user severely violated certain regulations or if she even committed a crime. We can achieve that in our framework by means of an identity escrow mechanism.

The user initially contacts the trusted third party \( EA \) acting as an escrow agent, which provides the user with a proof of the predicate \( EA \ says \ Escrowld(u_{idpub}, r) \), where \( u_{idpub} \) is the public identifier of the user and \( r \) is a number chosen by \( EA \) to identify the user.

The user creates an escrow proof by means of the \( mkIDRev \) function. This function takes as input the proof received from \( EA \) and the service, and it returns a proof of the predicate \( Escrowld(idr, x_r) \) and \( Escrowld(idr, x_r) \rightarrow \text{ScHW}(x_{idpub}, x_r) \), where \( idr \) is the user’s escrow identifier for the service \( s \). Given \( idr \) and \( s \), \( EA \) and only \( EA \) can extract the identity of the user. Thus, the user has simply to send a proof of \( \exists u_{idpub}, x_r \ says \ Escrowld(u_{idpub}, x_r) \wedge \ Escrowld(idr, x_r) \wedge \ Escrowld(idr, x_r) \) to the service provider, which hides the user’s identity and the value \( r \). Since, similarly to pseudonyms, the escrow identifiers of a user are unlinkable across different services, the identity escrow protocol preserves the inter-service unlinkability of user actions.

We stress that requiring a user action to enable the identity escrow service is an intentional feature of the API: the user has to give her explicit consent to engage in a service in which her anonymity might in principle be compromised.

**Example 6.** We show below how to extend the proof from [Example 5], assuming that \( pf'_{EA} \) is the proof that the patient previously received from the rating platform acting as an escrow agent.

... \let pf'_{escrow} = mkIDRev pf'_{EA} x_{Internist}; \let s' = \text{extractForm} \ pf'_{escrow}; \match s' \ with \ Escrowld(x_{Pat}, x_{Internist}, x_{idr}) \rightarrow \let pf''_{ES} = \text{mk}_{\lambda} \ (pf'_{ES}, pf'_{escrow}) \lbrack \begin{array}{l}
  \exists w_{Pat}, w_{date}, w_{results}, w_{pad'}, w_r, x_{Doc} \ 
  \quad \text{says Visit}(w_{Pat}, w_{date}, w_{results}) \wedge \\
  \quad w_{Pat} \text{ says Rating}(x_{opinion}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Internist}, x_{pad}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Dentist}, w_{pad'}d) \\
  \quad (w_{pad'}, \_ \notin x_t) \end{array} \rbrack; \let pf'' = \text{hide} \ pf''_{ES} \lbrack \begin{array}{l}
  \exists w_{Pat}, w_{results}, w_{pad'}, w_r, x_{Doc} \ 
  \quad \text{says Visit}(w_{Pat}, w_{date}, w_{results}) \wedge \\
  \quad w_{Pat} \text{ says Rating}(x_{opinion}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Internist}, x_{pad}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Dentist}, w_{pad'}d) \\
  \quad (w_{pad'}, \_ \notin x_t) \end{array} \rbrack\rbrack; \let pf'' = \text{hide} \ pf''_{ES} \lbrack \begin{array}{l}
  \exists w_{Pat}, w_{results}, w_{pad'}, w_r, x_{Doc} \ 
  \quad \text{says Visit}(w_{Pat}, w_{date}, w_{results}) \wedge \\
  \quad w_{Pat} \text{ says Rating}(x_{opinion}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Internist}, x_{pad}) \wedge \\
  \quad \text{SSP}(w_{Pat}, x_{Dentist}, w_{pad'}d) \\
  \quad (w_{pad'}, \_ \notin x_t) \end{array} \rbrack\rbrack.

**F. Open-endedness**

We finally remark that our API is well-suited for the development of open-ended systems, i.e., systems that can be extended with services sharing resources and interoperating with each other.

**Example 7.** We introduce an online pharmacy service (e.g., Medco [13]) that delivers medicines on request. To this end, the patient appends the order to the doctor’s attestation of her visit, hiding the doctor’s identity. Formally, she sends a validity proof of the following formula to the pharmacy:

\[ \exists w_{Doc}, w_{IPDoc}, x_{Hosp} \ says \ IsDoc(w_{IPDoc}, w_{Doc}) \wedge \ w_{Doc} \ says \ Visit(x_{Pat}, x_{date}, x_{results}) \wedge x_{Pat} \ says \ Buy(medicine) \]

**III. CRYPTOGRAPHIC REALIZATION**

Developing a cryptographic realization of the API described in Section II is a challenging task. Following prior work [11], our cryptographic realization builds on digital signatures and zero-knowledge proofs.
We cryptographically implement private user identifiers as handles to the corresponding signing keys. The keys themselves are not accessible by the interface and, thus, are invisible to the programmer. In particular, programmers cannot accidentally leak signing keys. The storage medium for the signing key is chosen depending on the security requirements: signing keys can be stored in files protected by the operating system or, to achieve better security guarantees, in cryptographic devices capable of computing digital signatures (e.g., cryptographic coprocessors [19]).

Public user identifiers are realized as verification keys and we rely on a public-key infrastructure (PKI) to bind users to their key. Section II implements a decentralized PKI that resembles webs of trust where the hospital vouches for the doctor, who in turn vouches for the user. A proof for the predicate $\exists x. B$ says $\text{Auth}(x) \land x$ says $\text{Access}(r)$ is implemented as a non-interactive zero-knowledge proof of knowledge [18] of two signatures such that the former is on (the bit-string encoding of) the predicate $\text{Auth}(\alpha)$ and verifies with $B$’s verification key, while the latter is on the predicate $\text{Access}(r)$ and verifies with some value $\alpha$. The verification key $\alpha$ (and the signature) is not revealed by the proof, thus achieving the anonymity of the requester.

Specifically, we adopt the automorphic signature scheme by Abe et al. [6] and the Groth-Sahai zero-knowledge proof scheme [7]. The former allows for signing verification keys without any extra encoding, which is crucial to realize efficient zero-knowledge proofs of predicates of the form $\exists x. B$ says $\text{Auth}(x) \land x$ says $\text{Access}(r)$, where the hidden verification key is both signed and used to verify a signature. The latter provides malleable zero-knowledge proofs, i.e., proofs that can be transformed to hide some of the witnesses, combined in conjunctive form, and so on. In order to achieve the other security properties supported by our API, we develop new cryptographic realizations of service-specific pseudonyms and of the identity escrow protocol that are fully compatible with the Groth-Sahai proof scheme.

In the following, we detail the cryptographic constructions used in the implementation of our API.

A. Cryptographic Realization of API Methods

Bilinear map. Elliptic curves with a bilinear map constitute a crucial building block of the cryptographic realization. The bilinear map is a function $e : G_1 \times G_2 \rightarrow G_T$ that takes as input two values from the groups $G_1$ and $G_2$, respectively, and returns a value in the group $G_T$. For our particular setup, we require that $|G_1| = |G_2| = |G_T| = p$, where $p$ denotes a large prime, and that $e$ is a type III pairing [21], i.e., $G_1 \neq G_2$ and there is efficiently computable homomorphism between $G_1$ and $G_2$. From $e$, we require that

1) $e$ is efficiently computable;
2) $e$ is bilinear (linear in both arguments), i.e., the condition
\[
\forall a, b \in \mathbb{Z}_p, \mathcal{X} \in G_1, \mathcal{Y} \in G_2 :
\]
\[
e(\mathcal{X} + a, \mathcal{Y} + b) = e(\mathcal{X} + a, \mathcal{Y})^b = e(\mathcal{X}, \mathcal{Y})^b
\]
holds;
3) $e$ is non-degenerate, i.e., if $\langle G \rangle = G_1$ and $\langle H \rangle = G_2$, then $\langle e(G, H) \rangle = G_T$.

In cryptography, groups are mostly written in multiplicative form $g^x$. For historical reasons, elliptic curves are written in additive form $xG$. We use the convention that elliptic curves are written in additive form; all other groups such as $G_T$ are written in multiplicative form. Here and throughout the rest of the paper, we let $G$ and $H$ denote the distinguished generators of $G_1$ and $G_2$, respectively. Furthermore, we let $p$ denote a large prime, calligraphic uppercase letters denote elliptic curve elements, lowercase letters denote elements from $\mathbb{Z}_p$.

Commitments. Commitments are an essential building block for the Groth-Sahai zero-knowledge proof scheme. Intuitively, a commitment is the digital equivalent of a message in a closed envelope lying on top of a table. The creator of the message cannot change it and no one can look inside until it is opened.

More formally, a principal commits to a value $x$ by applying the randomized commitment function to obtain a commitment $C_x$ on $x$ along with the so-called opening information $O$. Opening $C_x$ requires the opening information $O$, and $C_x$ itself. In our case, the opening information is the committed value $x$ and the randomness $r$ used in the commitment.

We use ElGamal encryptions as commitments in order to obtain proofs of knowledge. A proof of knowledge is formalized by a knowledge extractor [22] that, given a zero-knowledge proof, can extract the witnesses hidden by a zero-knowledge proof. Since instances of ElGamal encryptions [23] naturally have a decryption key, this key allows a knowledge extractor to open all commitments and to extract all values used in a zero-knowledge proof, including the hidden ones. The commitment scheme relies on the decisional Diffie-Hellman (DDH) assumption. Throughout the remainder of this paper, we let $C_x$ denote a commitment to value $x$ and $\|C\|$ the value committed to in $C$.

Groth-Sahai zero-knowledge proof scheme. We deploy the Groth-Sahai proof system [7] since it is a highly flexible and general scheme. Groth-Sahai proofs are non-interactive zero-knowledge proofs of knowledge, which capture relations among committed values that involve elliptic curve operations and bilinear map applications. For instance, the
equation \([C_x] \cdot [C_y] = [C_z]\) states that the value committed to in \(C_x\) multiplied by the value committed to in \(C_y\) equals the value committed to in \(C_z\), where \(c \cdot V\) denotes the scalar multiplication of \(c\) by \(V\). In general, Groth-Sahai proofs fulfill only the weaker notion of witness-indistinguishability. Our equations, however, are of a special form for which Groth-Sahai proofs are also zero-knowledge.

A Groth-Sahai proof on its own solely states that some values contained inside commitments satisfy a given set of equations. The expressive power of the Groth-Sahai scheme stems from the capability to selectively reveal and hide values occurring in these equations. For instance, if the proof for the equation above contains the opening information for values occurring in these equations. For instance, if the proof for the equation above contains the opening information for \(C_y\) and \(C_z\), then this proof shows the knowledge of the discrete logarithm \(x\) of \(H\) to the basis \(G\), keeping the discrete logarithm \(x\) hidden. Naturally, values can be hidden by removing the respective opening information from a proof. The zero-knowledge property ensures that no information about the hidden witnesses can be learned by the verifier, which faithfully captures the privacy property expressed by existential quantification.

Since Groth-Sahai proofs show the validity of a set of equations, concatenating two proofs shows the validity of the union of the equations proven by the two individual proofs; separating the set of proven equations creates two proofs, each showing the validity of its share of the equations. Realizing a logical disjunction is significantly more challenging since such a proof must hide which branch is valid. We use arithmetization techniques [7] but the prover must have all values appearing in the proof at her disposal. This explains why function mkv succeeds only if the proof passed as input has not previously been processed by function hide (see Section II-A). Finally, we mention that the Groth-Sahai scheme relies on a common reference string (CRS). We assume a global, trustworthy CRS. Such a CRS can be created by a TTP or by a distributed community effort, e.g., using secure multiparty computation schemes.

The Groth-Sahai proof system can be set up on different assumptions. We use the instantiation based on the SXDH assumption [7], i.e., the DDH problem is intractable in \(\mathbb{G}_1\) and \(\mathbb{G}_2\).

Automorphic signature scheme. We use the automorphic digital signature scheme proposed by Abe et al. [6]. This scheme is highly efficient and allows us to sign verification keys without encoding them. As previously mentioned, this is crucial to obtain efficient zero-knowledge proofs.

A verification key is of the form \(vk = x \cdot G\), where \(x \in \mathbb{Z}_p\) is the randomly-chosen signing key corresponding to \(vk\) and \(G\) is the public generator of \(\mathbb{G}_1\) (see the description of bilinear maps above). Here and throughout the remainder of the paper, we use the notation \(e \in R\) \(S\) to denote that element \(e\) is chosen uniformly at random from the set \(S\). Furthermore, the scheme is fully compatible with the Groth-Sahai proof system: we write

\[
\text{ver}([C_{\text{sig}}], [C_m], [C_{vk}])
\]

to denote a zero-knowledge proof showing that the value committed to in \(C_{\text{sig}}\) is a signature on the value committed to in \(C_m\), which can be verified using the verification key committed to in \(C_{vk}\) [6]. This proof realizes a proof for the formula \([C_{vk}]\) says \([C_m]\) and can be fine-tuned to open any of these commitments, revealing the respective values.

The digital signature scheme by Abe et al. is existentially unforgeable under chosen-message attacks, the standard notion of security for signature schemes. Its security relies on the \(q\text{-ADH-SDH}\) assumption [6] and the AWF-CDH assumption. The AWF-CDH assumption is implied by the SXDH assumption (see [6], Lemma 1).

Hashing into \(\mathbb{G}_1\). There are several ways to define hash functions \(h\) that maps arbitrary strings into the group \(\mathbb{G}_1\). The most straightforward way is to use any ordinary hash functions such as SHA1 and let \(h(x) := \text{SHA1}(x) \cdot G\). The drawback of this method is that it reveals the discrete logarithm of the hash value with respect to \(G\), which will break the security of our service-specific pseudonyms. There are several schemes in the literature (e.g., [24, 25, 26]) that solve this problem by encoding points directly on the curve. We use the method by Icart [26] as this scheme enjoys properties such as one-wayness and collision-resistance.

We idealize the hash function and assume the random oracle model, i.e., we assume that hash functions output a truly random string but answers consistently with previous queries.

Service-specific pseudonyms. Service-specific pseudonyms are a new cryptographic primitive that is compatible with the Groth-Sahai proof scheme. An SSP is computed from a service description \(S\) and a signing key. More precisely, the owner of verification key \(vk = x \cdot G\) computes her pseudonym \(psd\) for the service \(S\) as follows: \(psd := x \cdot S\) where \(S := h(S)\). The hash function maps arbitrary strings directly into \(\mathbb{G}_1\) (cf. above) i.e., the discrete logarithm of \(S\) to the basis \(G\) is unknown. The zero-knowledge proof for service-specific pseudonyms then shows the validity of the two equations

\[
[C_x] \cdot [C_y] = [C_{psd}].
\]

The left conjunct shows the well-formedness of the verification key. The right conjunct computes the service-specific pseudonym in zero-knowledge, using the commitment for the signing key. This proof shows the validity of the formula SSP(psd, vk, S). We stipulate that this proof always keeps the signing key \(x\) hidden and always reveals \(G\). In a proof comprising more than one pseudonym, the left conjunct needs to be shown only once since it is the same for all of the user’s pseudonyms.
The security of SSPs relies on the DDH assumption in $G_1$ and the random oracle model. The DDH assumption is implied by the SXDH assumption.

**Proving binary relations.** Proving binary relations in zero-knowledge is a well-studied problem and several approaches that are compatible with the Groth-Sahai zero-knowledge proof scheme exist. Proofs of equality are natively supported by the Groth-Sahai proof system and inequality proofs are well known (see, e.g., [27], §4.6). Respectively, we denote these proofs by

$$\|C\| = \|D\| \text{ and } \|C\| \neq \|D\|.$$  

For arithmetic relations $\text{op} \in \{<,\leq,\geq,\rangle\}$, we follow the approach proposed by Meiklejohn [23] to implement the zero-knowledge proofs $\|C_{\text{op}}\|$ of $\|C_{\text{op}}\|$. 

The construction of Meiklejohn requires the strong non-degeneracy from the bilinear map $e$, i.e., $e(\mathcal{X}, \mathcal{Y}) = 0$ if and only if $\mathcal{X} = \mathcal{O}$ or if $\mathcal{Y} = \mathcal{O}$, whereas the definition of bilinear map requires that $e(\mathcal{X}, \mathcal{Y}) \neq 0$ for all generator $\mathcal{X}$ and $\mathcal{Y}$ of $G_1$ and $G_2$, respectively. In the setting where $G_1$, $G_2$, and $G_T$ are prime-order groups, however, every non-neutral element (w.r.t. the group operation) is a generator and the non-degeneracy property implies the strong non-degeneracy requirement.

**Proving list non-membership.** For proving $(\text{psd}, \_ \not\in L)$, given a list $L = (\text{psd}_1, \text{attr}_1), \ldots, (\text{psd}_t, \text{attr}_t)$, we show that $\text{psd}$ is different from all pseudonyms in $L$:

$$\text{SSP}([C_{\text{psd}}], [C_{\text{attr}}], [C_S]) = [C_{\text{psd}}]$$

$$\wedge \bigwedge_{i=1}^{t} [C_{\text{psd}}] \neq [C_{\text{psd}}].$$

The complexity of this proof is linear in $t$.

**Proving list membership.** The proof of list membership assumes signatures $\text{sign}(\text{psd}, \_\_\text{attr}, \_\_\text{tag})$ on each of the individual list elements $(\text{psd}, \_\_\text{attr})$, where $\_\text{tag}$ uniquely identifies the list $L$. We exploit this particular list representation to make the list membership proof independent of the list size: we show the existence of a signature that belongs to the list without revealing the signature itself nor the pseudonym it signs. This construction closely resembles the signature-based set membership proof by Camenisch et al. [29], which we extend in order to prove statements of the form $(x, \_ \in L)$ as opposed to $x \in L$. Specifically, a proof for $(\text{psd}, \_\_\text{attr}) \in L$ shows the validity of the formula

$$\text{ver}([C_{\text{psd}}], [C_{\text{attr}}], [C_{\text{tag}}], [C_{\text{tag}}]).$$

We stipulate that this proof always reveals the tag $\_\text{tag}$ to show that the $(\text{psd}, \_\_\text{attr})$ pair indeed belongs to the list $L$.

As reputation lists are dynamic objects that change over time, one has to be careful in the choice of the tag uniquely identifying the list. Therefore, we propose to use a combination of a list description and an epoch number as tags. For instance, the list for a service $S$ would be tagged “List for service $S$, epoch 2” for the second epoch. Thus, adding elements does not require any re-signing, since only the newly added entries must be signed. Only removing elements causes an increase of the epoch number and requires the list administrator to re-sign all elements.

**Identity escrow.** The identity escrow proof exploits the idea of the service-specific pseudonyms. Since SSPs are designed to protect the identity of the users, however, we have to modify the protocol and add an extra piece of information to enable an escrow agent $EA$ to reveal the user’s identity. More precisely, the user obtains from the $EA$ a random value $t$ and a signature $s := \text{sign}(R)$, where $vk$ is the user’s verification key. We use this value to compute the escrow information $idr := t \cdot S$ for service $S$. The user has to prove the following statement:

$$\text{ver}([C_{\text{psd}}], [C_{\text{attr}}], [C_{\text{tag}}], [C_{\text{tag}}])$$

$$\land [C_{\text{psd}}] = [C_{\text{psd}}]$$

$$\land [C_{\text{attr}}] = [C_{\text{attr}}]$$

$$\land [C_{\text{tag}}] = [C_{\text{tag}}].$$

We stipulate that $t$ is never revealed. Hiding $r$ in the API translates into hiding $R$.

Akin to SSPs, this proof does not reveal the identity of the user. The $EA$, however, knows all the random value-user pairs, in our case $t$ and $vk$. If the $EA$ is informed of a cogent reason to reveal the identity of the user associated with the escrow information $idr$ for service $S$, the $EA$ can successively try all stored random values $t'$ to check whether $t' \cdot S = idr$; eventually, $t' = t$ and the user is identified.

It is also possible to employ several escrow agents to offer better trust guarantees. In this case, the user has to obtain a random value and the corresponding signature from each of the escrow agents and combine the random values into one. Identity escrow then requires the collaboration of all the escrow agents. They may use a secure multi-party computation (e.g., [30], [31]) to compute the joint value $t$ without revealing one’s share to other parties.

The security of escrow identifiers relies on the DDH assumption in $G_1$ and the random oracle model. The DDH assumption is implied by the SXDH assumption.

**Assumptions.** Our cryptographic setup is secure in the random oracle model and under the SXDH assumption and the $q$-ADH-SDH assumption.

## IV. Cryptographic Proofs

This section states the security results for the cryptographic primitives we developed in this work. The full proofs are given in Appendix A.

**Service-specific pseudonyms.** We state the three main properties of service-specific pseudonyms: uniqueness, anonymity, and unlinkability across services. In the following, we assume that the hash function $h$ is a random oracle, i.e., a truly random function that answers queries consistently with previous answers.
Theorem 1 (Uniqueness of SSPs). In the random oracle model, service-specific pseudonyms are unique, i.e., the following statements hold with overwhelming probability:

1) for any service $S$ and two honestly-generated verification keys $vk_1 := xG$ and $vk_2 := yG$, $xS \neq yS$;
2) for any verification key $vk := xG$ and service $S$, $xS$ is a unique value;
3) for any two different service descriptions $S_1$ and $S_2$ and verification key $vk := xG$, $x \cdot h(S_1) \neq x \cdot h(S_2)$.

The following theorems rely on the standard decisional Diffie-Hellman (DDH) assumption. Intuitively, the anonymity theorem states that given a set of candidate verification keys, a set of services, and a set of SSPs that all originated from one single verification key, it is computationally infeasible to decide which of the candidate verification keys was used to compute the pseudonyms. For instance, even knowing that several pseudonyms belong to the same user does not allow for identifying that user.

Theorem 2 (Anonymity for SSPs). In the random oracle model and under the DDH assumption for $G_1$, given a sequence of $k$ services $(S_1, \ldots, S_k)$ and $k$ corresponding service-specific pseudonyms $(psd_1 := xS_1, \ldots, psd_k := xS_k)$, and a set of $m$ verification keys $\{vk_1, \ldots, vk_m\}$, it is computationally infeasible to decide which $vk_i$ is associated with $psd_1, \ldots, psd_k$ (i.e., which $vk_i = xG$).

Identity escrow. The uniqueness, the anonymity, and the unlinkability results for escrow identifiers are similar to those for SSPs. We give all details in Appendix A.

V. STATIC ANALYSIS OF AUTHORIZATION POLICIES

This section formally proves that the cryptographic implementation enforces the authorization policies specified by the programmer. This is of paramount importance in our setting to make sure that the malleability of zero-knowledge proofs does not constitute an attack surface. Intuitively, we aim at showing that whenever a principal successfully verifies a validity proof for formula $F$, then $F$ holds true. First, we formally define what it means for a logical formula to hold true (Section V-A). We then show how to symbolically encode the semantics of malleable zero-knowledge proofs in RCF [3], a concurrent $\lambda$-calculus (Section V-B). This encoding allows us to leverage F7 [8], a type system for the static analysis of cryptographic protocol implementations. Specifically, we type-check the implementation of our API, showing that the successful verification of a validity proof for formula $F$ entails that $F$ holds true (Section V-C). Interestingly, the soundness results do not assume any typing discipline for the program, i.e., the usage of our API suffices to enforce the desired authorization policies (Section V-D), thus yielding security by construction guarantees.

A. Authorization Policies

Following a well-established methodology for the specification and verification of authorization policies in a distributed setting, we decorate the code with assumptions and assertions [8]: assumptions introduce logical formulas which are assumed to hold at a given point, while assertions specify logical formulas which are expected to follow from the previously introduced (i.e., active) assumptions.

Authorization policies (e.g., equation (1)) are explicitly assumed in the system. Furthermore, we place an assumption within the implementation of the mkSays method, reflecting the intention of the user to introduce a new logical formula in the system: for instance, executing the mkSays method on line 6 of Example 2 introduces an assumption of the form assume $x_{Pat}$ says Eval($x_{Opinion}$). Finally, assertions are placed immediately after each call to the verify method: for instance, the call to the verify method on line 4 of Example 2 is followed by

$$\text{assert} \left( x_{Hosp} \text{ says IsDoc} (x_{Doc}, x_{PI_{Doc}}) \wedge x_{Doc} \text{ says Visit} (x_{Pat}, x_{date}, x_{results}) \right)$$

B. Symbolic Cryptography

As usual in the static analysis of cryptographic protocol implementations, we rely on a symbolic abstraction of cryptographic primitives that captures their ideal behavior. Prior work showed how standard cryptographic primitives such as encryptions and signatures [6] as well as non-malleable zero-knowledge proofs [32] can be faithfully modeled using a sealing-based technique [33], which is purely based on standard language constructs. In a nutshell, 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 handle, and returns this handle; (ii) an unsealing function that takes as input a handle, scans the secret list in search for the associated message, and returns that message. The fundamental insight is that the only way to extract a sealed value is via the unsealing function. Sealing-based abstractions of encryptions and signatures have been proven computationally sound by Backes et al. [34], i.e., security results verified on these abstractions carry over to the actual cryptographic implementation.
Previous sealing-based abstractions for non-malleable zero-knowledge proofs \[32\] use one seal per proven statement. The sealing and unsealing functions can only be accessed by the functions to create and verify proofs. Since the number of proven statements in a protocol is finite in the non-malleable setting, the number of seals is finite as well. In a malleable setting, however, this approach yields an unbounded number of seals since proofs can be arbitrarily combined. We therefore devise a finite sealing-based library for malleable zero-knowledge proofs.

We model malleable zero-knowledge proofs using one seal for the proofs themselves and one seal to model the commitments used inside zero-knowledge proofs.

The seal for zero-knowledge proofs stores the proven statement and a random value; the random value corresponds to the randomness used during the computation of the zero-knowledge proof and the fresh handle corresponds to the zero-knowledge proof. The sealing and unsealing functions are only used inside the functions to create and verify zero-knowledge proofs, respectively.

The seal for commitments stores its own list of the committed values and the randomness used in the commitment; the fresh handle corresponds to the commitment. Only the sealing function to compute commitments, is public.

The proof creation function takes the formula to be proven as input (say, \(a \leq b\)), creates the commitments (\(C_a\) and \(C_b\)), and passes the zero-knowledge statement (\(\|C_a\| \leq \|C_b\|\)) to the sealing function, which outputs the zero-knowledge proof. The verification function takes as input the proof along with the zero-knowledge statement, internally opens the commitments, and executes the zero-knowledge statement on the witnesses to check its validity. The functions to manipulate zero-knowledge proofs (e.g., splitting of logical conjunctions) are straightforwardly implemented by using the sealing and unsealing functions for zero-knowledge proofs and for commitments.

This is a precise symbolic model of the Groth-Sahai proof system and, in concurrent work, we are establishing a formal computational-soundness result for it \[35\].

\[2\]

\[\exists x. \ x \text{ says } \text{Eval}(ev, course)\]

C. Typed Interface

We type-check the symbolic implementation of our API using F\(7\) \[8\], a type-checker for authorization policies. This type-checker works on RCF, a refined and concurrent \(\lambda\)-calculus that can be used to reason about a large fragment of ML and of Java by encoding. In F\(7\), the universal type \(\text{unit}\) describes values without a security import, i.e., values of type \(\text{unit}\) can be passed to and received from the attacker. All types used in the API interface (see Table I) are encoded as \(\text{unit}\). Signing keys are the only confidential data but, as previously discussed, they are not exported by the API. The interface types coincide with the ones shown in Table I except for the type of the verify method. This method is given a refinement type of the form:

\[\text{verify}_p : \text{proof} \rightarrow y : \text{formula} \rightarrow \{z : \text{bool} | \forall \exists. y = F \land z = \text{true} \implies F\}\]

Intuitively, a value \(v\) has type \(\{x : T \mid F\}\) if \(v\) has type \(T\) and, additionally, the logical formula \(F\{v/x\}\) (i.e., \(F\) where every occurrence of \(x\) is replaced by \(v\)) is entailed by the active assumptions. The type of verify ensures that if the returned value \(z\) is true and the formula \(y\) passed as input is the ML encoding \(F\) of the logical formula \(F\), then the formula \(F\) is entailed at run-time by the currently active assumptions. In other words, malleable zero-knowledge proofs constitute a sound implementation of our logic-based data processing API.

Example 8. The verify method on line \((s.4)\) from Example 2 is given the following type:

\[\text{verify} : \text{proof} \rightarrow y : \text{formula} \rightarrow \{z : \text{bool} | \forall \forall \exists. y = F \land z = \text{true} \implies F\}\]

Well-formedness of formulas. It is interesting to observe that not all proofs are meaningful. For instance, suppose that a principal receives a proof of the following formula:

\[\exists x. \ x \text{ says } \text{Eval}(ev, course)\]

We would be tempted to let this principal entail \(\exists x. \ x \text{ says } \text{Eval}(ev, course)\). The proof for this formula, however, does not reveal the identity of the person issuing the proof, nor is there any information about the origin of the creator of the proof. In fact, this proof might have been constructed by an attacker, using a fresh key-pair and, therefore, the formula \(\exists x. \ x \text{ says } \text{Eval}(ev, course)\) is not necessarily entailed by the formulas proved by 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 include public identifiers of principals that belong to principals of the system (as opposed to attacker’s keys) into their formulas. 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, formula \(\exists x. \ x \text{ says } \text{Eval}(ev, course)\) does not guarantee that the existentially quantified public identifier \(x\) is trustworthy. Conversely, the existentially quantified identifier \(w_{stud}\) in
Example 3 is endorsed by the professor and, therefore, is trustworthy. Hence, the proof $pf'$ justifies the corresponding formula.

In a nutshell, a formula is well-formed if it ensures that all public identifiers are trustworthy. Despite the simplicity of this intuition, the formal definition has to consider a number of complications, including the presence of logical disjunctions in the statement. For instance, the statement $\exists x_1. x_1 \text{ says } F \lor \text{Prof says } F$ is not well-formed, since we do not know which of the two disjuncts holds true. The idea is to transform a statement in disjunctive form and then to check that all identifiers in each sequence of conjunctions are registered. We formalize the notion of trustworthiness for keys in \textbf{Appendix C}.

Here and throughout the rest of the paper, we stipulate that all formulas are well-formed.

D. Soundness Result

We first formalize the notion of safety for authorization policies. Intuitively, safety states that assertions never fail at run-time, even in the presence of an opponent. The source code along with the proofs are given in \textbf{Appendix C}.

\textbf{Definition 1 (Safety, Opponent, and Robust Safety \textup{[8]}.)} A program $P$ is safe if and only if, in all executions of $P$, all assertions are entailed by the current assumptions.

A program $B$ is an opponent if and only if $B$ contains no assertions and the only type occurring in $B$ is $\text{unit}$.

A program $P$ is robustly safe if and only if the application $B \parallel P$ is safe for all opponents $B$.

The type system establishes judgments of the form $\Gamma \vdash P : T$ for some typing environment $\Gamma$, some program $P$, and some type $T$. Intuitively, $\Gamma$ tracks the types of variables in scope. The following theorem ensures that well-typed programs are robustly safe.

\textbf{Theorem 4 (Safety by Typing \textup{[8]})}. If $\emptyset \vdash P : T$, then $P$ is safe. If $\emptyset \vdash P : \text{unit}$, then $P$ is robustly safe.

Finally, our soundness theorem states that if a program is well-typed against the API interface $\Gamma_{API}$, then it is robustly safe when linked to the API implementation $P_{API}$.

\textbf{Theorem 5 (Soundness)}. If $\Gamma_{API} \vdash P : \text{unit}$, then $P_{API} \parallel P$ is robustly safe.

Notice that \textbf{Theorem 5} only applies to well-typed programs $P$. In the following, we formally demonstrate that our soundness result depends only on the well-typing of the API implementation and not on the program using the API.

This result is based on the opponent typability lemma \textup{[8]}, which says that all opponents are well-typed. Intuitively, this lemma captures our intuition that programs are safe if they only use values that can be sent to and received from the attacker, i.e., of type $\text{unit}$. In our case, the only values whose type is different from $\text{unit}$ are signing keys and the refined verification function. As discussed above, signing keys are concealed inside the API and accessible only via a public handle; in the symbolic library, they are stored in a secret reference. We construct a verification wrapper function $verify'_{F}$. This function calls the refined verification function $verify_{F}$ and executes the corresponding assertion (see \textbf{Appendix E}). We proved that $\Gamma_{API} \vdash verify'_{F} : \text{unit}$. Furthermore, as previously mentioned, all types except for $\text{unit}$ in $\Gamma_{API}$ are encoded as $\text{unit}$. We can then define a variant of our typed API, named $\Gamma'_{API}$, in which the $verify_{F}$ method is replaced by $verify'_{F}$. We call $P'_{API}$ the corresponding implementation. $\Gamma'_{API}$ is refinement-free and only exports $\text{unit}$ types. As a result, we can prove the following theorem, which says that any user program linked to $\Gamma'_{API}$ is robustly safe.

\textbf{Theorem 6 (Security by Construction).} Let $P$ be an assertion-free program such that $\text{unit}$ is the only type occurring therein and $\text{fnfo}(P) \subseteq \Gamma_{API}$ (i.e., $P$ only uses functions exported by $\Gamma_{API}$). Then $\Gamma_{API} \vdash P : \text{unit}$, i.e., $\Gamma'_{API}$; $P$ is robustly safe.

In other words, the programmer does not have to type-check her code with $F7$. Instead, the usage of our API suffices to yield security by construction guarantees.

VI. IMPLEMENTATION IN JAVA

We implemented the API methods as a Java library. We implemented the data processing primitives described in Section III in two abstraction layers. The low-level layer encapsulates the implementation of the cryptographic primitives, namely, automorphic signatures, Groth-Sahai zero-knowledge proofs, and pseudonyms. We rely on the jPBC library \textup{[36]} for the bilinear group operations. To the best of our knowledge, this is the first fully-fledged implementation of the SXDH instantiation of the Groth-Sahai proof system.

The high-level layer comprises the API methods. In our tests, we used the standard Java functionality to implement communication primitives and we use the Tor onion routing network \textup{[37]} to implement anonymous channels.

A. Implementation of a Course Evaluation System

In order to demonstrate the expressiveness of our API both for programmers and for users in the context of web applications, we implemented a course evaluation system. A demo is available online \textup{[38]}. In this demo, users interact with the system by creating proofs for logical formulas, which are straightforward to understand and do not require any knowledge in cryptography.

More precisely, the user can create a user identifier $u$ and pass it to the server. Impersonating a professor, the server issues a proof for the formula $\text{Prof says } \text{Reg}(u, \text{"Security2012"})$ to the user. After importing this proof, the user can evaluate the lecture, create a service-specific pseudonym for the lecture, and hide her
identity. If desired, the user can also create a list non-membership proof. The resulting overall proof is uploaded to the server and if successfully verified by the server listed in the evaluation board.

VII. EXPERIMENTS

We conduct an experimental evaluation of the Java implementation to demonstrate the feasibility of our approach. We evaluate our multi-threaded implementation on different MNT curves [69] with various security parameters $n$, namely, 112, 128, and 256 bits (NIST recommendations [40] deem 112 bit security parameters secure until the year 2030). For MNT curves, the group order $p$ is approximately $2^{2n}$, i.e., for $n = 112$, $G_1$ has more than $2^{224}$ elements.

We measure the proof generation time, proof verification time, and the proof size for the concrete implementation of Example 4, SSP proofs, list membership and non-membership proofs, and the identity escrow protocol. For the list membership and non-membership proofs, we fix the total number of list elements to 1000, which we distribute over various amounts of lists. We evaluate the identity escrow protocol for various security parameters and we determine how many escrow identifiers an escrow agent can check per second. We run our experiments on a computer with an Intel Xeon E5645 six core processor with 2.4 GHz and hyper-threading, and 4 GB of RAM.

Discussion. In the following, all quantities refer to a 112 bit security parameter. Figure 1 presents results for Example 4 which consists of two zero-knowledge proofs of signatures on message tuples and a pseudonym verification proof. The generation and the verification take 6 s and 10 s, respectively, and the proof is 266.4 KB in size. In general, zero-knowledge proofs of a signature on message tuples have a complexity linear in the arity of the tuple. For instance, the construction of proofs for tuples of arity 1, 2, and 3 take 1.28 s, 1.92 s, and 2.61 s, respectively. Proving conjunctions is very efficient since it is a concatenation of the sub-proofs. Figure 2 depicts the results for SSP proofs. Proof generation as well as proof verification are highly efficient and take 76 ms and 67 ms, respectively. The proof size is 2.6 KB.

Figure 3 shows that the list non-membership proof is practical, even for long lists. We vary the number of lists since users have to recompute their pseudonym in zero-knowledge for every list. The number of lists, however, plays only a small role as the proof is dominated by the computations for the list elements: the proof for one list with 1000 elements takes 109 s and the proof for 100 lists with a total of 1000 elements takes 116.5 s. The proof size varies between 3.2 MB for 1 list and 3.4 MB for 100 lists.

Figure 4 presents the results for the list membership proof. As expected, the proof for a single list is very efficient as it is independent of the size of the list. Creating a proof for many lists, however, is more expensive, since signatures on message tuples are computationally burdensome. The proof for one list and 1000 elements takes 3 s and the proof for 100 lists with a total of 1000 elements takes 309.1 s. The proof size varies between 133 KB for 1 list and 13.3 MB for 100 lists. We believe that these numbers do not undermine the practicality of our approach: typical users only participate in a small number of services and therefore are only confronted with a small number of list membership proofs.

As shown in Figure 5 the identity escrow proof constitutes a small computational burden for the prover and the verifier: the proof takes 360 ms to generate, requires 350 ms to verify, and takes 10.8 KB in size. The computation of the EA consists only of scalar multiplications and equality tests. These are extremely efficient and the EA can perform more than 10000 per second on a single core.

VIII. RELATED WORK

Although much work has been done to develop cryptographic protocols that achieve some of the security properties discussed in this paper, none of them can be seen as an off-the-shelf, general-purpose tool for the design of distributed systems: either, they put restrictions or assumptions on the structure of the system (e.g., the presence of a TTP or shared secret information), they hamper the system performance (e.g., by requiring additional setup phases or more interactions), they cannot be composed with other protocols to achieve a wider range of security properties, or they do not allow for the extension of the system with new components (open-endedness) and the sharing of data among them (interoperability). In the following, we discuss the cryptographic schemes most closely related to our work.

Security-oriented, declarative languages. The seminal works by Abadi et al. [41], [42] on access control in distributed systems paved the way for the development of a number of authorization logics and languages [12], [13], [14], [43], [44], which all rely on digital signatures to implement logical formulas based on the says modality. Maffei and Pecina extended this line of research with the concept of privacy-aware proof carrying authorization [17], showing how to cryptographically realize existential quantification by zero-knowledge proofs.

As an off-the-shelf, general-purpose tool for the design of distributed systems provided the way for the development of distributed systems. In the following, we discuss the cryptographic schemes most closely related to our work.

Building on that work, Backes et al. [11] have devised a framework for automatically deriving cryptographic implementations from a logic-based declarative specification language derived from evidential DKAL [10]. In their work, the programmer has to supply a logical derivation that is compiled piece by piece into executable code. In our framework, the high-level declarative API is directly embedded into the programming language, which allows programmers to devise systems without switching to an external logic-based language and to conveniently access the data exchanged in the protocol. Furthermore, besides authorization and privacy, our framework supports controlled linkability, accountability, and identity escrow.
G2C [45] is a goal-driven specification language for distributed applications capable of expressing secrecy, access control, and anonymity properties. These properties are enforced using broadcast encryption schemes and group signatures and the cryptographic details are automatically generated by a compiler. This compiler generates cryptographic protocol descriptions as opposed to executable implementations. Furthermore, the protocols are not open-ended and extending them often requires the re-generation of the whole system from scratch.

Pseudonyms. Chaum [46] initiated the research on pseudonyms and since then many schemes have been introduced (e.g., [47], [48], [49], [50], [51], [52], [53], [54], [55]). Many schemes do not consider the notion of service (e.g., [48], [49], [51], [47]), or incorporate a compulsory trusted third party (e.g., [53], [54]), or do not enforce the uniqueness property (e.g., [48], [49], [51]), or do not support any form of authorization policy unless the pseudonym owner is fully disclosed (e.g., [47]).

Martucci et al. [50] use a TTP only to register the real identity. Upon that, users generate pseudonyms on their own using a non-interactive publicly verifiable variant of a special signature scheme and then self-certify them by means of anonymous credentials and group signatures. A pseudonym is unique within a given context and a user is linkable for actions performed within this context. The compulsory presence of a trusted third party is a fundamental difference from the pseudonym system considered in the present paper, in which the presence of a TTP is optional and only needed to reveal the identity of misbehaving users.

Brands et al. [55] use a central authority to register users in a system: they receive a fixed number of pseudonyms that are used to register with a service provider, one pseudonym for every available service. Should a user misbehave, she can be completely revoked from the system but identity escrow is not possible. The central authority, the fixed number of services, and the absence of an identity escrow protocol significantly differentiate their work from ours.

Anonymous credential systems. We compare our work to the line of research on anonymous credential systems that support anonymous and delegatable authentication. All the following protocols, apart from the scheme by Belenkiy et al. [55], rely on $\Sigma$-protocols and, as a consequence lack the flexibility to selectively hide individual parts of the proven statement. This limitation is prohibitive for the design of open-ended systems. For instance, the protocol in Example 7 cannot be implemented using $\Sigma$-protocols, since it requires the hiding of the doctor’s identity and parts of the signed message from a given proof. Our work instead relies on the Groth-Sahai zero-knowledge proof system that is flexible and general enough to selectively hide and reveal any given part of the proven statement. Furthermore, our work supports an optional TTP-based identity-escrow functionality, which is offered by neither of the systems mentioned below.

The direct anonymous attestation (DAA) protocol [57] offers a pseudonymous-attestation functionality, which allows users to authenticate their trusted platform module (TPM) with a service provider using a pseudonym, derived from the TPM's secret value (chosen by an external party) and a base value chosen by the resource provider, yielding the notion of service. The TPM’s secret value is signed by a third party, called the issuer. In this work, we do not require trusted hardware and a trusted third party is only needed if identity escrow is desired.

Service-specific pseudonyms coincide with the concept of domain pseudonyms from Identity Mixer cryptographic library (idemix) [58], scope-exclusive pseudonyms from the attributed-based credentials for trust project (ABC4Trust) [59], and pseudonyms used in U-Prove [60]. The flexibility of our cryptographic setup based on Groth-Sahai proofs and the identity escrow protocol are the most prominent differences.

From the recently-proposed Nymble systems (e.g., [61], [62], [63]), BLACR [63] is the more expressive and efficient. In BLACR, users generate fresh private keys that get authenticated by a group manager. Users use their keys to generate tickets that are revealed to service providers. Service providers can blacklist or whitelist these tickets and assign scores to them. Users traverse all lists, adding up the scores in zero-knowledge and revealing the final result to the service provider who can use this result to allow or deny access. The complexity of such proofs is linear in the size and number of lists. For monotonically increasing lists, the user can ask the service provider for a token certifying her reputation for the current list, allowing the user to prove her reputation only for the subsequent part of the list for future requests. In our framework, every membership proof is independent of the list size (see Section III) and our construction is fully distributed, does not involve any group manager, and supports a much larger class of authorization policies, which may depend on (possibly anonymous) certificates released by any party of the system.

The delegatable anonymous credential scheme by Belenkiy et al. [56] is based on the Groth-Sahai proof system. There, a root authority issues anonymous credentials that can further be delegated. Delegatable credentials indicate the root authority and they reveal how often they have been delegated. For instance, in Example 1 the doctor has a level-1 credential and the patient has a level-2 credential, both rooted at the hospital. Although based on Groth-Sahai proofs, their scheme is not open-ended because the root is unalterably anchored in every credential and proofs originating from different root authorities cannot be combined. Additionally, it is not possible to change the root authority without re-issuing all delegated credentials, e.g., when the doctor switches to another hospital.

Group signatures and ring signatures. Group signatures
(e.g., [64], [65], [66], [67]) allow a member of a group to sign a message on behalf of the group. They often rely on a group manager to distribute keys and, in case of a dispute, reveal the signers of a message. The presence of a trusted third party that can reveal the identity of a signer is the most noticeable difference from our approach, where a third party (the escrow agent) is only needed to reveal the identity of misbehaving users. Further, group signatures increase the key management overhead since every group requires a different set of keys, whereas in our system, each user needs just one key-pair.

Ring signatures (e.g., [68], [69], [70]) allow a user to sign a message \( m \) on behalf of a set of users: the user gathers all the verification keys of the users in the set (including her own key) and uses the ring signature to create a signature on \( m \). The verifier of this signature will only learn the members of the set and that a member of the set created the signature but not which one. Consequently, to use ring signatures in our setting, it must be public knowledge which principal uses which service as otherwise, the set of principals for one particular service cannot be assembled. This poses serious privacy issues and even prohibits the specification certain systems: in decentralized social networks, users want to hide their friend list [49] which is not possible using ring signatures. Users take the role of service providers and their friends constitute the set of users using that service, which must be public to construct a ring signature.

**Accumulators.** Accumulators store an arbitrary number of values and are generally equipped with efficient membership and non-membership proofs, i.e., proofs of whether a value is stored in an accumulator or not. While accumulators seem to be ideal for implementing our reputation lists, incorporating them into our existing framework requires encoding pseudonyms into a special form that is compatible with the accumulator. Proving this encoding in zero-knowledge, however, makes the overall protocol very inefficient, outweighing the gains of accumulators over our reputation lists. For instance, there exist accumulators for numbers in \( \mathbb{Z}_n \) (e.g., [71]). The Groth-Sahai only supports quadratic equations and proving the computation of an SSP \( S^x \) in zero-knowledge takes time linear in the security parameter since we would have to resort to the square-and-multiply algorithm. The resulting computational overhead each time a user uses a pseudonym is significant. Other schemes that work on elliptic curves directly (e.g., [72], [73]) require a symmetric bilinear map, or they work in an RSA-like system (e.g., [73]), requiring composite-order groups. These requirements are incompatible with automorphic signatures.

IX. **Conclusion and Future Work**

We presented a framework for the declarative design of distributed systems, which supports a wide range of security properties, including authorization policies, privacy, controlled linkability, and accountability. The core component of the framework is an API that exports primitives for data processing. The programming abstraction represents the information known to principals as logical formulas and the messages exchanged by parties as validity proofs for logical formulas. The cryptographic implementation relies on a powerful combination of digital signatures, non-interactive zero-knowledge proofs of knowledge, service-specific pseudonyms, and reputation lists. Our framework constitutes an ideal plugin for proof-carrying authorization infrastructures [13], [14], [74], [7], [11].

We showed how to leverage an existing security type system for ML to statically enforce authorization policies in declarative specifications and we have proven that these policies are enforced by the cryptographic implementation. We also proved the security of the cryptographic constructions introduced in this paper (namely, service-specific pseudonyms and the identity escrow protocol).

We are currently exploring the design of an accumulator scheme compatible with our framework in order to make the complexity of the non-membership proof independent of the list length. Concerning the implementation of our cryptographic library, we plan to integrate several optimizations, including batch verification techniques, which may speed up the verification of zero-knowledge proofs by up to 90% [75]. Finally, we plan to extend our framework in a number of directions: for instance, 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 introduce designated verifier proofs in the cryptographic implementation and to extend the logic with a belief predicate in order to confine the validity of a formula to a specific individual.

**References**

http://privacybydesign.ca/

http://www.ftc.gov/opa/2012/03/privacyframework.shtm


[77] http://www.verisign.com/


Figure 1: The results for the example in Example 4.

Figure 2: The results for the computation of a service-specific pseudonym.

Figure 3: The results for the non-membership proof for $\ell$ lists, a total number of 1000 elements distributed over the lists, and a security parameter of $n = 112$ bits.

Figure 4: The results for the membership proof for $\ell$ lists, a total number of 1000 elements distributed over the lists, and a security parameter of $n = 112$ bits.

Figure 5: The results for the identity escrow protocol.
APPENDIX

A. Proofs for the Cryptographic Protocols

This section elaborates on all cryptographic proofs for the claims stated in Section III. We start with the service-specific pseudonym properties and then proceed to the properties of the escrow identifiers.

While defining uniqueness properties in Section III, we explicitly mentioned the discrete logarithm. In the following definitions of uniqueness, we use function Y is to define the desired properties.

Definition 2. We define the function $Y_X: \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_1$ as $Y_X(x, y) \rightarrow x \cdot y$.

Intuitively, $Y_X(A, B)$ extracts the discrete logarithm $x$ of $A$ to the basis $X$ and returns $xB$. Notice that the value $Y_G(vk, S)$ corresponds to the SSP of the owner of $vk$ and the service $S$, and the value $Y_{vk}(R, S)$ corresponds to the escrow identifier of the owner of $vk$ and the service $S$. In general, $Y_X$ cannot be efficiently computed and is only used for defining properties of SSPs and escrow identifiers. In fact, the DDH assumption implies that the discrete logarithm cannot be extracted and used as suggested by function $Y$.

Properties of Pseudonyms

Uniqueness of pseudonyms. To prove our uniqueness result, we first state basic facts about the distribution of hash values and of secret signing keys. The following proposition holds since the output of the hash function $h$ are uniformly random values from a set that is exponentially large in the security parameter $n$ (see Section III and Abe et al. [6]).

Proposition 1. The following probabilities are negligible in $n$:
- The output of the hash function $h: \{0,1\}^* \rightarrow \mathbb{G}_1$ is $O$, the neutral element of the group operation of $\mathbb{G}_1$.
- The output of the hash function $h: \{0,1\}^* \rightarrow \mathbb{G}_1$ for polynomially (in $n$) many different inputs coincide.
- A signing key $x$ is 0.
- A pseudonym $psd := xS$ is $O$.
- Two signing keys from a set of polynomially (in $n$) many coincide.

We can now proceed to prove the uniqueness theorem. First, we formally define the uniqueness for SSPs using the $Y$ function.

Definition 3 (Uniqueness of service-specific pseudonyms). We say that service-specific pseudonyms are unique if and only if, the following conditions hold with overwhelming probability:
1) for any service $S$ and two honestly-generated verification keys $vk_1$ and $vk_2$, $Y_G(vk_1, S) \neq Y_G(vk_2, S)$,
2) for any verification key $vk$ and service $S$, $Y_G(vk, S)$ is a unique value,
3) for any two different service descriptions $S_1$ and $S_2$ and verification key $vk$, $Y_G(vk, h(S_1)) \neq Y_G(vk, h(S_2))$.

Theorem 1: Uniqueness of SSP.

In the random oracle model service-specific pseudonyms and escrow identifiers are unique.

Proof of Theorem 1. First, we note that SSPs and escrow identifiers are computed in the same way. The difference is that for SSPs, the signing key $x$ used to compute the pseudonym is chosen by the user and the secret $r$ used to compute the escrow identifier is chosen by the $EA$. Both, however, are chosen uniformly at random from the set $\{1, \ldots, |\mathbb{G}_1|\}$ of exponents.

Condition 1: for any service $S$ and two verification keys $vk_1$ and $vk_2$, $Y_G(vk_1, S) = Y_G(vk_2, S)$ if and only if $S = O$, the only non-generator of $\mathbb{G}_1$, or $vk_1 = vk_2$; the two verification keys coincide if and only if the two corresponding, honestly-chosen signing keys coincide. These two events happen only with negligible probability by Proposition 1.

Condition 2: follows immediately since $Y$ is a deterministic function.

Condition 3: for any verification key and two different service descriptions $S_1$ and $S_2$, $Y_G(vk, h(S_1)) = Y_G(vk, h(S_2))$ if and only if $h$ maps $S_1$ and $S_2$ to the same hash value or the signing key is 0. These two events happen only with negligible probability by Proposition 1.

Anonymity of pseudonyms (intra-service unlinkability).

We now prove our theorem asserting that service-specific pseudonyms preserve the anonymity of users. We begin by stating our definition of anonymity.


Definition 4 (Pseudonyms-based Anonymity). A set of \( k \) pseudonyms \( \{\text{psd}_1 := xS_1, \ldots, \text{psd}_k := xS_k\} \) for \( k \) services (as constructed in Section III) provides anonymity if and only if, given a set \( \{vk_1, \ldots, vk_m\} \) of \( m \) verification keys, any polynomially-bounded attacker can determine which verification key \( vk \) from the set \( M \) was used for computing \( \text{psd}_1, \ldots, \text{psd}_k \) with probability at most \( \frac{1}{m} + \mu \), where \( \mu \) is negligible in \( n \).

Intuitively, pseudonyms provide anonymity if pure guessing essentially is as good as an attacker that tries to determine which verification key \( vk \) from the set \( M \) was used for computing \( \text{psd}_1, \ldots, \text{psd}_k \).

We now work our way towards the main theorem. The proof will be a reduction against the decisional Diffie-Hellman (DDH) problem. For the sake of completeness, we give all the necessary definitions.

Definition 5 (DDH and DDH Advantage). Given the tuple \((G, xG, yG, C)\), where \( \langle G \rangle = G_1 \) is a generator of \( G_1 \), and \( x, y \in \mathbb{Z}_p \) are randomly chosen, the DDH problem is to decide whether \( C = xyG \).

The advantage of a DDH attacker is defined as

\[
\text{Adv}_{\text{DDH}}(B) = \left| \Pr[1 \leftarrow B(1^n, G, xG, yG, xyG) \mid b = 1] - \Pr[1 \leftarrow B(1^n, G, xG, yG, Z) \mid b = 0] \right|
\]

where \( z \) is a random value in \( G_1 \) and \( b \) is randomly chosen from \( \{0,1\} \).

Intuitively, the advantage of a DDH attacker states how much better than pure guessing the attacker is.

Assumption 1 (Hardness of DDH). For all polynomially-bounded attackers \( B \), the advantage \( \text{Adv}_{\text{DDH}}(B) \) is negligible in \( n \).

Reviewing the construction of service-specific pseudonyms, we see that the values \((G, vk, S, psd)\) form a valid Diffie-Hellman tuple since \( vk = xG \) for a random \( x \), \( S = rG \) for a random \( r \), and \( psd = xS = xyG \). We now state and prove our main theorem about service-specific pseudonyms. This theorem is a formal restatement of Theorem 2.

Theorem 7 (Anonymity of Service-Specific Pseudonyms). In the random oracle model and under the DDH assumption, service-specific pseudonyms (as constructed in Section III) provide anonymity.

Proof: The proof is a reduction against DDH. Figure 6 visualizes the steps of this reduction proof. Intuitively, the set \( \{vk_1, \ldots, vk_k\} \) consists of the verification keys obtained by a service provider during the registration of \( k \) principals. Suppose there is an attacker \( A \) that, on input \((G, \{vk_1, \ldots, vk_m\}, \{S_1, \ldots, S_k\}, \{\text{psd}_1, \ldots, \text{psd}_k\})\), outputs \( \ell \) such that \( vk_\ell \) and \( \text{psd}_1, \ldots, \text{psd}_k \) are associated with probability \( 1/m + \eta \) where \( \eta \) is non-negligible. From this attacker, we construct an attacker \( B \) that breaks a decisional Diffie-Hellman problem with non-negligible probability.

The DDH challenger \( C \) uniformly at random draws a bit \( b \in \{0,1\} \). If \( b = 1 \), \( C \) generates a valid DDH tuple, if \( b = 0 \), \( C \) generates a fake DDH tuple, i.e., a tuple where \( C = zG \) for \( z \in R \mathbb{Z}_p \). The resulting tuple is sent to attacker \( B \).

Given that DDH challenge \((G, zG, yG, C)\), attacker \( B \) must give a perfect simulation to attacker \( A \), so that \( A \) cannot differentiate between a normal challenge and a challenge constructed by \( B \). We note that the value \( S \) in our service-specific pseudonym construction is a value that is indistinguishable from a random value in \( G_1 \); it is the output of a random oracle and hence, its discrete logarithm \( r \) with respect to \( G \) is also indistinguishable from a random number in \( \mathbb{Z}_p \). Hence, a computationally bounded attacker cannot notice the difference and \( S \) matches with \( yG \). Further we note that verification keys are constructed exactly as \( xG \). \( B \) chooses \( \ell' \in R \{1, \ldots, m\} \) and generates \( m \) random verification keys \( vk_1 \in G_1 \). Furthermore, \( B \) randomly draws \( s_i \in R \mathbb{Z}_p \) for \( i \in \{2, \ldots, k\} \). Since we are in a set with prime-order groups, every service and every pseudonym is a generator of the whole group \( G_1 \) (except for \( O \), which occurs only with negligible probability, cf. Proposition 1). Therefore, given a service \( S \) and a corresponding pseudonym \( psd := x \cdot S \), multiplying both with a random value \( s \) yields \( s \cdot S \) and \( spsd = x \cdot (s \cdot S) \). The products form another random service and the corresponding (random) pseudonym, justifying \( B \)’s action to draw random numbers and multiply them in the following call to \( A \).

\( B \) calls \( A(\{vk_1, \ldots, vk_{\ell'-1}, xG, vk_{\ell'+1}, \ldots, vk_m\}, G, (yG, y_2G, \ldots, y_kG), (C, s_2G, \ldots, s_kG)) \) and receives \( \ell \) as answer. \( B \) returns \( 1 \) if and only if \( \ell = \ell' \) where \( 1 \) denotes \( B \)’s decision that \( c = xyG \). In the following calculation, we let \( z \in R \mathbb{Z}_p \).

\[
\text{Adv}_{\text{DDH}}(B) = \left| \Pr[1 \leftarrow B(G, xG, yG, xyG) \mid b = 1] - \Pr[1 \leftarrow B(G, xG, yG, Z) \mid b = 0] \right| 
\]

\( \overset{\text{(1)}}{=} \left| \Pr[1 \leftarrow B(G, xG, yG, xyG) \mid b = 1] - \Pr[1 \leftarrow B(G, xG, yG, Z) \mid b = 0] \right| 
\]

\( \overset{\text{(2)}}{=} \left| \frac{1}{m} + \eta - \frac{1}{m} \right| = \eta \)
For equality (1), we substitute the attacker $B$ with its definition. Equality (2) holds for the following reason: the left part of the difference holds as we give a perfect simulation for attacker $A$ who, by assumption, can associate the verification key with the pseudonym with probability $1/k + \eta$, where $\eta$ is non-negligible. For the right part of the difference, there are only uniformly random values involved. More precisely, $G, xG, yG$, and $zG$ are values that are chosen independently and uniformly at random. As there is no structure that can be used to correlate the values, the best that $A$ can do is to return some number from $\{1, \ldots, k\}$. This number will hit $\ell'$ with probability $1/k$. Consequently, using $A$ enables $B$ to break the DDH problem with non-negligible probability. This contradicts our intractability assumption and we conclude that no such attacker $A$ can exist and our protocol provides anonymity.

**Unlinkability across services (inter-service unlinkability).** The last property of service-specific pseudonyms is unlinkability across services, i.e., it is computationally not possible to associate two pseudonyms from the same user but for different services with each other. We first define the desired security property and proceed directly with proving that our construction satisfies that definition.

**Definition 6 (Pseudonym-based Unlinkability across Services).** We say pseudonyms (as constructed in Section III) are unlinkable across services if and only if given a verification key $vk := xG$, an associated pseudonym $psd_1 := xS_1$ for service $S_1$, and a pseudonym $psd_2$ for service $S_2 \neq S_1$, it is computationally infeasible to decide whether $psd_2 = xS_2$, i.e., to decide whether the two pseudonyms belong to the same user or not.

**Theorem 8 (Unlinkability of Pseudonyms across Services).** In the random oracle model and under the DDH assumptions, pseudonyms (as constructed in Section III) are unlinkable across services.

**Proof:** Suppose $A$ is an attacker that takes as input a tuple of the form $(G, vk, S_1, psd_1 := x_1S_1, S_2, psd_2 := x_2S_2)$ where $S_1 \neq S_2$, and decides whether $x_1 = x_2$ with a probability $1/2 + \eta$, where $\eta$ is non-negligible; $1 \leftarrow A$ denotes that $x_1 = x_2$ and $0 \leftarrow A$ denotes that $x_1 \neq x_2$. We use this attacker to construct attacker $B$ against the decisional Diffie-Hellman assumption.

The DDH challenger randomly chooses $b \in R \{0, 1\}$. If $b = 1$, the challenger produces a valid DDH tuple $(G, xG, yG, zG)$ for uniformly random values $x$ and $y$, where $z = x \cdot y$. If $b = 0$, $z$ is randomly chosen. We construct attacker $B$ that uses $A$ to solve the given DDH challenge.

First, $B$ chooses $r \in R \{1, \ldots, |G_1|\}$, sets $vk := xG$, $S_1 := rG$, $psd_1 := rvk$, $S_2 := yG$, $psd_2 := zG$, calls $A(G, vk, S_1, psd_1, S_2, psd_2)$, and answers the challenge with $A$’s answer. We observe that $S_1$ looks like the output of a hash function (i.e., a random oracle) and that $psd_1$ is the pseudonym associated to $vk$ and $S_1$. Notice that $psd_2 = xS_2$ if and only if $z = x \cdot y$.

Let us now compute the success probability of our constructed adversary against the DDH challenge, where we let $z$
\[
\begin{align*}
\text{Adv}^{\text{DH}}(B) &= \Pr[1 \leftarrow B(\mathcal{G}, x_0 G, y_0 G, x_1 y_0 G) \mid b = 1] - \Pr[1 \leftarrow B(\mathcal{G}, x_0 G, y_0 G, z_0 G) \mid b = 0] \\
\overset{(1)}{=} & \Pr[r \in \mathbb{R} \{1, \ldots, |\mathcal{G}_1|\} : 1 \leftarrow A(\mathcal{G}, x_0 G, r G, x_0 G, y_0 G, x_1 y_0 G) \mid b = 1] \\
& - \Pr[r \in \mathbb{R} \{1, \ldots, |\mathcal{G}_1|\} : 1 \leftarrow A(\mathcal{G}, x_0 G, r G, r x_0 G, y_0 G, z_0 G) \mid b = 0] \\
\overset{(2)}{=} & \Pr[r \in \mathbb{R} \{1, \ldots, |\mathcal{G}_1|\} : 1 \leftarrow A(\mathcal{G}, x_0 G, r G, r x_0 G, y_0 G, z_0 G) \mid b = 1] \\
& - (1 - \Pr[r \in \mathbb{R} \{1, \ldots, |\mathcal{G}_1|\} : 0 \leftarrow A(\mathcal{G}, x_0 G, r G, r x_0 G, y_0 G, z_0 G) \mid b = 0]) \\
\overset{(3)}{=} & \frac{1}{2} + \eta - (1 - (\frac{1}{2} + \eta)) = 2\eta
\end{align*}
\]

In step (1), we substitute attacker \( B \) with its definition, and in step (2), we consider that if \( A \) decides if \( z = x_y \) then \( A \) also decides if \( z \neq x_y \). In step (3), we substitute \( A \) with its success probability. Thus, attacker \( B \) can use \( A \) to break the decisional Diffie-Hellman challenge with non-negligible probability, which violates our assumption. We conclude that no attacker \( A \) exists.

**Anonymity of Escrow Identifiers**

In parallel to the proofs for service-specific pseudonyms, we prove that escrow identifiers are unique, preserve the anonymity of users, and that they are unlinkable across services. The cryptographic construction strongly resembles that of service-specific pseudonyms but the amount of information available to outside parties is different and requires a different proof. We use most of the notation introduced above and begin by stating the uniqueness property.

**Definition 7 (Uniqueness of escrow identifiers).** We say that escrow identifier are unique if and only if, the following conditions hold with overwhelming probability:

1. For any service \( S \), two verification keys \( v k_1 \) and \( v k_2 \), and two honestly-generated escrow values \( \mathcal{R}_1 \) and \( \mathcal{R}_2 \), \( Y_{v k_1}(\mathcal{R}_1, S) \neq Y_{v k_2}(\mathcal{R}_2, S) \).
2. For any verification key \( v k \), corresponding escrow value \( \mathcal{R} \), and service \( S \), \( Y_{v k}(\mathcal{R}, S) \) (i.e., the escrow identifier) is a unique value.
3. For any two different service descriptions \( S_1 \) and \( S_2 \) and verification key \( v k \) and corresponding escrow value \( \mathcal{R} \), \( Y_{v k}(\mathcal{R}, h(S_1)) \neq Y_{v k}(\mathcal{R}, h(S_2)) \).

**Theorem 9 (Uniqueness of Escrow Identifiers).** In the random oracle model, escrow identifiers are unique.

**Proof:**

Condition 1: for any service \( S \), two verification keys \( v k_1 \) and \( v k_2 \), and two honestly-generated escrow values \( \mathcal{R}_1 := t_1 \cdot v k_1 \) and \( \mathcal{R}_2 := t_2 \cdot v k_2 \), \( Y_{v k_1}(\mathcal{R}_1, S) = Y_{v k_2}(\mathcal{R}_2, S) \) if and only if \( t_1 = t_2 \) or \( S = \mathcal{O} \), the two random values \( t_1 \) and \( t_2 \) are equal only with negligible probability and \( S = \mathcal{O} \) happens only with negligible probability by Proposition 1.

Condition 2: follows immediately since \( Y \) is a deterministic function.

Condition 3: for any two different service descriptions \( S_1 \) and \( S_2 \) and verification key \( v k \) and corresponding escrow value \( \mathcal{R} := t \cdot v k \), \( Y_{v k}(\mathcal{R}, h(S_1)) = Y_{v k}(\mathcal{R}, h(S_2)) \) if and only if \( h \) maps \( S_1 \) and \( S_2 \) to the same value or if \( t = 0 \). The former happens only with negligible probability by Proposition 1, the latter also happens only with negligible probability.
Theorem 11 (Unlinkability of Escrow Identifiers across Services). Let $M = \{ (vk_1, R_1 := r_1 tvk_1), \ldots, (vk_m, R_m := r_m tvk_m) \}$ be a set of verification keys and their corresponding random values (induced by the signing process of the EA). We say that escrow identifiers (as constructed in Section III) provide anonymity if, given $k$ services $S := (S_1, \ldots, S_k)$ and $k$ corresponding escrow identifier $I := (idr_1 := r_i S_1, \ldots, idr_k := r_i S_k)$, it is computationally infeasible to decide which $(vk, R)$ pair is associated with the services and escrow identifier (i.e., which $(vk, R)$ is such that $R = r_i v k$).

The following theorem states that escrow identifiers preserve the anonymity of users.

Theorem 10 (Anonymity for Escrow Identifiers). In the random oracle model and under the DDH assumptions, escrow identifiers (as constructed in Section III) provide anonymity.

Proof: We reduce this problem against the decisional Diffie-Hellman problem. Suppose we are given an attacker $A$ that breaks the anonymity of escrow identifiers, i.e., given $M$, $S$, and $I$ as in Definition 8 $A$ outputs the correct $i$ with probability greater than $1/m + \mu$ where $\mu$ is non-negligible in the security parameter $n$. From this attacker, we can construct attacker $B$ that breaks the DDH problem.

We are given the tuple $(G, uG, vG, uG)$ and are to decide whether $w = u \cdot v$. First we draw $x_i$, $r_i$, and $s_i$ uniformly at random from the set of exponents $\{1, \ldots, |G_1|\}$, and draw $\ell \in R \{1, \ldots, m\}$. Let $M := \{ (x_1 G, x_1 r_1 G), \ldots, (x_{m - 1} G, x_{m - 1} r_{m - 1} G), (x_m G, x_m r_1 G), \ldots, (x_{m - 1} G, x_{m - 1} r_{m - 1} G), \ldots, (x_{m - 1} G, x_{m - 1} r_{m - 1} G) \}$, let $S := (vG, s_2 vG, \ldots, s_k vG)$, and let $I := (wG, s_2 wG, \ldots, s_k wG)$. We run $A(G, M, S, I)$. The set $M$, $S$, and $I$ now have exactly the shape of $M$, $S$, and $I$ from the theorem, respectively, if $w = u \cdot v$. If $A$ can figure out $\ell$, then our attacker $B$ can break the given DDH challenge.

Let us now compute the advantage of our construction $B$ against the given DDH challenge:

$$\text{Adv}^{\text{DDH}}(B) = | \Pr[1 \leftarrow B(G, uG, vG, wG) \mid b = 1] - \Pr[1 \leftarrow B(G, uG, vG, wG) \mid b = 0] |$$

$$\stackrel{(1)}{=} | \Pr[| \{x_i, r_i, s_i \in R \{1, \ldots, |G_1|\}; \ell \in R \{1, \ldots, m\}: \ell' \leftarrow A(G, M, S, I) \mid \ell = \ell' \wedge b = 1] - \Pr[| \{x_i, r_i, s_i \in R \{1, \ldots, |G_1|\}; \ell \in R \{1, \ldots, m\}: \ell' \leftarrow A(G, M, S, I) \mid \ell = \ell' \wedge b = 0] |$$

$$\stackrel{(2)}{=} | \frac{1}{m} + \mu - \frac{1}{m} | = \mu$$

Equality (1) holds as we substituted $B$ by its definition. For equation (2), the left part holds by assumption, $A$ succeeds with a probability non-negligibly higher than $1/m$. For the right part, however, the DDH challenge consists of $4$ uniformly random values that are not correlated in any way and the best that $A$ can do is to output some number $k'$. With probability $1/m$, this number will coincide with $k$. Since $\mu$ is non-negligible, it contradicts our assumption that DDH is a computationally intractable. We conclude that no attacker $A$ exists.

Due to the very close construction of escrow identifiers and SSPs, the definition for unlinkability of escrow-identifiers across services is close to the definition of unlinkability of SSPs across services.

Definition 9 (Escrow-identifier-based Unlinkability across Services). We say escrow identifiers (as constructed in Section III) are unlinkable across services if and only if given a verification key $vk := xG$, an associated escrow value $R := t \cdot vk$, an escrow identifier $idr_1 := t S_1$ for service $S_1$, and an escrow identifier $idr_2$ for service $S_2 \neq S_1$, it is computationally infeasible to decide whether $idr_2 = x S_2$, i.e., to decide whether the two escrow identifiers belong to the same user or not.

The following theorem states the unlinkability of escrow identifiers across services.

Theorem 11 (Unlinkability of Escrow Identifiers across Services). In the random oracle model and under the DDH assumptions, escrow identifiers (as constructed in Section III) are unlinkable across services.

Proof: Suppose $A$ is an attacker that takes as input a tuple of the form $(D, vk, R := t \cdot vk, S_1, idr_1 := t S_1, S_2, idr_2)$ where $S_1 \neq S_2$, and decides whether $idr_2 = t \cdot S_2$ with a probability $1/2 + \eta$, where $\eta$ is non-negligible; $1 \leftarrow A$ denotes $A$’s decision that $idr_2 = t S_2$ and $0 \leftarrow A$ the decision that $idr_2 \neq t S_2$. We use this attacker to construct attacker $B$ against the decisional Diffie-Hellman assumption.

The DDH challenger randomly chooses $b \in R \{0, 1\}$. If $b = 1$, the challenger produces a valid DDH tuple $(G, xG, yG, zG)$ for uniformly random values $x$ and $y$, where $z = x \cdot y$. If $b = 0$, $z$ is randomly chosen. We construct attacker $B$ that uses $A$ to solve the given DDH challenge.

First, $B$ chooses $d, r \in R \{1, \ldots, |G_1|\}$ and sets $D := dG, vk := G, R := xG, S_2 := yG, idr_2 := zG, S_1 := r D,$ and $idr_1 := r \cdot d \cdot R$ and calls $A(D, vk, S_1, idr_1, S_2, idr_2)$ and answers the challenge with $A$’s answer. We observe that $vk = d^{-1} D$ is of the correct distribution and form for a verification key (since $d$ is chosen uniformly at random from a prime-order group, $d^{-1}$ exists and is a uniformly random value), $S_1 = r D$ and $idr_1 = r \cdot d \cdot R = r \cdot d \cdot xG = r xD = x S_1,$
where the form, i.e., formulas disjunctive normal form. While writing the verification functions, we assume that functions are written in disjunctive normal conjunctions of elementary formulas but does not contain disjunctions, and we write for some says formula for and sorted the explanations in topological order. We use knowledge proofs. We use the standard ML implementation and prove the desired properties about it. As suggested by Bengtson et al. [8], we use the standard ML notation to keep the code readable and we show how to encode this notation into RCF. We minimized forward references and sorted the explanations in topological order.

In step (1), we substitute attacker B with its definition, and in step (2), we consider that if A decides if \( z = xy \), then A also decides if \( z \neq xy \). In step (3), we substitute A with its success probability. Thus, attacker B can use A to break the decisional Diffie-Hellman challenge with non-negligible probability, which violates our assumption. We conclude that no attacker A exists.

B. Well-Typing Result for the API

For keeping the size of the present submission reasonable, we postpone the well-typedness proofs to the long version [9].

C. RCF Implementation of the API

This section details the implementation of our API (see Table I in RCF. The implementation of the API methods uses a sealing-based encoding of cryptographic primitives and relies on several standard library functions such as List.member. First, we introduce the necessary machinery including the RCF type system that we will ultimately use to type-check the implementation and prove the desired properties about it. As suggested by Bengtson et al. [8], we use the standard ML notation to keep the code readable and we show how to encode this notation into RCF. We minimized forward references and sorted the explanations in topological order.

In the remainder of this section, we use the following convention: we write \( F_e \) to denote an elementary formula, i.e., a formula for says, SSP and so on without conjunctions and disjunction. We write \( F^\land \) to denote a formula that may contain conjunctions of elementary formulas but does not contain disjunctions, and we write \( F^\lor \) to denote arbitrary formulas in disjunctive normal form. While writing the verification functions, we assume that functions are written in disjunctive normal form, i.e., formulas \( F^\lor \) are of the form

\[
F^\lor := \bigvee_{i=1}^{n} F^\land_i
\]

where the \( F^\land_i \) in turn are of the form

\[
F^\land_i := \bigwedge_{j=1}^{m} F_e^j
\]

for some \( n \) and \( m \). We justify this assumption below. Intuitively, formulas \( F_e, F^\land, \) and \( F^\lor \) can be proven with zero-knowledge proofs. We use \( F \) to denote general formulas that may also contain implications. These formulas naturally capture all statements provable with zero-knowledge proofs and additionally capture authorization policies that usually contain implications, i.e., negations.

1) Preliminaries: RCF Type System: This section briefly reviews the RCF type system components, namely, the types, subtyping and kinding. Bengtson et al. [8] describe all of these concepts in detail.

Typing environments. Type-checking and, in general, all judgments of a type system are always conducted relative to a typing environment. This typing environment \( E \) keeps track of bound variables, logical formulas, and so on. For instance, suppose we are given a typing environment \( E \) and we are to type-check the following code borrowed from Example 2.

\[
\text{let pf_s = mkSays y stud Eval(x ev, x sec)}
\]
After type-checking this line, the extended typing environment $E'$ inherits all information from $E$ and also records that $pf_s$ is a variable of type proof. More precisely, $E' := E, pf_s : proof$. How the entries affect the type-checking process depends on the currently proven judgment. In the case of RCF, the typing environment is monotonously increasing, i.e., there is no judgment that removes entries from the environment. In the following description, we assume that $E$ “fits” the current context; we formalize the meaning of “fits” and typing environment in Appendix D.

**RCF types and channels.** We give a brief overview of the RCF type system and the types that occur in our API. The RCF type system is a security type system that statically enforces authorization policies on distributed systems. As such, the types it supports are not integer or string. Rather, a type in RCF intuitively determines whom a value might originate from and whether it conveys information that can help to enforce authorization policies.

We overview the syntax of RCF types in Table II. The only basic type is the universal type unit. A value $v : unit$ does not convey any additional information besides the value itself. In general, the RCF type unit captures all concrete untrusted values, i.e., values can be send over and received from the Internet such as Boolean values, principal identifiers, strings and so on. The name unit is a little misleading since the RCF type unit is populated by a plethora of values and does not correspond to the unit type in OCaml and F#. In these two language, the empty tuple () is the only value of type unit.

Based on the basic type unit, more complicated RCF types are constructed as follows:

- **Refinement types $\{ x : T \mid F^v \}$:**
  Refinement types determine the type of a given value and additionally transport a logical formula that may depend on the value itself. More precisely, a value $v$ of type $v : \{ x : T \mid F^v \}$ is first of all a value of type $T$ and additionally, the formula $F^v \{(v/x)\}$ holds, i.e., the formula $F^v$ where every occurrence of $x$ is replaced by $v$. These refinement types are a salient tool to convey logical predicates from one principal in the system to another. For instance, let us reconsider the lecture evaluation system from Section II a value $v : \{ x : unit \mid Eval(x, course) \}$ carries the logical predicate $Eval(v, course)$. Concretely, $v$ is an actual evaluation for the course $course$. The logical predicate will hold for the student that turns in the evaluation $v$ and after the professor receives $v$, the logical predicate will also be valid for the professor. In a concrete programming language, the value $v : \{ x : unit \mid Eval(x, course) \}$ might correspond to a text file that contains the lecture evaluation; in RCF, the fact that the file contains this kind of information is reflected in its type.

- **Dependent functions $\Pi x : T, U$ and**
  RCF uses standard dependent functions $\Pi x : T, U$ and dependent pairs $\Sigma x : T, U$. If $x$ does not occur in the type $U$, i.e., the function or the pair is not dependent on the first component, then the dependent function and the dependent pair corresponds to the usual function and pair types. If $x$ occurs in $U$, then the return value of a function or the second component of a tuple depends on the first component.

- **Disjoint sum types $T + U$:**
  Disjoint sum types $T + U$ are used to encode the usual ML data types. For instance, common ML data types such as datatype bool = true | false is encoded as a disjoint sum type. More precisely, the type $T + U$ is constructed
by the type constructors inl and inr; inl takes as input a value of type $T$ (i.e., the left side) and returns a value of type $T + U$ and the type constructor inr takes as input a value of type $U$ (i.e., the right side) and returns a value of type $T + U$. In the following, we encode $\text{bool} \triangleq \text{unit} + \text{unit}$ where we define $\text{true} := \text{inr}(\cdot)$ and $\text{false} := \text{inl}(\cdot)$.

Technically, all sum types are build from the $\text{inl}(T, T + U)$ and $\text{inr}(U, T + U)$ type constructors, i.e., more complicated types have to be encoded using only these two type constructors. For instance, the type $T + U + V = T + (U + V)$ until no disjoint sum types are added (here, this means that neither $T$, nor $U$, nor $V$ are disjoint sum types). As suggested by Bengtson et al. [8], we assume a unique encoding and use arbitrary data type constructors as syntactic sugar and we will use the usual ML datatype notation.

Iso-recursive types $\mu \alpha. T$:

Iso-recursive types are constructed with the fold type constructor and allow to define recursive data structures such as lists. The core idea of iso-recursive types $T := \mu \alpha. U$ is that the type variable $\alpha$ that occurs in $U$ can be replaced with $T$.

For instance, the usual ML lists datatype $\text{List} \equiv \text{nil} + \text{Cons}(\text{List} \times \text{List})$ for values of type $T$ are defined in RCF as $T \equiv \mu \alpha. \text{unit} + T \times \alpha$. Initially, lists are constructed from a value $\text{nil} : \text{unit}$ by applying the constructor $\text{Cons}(\text{unit} + T \times \{\mu \alpha. \text{unit} + T \times \alpha\}, \mu \alpha. \text{unit} + T \times \alpha)$ to the value $\text{inl} \text{nil}$. Unfolding $T \equiv \mu \alpha. \text{unit} + T \times \alpha$ yields $\text{unit} + T \times (\mu \alpha. \text{unit} + T \times \alpha)$; the unfolded type is a disjoint sum and can be matched to check if the list is empty (left case) or whether the list has a head and a tail (right case).

For a more formal and exhaustive discussion, we refer to Bengtson et al. [8] and the book by Gunter [76].

Concretely, we use iso-recursion to specify the types for our signature scheme. Intuitively, we verify signatures on verification keys (which encode principal identifiers) and therefore, the type of a verification key must be able to describe verification keys. As for the disjoint sum types, we use the standard ML notation as syntactic sugar to describe iso-recursive types.

We use the abbreviation $\{F\} := \{\_ | F\}$ to denote that a refinement type where the variable (denoted by the anonymous variable “\_”) does not occur in the formula and, as mentioned above, we use the convention that $\text{bool} := \text{unit} + \text{unit}$ where $\text{inl}(\cdot) := \text{true}$ and $\text{inl}(\cdot) := \text{false}$.

RCF does not support polymorphic types. Since in a program, only a finite number of types can occur, it is possible to encode this polymorphism by instantiating the type variable as needed [8]. For instance, suppose a program uses the polymorphic function $\text{fail}(\alpha) : \text{unit} \rightarrow \alpha$ once with the return type $\alpha := T$ and once with the return type $\alpha := U$, for some types $T$ and $U$; the $(\alpha)$ denotes that the $\alpha$ is all-quantified. This can be translated into two functions $\text{fail}^T : \text{unit} \rightarrow T$ and $\text{fail}^U : \text{unit} \rightarrow U$ that instantiate $\alpha$ with $T$ and $U$, respectively. Since an implementation has only finitely many occurrences of polymorphic functions, the translation is also finite and well-founded. We use the same convention for polymorphic data types and we write datatype $(\alpha)T = U$ to denote the data type $T$ that is parameterized in $\alpha$; the scope of $\alpha$ is $U$.

Although channels do not occur in the API, we explain them for the sake of completeness. Communication in RCF is modeled via channels that represent public communication means such as the Internet or private communication means such as secured and authenticated TLS connections. Channels constitute the only possibility for protocol participants to exchange information: the sender puts a message $m$ onto the channel $a$, written “$\text{alm}$”, and the receiver can read a message from $a$, written “$\text{a}?$”. RCF channels are anonymous, i.e., the identity of a principal reading from or writing on a channel is not observable. The key insight is that channels are typed and the types assigned to values must be respected when these values are exchanged over communication channels. Therefore, a typed channel $a \downarrow T$ for values of type $T$ can only be used to exchange values of type $T$, i.e., a message $m$ can only be sent over $a \downarrow T$ if it is possible to prove that $m : T$ in the context of the sender. Consequently, all messages read from $a \downarrow T$ are always given type $T$ in the context of the receiver.

**Subtyping and kinding.** Type systems without the possibility to compare and subtype types are small but they severely lack expressiveness. For instance, a channel $a \downarrow \text{unit}$ for values of type $\text{unit}$ could not be used to transmit values of type $v : \{x : \text{unit} | F\}$, even though, $v$ is a value of type $\text{unit}$. RCF solves this restriction by means of an elaborate subtyping mechanism. Table VIII in Appendix D states all kinding rules.

Subtyping in RCF relies on two concepts: standard subtyping (i.e., subtyping rules without the SUB PUBLIC TAINTED rule) purely based on types and a mechanism based on kinding (i.e., the subtyping relation characterized by the SUB PUBLIC TAINTED rule and the kinding rules). For instance, standard subtyping allows us to deduce that values of type $\{x : \text{unit} | F^V\}$ are also of type $\text{unit}$. We write $E \vdash T <: U$ to denote that type $T$ is a subtype of $U$ under typing environment $E$, i.e., $E$ proves that a value of type $T$ can be used in place of a value of type $U$. Intuitively, kinding decides subtyping based on whether a type is of kind public, i.e., it can be send to the attacker, or whether a type is of kind tainted,
Indeed, the kinding rule for functions $K_{\text{IND}}$ of kind $\text{intuitively safe to give the attacker access to function } f$. $E$ relation, however, can decide $\exists y. P_k(x, \ldots, x_{\ell_k})$.

For instance, purely type-based subtyping cannot decide whether a functional type $T \to U$ can be sent over a channel $\alpha \triangleleft \text{unit}$. The reason is that the subtyping rule $\text{SUB FUN}$ for function types only relates two function types. The kinding relation, however, can decide $E \vdash T \to U <: \text{unit}$ if the function $f$ of type $T \to U$ can be given kind pub (public), i.e., it is intuitively safe to give the attacker access to function $f$. Intuitively, if a function takes attacker-provided values (i.e., values of kind tnt) as input and outputs a value that can be given to the attacker (i.e., a value of kind pub), then this function can be passed to the attacker. Indeed, the kinding rule for functions $\text{KIND FUN}$ formalizes this intuition.

It states that a function type is public if the arguments are tainted and the output is public.

2) RCF Types of the API Methods: The API methods introduced in Table I and the refined verify method from Section V contain several abstract types. In the remainder of this section, we instantiate these abstract, unspecified types with RCF types and state the RCF code of the cryptographic realization of the API methods. We start with the simple types such as $\text{bitstring}$. These types are defined to be $\text{unit}$ but for the sake of readability and to emphasize the intended use of the corresponding values, we leave the more meaningful names. Table III gives a complete overview of these types.

Table IV: Definition of the signing key type $\text{sigkey}$ and verification key type $\text{verkey}$.

Table III: Instantiation of types in RCF.

$$
T_{\text{vk}}^k := \sum_{k=1}^n P_{\text{sk}}^S(x_1 \cdot T_{\ell_k}^k \cdots \times x_{\ell_k-1} : T_{\ell_k}^k \{x_{\ell_k} : (y \text{ says } P_k(x_1, \ldots, x_{\ell_k}))\})
$$

$$
T_{\text{sk}} := y : \text{bitstring} \times T_{\text{vk}}^k
$$

$$
U_{\text{sk}}^k := \sum_{k=1}^n P_{\text{sk}}^S(x_1 : T_{\ell_k}^k \cdots \times x_{\ell_k-1} : T_{\ell_k}^k \{x_{\ell_k} : (y \text{ says } P_k(x_1, \ldots, x_{\ell_k}))\})
$$

where $T_{\ell_k}^i \in \{\text{bitstring}, \alpha\}$

$$
\text{verkey} := \mu\alpha. \text{signature} \to T_{\text{vk}}
$$

$$
\text{sigkey} := (\mu\alpha. T_{\text{vk}} \to \text{signature}) \times \text{verkey} (T_{\text{vk}}\{\text{verkey}/\alpha\} \to \text{signature}) \times \text{verkey}
$$

Table IV: Definition of the signing key type $\text{sigkey}$ and verification key type $\text{verkey}$.

i.e., it can originate from the attacker.

\[
\begin{array}{c}
\text{SUB PUBLIC TAINTED}
\end{array}
\]

\[
\begin{array}{c}
E \vdash T :: \text{pub} \quad E \vdash U :: \text{tnt}
\end{array}
\]

\[
E \vdash T <: U
\]

It states that a function type is public if the arguments are tainted and the output is public.

For instance, purely type-based subtyping cannot decide whether a functional type $T \to U$ can be sent over a channel $\alpha \triangleleft \text{unit}$. The reason is that the subtyping rule $\text{SUB FUN}$ for function types only relates two function types. The kinding relation, however, can decide $E \vdash T \to U <: \text{unit}$ if the function $f$ of type $T \to U$ can be given kind pub (public), i.e., it is intuitively safe to give the attacker access to function $f$. Intuitively, if a function takes attacker-provided values (i.e., values of kind tnt) as input and outputs a value that can be given to the attacker (i.e., a value of kind pub), then this function can be passed to the attacker. Indeed, the kinding rule for functions $\text{KIND FUN}$ formalizes this intuition.

It states that a function type is public if the arguments are tainted and the output is public.

2) RCF Types of the API Methods: The API methods introduced in Table I and the refined verify method from Section V contain several abstract types. In the remainder of this section, we instantiate these abstract, unspecified types with RCF types and state the RCF code of the cryptographic realization of the API methods. We start with the simple types such as $\text{bitstring}$. These types are defined to be $\text{unit}$ but for the sake of readability and to emphasize the intended use of the corresponding values, we leave the more meaningful names. Table III gives a complete overview of these types.

Table III: Instantiation of types in RCF.

\[
\begin{array}{c}
T_{\text{vk}}^k := \sum_{k=1}^n P_{\text{sk}}^S(x_1 : T_{\ell_k}^k \cdots \times x_{\ell_k-1} : T_{\ell_k}^k \{x_{\ell_k} : (y \text{ says } P_k(x_1, \ldots, x_{\ell_k}))\})
\end{array}
\]

\[
T_{\text{sk}} := y : \text{bitstring} \times T_{\text{vk}}^k
\]

\[
U_{\text{sk}}^k := \sum_{k=1}^n P_{\text{sk}}^S(x_1 : T_{\ell_k}^k \cdots \times x_{\ell_k-1} : T_{\ell_k}^k \{x_{\ell_k} : (y \text{ says } P_k(x_1, \ldots, x_{\ell_k}))\})
\]

where $T_{\ell_k}^i \in \{\text{bitstring}, \alpha\}$

\[
\text{verkey} := \mu\alpha. \text{signature} \to T_{\text{vk}}
\]

\[
\text{sigkey} := (\mu\alpha. T_{\text{vk}} \to \text{signature}) \times \text{verkey} (T_{\text{vk}}\{\text{verkey}/\alpha\} \to \text{signature}) \times \text{verkey}
\]

Table IV: Definition of the signing key type $\text{sigkey}$ and verification key type $\text{verkey}$.

\[
\begin{array}{c}
\text{SUB PUBLIC TAINTED}
\end{array}
\]

\[
\begin{array}{c}
E \vdash T :: \text{pub} \quad E \vdash U :: \text{tnt}
\end{array}
\]

\[
E \vdash T <: U
\]

It states that a function type is public if the arguments are tainted and the output is public.

2) RCF Types of the API Methods: The API methods introduced in Table I and the refined verify method from Section V contain several abstract types. In the remainder of this section, we instantiate these abstract, unspecified types with RCF types and state the RCF code of the cryptographic realization of the API methods. We start with the simple types such as $\text{bitstring}$. These types are defined to be $\text{unit}$ but for the sake of readability and to emphasize the intended use of the corresponding values, we leave the more meaningful names. Table III gives a complete overview of these types.

Table III: Instantiation of types in RCF.

\[
\begin{array}{c}
T_{\text{vk}}^k := \sum_{k=1}^n P_{\text{sk}}^S(x_1 : T_{\ell_k}^k \cdots \times x_{\ell_k-1} : T_{\ell_k}^k \{x_{\ell_k} : (y \text{ says } P_k(x_1, \ldots, x_{\ell_k}))\})
\end{array}
\]

\[
T_{\text{sk}} := y : \text{bitstring} \times T_{\text{vk}}^k
\]

\[
U_{\text{sk}}^k := \sum_{k=1}^n P_{\text{sk}}^S(x_1 : T_{\ell_k}^k \cdots \times x_{\ell_k-1} : T_{\ell_k}^k \{x_{\ell_k} : (y \text{ says } P_k(x_1, \ldots, x_{\ell_k}))\})
\]

where $T_{\ell_k}^i \in \{\text{bitstring}, \alpha\}$

\[
\text{verkey} := \mu\alpha. \text{signature} \to T_{\text{vk}}
\]

\[
\text{sigkey} := (\mu\alpha. T_{\text{vk}} \to \text{signature}) \times \text{verkey} (T_{\text{vk}}\{\text{verkey}/\alpha\} \to \text{signature}) \times \text{verkey}
\]
logical predicate $P_k$ itself, namely, $y$ says $P_k(x_1, \ldots, x_{\ell_k})$. This refinement establishes the connection between the logical predicate and its representation in the disjoint sum. In this definition of $\mathcal{T}_v^\alpha$, $y$ is intentionally left unbound. Intuitively, the reason is that $y$ corresponds to the verification key of the principal running the code. The RCF types, however, cannot statically characterize dynamic properties such as ownership values. To accommodate the free variable $y$ in $\mathcal{T}_v^\alpha$, we bind $y$ in the type $\mathcal{T}_v$. The value $y$ will be instantiated by a verification key (we will later see that $\text{verkey} < \text{unit}$, i.e., a verification key is a sub-type of $\text{unit}$ and therefore, we can use $y$ to denote a verification key). Intuitively, the code for the signature construction enforces that $y$ corresponds to the principal that states a logical predicate (more precisely, the signing function will place the unsealing function of a signing key as $y$ while creating signatures).

We defined $\text{verkey}$ as $\mu \alpha. \text{signature} \to \mathcal{T}_v$ and $\text{sigkey}$ as $\mu \alpha. \mathcal{T}_v \to \text{signature}$. As described in Section V, we use a sealing-based encoding of cryptographic primitives and $k : \text{sigkey}$ is a seal and $\text{verkey}$ is the unsealing function of $k$. If a value is applied to the sealing function of a value $k : \text{sigkey}$, the resulting value of type $\text{signature}$ is interpreted as signature. Applying the unsealing function, i.e., the verification key, to a signature returns the sealed value, i.e., the signed message, with the original type.

The type $\mathcal{U}_v^\alpha$ corresponds to $\mathcal{T}_v^\alpha$ for the purpose of signing, i.e., it connects the verification key used in a says modality during the signing process with the signing key $sk$. As in the definition of $\mathcal{T}_v^\alpha$, the type $\mathcal{U}_v^\alpha$ is a large disjoint sum type that contains one case for every possible predicate $P_k$ that can occur. The main addition is that the refinement of the last element in the tuple: there, we make a statement about the structure of signing keys and extract the verification key, i.e., the second component of the signing key, and use it to instantiate the actor in the says modality (denoted by $y$). Since every principal must split (using rule EXP SPLIT) her signing key in order to obtain her verification key, every principal will know the logical predicate $sk = (\_, \text{vk})$ where $sk$ is her signing key and $\text{vk}$ is the verification key contained in $sk$. Using this logical formula and the existential quantification occurring in the type $\mathcal{U}_v^\alpha$, a principal can derive the logical refinement required by the type $\mathcal{T}_v$. More precisely, given the logical predicates $\exists z. x. sk = (z, y) \wedge y$ says $P_k(x_1, \ldots, x_{\ell_k})$ and $sk = (\_, \text{vk})$, one can derive $\text{vk}$ says $P_k(x_1, \ldots, x_{\ell_k})$.

**Instantiating types.** We discuss our instantiation of the types we left abstract in the definition of the API. As stated in Section III, we instantiate the types $\text{uid}$ and $\text{uid}_\text{pub}$ as signing and verification key, respectively, i.e., we define $\text{uid} := \text{sigkey}$ and $\text{uid}_\text{pub} := \text{verkey}$. We will prove later that the verification key type is public, i.e., verification keys can be sent on a public channel and, in particular, verification keys can be given to the attacker.

Similar to verification keys, proof type $\text{proof}$ and formula type $\text{formula}$ have to be public, i.e., accessible by the attacker; otherwise, the type system would prevent us from sending values of this type over an attacker-readable channel, i.e., the Internet. The instantiations of the remaining abstract types are stated in Table V and we argue in Appendix D that $\text{proof}$ and $\text{pred}$ are indeed public.

$\alpha \text{ RevHid}$:

The type $\alpha \text{ RevHid}$ represents values that will occur in a formula. The values can be either revealed, denoted by the type constructor Revealed $x$, or they can be hidden, denoted by Hidden $z$. The argument $z$ of the Hidden and type constructor acts as a positional index and is used to establish the equality among hidden values. For instance, consider the formula Revealed $\text{vk}$ says $\text{Good}^F$ (Revealed $m$, Revealed $m$). A proof for this formula can be changed into a proof for the formula Revealed $\text{vk}$ says $\text{Good}^F$ (Hidden $1$, Hidden $1$), i.e., the first and the second argument to the predicate $\text{Good}$ are the same, indicated by the same index. We stress that the converse is not true, since this proof would also verify for the formula Revealed $\text{vk}$ says $\text{Good}^F$ (Hidden $1$, Hidden $2$), i.e., different arguments to the type constructor Hidden do not imply that the hidden values are different.

$\text{predicate}^F$ and $\text{predicate}^P$:

The types $\text{predicate}^F$ and $\text{predicate}^P$ describe logical predicates that are used in combination with the says modality.

The type $\text{predicate}^F$ is a sum type, one case for each predicate proven in a protocol. The arity of each case $\text{P}_i^F$ matches the corresponding case $\text{P}_i^S$ of the verification key type. Each element, however, is of type $\alpha \text{ RevHid}$ to allow the distinction between hidden and revealed user identifiers ($\alpha := \text{uid}_\text{pub}$) and between hidden and revealed values ($\alpha := \text{bitstring}$).

The type $\text{predicate}^P$ is also a sum type with one case for each predicate proven in a protocol. The representation of the elements differs when compared to $\text{predicate}^F$: the high-level type $\text{predicate}^F$ describes if a value is revealed or hidden, the type $\text{predicate}^P$ closely models the corresponding cryptographic implementation. As such, each element Revealed $x$ in a case $\text{P}_i^F$ is represented by three values: the commitment $c$ to $x$, and the opening information $(w, r)$ for $c$ where $x = w$. If a value is hidden with the hide API method, then the opening information is removed, i.e., the representation $(c, (w, r))$ is changed to $(c, (\_, \_))$. 

<table>
<thead>
<tr>
<th>datatype</th>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>α list = Cons of α * α list</td>
<td>NIL</td>
</tr>
<tr>
<td>datatype α RevHid =</td>
<td>Revealed of α</td>
</tr>
<tr>
<td></td>
<td>Hidden of integer</td>
</tr>
</tbody>
</table>

\[
predicatex := \begin{align*}
| P^1 \text{ of } T^1_1 \cdots T^1_{n_1} & \quad \cdots \quad \text{(commitment} \ast (T^1_1 \ast \text{random})) \\
| \cdots & \\
| P^m \text{ of } T^m_1 \cdots T^m_{n_m} & \quad \cdots \quad \text{(commitment} \ast (T^m_{n_m} \ast \text{random}))
\end{align*}
\]

where \( T^i_j \in \{\text{bitstring RevHid, uid}_{pub} \text{ RevHid}\} \)

<table>
<thead>
<tr>
<th>datatype</th>
<th>formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Says of} \ (z : \text{uid}_{pub} \ \text{RevHid}) \ast \text{predicatex}</td>
<td></td>
</tr>
<tr>
<td>\text{SSP of} \ (z : \text{uid}_{pub} \ \text{RevHid}) \ast (s : \text{bitstring RevHid}) \ast (\text{psd} : \text{bitstring RevHid})</td>
<td></td>
</tr>
<tr>
<td>\text{LM of} \ (x : \text{bitstring RevHid}) \ast (b : \text{bitstring RevHid}) \ast (\ell : (\text{bitstring} \ast \text{bitstring}) \text{ list})</td>
<td></td>
</tr>
<tr>
<td>\text{LNM of} \ (x : \text{bitstring RevHid}) \ast (\ell : (\text{bitstring} \ast \text{bitstring}) \text{ list})</td>
<td></td>
</tr>
<tr>
<td>\text{REL of} \ (x : \text{bitstring RevHid}) \ast (op : \text{string}) \ast (y : \text{bitstring RevHid})</td>
<td></td>
</tr>
<tr>
<td>\text{EscrowInfo of} \ (z : \text{uid}<em>{pub}) \ast (x : \text{uid}</em>{pub} \ \text{RevHid})</td>
<td></td>
</tr>
<tr>
<td>\text{And of} \ \text{formula} \ast \text{formula}</td>
<td></td>
</tr>
<tr>
<td>\text{Or of} \ \text{formula} \ast \text{formula}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>datatype</th>
<th>statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Says of} \ ((c_z : \text{commitment}) \ast ((z : \text{bitstring}) \ast (r_z : \text{random})))</td>
<td></td>
</tr>
<tr>
<td>\text{SSP of} \ ((c_z : \text{commitment}) \ast ((z : \text{bitstring}) \ast (r_z : \text{random}))) \ast \text{predicatex}</td>
<td></td>
</tr>
<tr>
<td>\text{LM of} \ ((c_x : \text{commitment}) \ast ((s : \text{string}) \ast (r_x : \text{random})))</td>
<td></td>
</tr>
<tr>
<td>\text{LNM of} \ ((c_x : \text{commitment}) \ast ((x : \text{bitstring}) \ast (r_x : \text{random}))) \ast (\ell : (\text{bitstring} \ast \text{bitstring}) \text{ list})</td>
<td></td>
</tr>
<tr>
<td>\text{REL of} \ ((c_x : \text{commitment}) \ast ((x : \text{integer}) \ast (r_x : \text{random}))) \ast (op : \text{string})</td>
<td></td>
</tr>
<tr>
<td>\text{And of} \ (p_1 : \text{statement}) \ast (p_2 : \text{statement})</td>
<td></td>
</tr>
<tr>
<td>\text{Or of} \ (p_1 : \text{statement}) \ast (p_2 : \text{statement})</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>datatype</th>
<th>proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ZK of} \ (zkv : \text{zero-knowledge}) \ast (s : \text{statement})</td>
<td></td>
</tr>
</tbody>
</table>

Table V: Data type declarations used by the API methods.
**formula and statement:**

The type `formula` captures the logical formula proven by a zero-knowledge proof. The type `statement` captures the corresponding statement. For instance, a proof that shows a signature verification statement conveys a logical says formula (see Section III-A). As such, the type `formula` focuses on the logical formulas while the type `statement` is centered about the cryptographic implementation of the corresponding proofs. Every logical formula provable by the API is represented by a case in `formula` and `statement`. In all the cases, the encoding of values in type `formula` uses the type \( \alpha \ \text{RevHid} \) to express whether a value is revealed or hidden and whether a value is a principal identifier or not. The translation of values from formulas into statements is the same as for `predicate` and `predicate`: every value \( v \) in `formula` is translated into three values \( \langle c, (w, r) \rangle \) in `statement` (cf. above). Since `statement` corresponds closely to the cryptographic implementation, it contains more entries than `formula`. We highlight these differences in the following description.

Says modality proofs `Says` and `Says_p`:
- `Says` and `Says_p` canonically encode the formula \( z \ \text{says} \ p \) where \( p \) corresponds to the predicate encoded by the `predicateF` in `Says` and `predicateP` in `Says_p`. The statement `Says_p` additionally contains the cryptographic signature (i.e., commitment plus opening information).

Pseudonym ownership proofs `SSP` and `SSP_p`:
- The encoding of service-specific pseudonyms contains the principal identifier \( z \) of the owner of the pseudonym, the service \( s \), and the pseudonym `psd` itself. The proof `SSP_p` additionally contains a commitment \( c_x \) to the signing key used in the computation of the pseudonym. Since we stipulate that the signing key is never be revealed, no opening information for the commitment \( c_x \) is present.

List membership proofs `LM` and `LM_p`:
- The list-membership branch of the types `formula` and `statement` consists of the pseudonym \( x \), the attribute \( b \), and the list \( \ell \). Since we disallow the list to be hidden, the list occurs in plain in `formula` and in `statement` (i.e., only the plain value without commitment and opening information).

List non-membership proofs `LNM` and `LNM_p`:
- The list-nonmembership branch of the types `formula` and `statement` consists of the pseudonym \( x \) and the list \( \ell \). Since we disallow the list to be hidden, the list occurs plainly in `formula` and in `statement` (i.e., only the plain value without commitment and opening information).

Relational proofs `REL` and `REL_p`:
- The encoding of relational proofs contains the two operands \( x \) and \( y \) as well as the relation \( \text{op} \). Since the proven Groth-Sahai equations allow for deducing the proven operation, the operation occurs in plain in `formula` and `statement`.

Identity escrow proofs `EscrowInfo` and `EscrowInfo_p`:
- The identity escrow case of `formula` consists of the following components: the verification key of the trusted third party \( z \), the verification key \( x \) of the user, the value \( R \) that was signed and issued by the TTP, the service \( s \) for which the escrow identifier is issued, and the escrow identifier `idr` itself. The corresponding case of `statement` additionally contains the commitment \( c_r \) on the value \( r \) used to compute the escrow identifier. Since we stipulate that \( r \) is never revealed and that \( z \) (the public identifier of the trusted party) is never hidden, there is no opening information for \( c_r \) and no commitment for \( z \) in the proof.

Conjunctive proofs `And` and `And_p`:
- The conjunction case of types `formula` and `statement` consist only of the two sub-proofs.

Disjunctive proofs `Or` and `Or_p`:
- The disjunction case of types `formula` and `statement` consist only of the two sub-proofs.

**proof:**

A zero-knowledge proof \( \text{ZK}(zkv, stm) \) of type `proof` consists of a value `zkv` : `zero-knowledge`, modeling the actual zero-knowledge proof for the proven statement `stm` of type `statement`.

Several possibilities for implementing `zkv` exist. The implementation, however, has to exclude the following attack: given a valid proof for the statement `Or_p(p1, p2)`, it must not be possible to determine if the statement

\[2\text{ The value } zkv \text{ corresponds the pair } (\pi, \theta) \text{ in the concrete Groth-Sahai proof (cf. Groth and Sahai [7]).} \]
\( p_1 \) is valid, if the statement \( p_2 \) is valid, or if \( p_1 \) and \( p_2 \) both are valid. Without the \( zkv \) value, an attacker could pattern-match the type-constructor and extract the two sub-statements \( p_1 \) and \( p_2 \), call the verification method on either of these sub-statements and determine which of the two hold.

To counter this attack, we implement \( zkv \) using a dedicated seal. This seal takes as input the statement \( stm \) and a randomness. The returned handle is used as \( zkv \). Different randomness results in different \( zkv \) values, reflecting that many different zero-knowledge proofs for the same statement exists. The verification function, having access to the unsealing function, will use \( zkv \) to retrieve the sealed statement \( stm' \) and enforce that \( stm' \) and \( stm \) match. More precisely, the structure as well as the commitments of \( stm' \) and \( stm \) have to be equal.

The computational soundness result detailed below shows that this implementation and the matching of \( stm' \) and \( stm \) suffice for obtaining a computationally-secure zero-knowledge implementation.

### Creating fresh values and seals.

In RCF, we use only the function \( \text{mkUn} \) to create fresh values of type \( \text{unit} \); these values can be combined to form more complex types, for instance, using disjoint unions, tuples, or logical refinements. \( \text{mkUn} \) takes as input a value of type \( \text{unit} \) and outputs a fresh value of type \( \text{unit} \).

If the empty tuple \( () \) is the only non-functional value that occurs in the API, one may wonder how equality checks that occur in the API methods in form of if-statements, can ever fail. The intuitive reason is that RCF does not actually perform any equality checks but enters the then-branch of an if-statement under the premise that the equality check succeeded and it enters the else-branch under the premise that the check failed. Since \( \text{unit} \) together with all the other RCF types capture all values occurring in a concrete program, in a concrete execution of such a program, the equality checks will be performed on possibly different values, and then succeed or fail, respectively.

\[
\text{mkUn}: \text{unit} \rightarrow \text{unit}
\]

We encode cryptographic primitives using a sealing-based encoding. In RCF, seals are created using the polymorphic function \( \text{mkSeal}(\alpha) \) that takes as input a value of type \( \text{unit} \) and returns a seal for type \( \alpha \). The argument to \( \text{mkSeal} \) can be a (textual) description of the seal though. The first component of the returned pair is the sealing function, the second component is the unsealing function.

\[
\text{mkSeal}(\alpha): \text{unit} \rightarrow ((\alpha \rightarrow \text{unit}) \ast (\text{unit} \rightarrow \alpha))
\]

### External library functions and internal auxiliary functions.

The API uses several external standard library functions and internal auxiliary functions. These provide commonly used functionality such as list operations and computing hash values. Generally, the former functions are available via standard libraries that are shipped with a programming language. [Table VI] overviews the used functions with their type.

In the following, we describe each of the standard library functions and the auxiliary functions. We provide implementations for the auxiliary functions if they help to understand the API methods. Short implementations are inlined.

\[
\text{fail}(\alpha): \text{unit} \rightarrow \alpha
\]

While type-checking RCF code, it is crucial that all branches of the code have the same return type. Different branches are, for instance, introduced by a match-statement. For instance, the signature verification returns a strong type but it also has to return this type if the verification fails. The function \( \text{fail} \) enables us to write this verification function because \( \text{fail} \) can be typed to any return type. In a concrete implementation, \( \text{fail} \) throws an exception and stops the program execution.

\[
\text{rand}: \text{unit} \rightarrow \text{random}
\]

Randomness is an essential ingredient in zero-knowledge proofs: the commitments used in a proof require randomness and even the proof itself contains randomness. In RCF, randomness is generated by calling the \( \text{rand} \) function that takes as argument of type \( \text{unit} \) and return a fresh (random) value of type \( \text{unit} \).

\[
\text{let} \ \text{rand}: \text{unit} \rightarrow \text{random} =
\text{fun} \ _ \rightarrow \text{mkUn}()
\]

\[
\text{sign}: (sk : \text{sigkey}) \rightarrow (m : \mathcal{U}_{sk}^\text{\#}) \rightarrow \text{signature}
\]

The signing function takes as input a signing key \( sk : \text{sigkey} \) and a message \( m : \mathcal{U}_{sk}^\text{\#} \) and returns a signature of type \( \text{signature} \) for \( m \).
fail(α) : unit → α
rand : unit → random
sign : (sk : sigkey) → (m : U^α sk) → signature
checkSig : (y : verkey) → (sig : signature) → T_{vk}(verkey/α)
storeSK : sigkey → uid
restoreSK : uid → sigkey

List.member^i,j(α_1, ..., α_i, β_{i+1}, ..., β_j) :
(y_1 : α_1) → ... → (y_i : α_i) → (ℓ : α_1 * ... * α_i * β_{i+1} * ... * β_j list) →
{x : bool | x = true ⇔ ∃y_{i+1}, ..., y_j, (y_1, ..., y_j) ∈ ℓ}

List.get^i,j(α_1, ..., α_i, β_{i+1}, ..., β_j) :
(y_1 : α_1) → ... → (y_i : α_i) → (ℓ : α_1 * ... * α_i * β_{i+1} * ... * β_j list) →
{(y_1, ..., y_i, ..., y_j) : α_1 * ... * α_i * β_{i+1} * ... * β_j | ∃y_{i+1}, ..., y_j, (y_1, ..., y_j) ∈ ℓ}

getOperation : (op : string) → ((x : integer) → (y : integer) → bool)
cmp_op : (x : integer) → (y : integer) → {z : bool | z = true ⇔ x op y}

computeR : (x : bitstring) → (r : bitstring) → bitstring
computePsd : (sk : sigkey) → (s : string) → {x : pseudo | ∃y, z, sk = (y, z) ∧ SSP(z, s, x)}
computeIDR : (vk, E_A : bitstring) → (vk : bitstring) → (r : bitstring) → (R : bitstring) →
(s : string) → {idr : pseudo | EscrowInfo(vk, E_A, vk, R, s, idr)}

commit : bitstring * random → commitment
openCommit : commitment → bitstring * random
commit_{sk} : sigkey * random → commitment
openCommit_{sk} : commitment → sigkey * random
stripStm : statement → statement
checkZK : proof → bool

commitZK : statement * random → zero-knowledge
openZK : zero-knowledge → statement * random

fakestm : formula → statement
createZK^c : statement → random → proof * zero-knowledge * statement
createZK : statement → random → proof * zero-knowledge * statement

erand_{stm} : statement → statement → statement
rerand : statement → random → statement
proof : proof → statement → random → proof

checkEq(α) : α RevHid → bitstring → (integer * bitstring) list ref → bool
verify_{stm} : statement → formula → (integer * bitstring) list ref → bool
verify : proof → formula → bool
hide_{stm} : statement → formula → statement
combineOr : proof → formula → random → proof
commuteOr : proof → statement → proof
commuteAnd : proof → statement → proof
PKI : (x : uid_{pub}) → {y : verkey | x = y}

Table VI: Typed library and auxiliary functions used by the API methods.
let sign (sk : sigkey) (m : U_{sk}^*) : signature =
1  let (x, y) = sk in
2  x (y, m)

The type \( U_{sk}^* \) by itself not closed because the variable \( sk \) is free in \( U_{sk}^* \). Due to the dependent function that binds \( sk \) in the type \( U_{sk}^* \), the type of the signing function is closed. The user identifier \( y \) that is signed along with the message occurs in the logical refinement and determines the action of the says modality. Intuitively, the logical refinement is of the form \( y \) says \( m \).

check_{\text{sig}} : (y : \text{verkey}) \to (\text{sig} : \text{signature}) \to T_{\text{sk}} \{ \text{verkey}/\alpha \}  

The verification function takes as input a verification key \( y : \text{verkey} \) and a signature \( \text{sig} : \text{signature} \) and returns a value of type \( T_{\text{sk}} \{ \text{verkey}/\alpha \} \), i.e., the type \( T_{\text{sk}}^\alpha \) where all occurrences of \( \alpha \) are replaced by \( \text{verkey} \).

let check_{\text{sig}} (y : \text{verkey}) (\text{sig} : \text{signature}) : T_{\text{sk}} \{ \text{verkey}/\alpha \} =
1  let (x, m) = y \text{ sig};
2  if x = y then
3    m
4  else
5    \text{fail} (\text{T}_{\text{sk}} \{ \text{verkey}/\alpha \} )()

In particular, the implementation ensures that the verification key used to verify the signature is also the verification key that is signed along with the message.

storeSK : sigkey \to uid and

restoreSK : uid \to sigkey:

The functions storeSK and restoreSK are internally used to store and restore a signing key to and from a handle, respectively. The function storeSK takes as input a signing key and returns a handle for that key, the function restoreSK takes as input a handle and returns the stored signing key. We symbolically implement these functions as a seal.

let (storeSK, restoreSK) = mkSeal(sigkey) "Signing Key Storage";

List.member^{(i,j)}(\alpha_1, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_j) : (y_1 : \alpha_1) \to \cdots \to (y_i : \alpha_i) \to (\ell : \langle \alpha_1, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_j \rangle \text{list})

\to \{ x : \text{bool} \mid x = \text{true} \iff \exists y_{i+1}, \ldots, y_j : (y_1, \ldots, y_j) \in \ell \} \text{ and}

List.get^{(i,j)}(\alpha_1, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_j) : (y_1 : \alpha_1) \to \cdots \to (y_i : \alpha_i) \to (\ell : \langle \alpha_1, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_j \rangle \text{list})

\to \{ (y_1, \ldots, y_i, \ldots, y_j) : \alpha_1 \star \cdots \star \alpha_i \star \beta_{i+1} \star \cdots \star \beta_j \mid \exists y_{i+1}, \ldots, y_j : (y_1, \ldots, y_j) \in \ell \}:

The list operations are necessary for the list membership and the list non-membership proof. Both functions work on lists of tuples of any arity. For both functions, the index \( i \) expresses how many arguments are given to the function and the index \( j \) expresses the arity of the tuples stored in the list. We use the convention that the \( i \) arguments always represent the first \( i \) components of the tuples stored in the list. For instance, let \( \ell \) be a list that contains triples. Then List.member^{(1,3)} takes as argument a value \( a \) and returns a triple \( (a, b, c) \) such that \( (a, b, c) \in \ell \). The semantic of List.get^{(i,j)} is analogous.

getOperation : (op : string) \to ((x : integer) \to (y : integer) \to bool)

This helper function translates a function description into the corresponding function and it is used in the implementation of the general verification method, i.e., the verification method that returns a plain, non-refined Boolean. It takes as input a description \( x \in \{ =, \neq, \leq, \leq, <, >, >, \geq \} \) and returns the function \( f \) that implements the respective relational test. For instance, getOperation \( \leq \) returns a function that takes as input two arguments \( x \) and \( y \) and returns the Boolean result of \( x \leq y \). Internally, getOperation is a cascade of if statements.

cmp_{op} : (x : integer) \to (y : integer) \to \{ z : \text{bool} \mid z = \text{true} \iff x \text{ op } y \}:

The function \( \text{cmp}_{op} \) is functionally equivalent to getOperation \(" op"). The result is refined with a logical predicate that logically states the result. For instance, if \( x \leq y \), then \( \text{cmp}_< x \sim y \) returns the Boolean value \( z = \text{true} \) that is refined with the logical predicate \( z = \text{true} \iff x \leq y \). This method is used in the API verification method, which relies on the logical refinement.
computeR : (x : bitstring) → (r : bitstring) → bitstring,
computePsdf : (sk : sigkey) → (s : string) → {x : pseudo | ∃y, z. sk = (y, z) ∧ SSP(z, s, x)}, and
computeIDR : (vkEA : bitstring) → (vk : bitstring) → (r : bitstring) → (R : bitstring) → (s : string)
→ {idr : pseudo | EscrowInfo(vkEA, vk, R, s, idr)}:
The functions computeR, computePsdf, and computeIDR model the mathematical operations detailed in Section III. Although computeIDR and computePsdf perform the same mathematical operations, we use different RCF functions to distinguish between their different application scenarios and, in particular, between their different input and output types.

computeR takes as input two values r and sk of type bitstring and returns a value R of type bitstring.
computePsdf takes as input the signing key sk : sigkey and a service description s : string, and it returns the SSP psdf : pseudo for which additionally the logical predicate ∃y, z. sk = (y, z) ∧ SSP(z, s, psdf) holds.
computeIDR takes as input five values vkEA, vk, r, and R of type bitstring, and s of type string; it returns a value idr of type pseudo that additionally carries the logical formula EscrowInfo(vkEA, vk, R, s, idr). The inputs is necessary to bind the corresponding values occurring in the refinement; the value r occurs only implicitly in the formula since it is used to mathematically compute the value idr.

commit : bitstring * random → commitment and
openCommit : commitment → bitstring * random:
The functions commit and openCommit are sealing and unsealing functions, respectively, for values of type bitstring. We use the seal directly to model the commitment function and the handle to model commitments. The unsealing function openCommit is used only internally in the API methods; it cannot be accessed from the outside and it is not exported by the API.

1 let (commit, openCommit) = mkSeal(bitstring * random) "Commitment Seal";

commitak : sigkey * random → commitment and
openCommitak : commitment → sigkey * random:
The functions commitak and openCommitak are sealing and unsealing functions, respectively, of a seal for values of type sigkey * random. We use the seal directly to model the dedicated commitment function for signing keys and the handle to model the commitments to signing keys; the random part corresponds to the randomness used in the creation of the commitment. The unsealing function openCommitak is not available outside of the API implementation (in fact, the unsealing function is not of kind public, so the type system prevents us from giving the function to the attacker).

1 let (commitak, openCommitak) = mkSeal(sigkey * random) "Signing key Seal";

stripStm : statement → statement and
checkZK : proof → bool:
During the zero-knowledge verification, we will have to check whether the statement stmSkv stored in the zero-
knowledge value zkv of a proof p corresponds to the statement stm stored in p itself. An equality test, however,
does not suffice, since stm might have been modified, for instance, some opening information could have been
removed. The function stripStm enables us to check for the equality of stmSkv and stm without coping with the
opening information. We will see below that the consistency of the opening information and the corresponding
commitments is enforced by the zero-knowledge verification method. The function checkZK takes as argument a
proof and uses stripStm to conduct the check whether the zero-knowledge proof and its statement correspond.

The function stripStm takes as input a value of type statement and returns that value where all the opening
information are removed, i.e., all opening information are set to ()). The implementation is given in Listing 14.

The function checkZK takes as input a value of type proof and returns true if and only if the zero-knowledge
value contained inside the proof matches the statement contained inside the proof.

1 let checkZK (p : proof) =
2  match p with

Listing 14
commitZK : statement * random -> commitment and
openZK : commitment -> statement * random:
The commitZK and the openZK functions are sealing and unsealing functions for values of type statement * random, respectively. They are used to create the zkv value, the first component of values of type proof. The random part corresponds to the randomness used in a zero-knowledge proof. Both functions are only used internally and are not exported.

createZK : statement -> random -> proof * zero-knowledge * statement and
createZK : statement -> random -> proof * zero-knowledge * statement
The API methods give users and the attacker the possibility to create zero-knowledge proofs. For instance, the mkSays API method requires a signing key as input and performs all the necessary cryptographic operations to create a zero-knowledge proof, including the computation of a signature.

While mkSays method conveniently encapsulates the necessary operations, the generation of a zero-knowledge proof in principle only requires the knowledge of the witnesses (hence the name proofs of knowledge). Since only the verification key but not the signing key occurs in the generated zero-knowledge proof, we must provide users and, in particular, the attacker with the means to compute zero-knowledge proofs given only the values occurring inside the proof. In fact, this possibility is a necessary condition to obtain computational soundness (see below). Intuitively, the reason is as follows: Suppose there is an attack against a protocol that exploits that users send out witnesses of a zero-knowledge proof in an attacker-readable form, for instance, in a different part of the protocol. The attacker uses these witnesses to compute zero-knowledge proofs to break some security property. Without the possibility to compute abstract zero-knowledge proofs from the witnesses only, our abstract zero-knowledge model would deem this protocol save although there are attacks. In particular, the security guarantees obtained with our abstract model would not imply the security of a concrete implementation.

The createZK function takes as input a non-conjunctive, non-disjunctive statement stm, a randomness r and creates a zero-knowledge proof p from the opening information contained in stm; the randomness is used to compute the corresponding zero-knowledge value. More precisely, createZK recomputes the commitments in the stm from scratch using the opening information present in the statement to yield stm', it generates the corresponding zkv value by applying the commitZK function to stm' and r, and it finally assembles the proof p by applying the ZK constructor to zkv and stm'. The proof p is returned. The implementation is given in Listing 17.

Although seemingly related with the fakestm function, createZK can selectively fill the cryptographic objects
that exist in a statement (e.g., digital signatures in a says proof) whereas fakestm chooses random elements for these objects since they do not occur in type formula.

Statements such as SSP\(_p\) contain commitments without opening information. We describe the consequences of this imprecision in detail below.

### rerand\(_{stm}\) : \text{statement} → \text{statement} → \text{statement}:

The function rerand\(_{stm}\) takes as input a statement \(stm\) to be rerandomized, a statement \(r\) that determines which commitments in \(stm\) to randomize, and returns the rerandomization of \(stm\). We provide the implementation in

```ocaml
let rerand \((p: \text{proof}) \ (stm: \text{statement}) \ (r: \text{random})\) : \text{proof} =

match \(p\) with
| \(\text{ZK}(zkv, stm_2)\) ⇒
  
  let \((stm_1, r_1)\) = openZK \(zkv\);
  
  let \(stm'_1 = \text{rerand}_{stm} \text{stm}_1 \text{stm}\);
  
  let \(stm'_2 = \text{rerand}_{stm} \text{stm}_2 \text{stm}\);

let \(r' = \text{if } r = () \text{ then } r_1 \text{ else } (r_1, r)\);

let \(zkv' = \text{commitZK} (stm'_1, r')\);

| _ ⇒ \(\text{fail}\langle\text{proof}\rangle\) ()
```

We symbolically implement rerandomization as tuple of the old randomness with the new randomness. In particular, we assume that the provided randomness is freshly chosen (or () to denote the randomness to remain unchanged) and we do not consider arithmetic properties of randomness, i.e., we do not model protocols that rely on the fact that randomness can be cancelled.

### checkEq\(_\alpha\) : \(\alpha \ \text{RevHid} \rightarrow \text{bitstring} \rightarrow (\text{integer} \ * \ \text{bitstring}) \ \text{list ref} \rightarrow \text{bool}\)

While verifying a proof against a formula, we need to enforce the equality of certain values. More precisely, we must ensure that hidden values with the same index (e.g., values Hidden \(x\) and Hidden \(y\) such that \(x = y\)) are equal and that the values in the proof and revealed values in the formula match.

The function checkEq takes as input a value \(v_1\) from a formula, a committed value \(v_2\) from a proof statement (the verification methods have access to the commitment unsealing functions), and a list reference \(\ell\); \(v_1\) can either be of type \(\text{uid}_{\text{pub}}\) (\(\alpha := \text{uid}_{\text{pub}}\)) or of type \(\text{bitstring}\) (\(\alpha := \text{bitstring}\)). checkEq verifies that if \(v_1\) is revealed, then it matches \(v_2\) and that if \(v_1\) is hidden (e.g., \(v_1 = \text{Hidden } x\) then \(v_2\) is equal to all values also hidden with the index (e.g., \(x\)) specified in \(v_1\). The list reference \(\ell\) is used to keep track of the index-value pairs throughout the calling verification function and \(\ell\) is updated in the process accordingly.

```ocaml
let checkEq\(_\alpha\) \(\alpha \ (v_1: \text{RevHid}) \ (v_2: \text{bitstring}) \ (\ell: (\text{integer} \ * \ \text{bitstring}) \ \text{list ref})\) : \text{bool} =

match \(v_1\) with
| \text{Revealed } x ⇒
  
  if \(v_1 = v_2\) then true
  
  else false

| \text{Hidden } x ⇒
  
  if \(\text{List.member\(_{1,2}\)}\langle\text{integer} \ * \ \text{bitstring}\rangle \ x \ (!\ell)\) = true then

  let \( (_, v) = \text{List.get\(_{1,2}\)}\langle\text{integer} \ * \ \text{bitstring}\rangle \ x \ (!\ell)\);

  if \(v = v_2\) then true

  else
```

Belenkiy et al. [56]. It takes as input a proof \(p\), statement \(stm\), and it returns the rerandomization of \(p\). The rerandomization is guided by \(stm\) (see rerand\(_{stm}\)).
If the value \( v_1 \) (from the formula) is revealed (line 8), then it must match the value \( v_2 \) from the statement (line 9). Otherwise, checkEq checks whether the index \( x \) of \( v_1 \) occurs in the list reference \( \ell \) (line 10). If it does, then the value stored for \( x \) must match \( v_2 \) (line 11). If \( x \) does not occur in the list, then the pair \((x, v_2)\) is added to the list stored in the reference \( \ell \) (line 15).

\[
\text{verify}_{\text{stm}} : \text{statement} \rightarrow \text{formula} \rightarrow (\text{integer} \ast \text{bitstring}) \text{ list ref ref} \rightarrow \text{bool}
\]

The \( \text{verify}_{\text{stm}} \) function takes as input a statement \( \text{stm} : \text{statement} \), a formula \( f : \text{formula} \), and a list reference \( \ell : (\text{integer} \ast \text{bitstring}) \text{ list ref} \). The function verifies that \( \text{stm} \) is a statement for formula \( f \). It uses \( \ell \) as input for checkEq to globally keep track of equality of hidden values to guarantee the intended proof properties. It returns true if \( \text{stm} \) is a valid statement for formula \( f \). The implementation is shown in Listing 19.

\[
\text{verify} : \text{proof} \rightarrow \text{formula} \rightarrow \text{bool}
\]

The \( \text{verify} \) API method takes as input a proof \( p : \text{proof} \), a formula \( f : \text{formula} \), and returns true if \( p \) is a proof for \( f \). This method is generally accessible and, unlike the \( \text{verify}_{\text{STM}} \) method, does not depend on a certain logical formula. However, it returns only a Boolean value without a logical refinement.

Internally, \( \text{verify} \) first creates the list reference \( \ell \) (line 1) that will be passed to \( \text{verify}_{\text{stm}} \) and is used to keep track of the equality of hidden values with the same index. Next, we check that the statement contained inside \( p \) is a statement for the zero-knowledge proof (lines 2 to 3); intuitively, this check enforces that the commitments in both parts are equal (see checkEq above). Finally, \( \text{verify}_{\text{stm}} \) verifies that the statement \( \text{stm} \) in \( p \) is a valid zero-knowledge statement for formula \( f \). Since we checked that \( \text{stm} \) and the zero-knowledge proof also match, this method verifies that \( p \) is a valid zero-knowledge proof for formula \( f \).

\[
\text{hide}_{\text{stm}} : \text{statement} \rightarrow \text{formula} \rightarrow \text{statement}
\]

The \( \text{hide}_{\text{stm}} \) function takes as input a statement \( \text{stm} \) and a formula \( f \) and returns a statement where the values specified by \( f \) are hidden. The implementation is given in Listing 20.

\[
\text{combineOr} : \text{proof} \rightarrow \text{formula} \rightarrow \text{random} \rightarrow \text{proof}
\]

The \( \text{combineOr} \) function performs the same operation as the \( \text{mk}_{\lor} \) API method (in fact, the \( \text{mk}_{\lor} \) implementation uses \( \text{combineOr} \)) but allows for a custom randomness used in the creation of the zero-knowledge value \( \text{zkv} \). The \( \text{combineOr} \) function is necessary to achieve computational soundness.

The \( \text{combineOr} \) function takes as input a proof \( p : \text{proof} \), a disjunctive formula \( f : \text{formula} \), and randomness \( r : \text{random} \). It returns a proof for the formula \( f \) that is valid only if \( p \) is a valid proof for either the left or the right branch of the disjunctive formula \( f \).

\[
\text{combineOr} \ (p : \text{proof}) \ (f : \text{formula}) \ (r_{\text{fake}} : \text{random}) : \text{proof} \ast \text{zero-knowledge} \ast \text{statement} =
\]

\[
\text{match } p \text{ with } \text{ZK}(\text{zkv}, \text{stm}) \implies
\]

\[
\text{let } (t, r) = \text{openZK} \text{ zkv};
\]

\[
\text{match } f \text{ with } \text{Or}(f_1, f_2) \implies
\]

\[
\text{if } \text{verify } f_1 \text{ then }
\]

\[
\text{let } \text{stm}_{\text{fake}} = \text{fakestm } f_2;
\]

\[
\text{let } \text{stm}' = \text{Or}_{\scriptscriptstyle \ell}(\text{stm}, \text{stm}_{\text{fake}});
\]

\[
\text{let } \text{zkv}' = \text{commitZK}\((t, \text{stm}_{\text{fake}}), (r, r_{\text{fake}}))\);
Computational soundness results. In concurrent work [35], we are proving the computational soundness of the abstract PKI.
zero-knowledge model, i.e., of the encoding of zero-knowledge proofs, of the API methods, and of the auxiliary helper functions. Intuitively, the computational soundness result connects the abstract zero-knowledge model (i.e., the sealing-based implementation detailed above) with a concrete implementation (e.g., the implementation based on the Groth-Sahai zero-knowledge proof scheme detailed in Section III) and allows us to deduce security guarantees of an actual concrete implementation from the successful type-checking of the API methods.

More precisely, the computational soundness result states that for every attack against a concrete implementation of the API methods, there is an attack against the abstract zero-knowledge model that captures the concrete attack. On the converse, this means that if we can exclude attacks on the abstract model, then we also rule out attacks on a concrete implementation. In other words, the security guarantees obtained by verifying the abstract implementation imply the security of concrete implementations.

**Precision of the abstract zero-knowledge model.** The proposed typed API interface type-checks with the RCF type system, it satisfies the preconditions of the computational soundness results, and it is easy-to-use and is expressive. To keep the API easy-to-use and short, the instantiation of the types proof and statement are slightly imprecise.

It is mandatory that the API types proof and statement are public. Certain kinds of computational proofs, however, require information whose abstract counter part is neither public nor tainted. For instance, the pseudonym proofs computationally require the signing key of type sigkey and the type sigkey is neither public nor tainted. Intuitively, the reason is that a signing key is a tuple. The second component, the verification key is public (see Lemma 10) and the first component, the sealing function, is tainted (using KIND FUN). The kinding rule for tuples requires that both components are either public or tainted. Consequently, type sigkey is neither public nor tainted. The RCF kinding mechanism propagates this property and deems all types that use such non-public, non-tainted types as also non-public and non-tainted. Thus, if the type statement contains type sigkey, then statement would also be neither public nor tainted and, as a result, type proof would not be public and the type system would prevent proofs to be sent over a publicly readable network (i.e., the internet). The computational soundness result, however, requires the abstract model to contain all the values needed for an actual cryptographic computation to be present also in the abstract model.

We address the problem by keeping the commitments to signing keys in the type statement but we do not keep the corresponding opening information. With this relaxation, we still obtain computational soundness for our abstract zero-knowledge model, i.e., we do not prevent attacks in the abstract model that are possible in a concrete implementation. In fact, quite the opposite is the case: an attacker can create zero-knowledge proofs in our abstract model that the attacker could not create computationally. For instance, the implementation of createZK recomputes all commitments in a statement from the given opening information and returns a proof for the resulting value of type statement; the commitment to signing keys is used as given in the original statement. Thus, an attacker can create a proof, for instance, an SSP proof, without supplying the opening information of some commitments. Since an abstract attacker can create zero-knowledge proofs without knowing the opening information to the signing key, we technically lose the abstract of-knowledge property for these values. However, the predicates we transfer do not rely on a principal knowing the witnesses but only on the validity of the proven formula. Despite this imprecision, our model is computationally sound and type-checking a program implies the security of a concrete implementation.

**Implementation of stripStm.**

The stripStm function takes as input a statement stm and returns the statement where all the opening information are replaced by ()

```
stripStm (stm : statement): statement =
1 match stm with
2   | Saysₚ(c₂, (...), cₓ₁, (...), P) ⇒
3       let P' = match P with
4           | P₁(c₁, (...), ..., cₓₙ₁, (...)) ⇒
5                P₁(P₁(c₁, (...), ..., cₓₙ₁, (...)))
6           | P₂(c₁, (...), ..., cₓₙ₂, (...)) ⇒
7                      ...;
8       | Pₘ(c₁, (...), ..., cₓₙₘ, (...)) ⇒
9                       ...
10                  | Pₘ(c₁, (...), ..., cₓₙₘ, (...));
11       Sayₚ(c₂, (()), cₓ₁, (()), P')
```
Implementation of the fakestm function.

The fakestm function takes as input a formula \( f \) and produces a statement for that formula. Intuitively, fakestm pattern-matches the given formula, extracts all the contained information to create commitments at the proper places. All cryptographic values such as digital signatures that occur in the proof but not in the formula are chosen randomly and are inserted into the proof. Consequently, a proof using a statement that was created by fakestm will not verify but they can be used as the “false” branch in a disjunctive proof.

```plaintext
fakestm \( (f : \text{formula}) \) : \text{statement} =
match \ f \ with
| \ And\( (f_1, f_2) \) \Rightarrow
  \ And_p\( (\text{fakestm} f_1, \text{fakestm} f_2) \)
| \ Or\( (f_1, f_2) \) \Rightarrow
  \ Or_p\( (\text{fakestm} f_1, \text{fakestm} f_2) \)
| \ Says\( (vk, P(\text{Revealed} y_1, \ldots , \text{Revealed} y_n)) \) \Rightarrow
  \ let \ sig = \ rand();
  \ let \ r_{sig} = \ rand();
  \ let \ c_{sig} = \ commit(sig, r_{sig});
  \ let \ r_{vk} = \ rand();
  \ let \ c_{vk} = \ commit(vk, r_{vk});
  \ let \ r_1 = \ rand();
  \ let \ c_1 = \ commit(y_1, r_1);
  \;
  \ let \ r_n = \ rand();
  \ let \ c_n = \ commit(y_n, r_n);
  \ let \ \text{Says}_p\( (c_{sig}, (sig, r_{sig}), c_{vk}, (vk, r_{vk}), P(c_1, (y_1, r_1), \ldots , c_n, (y_n, r_n))) \) \Rightarrow
  \ let \ r_{vk} = \ rand();
  \ let \ c_{vk} = \ commit(vk, r_{vk});
  \ let \ r_s = \ rand();
  \ let \ c_s = \ commit(s, r_s);
  \ let \ r_{psd} = \ rand();
  \ let \ c_{psd} = \ commit(psd, r_{psd});
```

Listing 14: The implementation of stripStm.
let \( x = \text{rand}(); \)
let \( r_x = \text{rand}(); \)
let \( c_x = \text{commit}(x, r_x); \)
SSP_p(c_{ok}, (vk, r_{ok}), c_x, (s, r_s), c_{psd}, (psd, r_{psd}), c_x)
| LM(\text{Revealed } x, \text{Revealed } b, \ell) \implies 
let \( r_x = \text{rand}(); \)
let \( c_x = \text{commit}(x, r_x); \)
| LNM(\text{Revealed } x, \ell) \implies 
let \( r_x = \text{rand}(); \)
let \( c_x = \text{commit}(x, r_x); \)
REL_p(c_x, (x, r_x), op, c_y, (y, r_y))
| \text{EscrowInfo}(z, \text{Revealed } x, \text{Revealed } r, \text{Revealed } s, \text{Revealed } idr) \implies 
let \( r_x = \text{rand}(); \)
let \( c_x = \text{commit}(x, r_x); \)
| \text{EscrowInfo}_p(z, c_z, (x, r_x), c_R, (R, r_R), c_s, (s, r_s), c_{iddr}, (iddr, r_{iddr}), c_R)

Listing 15: The implementation of the fakestm function.

Implementation of \text{createZK^e} \text{ and } \text{createZK}.

Intuitively, the zero-knowledge creation functions \text{createZK^e} \text{ and } \text{createZK} take as input a statement \text{stm} together with randomness \( r \) and return a zero-knowledge proof for \text{stm} that uses randomness \( r \). More precisely, \text{createZK^e} takes as input a non-conjunctive and non-disjunctive statement, \text{createZK} takes as input any statement. For implementational convenience, the functions do not return a zero-knowledge proof but a triple that consists of a zero-knowledge proof for the input statement and the zero-knowledge value as well as the statement contained within that proof.

Internally, the \text{createZK^e} function recomputes all commitments using the opening information contained in the input statement; the commitments for values without opening information (e.g., signing keys) are taken as is. The \text{createZK} function relies on the \text{createZK^e} function for all non-conjunctive and non-disjunctive statements. The cases for the other two kinds of statements recursively call the \text{createZK} function, as expected.

\text{createZK^e} \text{ (stm : statement) (r : random): proof * zero-knowledge * statement =}
mismatch \text{stm} with
| \text{Says}_p(_, (z, r_z), _, (sig, r_{sig}), P) \implies 
let \( c_z = \text{commit}(z, r_z); \)
let \( c_{sig} = \text{commit}(sig, r_{sig}); \)
let \( P' = \text{match } P \text{ with} \)
| \text{P'}(_, (arg_1, r_{arg_1}), \ldots, (_, (arg_n, r_{arg_n})) \implies 
let \( c_{arg_1} = \text{commit}(arg_1, r_{arg_1}); \)
let \( c_{\text{arg}_{n_1}} = \text{commit}(\text{arg}_{n_1}, \text{r}_{\text{arg}_{n_1}}); \)

\[ P^1_m(\text{arg}_{n_1}, (\text{arg}_{1}, \text{r}_{\text{arg}_{1}}), \ldots, \text{arg}_{n_1}, (\text{arg}_{n_1}, \text{r}_{\text{arg}_{n_1}})) \]

\[ \implies \]

\[ P^m_m(\text{arg}_{1}, \ldots, \text{arg}_{n_1}, r_{\text{arg}_{n_1}}) \]

\[ \implies \]

\[ P^m_m(\text{arg}_{1}, \ldots, \text{arg}_{n_1}, r_{\text{arg}_{n_1}}); \]

let \( \text{stm}' = \text{Says}_p(c_z, (z, r_z), c_{\text{sig}}, (\text{sig}, r_{\text{sig}}), P^p); \)

let \( \text{zkv} = \text{commitZK}(\text{stm}', r); \)

let \( p = \text{ZK}(\text{zkv}, \text{stm}'); \)

\( (p, \text{zkv}, \text{stm}') \)

\[ \implies \]

\[ \text{SSP}_p(z, r_z), (s, r_s), (\text{psd}, r_{\text{psd}}), c_z \]

\[ \implies \]

\[ \text{LM}_p(x, r_x), (h, r_h), \ell \]

\[ \implies \]

\[ \text{LNM}_p(x, r_x), \ell \]

\[ \implies \]

\[ \text{REL}_p(x, r_x), (y, r_y) \]

\[ \implies \]

\[ \text{EscrowInfo}_p(z, x, r_x), (R, r_R), (s, r_s), (\text{idr}, r_{\text{idr}}), c_r \]

\[ \implies \]

\[ \_ \implies \]

\[ \text{fail}(\text{proof} \ast \text{zero-knowledge} \ast \text{statement}) \]

---

**Listing 16** The implementation of \( \text{createZK}^\ast \).
let \((p_1, zk'_1, stm'_1) = createZK \; stm_1 \; r_1;\)
let \((p_2, zk'_2, stm'_2) = createZK \; stm_2 \; r_2;\)
let \(stm' = Or_p(stm'_1, stm'_2);\)
lет \(zk = commitZK(stm', r);\)
lет \(p = ZK(zk, stm');\)

\((p, zk, stm')\)

| _ \(\implies\)
| fail(proof * zero-knowledge * statement) ()

Listing 17: The implementation of createZK.

**Implementation of the rerand_{stm} function.**

The rerand_{stm} function takes as input a statement \(stm\) to be rerandomized and a statement \(g\) that guides this process, i.e., \(g\) determines which values in \(stm\) are to be rerandomized. More precisely, if a randomness in \(g\) is different from \(()\), then the corresponding value in \(stm\) is rerandomized with freshly-chosen randomness; commitments for values that do not contain opening information (e.g., signing keys) are always rerandomized.

Notice that using the Groth-Sahai implementation, it is possible to rerandomize values and to choose the randomness that will be added to the randomness contained in the corresponding commitment \([56]\). Since a symbolic model of such algebraic structures in inherently unsound \([80]\), we do not consider this selective re-randomization.

rerand_{stm} \((stm : \text{statement}) \rightarrow (g : \text{statement}) : \text{statement} = \)

match \(stm\) with
| \(\text{Says}_p(c_z, (z, r_z), c_{sig}, (sig, r_{sig}), P) \implies\)
| \(:\)
| \(\text{SSP}_p(c_z, (z, r_z), c_s, (s, r_s), c_{psd}, (psd, r_{psd}), c_x) \implies\)
match \(g\) with \(\text{SSP}_p(\_\_, (r'_{z}), \_\_, (r'_{s}), \_\_, (r'_{psd}), \_\_) \implies\)
let \(r'_x = \text{rand} ();\)
let \((z_ sol, r_{sol}) = \text{openCommit} c_z;\)
let \((s_ sol, r_{sol}) = \text{openCommit} c_s;\)
let \((psd, r_{psd}) = \text{openCommit} c_{psd};\)
let \((x, r_x) = \text{openCommit} c_x;\)
let \((c_{sn}, s_n, r_{sn}) =\)
if \(r'_z = ()\)
\((c_z, z, r_z);\)
else
if \(r_z = ()\)
\((\text{commit} (z_ sol, (r_{sol}, r'_z)), ()),())\)
else
\((\text{commit} (z_ sol, (r_{sol}, r'_z)), z, (r_z, r'_z));\)
let \((c_{sn}, s_n, r_{sn}) =\)
if \(r'_s = ()\)
\((c_s, s, r_s);\)
else
if \(r_s = ()\)
\((\text{commit} (s_ sol, (r_{sol}, r'_s)), ()),())\)
else
\((\text{commit} (s_ sol, (r_{sol}, r'_s)), s, (r_s, r'_s));\)
let \((c_{psd}, psd_ n, r_{psd}) =\)
if \(r'_{psd} = ()\)
\((c_{psd}, psd, r_{psd});\)
else
if \( r_{psd} = () \) then

\[
\text{commit} \ (psd_{o}, (r_{psd_{o}}, r'_{psd}))), (, ())
\]

else

\[
\text{commit} \ (psd_{o}, (r_{psd_{o}}, r'_{psd})), psd_{i}, (r_{psd}, r'_{psd}))\]

let \( c_{zn} = \text{commit} \ (x, (r_{x}, r'_{x})) \);

\[
\text{SSP}_{p}(c_{zn}, (c_{zn}, r_{zn}), c_{sn}, (s_{n}, r_{sn}), c_{psd_{n}}, (psd_{n}, r_{psd_{n}}), c_{zn})
\]

| _ | \( \Rightarrow \)
fail(proof) ()

| LMP_{p}(c_{x}, (x, r_{x}), c_{y}, (b, r_{b}), \ell) | \( \Rightarrow \)

| LNM_{p}(c_{x}, (x, r_{x}), \ell) | \( \Rightarrow \)

| REL_{p}(c_{x}, (x, r_{x}), \ell, c_{y}, (b, r_{b}), \ell) | \( \Rightarrow \)

| EscrowInfo_{p}(c_{x}, (x, r_{x}), c_{R}, (R, r_{R}), c_{s}, (s_{r}, c_{idr}, (idr, r_{idr}), c_{r}) | \( \Rightarrow \)

| And_{p}(stm_{1}, stm_{2}) \( \Rightarrow \)

match \( g \) with \( \text{And}_{p}(g_{1}, g_{2}) \) \( \Rightarrow \)

And_{p}(rerand_{stm} \ stm_{1} \ g_{1}, \rerand_{stm} \ stm_{2} \ g_{2})

| _ | \( \Rightarrow \)
fail(statement) ()

| Or_{p}(stm_{1}, stm_{2}) \( \Rightarrow \)

match \( g \) with \( \text{Or}_{p}(g_{1}, g_{2}) \) \( \Rightarrow \)

Or_{p}(rerand_{stm} \ stm_{1} \ g_{1}, \rerand_{stm} \ stm_{2} \ g_{2})

| _ | \( \Rightarrow \)
fail(statement) ()

| _ | \( \Rightarrow \)
fail(statement) ()

The rerand function is a large case split. It uses the rerandomization guide \( g : \text{formula} \) to determine whether to rerandomize a specific commitment or not. More precisely, if the randomness part of the opening information for a certain part in \( g \) is different from () (line 27), then we choose new randomness and apply it to the respective commitment (line 31 and 33). If the original opening information was removed (line 30), then the new proof also does not contain the opening information (line 31), otherwise the original opening information with the modified randomness is propagated into the new proof (line 33).

Here, we modify the randomness using the tuple operation. In a cryptographic implementation, this will be implemented using a mathematical operation.

**Implementation of the \( \text{verify}_{stm} \) function.**

The \( \text{verify}_{stm} \) function takes as input a statement \( stm \), a formula \( f \) and an \((\text{integer} * \text{bitstring})\) list ref \( \ell \) and verifies that \( stm \) is a valid statement for formula \( f \).

Internally, \( \text{verify}_{stm} \) matches \( p \) and in each case checks the validity of the case against formula \( f \). The list reference \( \ell \) is used to enforce that hidden values with the same index (via the Hidden constructor in type \( \alpha \ RevHid \)) are indeed equal.

\[
\text{verify}_{stm} \ (p: \text{statement}) \ (f: \text{formula}) \ (\ell: (\text{integer} * \text{bitstring}) \ \text{list ref}) : \text{bool} =
\]

1 match \( p \) with
2 | And_{p}(p_{1}, p_{2}) \( \Rightarrow \)
3 | match \( f \) with
4 | | And(f_{1}, f_{2}) \( \Rightarrow \)
5 | | if \( \text{verify}_{stm} \ p_{1} \ f_{1} \) = true then
6 | | \text{verify}_{stm} \ p_{2} \ f_{2}
7 | else
8 | false
9 | _ | \( \Rightarrow \) false
\( \text{Or}(p_1, p_2) \implies \)
match \( f \) with
\( \text{Or}(f_1, f_2) \implies \)
if \( \text{verify}_{stn} p_1 f_1 = \text{true} \) then
true
else
\( \text{verify}_{stn} p_2 f_2 \)
\( \_ \implies \text{false} \)
\( \text{REL}_{p}(c_x, f_1, \_), \text{op}, (c_y, f_2) \implies \)
match \( f \) with
\( \text{REL}(x', \text{op}', y') \implies \)
let \( (x, r_x) = \text{openCommit}(c_x) \);
let \( (y, r_y) = \text{openCommit}(c_y) \);
let \( b_x = \text{checkEq} x' x \ell \);
let \( b_y = \text{checkEq} y' y \ell \);
if \( b_x = \text{true} \) then
if \( b_y = \text{true} \) then
if \( \text{op} = \text{op}' \) then
let \( \text{op}'' = \text{getOperation} \text{op} \);
\( \text{op}'' x y \)
else
false
else
false
else
false
\( \_ \implies \text{false} \)
\( \text{Says}_{p}(c_{sig}, c_z, (c_1, \ldots, (c_n, \_))) \implies \)
match \( f \) with
\( \text{Says}(z', P(y_1', \ldots, y_n')) \implies \)
let \( (\text{sig}, r_{\text{sig}}) = \text{openCommit}(c_{\text{sig}}) \);
let \( (z, r_z) = \text{openCommit}(c_z) \);
let \( (y_1, r_1) = \text{openCommit}(c_1) \);
\vdots
let \( (y_n, r_n) = \text{openCommit}(c_n) \);
let \( (z'', q) = z_{\text{sig}} \);
let \( b_z = \text{checkEq} z' z \ell \);
let \( b_1 = \text{checkEq} y_1' y_1 \ell \);
\vdots
let \( b_n = \text{checkEq} y_n' y_n \ell \);
if \( z = z'' \) then
if \( q = P(y_1, \ldots, y_n) \) then
if \( b_z = \text{true} \) then
if \( b_1 = \text{true} \) then
\( \ldots \)
if \( b_{n-1} = \text{true} \) then
\( b_n \)
else
false
\( \ldots \)
else
false
else false
else false
else false
| _ ⇒ false

| EscrowInfo_p(z, (c_x, _), (c_R, _), (c_s, _), (c_idr, _), c_r) ⇒
match f with
| EscrowInfo(z', x', r', s', idr') ⇒
  let (s, r_s) = openCommit(c_s);
  let (idr, r_idr) = openCommit(c_idr);
  let (x, r_x) = openCommit(c_x);
  let (r, r_r) = openCommit(c_r);
  let (R, r_R) = openCommit(c_R);
  let R'' = computeR x r;
  let idr'' = computeIDR z x r R service;
  let b_x = checkEq x' x ℓ;
  let b_r = checkEq r' r ℓ;
  let b_s = checkEq s' s ℓ;
  let b_idr = checkEq idr' idr ℓ;
  if R = R'' then
    if idr = idr'' then
      if z = z' then
        if b_x = true then
          if b_r = true then
            if b_s = true then
              b_idr
            else
              false
        else
          false
      else
        false
    else
      false
  else
    false

| SSP_p((c_y, _), (c_s, _), (c_psd, _), c_x) ⇒
match f with
| SSP(y', s', psd') ⇒
  let (y, r_y) = openCommit(c_y);
  let (s, r_s) = openCommit(c_s);
  let (psd, r_psd) = openCommit(c_psd);
  let (x, r_x) = openCommit(sk(c_x));
  let (_, w) = x;
  let psd'' = computePsd x s;
  let b_y = checkEq y' y ℓ;
  let b_s = checkEq s' s ℓ;
  let b_psd = checkEq psd' psd ℓ;
  if b_y = true then
    if b_s = true then
      if psd'' = psd then
        if w = y then
          true
        else
          false
      else
        false
    else
      false
  else
    false
else
false

else
false

| _  ⇒ false

| LM_p((c_x,_,),(c_b,_,),ℓ) ⇒
match f with
| LM(x',b',ℓ') ⇒
let (x, r_x) = openCommit(c_x);
let (b, r_b) = openCommit(c_b);
let b_x = checkEq x' x ℓ;
let b_b = checkEq b' b ℓ;
let b_ℓ = List.member(2,2)(pseudo * bitstring) x b ℓ;
if b_x = true then
  if b_b = true then
    if ℓ = ℓ' then
      b_ℓ
    else
      false
  else
    false
else
false

else
false

| _  ⇒ false

| LNM_p((c_x,_,),ℓ) ⇒
match f with
| LNM(x',ℓ') ⇒
let (x, r_x) = openCommit(c_x);
let b_ℓ = List.member(2,2)(pseudo * bitstring) x ℓ;
if b_ℓ = true then
  false
else
  if ℓ = ℓ' then
    checkEq x x'
  else
    false
| _  ⇒ false

| _  ⇒ false

Listing 19: The implementation of the verify_stm function.

Implementation of hide_stm.
The hide_stm function takes as input a statement stm and a formula f, and it returns the statement that is derived from stm by hiding all values that are hidden in f.

Internally, hide_stm matches stm and f and if a value in f is hidden (via the Hidden constructor of the α RevHid type), then the corresponding opening information are replaced by (), otherwise the opening information are not modified.
match \( f \) with \( SSP(z', s', psd') \) \( \implies \)
  let \( (s'', r''_{psd}) = \text{match } s' \text{ with} \)
  | Revealed _ \( \implies \)
  (s, r_s) \( \implies \)
  ((),());

let \( (z'', r''_{z}) = \text{match } z' \text{ with} \)
  | Revealed _ \( \implies \)
  (z, r_z) \( \implies \)
  ((),());

let \( (psd''_p, r''_{psd}) = \text{match } psd' \text{ with} \)
  | Revealed _ \( \implies \)
  (psd, r_{psd}) \( \implies \)
  ((),());

\( SSP_p(c_z, (z'', r''_{z}), c_s, (s'', r''_{s}), c_{psd}, (psd'', r''_{psd}), c_x) \) \( \implies \)

\( \text{fail(proof)} () \)

\( \text{LM}_p(c_x, (x, r_x), c_b, (b, r_b), \ell) \implies \)

\( \text{LNM}_p(c_x, (x, r_x), \ell) \implies \)

\( \text{REL}_p(c_x, (x, r_x), op, c_y, (y, r_y) \implies \)

\( \text{EscrowInfo}_p(z, c_z, (x, r_x), c_R, (R, r_R), c_s, (s, r_s), c_{idr}, (idr, r_{idr}), c_r) \implies \)

\( \text{And}_p(stm_1, stm_2) \implies \)
match \( f \) with \( \text{And}(f_1, f_2) \implies \)
\( \text{And}_p(hide_{stm} \, stm_1 \, f_1, hide_{stm} \, stm_2 \, f_2) \)
  | _ \( \implies \)
  \( \text{fail(statement)} () \)

\( \text{Or}_p(stm_1, stm_2) \implies \)
match \( f \) with \( \text{Or}(f_1, f_2) \implies \)
\( \text{Or}_p(hide_{stm} \, stm_1 \, f_1, hide_{stm} \, stm_2 \, f_2) \)
  | _ \( \implies \)
  \( \text{fail(statement)} () \)
  | _ \( \implies \)
  \( \text{fail(statement)} () \)

Main API methods. The rest of the section is organized in two parts.
- first, we implement the main API methods except for the verification method. The implementation is often surprisingly short because the creation functions only perform checks regarding the well-formedness of the inputs;
- secondly, we develop important concepts for the implementation of the verify method and finally implement the verify method. From a type-checking and from a logical point of view, the verification method is the most important and most complex function because it allows principals to deduce logical formulas from data received from an untrusted public network.

We start with the mkId API method.

\( \text{mkId} : (x : \text{string}) \rightarrow (uid * uid_{pub}) \):

\( \text{mkId} \) takes a textual description \( x : \text{string} \) as input and returns a freshly generated signing key handle/public key
pair.

mkId (x : string) : uid * verkey = 
1 let sk = mkSeal⟨T_{sk}{verkey/α}⟩ x;
2 let (_, vk') = sk;
3 let hdl = storeSK sk;
4 (hdl, fold⟨T_{sk}{verkey/α}, verkey⟩ vk')

The signing keys are the pairs of sealing and unsealing functions, and verification keys are the unsealing function. We apply the fold constructor to obtain the desired recursive type verkey for the verification key.

\[
\text{mkSays} : \text{uid} \rightarrow \text{predicate}^F \rightarrow \text{proof}:
\]

\text{mkSays} takes as input a signing key \( x \) of the principal \( A \) executing the API method and a predicate \( y \). For the sake of readability, we use a predicate rather than a formula, since the core insight lies in the transport of the logical formula and not the complexity thereof. \text{mkSays} outputs a proof that, if verified by a principal \( B \), will allow \( B \) to logically entail \( A \) says \( F \) where \( y \) is the RCF encoding of predicate \( F \). Since calling the \text{mkSays} method expresses the intention of the executing principal to state the provided formula, we internalized the necessary assumption into the code of the API method.

\[
\text{mkSays} (x' : \text{uid}) (y : \text{predicate}^F) : \text{proof}
\]

1 let \( x = \text{restoreSK} x' \);
2 let \((w, z) = x\);
3 match \( y \) with
4 | \( P_1^F(\text{Revealed } y_1, \ldots, \text{Revealed } y_{n_1}) \implies \)
5 let \( y' = P_1^S(y_1, \ldots, y_{n_1}) \);
6 let \( t = \text{assume } z \text{ says } P_1(y_1, \ldots, y_{n_1}) \);
7 let \( \text{sig} = \text{sign } x (z, y') \);
8 let \( \text{r}_{\text{sig}} = \text{rand}() \);
9 let \( \text{c}_{\text{sig}} = \text{commit}(\text{sig}, \text{r}_{\text{sig}}) \);
10 let \( \text{r}_z = \text{rand}() \);
11 let \( \text{c}_z = \text{commit}(z, \text{r}_z) \);
12 let \( \text{r}_1 = \text{rand}() \);
13 let \( \text{c}_1 = \text{commit}(y_1, \text{r}_1) \);
14 :
15 let \( \text{r}_{n_1} = \text{rand}() \);
16 let \( \text{c}_{n_1} = \text{commit}(y_{n_1}, \text{r}_{n_1}) \);
17 let \( \text{stm} = \text{Says}_y((\text{c}_{\text{sig}}, (\text{sig}, \text{r}_{\text{sig}})), (\text{c}_z, (z, \text{r}_z)), P_1^P((\text{c}_1, (y_1, \text{r}_1)), \ldots, (\text{c}_{n_1}, (y_{n_1}, \text{r}_{n_1})))) \);
18 let \( \text{r}_{\text{zkv}} = \text{rand}() \);
19 let \( \text{zkv} = \text{commitZK} (\text{stm}, \text{r}_{\text{zkv}}) \);
20 ZK (\text{zkv}, \text{stm})
21 | \( P_2^F(\text{Revealed } y_1, \ldots, \text{Revealed } y_{n_2}) \implies \)
22 :
23 | \( P_m^F(\text{Revealed } y_1, \ldots, \text{Revealed } y_{n_m}) \implies \)
24 let \( y' = P_m^S(y_1, \ldots, y_{n_m}) \);
25 let \( t = \text{assume } z \text{ says } P_m(y_1, \ldots, y_{n_m}) \);
26 let \( \text{sig} = w (z, y') \);
27 let \( \text{r}_{\text{sig}} = \text{rand}() \);
28 let \( \text{c}_{\text{sig}} = \text{commit}(\text{sig}, \text{r}_{\text{sig}}) \);
29 let \( \text{r}_z = \text{rand}() \);
30 let \( \text{c}_z = \text{commit}(z, \text{r}_z) \);
31 let \( \text{r}_1 = \text{rand}() \);
32 let \( \text{c}_1 = \text{commit}(y_1, \text{r}_1) \);
33 :
34 let \( \text{r}_{n_m} = \text{rand}() \);
We implement `mkSays` as a match statement over all possible predicates in the system. As a result, `mkSays` is general and does not require a formula annotation as is needed for the verification function (see below). `mkSays` first splits the signing key (line 3) to obtain the signing component used in creating the signature (line 4), matches predicate \( y \) with the type `predicate` to access the arguments for the \( P^F \) (lines 4, 19, \ldots, 20) respectively, and creates the signature. The matching of the supplied formula without any hidden components enforces that all values in the formula are revealed. This corresponds to the of-knowledge property of the zero-knowledge proofs.

The rest of the proof draws the randomness, computes the commitments, and builds the proof. The code for the different cases only differs in the matched pattern and on the number of calls to `rand` and `commit` (they depend on the arity of the matched pattern).

```
31 let \( c_{n,m} = \text{commit}(y_{n,m}, r_{n,m}); \)
32 let \( \text{stm} = \text{Says}_p((c_{n,m}, (s, r_{n,m})), (c_{n,m}, (y_{n,m}, r_{n,m}))); \)
33 let \( \text{r}_{zkw} = \text{rand}(); \)
34 let \( \text{zkv} = \text{commitZK} (\text{stm}, \text{r}_{zkw}); \)
35 \( \text{ZK} (\text{zkv}, \text{stm}) \)  
\( \text{fail}\langle \text{proof}\rangle () \)
```

```
mkSays : \( \text{uid} \to \text{bitstring} \to \text{proof} \)
mkSays takes as input the signing key \( x \) of the principal running the code and the service description \( s \). It extracts the verification key \( y \) from \( x \) and uses the \( \text{computePsd} \) function to compute the pseudonym \( psd \).
The remainder of the function draws the randomness for the commitments, computes the commitments, and builds the proof. In accordance with the API description, the opening information for commitment \( c_x \) on the signing key are not included in the proof. This prevents accidental or intentional leaking of secret key material.
```

```
\( \text{mkSays} (x' : \text{uid}) (s : \text{bitstring}) : \text{proof} = \)
\( \text{let } x = \text{restoreSK } x'; \)
\( \text{let } (\_, y) = x; \)
\( \text{let } \text{psd} = \text{computePsd} x \ s; \)
\( \text{let } \text{r}_y = \text{rand}(); \)
\( \text{let } \text{c}_y = \text{commit}(y, \text{r}_y); \)
\( \text{let } \text{r}_s = \text{rand}(); \)
\( \text{let } \text{c}_s = \text{commit}(s, \text{r}_s); \)
\( \text{let } \text{r}_{psd} = \text{rand}(); \)
\( \text{let } \text{c}_{psd} = \text{commit}(\text{psd}, \text{r}_{psd}); \)
\( \text{let } \text{r}_x = \text{rand}(); \)
\( \text{let } \text{c}_x = \text{commit}_{\text{ak}} (x, \text{r}_x); \)
\( \text{let } \text{stm} = \text{SSP}_p((\text{c}_y, (y, \text{r}_y)), (\text{c}_s, (s, \text{r}_s)), (\text{c}_{psd}, (\text{psd}, \text{r}_{psd})), \text{c}_x) \)
\( \text{let } \text{r}_{zkw} = \text{rand}(); \)
\( \text{let } \text{zkv} = \text{commitZK} (\text{stm}, \text{r}_{zkw}); \)
\( \text{ZK} (\text{zkv}, \text{stm}) \)  
\( \text{fail}\langle \text{proof}\rangle () \)
```

```
mkREL : \( \text{formula} \to \text{proof} \)
mkREL takes as input the formula \( y \) that describes an (in)equality relation. The function draws randomness, builds the corresponding commitments, and creates the zero-knowledge proof. Since the operation can be deduced from the performed cryptographic operation, the proven operation occurs inside the abstract proof in plain.
```

```
mkREL : \( \text{formula} \to \text{proof} \)
mkREL takes as input the formula \( y \) that describes an (in)equality relation. The function draws randomness, builds the corresponding commitments, and creates the zero-knowledge proof. Since the operation can be deduced from the performed cryptographic operation, the proven operation occurs inside the abstract proof in plain.
```
```
\( \text{mkREL} (y : \text{formula}) : \text{proof} = \)
\( \text{match } y \text{ with } \text{REL}(\text{Revealed } x, op, \text{Revealed } y) \)
\( \text{let } \text{r}_x = \text{rand}(); \)
\( \text{let } \text{c}_x = \text{commit}(x, \text{r}_x); \)
\( \text{let } \text{r}_y = \text{rand}(); \)
\( \text{let } \text{c}_y = \text{commit}(y, \text{r}_y); \)
\( \text{let } \text{stm} = \text{REL}_p((\text{c}_x, (x, \text{r}_x)), op, (\text{c}_y, (y, \text{r}_y))) \)
\( \text{let } \text{r}_{zkw} = \text{rand}(); \)
\( \text{let } \text{zkv} = \text{commitZK} (\text{stm}, \text{r}_{zkw}); \)
```

```
```
mkIDRev takes as input a special proof \( p \) constructed by a trusted third party and a service description \( s \). The TTP can, in case of a dispute or another cogent reason reveal the identity of the user. The proof \( p \) shows that the principal \( A \) running the code is registered with the trusted third party and contains the values used by the TTP to identify \( A \) (see \textsection \textbf{III}). Since mkIDRev creates a proof from scratch and all values in \( p \) are used, \( p \) must contain all the opening information.

The escrow value \( R \) is computed using the computeR function, the escrow identifier \( idr \) is computed by applying the function computeIDR to \( r, x, R, s \), and the service description \( s \).

After the computation, the method draws the randomness to compute the commitments and creates the proof. As the value \( r \) is considered secret, we only include the commitment \( c_r \) on \( r \) in the proof.
let zkv = commitZK(stm, rzkv);
ZK(zkv, stm)
\[ \_ \Rightarrow \text{fail} \langle \text{proof} \rangle () \]

\[ \text{mk}_\land : (\text{proof} * \text{proof}) \rightarrow \text{proof} : \]
\[ \text{mk}_\land \text{ takes as input two proofs } p_1 \text{ and } p_2, \text{ and returns the proof for the logical conjunction } p_1 \land p_2. \]

\[ \text{mk}_\land (p : \text{proof} * \text{proof}) : \text{proof} = \]
1 let (p1, p2) = p;
2 match p1 with ZK(zkv1, stm1) \[ \Rightarrow \]
3 match p2 with ZK(zkv2, stm2) \[ \Rightarrow \]
4 let (t1, r1) = openZK zkv1;
5 let (t2, r2) = openZK zkv2;
6 let zkv = commitZK(And_p(t1, t2), (r1, r2));
7 ZK(zkv, And_p(stm1, stm2))
\[ \_ \Rightarrow \text{fail} \langle \text{proof} \rangle () \]

The implementation performs the following steps: it matches the given proof \( p \) against a conjunction proof (line 1), splits the statement to retrieve the two sub-statements of the individual sub-proofs (line 2), opens the zkv value (line 3), splits the randomness (line 4) and reassembles the zero-knowledge proofs for the two sub-proofs, accordingly (lines 5 through 8).

\[ \text{split}_\land : \text{proof} \rightarrow (\text{proof} * \text{proof}) : \]
\[ \text{split}_\land \text{ takes as input a proof. If this proof is a conjunction, it returns the two conjuncted proofs by reversing the operations conducted by } \text{mk}_\land. \]

\[ \text{split}_\land (p : \text{proof}) : \text{proof} * \text{proof} = \]
1 match p with ZK(zkv, And_p(stm)) \[ \Rightarrow \]
2 let (stm1, stm2) = stm;
3 let (t, r) = openZK zkv;
4 let (r1, r2) = r;
5 match t with And_p(t1, t2) \[ \Rightarrow \]
6 let zkvl = commitZK(t1, r1);
7 let zkvr = commitZK(t2, r2);
8 (ZK(zkvl1, stm1), ZK(zkvr2, stm2))
\[ \_ \Rightarrow \text{fail} \langle \text{proof} * \text{proof} \rangle () \]

The construction of this proof takes into account that the zero-knowledge value zkv is set-up correctly and that the cryptographic implementation of a conjunction is a concatenation of two proofs. More precisely, the \( \text{mk}_\land \) function matches the two proofs \( p_1 \) and \( p_2 \) to obtain the two corresponding zero-knowledge values zkv1 and zkv2 as well as the two corresponding statements \( \text{stm}_1 \) and \( \text{stm}_2 \). For the unsealed contents \( (t_1, r_1) \) and \( (t_2, r_2) \) of zkv1 and zkv2, respectively, it holds that \( t_1 \) equals \( \text{stm}_1 \) and \( t_2 \) equals \( \text{stm}_2 \) up to opening information that has been removed via the hide API method from \( \text{stm}_1 \) and \( \text{stm}_2 \). Consequently, the correct content of the zkv value consists of the constructed value And_p(t1, t2).

The randomness for the zero-knowledge proof is a subtle matter. Since the cryptographic realization does not require any special randomness (the two proofs are concatenated), we reflect this implementation by using the paired randomness of the two sub-proofs as randomness for the conjunction. Thus, exactly as in the cryptographic implementation, the proof for a logical conjunction only depends on its two sub-proofs. The treatment of the randomness of the logical conjunction is in accordance with the computational soundness result.

\[ \text{split}_\land (p : \text{proof}) : \text{proof} * \text{proof} = \]
1 match p with ZK(zkv, And_p(stm)) \[ \Rightarrow \]
2 let (stm1, stm2) = stm;
3 let (t, r) = openZK zkv;
4 let (r1, r2) = r;
5 match t with And_p(t1, t2) \[ \Rightarrow \]
6 let zkvl = commitZK(t1, r1);
7 let zkvr = commitZK(t2, r2);
8 (ZK(zkvl1, stm1), ZK(zkvr2, stm2))
\[ \_ \Rightarrow \text{fail} \langle \text{proof} * \text{proof} \rangle () \]
hide : proof \rightarrow formula \rightarrow proof:

The hide method takes as input a proof \( p \) and formula \( f \) and outputs the proof where all the values specified by \( f \) are hidden. hide takes as input a proof \( p \) and a formula \( f \), and it returns the proof obtained by hiding all variables specified by \( f \). Internally, hide relies on the \( hide_{stm} \) function.

\[
\text{hide} \quad (p : \text{proof}) \rightarrow (f : \text{formula}) \rightarrow \text{proof}
\]

1. \text{match } p \text{ with } \text{ZK}(zkv, stm) \Rightarrow
2. \quad \text{let } stm' = \text{hide}_{stm} stm f;
3. \quad \text{ZK}(zkv, stm')
4. \quad \_ \Rightarrow
5. \quad \text{fail}(\text{proof}) ()

The verification function \( \forall \text{verify} \). We proceed to detailing the code for the proof verification function. Before we start giving the code, we define all the necessary ingredients of the verification function.

For technical type-checking reasons, we cannot provide an implementation. Intuitively, one implementation would need to call itself recursively in case of a conjunction proof and the intermediate representation for names is lost in the recursive call. For instance, consider the following formula from Section II:

\[
\exists x. \quad x_{Prof}\ says\ Reg(x, x_{course}) \\
\land x\ says\ Eval(x_{ev}, x_{course}) \\
\land SSP(x, x_{course}, x_{psd})
\] (3)

If we verified this formula recursively, we would logically obtain the following formula:

\[
\exists x_1. \quad x_{Prof}\ says\ Reg(x_1, x_{course}) \\
\land \exists x_2. \quad x_{Prof}\ says\ Reg(x_2, x_{course}) \\
\land \exists x_3. \quad SSP(x_3, x_{course}, x_{psd})
\] (4)

At this point, however, all the intermediate representations of the hidden variables for the \( x_i \) are lost inside the function calls and we cannot retrieve the equality between the \( x_i \) anymore. Thus, instead of giving one implementation for the verification function, we provide code macros that are assembled into the final implementation.

Technically, the code macros are contexts. Intuitively, a context is an expression with a unique hole and into this hole, another context or another expression can be inserted. We write \( C[\_] \) to denote the context \( C \) and the hole is denoted by \( \_ \). Let \( C' \) be another context or an expression. \( C[ C' ] \) is the result of replacing the unique \( \_ \) with \( C' \). If \( C' \) is a context, the result is a context again, and if the \( C' \) is an expression, the result will be an expression.

Clearly, the code macros depend on the proven formula, e.g., the verification code for a list membership proof differs from a identity relevation proof (as indicated by the subscript \( F \) of the verification function). The dependence of the code macros on the previously assembled code, however, is more subtle and crucial to unify existentially quantified variables as outlined in equations (3) and (4). We use logical maps to keep record of the respective equalities within the code macros. The maps rely on the following definition that identifies variables with their canonical positional index.

\textbf{Definition 10} (Formula offset, verification key indices, and variables of conjunctive formulas). Let \( el(f : \text{formula}) \) be the
function that extracts information from a \( f \) as follows:

\[
\begin{align*}
el \left(\text{Says}(x_0 : \text{uidpub} \text{ RevHid}, P^v_k(x_1 : T^1_k, \ldots, x_n : T^n_k | \mathcal{F}))\right) &:= (n + 1, \{0\} \cup \{i \mid T^i_k = \text{uidpub} \text{ RevHid}\}, (x_0, \ldots, x_n)) \\
el(\text{SSP}(v_k : \text{uidpub} \text{ RevHid}, s : \text{bitstring} \text{ RevHid}, psd : \text{bitstring} \text{ RevHid})) &:= (3, 0, (v_k, s, psd)) \\
el(\text{REL}(x : \text{bitstring} \text{ RevHid}, o : \text{string}, y : \text{bitstring} \text{ RevHid})) &:= (3, 0, (x, o, y)) \\
el(\text{LM}(x : \text{bitstring} \text{ RevHid}, b : \text{bitstring} \text{ RevHid}, \ell : \text{list})) &:= (3, 0, (b, \ell)) \\
el(\text{LM}(x : \text{bitstring} \text{ RevHid}, \ell : \text{list})) &:= (2, 0, (x, \ell)) \\
el(\text{EscrowInfo}(v_{k,EA} : \text{uidpub}, v_k : \text{uidpub} \text{ RevHid}, \\ R : \text{bitstring} \text{ RevHid}, s : \text{bitstring} \text{ RevHid}, idr : \text{bitstring} \text{ RevHid})) &:= (5, 0, (v_{k,EA}, v_k, R, s, idr)) \\
el(\text{And}(f_1, f_2)) &:= (n + n', S \cup (n + S'), V \oplus V')
\end{align*}
\]

where \((n, S, V) = el(f_1), \quad (n', S', V') = el(f_2), \quad i + M := \{i + m \mid m \in M\}, \quad \text{and} \quad (x_0, \ldots, x_n) \oplus (y_0, \ldots, y_m) := (x_0, \ldots, x_n, y_0, \ldots, y_m).

Let \((n, S, V) = el(f). We call \( \Delta^+ (x) := n \) the formula offset of \( x \). We call \( I_{vk}(x) := S \) the verification key indices of \( x \), and we call \( \text{Vars}(x) := V \) the variables occurring in \( x \).

The verification key indices report the positions in a formula that are filled with public identifier. This notion is important because we use it to identify variables that have the strong type \( \text{verkey} \). More precisely, we will later require that for each variable \( x : \text{uidpub} \) at these positions, there is a variable \( y \) such that \( x = y \) and \( y : \text{verkey} \) w.r.t. the current typing environment. The formula offset indicates how many elements a formula contains and is important to keep the logical maps aligned. The variables of a formula allow us to name variables in a formula.

For instance, let

\[
\mathcal{F}^\wedge := \exists y. \text{Prof says } \text{Reg}(y) \land y \text{ says } \text{Eval}(\text{sec, good})
\]

with the encoding

\[
\begin{align*}
\mathcal{F}^\wedge &:= \text{And} \\
&= \text{Says} (\text{Revealed } x_{\text{Prof}}, \text{Reg}^F (\text{Hidden } y)) \\
&\quad \text{Says} (\text{Hidden } y, \text{Eval}^F (\text{Revealed } x_{\text{sec}}, \text{Revealed } x_{\text{good}})) \\
\end{align*}
\]

Then

- \( \Delta^+ (x) = 2, \Delta^+ (y) = 3, \) and \( \Delta^+ (\mathcal{F}^\wedge) = 5 \)
- \( I_{vk}(\mathcal{F}^\wedge) = \{0, 1, 2\} \)
- \( \text{Vars}(\mathcal{F}^\wedge) = \{\text{Revealed } x_{\text{Prof}}, \text{Hidden } y, \text{Hidden } y, \text{Revealed } x_{\text{sec}}, \text{Revealed } x_{\text{good}}\} \)

We deploy three maps: \( E_F, \psi, \) and \( \phi \). Intuitively,

- \( E_F \) takes as input a positional index \( i \) of a variable and returns the smallest positional index \( j \leq i \) such that the value at position \( j \) is equal to the value at position \( i \) (in the formula). For instance, the example in equation \([5]\) yields the following map

\[
\begin{align*}
E_{F^\wedge}(0) &= 0 \quad E_{F^\wedge}(1) = 1 \quad E_{F^\wedge}(2) = 1 \\
E_{F^\wedge}(3) &= 3 \quad E_{F^\wedge}(4) = 4
\end{align*}
\]

- \( \psi \) partitions the variables occurring in the verification function \( \text{verify}_F \) w.r.t. equality, i.e., variables in one partition are pairwise equal. This is of paramount importance: the sealing mechanism will provide access to the witnesses (i.e., the hidden values) in the verification function. We use \( \psi \) to prove equality of these witnesses and these equalities allow us to logically represent equal witnesses by a single existentially quantified variable.
- \( \phi \) tracks variables that can be typed with the strong type \( \text{verkey} \) under the current typing environment. The reason for the tracking is of a very technical nature and inherently necessary for the type-checking process. For instance, the signature verification function requires as argument a value of type \( \text{verkey} \). If a value, however, is hidden in a formula, then the formula does not provide us with a value of that type and the proof only contains values of type \( \text{bitstring} \). In these situations, \( \phi \) will point us to a variable \( y : \text{verkey} \) that can be used as argument to the verification function (the code performs the necessary equality checks to justify using \( y \) as verification key).

Using \textbf{Definition 10} we formally define the maps.
Definition 11 ($E^{=} \psi$, $\psi$, and $\phi$). Let $f := F^{=} be a formula without disjunctions and $(x_0, \ldots, x_n) = \text{Vars}(F^{=})$. We define the index map function $E^{=} : \mathbb{N} \rightarrow \mathbb{N}$ as follows:

$$E^{=} : \mathbb{N} \rightarrow \mathbb{N}, \quad E^{=} (i) = \min_{0 \leq j \leq m} x_j = x_i$$

where $x_i \neq x_j$ denotes that the values or variables corresponding to $x_i$ and $x_j$ in the logical formula $F^{=}$ are equal, i.e., the variables $x_i$ and $x_j$ correspond to the same existentially quantified variable in $F^{=}$ or that the variables $x_i$ and $x_j$ correspond to the same value in $F^{=}$.

$$\psi : \mathbb{N} \rightarrow \varphi(\text{Vars}) \quad \phi : \mathbb{N} \rightarrow \text{Vars}.$$ 

where $\text{Vars}$ denotes the set of all RCF variables and $\varphi(S)$ denotes the power set of the set $S$. Additionally, we define the usual update:

$$\phi[x \mapsto y](z) = \begin{cases} y & \text{if } x = z \\ \phi(z) & \text{otherwise} \end{cases} \quad \psi[x \mapsto y](z) = \begin{cases} y & \text{if } x = z \\ \psi(z) & \text{otherwise} \end{cases}$$

Initially (i.e., if no update is applied) $\forall x. \phi(x) = \bot$ and $\forall x. \psi(x) = \emptyset$.

The updates are contained in the proper locations in the code macros and are marked in the special line numbers.

The following proposition states that extending a formula $F^{=}$ with a logical conjunction does not change the index map restricted to the old formula $F^{=}$.

Proposition 2. Let $F_1^{=}$ and $F_2^{=}$ be a conjunctive formulas. Then, $\forall i < \Delta^{=}(F_1^{=}), E^{=}_{F_1^{=}}(i) = E^{=}_{F_1^{=} \land F_2^{=} }(i)$.

Notation (Extending formulas with logical conjunctions). In the following, we will discuss formulas that are extended by applying a logical conjunctions to it. To simplify our soundness proof, we stipulate that formulas are in a normal form. More precisely, we extended formulas only with elementary formulas, i.e., a formula $F^{=}$ is extended to $F^{=}':= F^{=} \land F$ for some elementary formula $F$. Since logical conjunction is commutative and associative, this restricts only the syntax but does not change the logical meaning.

Notation (Index map $E$ subscript). As stated by Proposition 2 the index map $E$ induced by a formula $F^{=}$ and by a formula $F^{=} \land F$ is the same for all indices $i \leq \Delta^{=}(F^{=})$. Therefore, if the formula is clear from the context, we will drop it from the subscript for the sake of readability in the rest of the paper.

Above, we stated that we use the logical map $\psi$ to keep track of the variables that are equal to each other. We now define the code that will enforce these equality so that we can use them while type-checking later on.

Definition 12 ($\psi$-induced equalities). We define the equality constraints $(S)^= and (\{S_1, \ldots, S_n\})^= as (\{m_1, \ldots, m_n\})^:= m_1 = m_2, \ldots, m_1 = m_n and (\{S_1, \ldots, S_n\})^= := (S_1)^=, \ldots, (S_n)^=.

We define the equality constraint context $\text{cont}^= (\{m_1, \ldots, m_n\})[\bullet]$ as

$$\text{cont}^= (\{m_1, \ldots, m_n\})[\bullet] := \begin{cases} \text{if } m_1 = m_2 \text{ then} & \\ \vdots & \\ \text{if } m_1 = m_n \text{ then} & \\ \bullet & \\ \text{else} & \\ \text{false} & \\ \vdots & \\ \text{else} & \\ \text{false} \end{cases}$$

and $\text{cont}^= (\{S_1, \ldots, S_n\})[\bullet]$ as

$$\text{cont}^= (\{S_1, \ldots, S_n\})[\bullet] := \text{cont}^= (S_1)[\text{cont}^= (S_2)[\ldots [\text{cont}^= (S_n)[\bullet]] \ldots]]$$

Let $M_\psi = \{S \mid S = \psi(i), i \in \mathbb{N}\}$. We define $(\psi)^= := (M_\psi)^=$ the equalities induced by $\psi$ and $\text{cont}^= (\psi) := \text{cont}^= (M_\psi)$ the context induced by $\psi$. 
We will prove the anticipated relation between \( \text{cont}^{-}(\psi) \) and \( (\psi)^{=} \) in \textbf{Lemma 20}.

The last piece missing before we can finally define the verification function is the formula translation \( \llbracket f \rrbracket \). This translation defines how we assemble the individual code macros into the final, formula-specific verification function.

\textbf{Definition 13} (Formula translation). We define the formula translation \( \llbracket f, p, i \rrbracket \) for an arbitrary formula \( f : \text{formula} \), proof \( p : \text{proof} \) and positional index \( i \).

\[
\begin{align*}
[\text{ Says}(x_{v_k}, f'), p, \omega] &::= \text{ Says-Macro}(f, p, \omega) \\
[\text{ SSP}(x_{v_k}, x, x_{p_{\text{pred}}}), p, \omega] &::= \text{ SSP-Macro}(f, p, \omega) \\
[\text{ REL}(x, op, y), p, \omega] &::= \text{ REL-Macro}(f, p, \omega) \\
[\text{ LM}(x, b, t), p, \omega] &::= \text{ LM-Macro}(f, p, \omega) \\
[\text{ LNM}(x, t), p, \omega] &::= \text{ LNM-Macro}(f, p, \omega) \\
[\text{ EscrowInfo}(x_{v_k,x}, x_{v_k}, x_{f_t}, x_{s_t}, x_{s_{\text{idr}}}), p, \omega] &::= \text{ Escrow-Macro}(f, p, \omega) \\
[\text{ And}(f_1, f_2), p, \omega] &::= \text{ And-Macro}(f, p, \omega, \omega + \Delta^+(f_1)) \\
[\text{ Or}(f_1, f_2), p, \omega] &::= \text{ Or-Macro}(f, p) \\
[\text{ true}] &::= \star
\end{align*}
\]

The case \( f = \text{ true} \) is by definition not a formula (cf. Table V) and will only serve as our induction base case in the soundness proof.

\textbf{Listing 32: verify top-level structure}

\begin{verbatim}
1 match p with Or(p1, p2) =>
2   match f with Or(f1, f2) =>
3     let r1 = verifyf1, p1, f1;
4     if r1 = true then
5       r1
6     else
7       verifyf2, p2, f2
8     | _  => false
9     | _  => false
\end{verbatim}

\textbf{Listing 33: Or-Macro}(f, p)

\begin{verbatim}
1 match p with And(p1, p2) =>
2   match f with And(f1, f2) =>
3     \llbracket f1, p1, \omega \rrbracket, \llbracket f2, p2, \omega \rrbracket
4     | _  => false
5     | _  => false
\end{verbatim}

\textbf{Listing 34: And-Macro}(f, p, \omega, j)

\begin{verbatim}
1 match p with Says_s((c_{sig}, _), (c_{z}, _), P_k((c_{arg1}, _), ..., (c_{arg_n}, _))) =>
2   match f with Says(arg'_0, P'_k(arg'_1, ..., arg'_n)) =>
3     let (sig', r_{sig}) = openCommit(c_{sig});
4     let (arg0', r_{arg0}) = openCommit(c_{z});
5     let (arg1', r_{arg1}) = openCommit(c_{arg1});
\end{verbatim}
let \((\text{arg}_n, r_{\text{arg}_n}) = \text{openCommit}(c_{\text{arg}_n})\);

Add lines 8–10, \(U_0^\psi\) for each \(i\) if \(\exists x. \text{arg}'_i = \text{Revealed} x\), i.e., \(i\)-th argument is revealed

let \(\text{arg}'_i = \text{match} \ \text{arg}'_i\) with

| Revealed \(x\) \(\implies\) \(x\)
| Hidden \(-\) \(\implies\) fail\(\langle T_k \rangle ()\);

\(U_0^\psi\)

\(\psi := \psi[\mathcal{E}(\omega + i) \mapsto \psi(\mathcal{E}(\omega + i)) \cup \{\text{arg}'_i\}]\)

EndAdd

let \(\text{arg}'_0 = \text{match} \ \text{arg}'_0\) with

| Revealed \(x\) \(\implies\) \(x\)
| Hidden \(-\) \(\implies\) \(\phi(\mathcal{E}(\omega))\);

if \(\text{arg}'_0 = \text{arg}'_0\) then

let \(m = \text{check}_\text{sig} \ \text{arg}'_0\) sig;

match \(m\) with

| PS\((y'_1, \ldots, y'_n)\) \(\implies\)
| if \(\text{arg}_1 = y'_1\) then ...
| ...

if \(\text{arg}_n = y'_n\) then

\(U_1^\phi\)

\(\phi := \phi[\mathcal{E}(\omega + 1) \mapsto y'_1]\)

...

\(U_n^\phi\)

\(\phi := \phi[\mathcal{E}(\omega + n) \mapsto y'_n]\)

\(U_1^\psi\)

\(\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{\text{arg}_0\}]\)

\(U_2^\psi\)

\(\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{\text{arg}_1, y'_1\}]\)

...

\(U_{n+1}^\psi\)

\(\psi := \psi[\mathcal{E}(\omega + n) \mapsto \psi(\mathcal{E}(\omega + n)) \cup \{\text{arg}_n, y'_n\}]\)


else

false

else

false

\| _ \(\implies\) \text{false}

else

false

\| _ \(\implies\) \text{false}

\| _ \(\implies\) \text{false}

Listing 35: \textit{Says-Macro}(f, p, \omega)

match \(p\) with \(\text{SSP}_p((c_z, _), (c_s, _), (c_{psd}, _), c_x)\) \(\implies\)

match \(f\) with \(\text{SSP}((z', s', psd'), \_\_\_\_\_)\) \(\implies\)

let \((x, r_x) = \text{openCommit}_{s_k}(c_x);\)

let \((z, r_z) = \text{openCommit}(c_z);\)

let \((s, r_s) = \text{openCommit}(c_s);\)

let \((\_ , r_{psd}) = \text{openCommit}(c_{psd});\)

let \((\_ , r_x') = x;\)

Add lines 8–10 and \(U_0^\psi\) if \(\exists y. \text{arg}' = \text{Revealed} y\), i.e., the verification key is revealed
let $z^o = \text{match } z' \text{ with}$

| Revealed $y \Rightarrow y$
| Hidden $\_ \Rightarrow \text{fail}(\text{wid}_{psd})$

$U^0_\psi$

$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z^o\}]$

Add lines 12-14 and $U^0_\psi$ if $\exists y. s' = \text{Revealed } y$, i.e., the service is revealed

let $s^o = \text{match } s' \text{ with}$

| Revealed $y \Rightarrow y$
| Hidden $\_ \Rightarrow \text{fail}(\text{string})$

$U^1_\psi$

$\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{s^o\}]$

Add lines 16-18 and $U^1_\psi$ if $\exists y. psd' = \text{Revealed } y$, i.e., the pseudonym is revealed

let $psd'^o = \text{match } psd' \text{ with}$

| Revealed $y \Rightarrow y$
| Hidden $\_ \Rightarrow \text{fail}(\text{bitstring})$

$U^2_\psi$

$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{psd'^o\}]$

End Add

$U^3_\psi$

$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z, x'\}]$

$U^1_\psi$

$\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{psd\}]$

$U^2_\psi$

$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{s\}]$

End Add

let $psd'' = \text{computePsd } x \ s$;

if $psd'' = psd$ then

else

| false
| $\_ \Rightarrow \text{false}$

Listing 36: SSP-Macro$(f, p, \omega)$

match $p$ with $\text{REL}_p((c_x, \_), (c_y, \_)) \Rightarrow$

match $f$ with $\text{REL}(x', op', y') \Rightarrow$

let $(x, r_x) = \text{openCommit}(c_x)$;

let $(y, r_y) = \text{openCommit}(c_y)$;

let $b = \text{cmp}_{op} x \ y$;

Add lines 6-8 and $U^0_\psi$ if $\exists z. x' = \text{Revealed } z$, i.e., the left operand is revealed

let $x^o = \text{match } x' \text{ with}$

| Revealed $z \Rightarrow z$
| Hidden $\_ \Rightarrow \text{fail}(\text{bitstring})$

$U^0_\psi$

$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x^o\}]$

End Add

Add lines 10-12 and $U^1_\psi$ if $\exists z. y' = \text{Revealed } z$, i.e., the right operand is revealed

let $y^o = \text{match } y' \text{ with}$

| Revealed $z \Rightarrow z$
| Hidden $\_ \Rightarrow \text{fail}(\text{bitstring})$

$U^1_\psi$

$\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{y^o\}]$

End Add

$U^2_\psi$

$\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]$
\[
U_3^\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{ y \}]
\]

if \( b = \text{true} \) then

if \( op = op' \) then

\[
\begin{align*}
\quad & \text{false} \\
\quad & \text{false}
\end{align*}
\]

\[ \left| \_ \right| \leftarrow \text{false} \]

\[ \left| \_ \right| \leftarrow \text{false} \]

Listing 37: \( \text{Rel-Macro}(f, p, \omega) \)

---

1 match \( p \) with \( \text{LM}_p((c_x, \_), (c_b, \_), \ell) \) \( \Rightarrow \)
2 match \( f \) with \( \text{LM}(x', b', \ell') \) \( \Rightarrow \)
3 let \((x, r_x) = \text{openCommit}(c_x)\);
4 let \((b, r_b) = \text{openCommit}(c_b)\);

Add lines 5–7 and \( U_0^\psi \) if \( \exists y. x' = \text{Revealed} y \), i.e., the pseudonym is revealed

5 let \( x^o = \text{match} x' \text{ with} \)
6 \[ \text{Revealed} y \mapsto y \]
7 \[ \text{Hidden} \_ \mapsto \text{fail} \langle \text{bitstring} \rangle () \];

\( U_0^\psi \)

\[
\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{ x^o \}]
\]

End Add

Add lines 9–11 and \( U_1^\psi \) if \( \exists b'. b'' = \text{Revealed} y \), i.e., the attribute is revealed

9 let \( b^o = \text{match} b' \text{ with} \)
10 \[ \text{Revealed} y \mapsto y \]
11 \[ \text{Hidden} \_ \mapsto \text{fail} \langle \text{bitstring} \rangle () \];

\( U_1^\psi \)

\[
\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{ b^o \}]
\]

End Add

\( U_2^\psi \)

\[
\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{ x \}]
\]

\( U_3^\psi \)

\[
\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{ b \}]
\]

\( U_4^\psi \)

\[
\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{ \ell, \ell' \}]
\]

16 let \( r = \text{List.member}^2(2, 2)(\text{pseudo} * \text{bitstring}) x b \ell \);
17 if \( r = \text{true} \) then
18 \[
\quad & \text{false} \\
\quad & \text{false}
\]

\[ \left| \_ \right| \leftarrow \text{false} \]

\[ \left| \_ \right| \leftarrow \text{false} \]

Listing 38: \( \text{LM-Macro}(f, p, \omega) \)

---

1 match \( p \) with \( \text{LM}_p((c_x, \_), \ell) \) \( \Rightarrow \)
2 match \( f \) with \( \text{LM}(x', \ell') \) \( \Rightarrow \)
3 let \((x, r_x) = \text{openCommit}(c_x)\);

Add lines 4–6 and \( U_0^\psi \) if \( \exists y. x' = \text{Revealed} y \), i.e., the pseudonym is revealed

4 let \( x^o = \text{match} x' \text{ with} \)
5 \[ \text{Revealed} y \mapsto y \]
6 \[ \text{Hidden} \_ \mapsto \text{fail} \langle \text{bitstring} \rangle () \];

\( U_0^\psi \)

\[
\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{ x^o \}]
\]
\(U_1^\psi\)
\[
\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{x\}]
\]
\(U_2^\psi\)
\[
\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{\ell, \ell'\}]
\]
let \(b = \text{List.member}_{\text{sub}}(\psi \ast \text{bitstring}) x \ell;\)
if \(b = \text{false}\) then
\[
\bullet
\]
else
\[
\text{false}
\]
\[
| _ \implies \text{false}
\]
\[
| _ \implies \text{false}
\]

\[\text{Listing 39: } \text{LNM-Macro}(f, p, \omega)\]

1. \(\text{match } p \text{ with } \mathcal{E} \in \text{EscrowInfo}_{\text{pub}}(\mathcal{E}, c_x, \ldots, (c_{R}, \ldots, (c_s, \ldots, (c_{	ext{idr}}, \ldots, c_r) \implies}\)
2. \(\text{match } f \text{ with } \mathcal{E} \in \text{EscrowInfo}(\mathcal{E}, x', R', s', \text{idr'}) \implies}\)
3. \(\text{let } (x, r_x) = \text{openCommit}(c_2);\)
4. \(\text{let } (R, r_R) = \text{openCommit}(c_R);\)
5. \(\text{let } (s, r_s) = \text{openCommit}(c_s);\)
6. \(\text{let } (\text{idr}, r_{\text{idr}}) = \text{openCommit}(c_{\text{idr}});\)
7. \(\text{let } (r, r_r) = \text{openCommit}(c_r);\)
8. \(\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z'\}];\)

Add lines 10–12 and \(U_1^\psi\) if \(\exists y. x' = \text{Revealed } y,\)
i.e., if the user’s verification key is revealed

\(U_0^\psi\)
\[
\text{let } x^0 = \text{match } x' \text{ with}
\]
\[
| \text{Revealed } y \implies y
\]
\[
| \text{Hidden } _ \implies \text{fail}(\text{uid}_{\text{pub}}) ()
\]
\[
\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{x^0\}]
\]

End Add

Add lines 14–16 and \(U_2^\psi\) if \(\exists y. R' = \text{Revealed } y, \text{ i.e., if the value } R \text{ is revealed}\)

\(U_1^\psi\)
\[
\text{let } R^0 = \text{match } R' \text{ with}
\]
\[
| \text{Revealed } y \implies y
\]
\[
| \text{Hidden } _ \implies \text{fail}(\text{bitstring}) ()\text{Error};
\]
\[
\psi := \psi[\mathcal{E}(\omega + 2) \mapsto \psi(\mathcal{E}(\omega + 2)) \cup \{R^0\}]
\]

End Add

Add lines 18–20 and \(U_3^\psi\) if \(\exists y. s' = \text{Revealed } y, \text{ i.e., if the service is revealed}\)

\(U_2^\psi\)
\[
\text{let } s^0 = \text{match } s' \text{ with}
\]
\[
| \text{Revealed } y \implies y
\]
\[
| \text{Hidden } _ \implies \text{fail}(\text{string}) ()
\]
\[
\psi := \psi[\mathcal{E}(\omega + 3) \mapsto \psi(\mathcal{E}(\omega + 3)) \cup \{s^0\}]
\]

End Add

Add lines 22–24 and \(U_4^\psi\) if \(\exists y. \text{idr}' = \text{Revealed } y, \text{ i.e., the escrow identifier is revealed}\)

\(U_3^\psi\)
\[
\text{let } \text{idr}^0 = \text{match } \text{idr}' \text{ with}
\]
\[
| \text{Revealed } y \implies y
\]
\[
| \text{Hidden } _ \implies \text{fail}(\text{bitstring}) ()
\]
\[
\psi := \psi[\mathcal{E}(\omega + 4) \mapsto \psi(\mathcal{E}(\omega + 4)) \cup \{\text{idr}^0\}]
\]

End Add

\(U_4^\psi\)
\[
\psi := \psi[\mathcal{E}(\omega) \mapsto \psi(\mathcal{E}(\omega)) \cup \{z\}]
\]
\(U_5^\psi\)
\[
\psi := \psi[\mathcal{E}(\omega + 1) \mapsto \psi(\mathcal{E}(\omega + 1)) \cup \{x\}]
\]
Listing 40: Escrow-Macro(f, p, ω)

Full implementation of the verification function wrapper. We finally write the verification function wrapper. The function outlined in Appendix E

```
verify' (p : proof) (F : formula) =
let r = verify p F;
if r = true then
  assert F;
  r
else
  r
```

D. Proof of Well-Typedness of API Methods

We start with the definitions and lemmas that will ultimately pave our way to showing that our API methods are well-typed. We start formalizing our notion of well-formed formula. Intuitively, a formula is well-formed if we can check that it originates from a trustworthy principal of the system. Later in the type-checking proofs, the well-formedness will establish that all values that are used as verification keys can be given the verification key type verkey. Since type-checking depends on the types of values, well-formedness is a central notion and is of paramount importance in our proofs.

We call a key u registered if it is publicly verifiable that u belongs to a principal of the system. Typically, the owner of a key u registers her key in a public-key infrastructure (PKI) to establish a publicly-verifiable connection between her identity and her public key. This infrastructure can be hierarchically-ordered such as VeriSign [77] or it can be distributed such as webs of trust [78].

**Definition 14** (Trustworthiness of keys). A key u is trustworthy in a monomial \( M = \bigwedge_{i=1}^{m} ap_i \) iff one of the following conditions holds:

- u = vk is registered;
- there exists \( ap_j = \text{Says}(u_k, F) \) such that u is a variable occurring free in F and \( u_k \) is trustworthy in \( M^{<j} := \bigwedge_{i=1}^{j-1} ap_i \).

If we apply this definition to our zero-knowledge proofs, the intuitive meaning is that a key is trustworthy if the key is not hidden or if it is hidden but it is authenticated (via a says-predicate) that is issued by a trustworthy key.

**Definition 15** (Disjunctive form). We say a formula 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 formula can be rewritten in disjunctive form. In the following, we assume a disjunctive normal form for each formula S, written as \( \text{dnf}(S) \). The disjunctive normal form can be obtained, for instance, by lexicographical order.

**Definition 16** (Well-formedness of formulas). A monomial \( M = \bigwedge_{i=1}^{m} ap_i \) is well-formed iff for every \( ap_i = \text{Says}(u_k, F) \), and for every \( ap_j = \text{SSP}(u_k, s, psd) \), u_k and u_i are trustworthy in \( M^{<i} \) and \( M^{<j} \), respectively.

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

**Lemma 1** (Representation lemma). For every Boolean formula f, there is a formula g in disjunctive form such that \( f \iff g \).

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

In the following, we assume that all formulas are well-formed and in disjunctive normal form.
Notation and auxiliary lemmas. Logical formulas that are tracked in typing environments \( E \) are a substantial part of our proofs. Technically, a formula \( F \) does not occur in \( E \) but it is always attached to a refined variable \( \_ : \text{unit} \mid F \) where \( \_ \) does not occur in \( F \). We use the abbreviated notation \( \{ F \} \) to denote \( \_ : \text{unit} \mid F \). In this type, the variable and the variable of the refinement type are unnamed. Formally, we use fresh names that occur nowhere else to instantiate the placeholders \( \_ \).

To justify this notation, let us see where our notation influences the type-checking process. While type-checking, the only place where placeholders play a role is the well-formedness check of the typing environment. Intuitively, a well-formed typing environment \( E \) is closed if all formulas are closed and variables are only bound once. Since placeholders are instantiated by fresh variables that occur nowhere else, they do not influence formulas (because they do not occur therein) and they are only occur in the typing environment once. As a result, our notation does not interfere with the type-checking process and we can safely use it.

Regarding the well-typedness of an expression, we state the following lemma which will help us with the later proofs. Intuitively, the lemma says that if an expression type-checks under a typing environment \( E \), then \( E \vdash \circ \), i.e., \( E \) is well-formed.

Lemma 2 (Type-checking implies well-formedness). Let \( E \) be a typing environment and \( J \) be a judgment. If \( E \vdash J \), then \( E \vdash \circ \).

The proof of our lemma uses the derived judgments lemma.

Lemma 3 (Derived judgments (Lemma 2 [8])).

1. If \( E \vdash T \), then \( E \vdash \circ \) and \( \text{fnfv}(T) \subseteq \text{dom}(E) \).
2. If \( E \vdash C \), then \( E \vdash \circ \) and \( \text{fnfv}(C) \subseteq \text{dom}(E) \).
3. If \( E \vdash T :: \_ \), then \( E \vdash T \).
4. If \( E \vdash T <: U \), then \( E \vdash T \) and \( E \vdash U \).
5. If \( E \vdash A :: T \), then \( E \vdash T \) and \( \text{fnfv}(A) \subseteq \text{dom}(E) \).

Proof of Lemma 2. Let us consider all the possible judgments \( J \) that the RCF type-system can verify (cf. Bengtson et al. [8], Table “Judgments”, Section 4).

\( E \vdash \circ \):

- Our claim follows directly since \( J = \circ \).

\( E \vdash T \):

- Our claim follows as direct consequence of Lemma 3 (1).

\( E \vdash C \):

- Our claim follows as direct consequence of Lemma 3 (2).

\( E \vdash T :: \_ \):

- Our claim follows as consequence of Lemma 3 (5) and the case \( E \vdash T \).

\( E \vdash T <: U \):

- Our claim follows as consequence of Lemma 3 (4) and the case \( E \vdash T \).

\( E \vdash A :: T \):

- Our claim follows as consequence of Lemma 3 (5) and the case \( E \vdash T \).

In our proofs, we will often find ourselves in a situation where we know that certain facts can be proven by a typing environment \( E \), but we need to prove these facts with an environment of the form \( E, E' \), i.e., the environment \( E \) extended with the entries in \( E' \). Intuitively, we would like to drop the entries in the extended environment until only those that we need are left. This intuition is formalized and proven by the following weakening lemma.

Lemma 4 (Weakening, Lemma 6 [8]). If \( E, E' \vdash J \) and \( E, \mu, E' \vdash \circ \), then \( E, \mu, E' \vdash J \), where \( \mu \) corresponds to one single entry in the typing environment.

Similar to weakening, it will be handy to be able to add to our typing environment \( E \) formulas that we can logically derive from \( E \). This is called strengthening and formalized as follows:
Lemma 5 (Anon Variable Strengthening, Lemma 4 [8]). If \( E, \{C\}, E' \vdash \mathcal{J} \) and \( \text{forms}(E, E') \vdash C \), then \( E, E' \vdash \mathcal{J} \), where \( \text{forms}(E'') \) returns all the logical formulas (i.e., formulas in refinement types) contained in \( E'' \).

In the code, we make heavy use of if-statements. They are syntactic sugar that we encode into the RCF calculus as follows:

**Definition 17** (Encoding and type-checking of conditionals). We encode conditionals as follows:

- \( \text{if } M = N \text{ then } A \quad \text{else } B \quad : V \)
- \( \text{let } x = (M = N) \text{ in } A \text{ else } B \)
- \( \text{match } x \text{ with } \text{true} \rightarrow A \text{ else } B \)

Additionally, we add the following (derived) rule to the RCF type-system.

\[
\frac{\text{EXP IF}}{E \vdash \circ \quad E \vdash M : T \quad E \vdash N : U \quad E, \{M = N\} \vdash A : V \quad E \vdash B : V}{E \vdash \text{if } M = N \text{ then } A \text{ else } B : V}
\]

The following lemma justifies that we can extend the RCF type system by the EXP IF rule.

**Lemma 6.** Rule EXP IF is derivable in \( F7 \).

**Proof:** We show that the hypotheses of EXP IF are strong enough to imply the premises deriving from type-checking the de-sugared code for the if statement. We start by type-checking the de-sugared version of the code under a typing environment \( E \).

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

where \( x \) is a fresh name that occurs nowhere else.

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

\[
\begin{align*}
\text{EXP IF} & \quad \frac{E \vdash T \quad E \vdash N : U \quad x \notin \text{fv}(M, N)}{E \vdash (M = N) : W} \\
\text{EXP LET} & \quad \frac{E, x : W \vdash C : V \quad x \notin \text{fv}(V)}{E \vdash \text{let } x = (M = N) \text{ in } C : V}
\end{align*}
\]

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

\[
\begin{align*}
\text{EXP MATCH} & \quad \frac{E' \vdash x : \text{bool} \quad \text{inr} : (\text{unit}, \text{bool}) \quad E', (\text{inr}) = x \vdash A : V \quad E', \{\forall y, \text{inr } y \neq x\} \vdash B : V}{E' \vdash \text{match } x \text{ with true } \rightarrow A \text{ else } B : V}
\end{align*}
\]

We now show that the hypotheses of EXP IF imply the hypotheses of the derivation for the de-sugared if-statement, i.e., we show that \( (\text{a}) \sim (\text{c}) \) imply \( (\text{1}) \sim (\text{6}) \).

\[
\begin{align*}
(\text{a}) & \quad E \vdash \circ \\
(\text{b}) & \quad E \vdash M : T \\
(\text{c}) & \quad E \vdash N : U \\
(\text{d}) & \quad E, \{M = N\} \vdash A : V \\
(\text{e}) & \quad E \vdash B : V
\end{align*}
\]

\[
\begin{align*}
(\text{1}) & \quad E \vdash M : T \\
(\text{2}) & \quad E \vdash N : U \\
(\text{3}) & \quad x \notin \text{fv}(M, N, V) \\
(\text{4}) & \quad E, x : W \vdash x : \text{bool} \\
(\text{5}) & \quad E, x : W, (\text{inr}) = x \vdash A : V \\
(\text{6}) & \quad E, x : W, \{\forall y, \text{inr } y \neq x\} \vdash B : V
\end{align*}
\]

(\text{1}) and (\text{2}): \( E \vdash M : T, E \vdash N : U \)

Immediately by (\text{b}) \( (E \vdash M : T) \) and (\text{c}) \( (E \vdash N : U) \).

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

Follows from our convention that the variable \( x \) in the de-sugared version is fresh and occurs nowhere else.

(\text{4}): \( E, x : W \vdash x : \text{bool} \)
Follow since $E, x : W \vdash W <: \text{bool}$ by \textsc{Sub Refine Left} and \textsc{Sub Refl}.

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

This case is the most involved one in the proof.

Proof steps: 

This is proven as hypothesis of rule \textsc{Exp If}.

The shape of the current typing environment is almost the one from (4). We apply \textsc{Lemma 4} (Weakening) thrice to add the entries $\{ \text{inr}() = x \}$, $() : \text{unit}$, and $x : W$ in that order to our current typing environment. The order is important to maintain the well-formedness condition required by the weakening lemma. We stress that adding $x$ does not break the well-formedness guaranteed by (a) since $x$ is fresh and does not occur anywhere else.

We note that $\text{forms}(E_1) \vdash M = N$, 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 \textsc{Lemma 5} (Strengthening) 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): $E, x : W, \{ \forall y. \text{inr} y \neq x \} \vdash B : V$

We apply \textsc{Lemma 4} (Weakening) twice on (e) $(E \vdash B : V)$, adding the $x : W$ and $\{ \forall y. \text{inr} y \neq x \}$ in that order, and obtain premise (6). Since $x$ is fresh, well-formedness of the extended environment is not affected.

We have proven that the premises of rule \textsc{Exp If} imply the premises required to hold by the de-sugared version of an if-statement. This concludes our proof.

Also, in our code we often have large cascades of if-statements. The following lemma eases the type-checking effort for these constructions.

\textbf{Lemma 7} (Lemma 9 in [81]). Let $E$ be basic 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

$$C := \begin{cases} 
\text{if } M_1 = N_1 \text{ then } & \\
\text{if } M_2 = N_2 \text{ then } & \\
\vdots & \\
\text{if } M_\ell = N_\ell \text{ then } & \\
C' & \\
\text{else } & \\
\text{fail}(V) () & \\
\vdots & \\
\text{else } & \\
\text{fail}(V) () & \\
\text{else } & \\
\text{fail}(V) ()
\end{cases}$$

If $E, \_, \{ M_1 = N_1 \}, \ldots, \{ M_\ell = N_\ell \} \vdash C' : V$ 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 $E \vdash C : V$.

\textbf{Proof:} The proof is by induction on the number of cascaded if statements.

RCF only allows pairs rather than $n$-ary tuple. We use the following abbreviation to encode arbitrary tuples in RCF.
Definition 18 (General tuples). We use the convention that pairs are right-associative and we write
\[ x_1 : T_1 \ast \cdots \ast x_{n-1} : T_{n-1} \ast T_n \]
to denote \( \sum x_1 : T_1. \sum x_2 : T_2. \cdots \sum x_{n-1} : T_{n-1}. T_n \), where the scope of \( x_i \) is \( T_{i+1}, \ldots, T_n \), and we write
\[ T_1 \ast \cdots \ast T_n \]
to denote \( \sum x_1 : T_1, \ldots, x_n : T_n \) for some \( x_i \) if the pair is not dependent.

The RCF calculus allows for splitting of pairs, however, not for splitting tuples of arbitrary length. We introduce syntactic sugar for such a construct.

Definition 19 (Splitting tuples). We use the following syntactic sugar:
\[ \text{let } (x_1, \ldots, x_n) = x \text{ in } A \]
as abbreviation for
\[ \text{let } (x_1, x^1) = x \text{ in } \]
\[ \text{let } (x_2, x^2) = x^1 \text{ in } \]
\[ \vdots \]
\[ \text{let } (x_{n-1}, x_n) = x^{n-1} \text{ in } A \]
Additionally, we add the following (derived) rule to the type system:
\[
\text{Exp Split}^n \\
\frac{E \vdash \phi}{E, x_1 : T_1, \ldots, x_n : T_n, \{(x_1, \ldots, x_n) = M\} \vdash A : V} \\
\frac{E \vdash M : x_1 : T_1 \ast \cdots \ast x_{n-1} : T_{n-1} \ast T_n, \{x_1, \ldots, x_n\} \cap \text{fv}(V) = \emptyset}{E \vdash \text{let } (x_1, \ldots, x_n) = M \text{ in } A : V}
\]

Lemma 8. \text{Exp Split}^n \text{ is derivable in F7.}

Proof: We show that the hypotheses of \text{Exp Split}^n imply the hypotheses of the de-sugared code. We start by type-checking the de-sugared version of the code under the typing environment \( E \).

\[ \text{let } (x_1, x^1) = x \text{ in } \]
\[ \text{let } (x_2, x^2) = x^1 \text{ in } \]
\[ \vdots \]
\[ \text{let } (x_{n-1}, x_n) = x^{n-1} \text{ in } A \]
where \( x_1, \ldots, x_n \) and \( x^1, \ldots, x^{n-2} \) are fresh variables that occur nowhere else.

We denote the hypothesis of rule \text{Exp Split}^n as follows:
(a) \( E \vdash \phi \)
(b) \( E \vdash M : U_1 \)
(c) \( E, x_1 : T_1, \ldots, x_n : T_n, \{(x_1, \ldots, x_n) = M\} \vdash A : V \)
(d) \( \{x_1, \ldots, x_n\} \cap \text{fv}(V) = \emptyset \)

We define \( U_i = x_i : T_i \ast \cdots \ast x_{n-1} : T_{n-1} \ast T_n \), type-check the desugared code as illustrated below (technically, we apply rule \text{Exp Split} n - 1 times).

\[
\text{Exp Split} \\
\frac{E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots, x^{n-2} : U_{n-1}, \{x_{n-1}, x_n : T_{n-1} \ast T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \vdash A : V}{E \vdash \text{let } (x_1, x^1) = x \text{ in } \}
\]
\[
\text{Exp Split} \\
\frac{E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = x\} \vdash \text{let } (x_2, x^2) = x^1 \text{ in } \ldots \text{let } (x_{n-1}, x_n) = x^{n-2} \text{ in } A : V}{E \vdash x : (x_1 : T_1 \ast U_2, \{x_1, x^1\} \cap \text{fv}(V) = \emptyset)}
\]
\[
\text{Exp Split} \\
\frac{E \vdash \text{let } (x_1, x^1) = x \text{ in } \text{let } (x_2, x^2) = x^1 \text{ in } \ldots \text{let } (x_{n-1}, x_n) = x^{n-2} \text{ in } A : V}{E \vdash (x_1, x^1) \ast x \ast U_2}
\]

(1) \( E \vdash M : x_1 : T_1 \ast U_2 \)
(2) \( \forall i < n. E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots, x^{i-2} : U_{i-1}, \{(x_{i-2}, x^{i-2}) = x^{i-3}\} \vdash x^{i-2} : x_{i-1} : T_{i-1} \ast T_i \)
(3) \( \{x_1, x^1\} \cap \text{fv}(V) = \emptyset \)
(4) \( \forall i < n. \{x_{i-1}, x^{i-1}\} \cap \text{fv}(V) = \emptyset \)
(5) \( E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots, x^{n-2} : U_{n-1}, x_{n-1} : T_{n-1} \ast T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \vdash A : V \)
We prove that the hypothesis of rule \( \text{EXP SPLIT}^n \) are strong enough to entail the obligations collected above.

\[ \begin{align*}
(1) & : E \vdash M : x_1 : T_1 * U_2 \\
& \quad \text{Follows immediately from (b) \( (E \vdash M : U_1) \) since \( U_1 := x_1 : T_1 * U_2 \).}

(2) & : \forall 3 < i \leq n. E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots , x^{i-2} : U_{i-1}, \{(x_{i-2}, x^{i-2}) = x^{i-3}\} \vdash x^{i-2} : x_1 : T_{i-1} * T_i
\\
& \quad \text{Since \( E \) is well-formed by (a) \( (E \vdash \emptyset) \) and all variables added are fresh, the extended typing environment in \( (2) \) is also well-formed. This extended environment contains the statement to be proven so we can apply VAL \ VAR and obtain the desired result immediately.}

(3) & : \{x_1, x^1\} \cap \text{fv}(V) = \emptyset, \forall 1 < i \leq n. \{x_{i-1}, x^i\} \cap \text{fv}(V) = \emptyset
\\
& \quad \text{Follows from (d) \( (\{x_1, \ldots , x_n\} \cap \text{fv}(V) = \emptyset) \) and the fact that \( x^1, \ldots , x^{n-2} \) are fresh and occur nowhere else.}

(5) & : E, x_1 : T_1, x^1 : U_2, \{(x_1, x^1) = M\}, \ldots , x^{n-2} : U_{n-1}, x_{n-1} : T_{n-1}, x_n : T_n, \{(x_{n-1}, x_n) = x^{n-2}\} \vdash A : V
\\
& \quad \text{This one is the most involved one in the proof.}

\text{Proof steps:}

This is proven as hypothesis (d) of \( \text{EXP SPLIT}^n \).

We apply \textbf{Lemma 4} (Weakening) \( 2n - 3 \) times \( (n - 1) \) times for entries of the form \( \{(x_i, x^i) = x^{i-1}\} \) and \( n - 2 \) times for variables of the form \( x^i \) to introduce \( x^1, \ldots , x^{n-2} \) and the entries \( \{(x_1, x^1) = M\}, \ldots , \{(x_{n-1}, x_n) = x^{n-2}\} \).

We can see that \( \text{forms}(E_1) \vdash (x_1, \ldots , x_n) = M \) (viewing the \( n \)-tuple as nested pairs). This is the case because we can use the substitution property of the logic and derive \( (x_1, \ldots , x_n) = M \) from \( \{(x_1, x^1) = M\}, \{(x_2, x^2) = x^1\}, \ldots , \{(x_{n-1}, x_n) = x^{n-2}\} \). Hence, by using strengthening \( \textbf{Lemma 5} \), we can drop the last entry in the environment.

This is the required premise (5) of the desugared version.

The RCF calculus does not allow to match type-constructors and, at the same time split the arguments if it is a tuple. We use the following syntactic sugar.

\textbf{Definition 20} (Matching of constructor with tuples as arguments). We use the following syntactic sugar:

\[
\text{match } M \text{ with } \\
| h_1 (x^1_1, \ldots , x^1_{m_1}) \rightarrow A_1 \\
| h_2 (x^2_1, \ldots , x^2_{m_2}) \rightarrow A_2 \\
| \quad \vdots \\
| h_n (x^n_1, \ldots , x^n_{m_n}) \rightarrow A_n
\]
as abbreviation for

\[
\begin{align*}
\text{match } M \text{ with } h_1 \ x^1 & \rightarrow \\
& \quad \text{let } (x_1^1, \ldots, x_{m_1}^1) = x^1 \text{ in } A_1 \\
\text{else match } M \text{ with } h_2 \ x^2 & \rightarrow \\
& \quad \text{let } (x_1^2, \ldots, x_{m_2}^2) = x^2 \text{ in } A_2 \\
& \quad \vdots \\
\text{else match } M \text{ with } h_n \ x^n & \rightarrow \\
& \quad \text{let } (x_1^n, \ldots, x_{m_n}^n) = x^n \text{ in } A_n
\end{align*}
\]

where the variables \( x^i \) and \( x_j^i \) are fresh.

Additionally, we add the following (derived) rule to the type system:

\[
\begin{align*}
\textbf{Exp Match-Split}^{\alpha}_{(m_1, \ldots, m_n)} \\
E \vdash \diamond \quad E \vdash M: T \\
\forall 0 < i \leq n, h_i : (H_i, T) \\
\forall 0 < i \leq n, H_i = x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, (M = h_i(x_1^i, \ldots, x_{m_i}^i)) \vdash A_i : U \\
\hline
E \vdash \text{match } M \text{ with } | h_1 (x_1^1, \ldots, x_{m_1}^1) \rightarrow A_1 | h_n (x_1^n, \ldots, x_{m_n}^n) \rightarrow A_n : U
\end{align*}
\]

For the sake of readability, we will often omit the parameters \( n \) and \((m_1, \ldots, m_n)\).

**Lemma 9.** \textbf{Exp Match-Split}^{\alpha}_{(m_1, \ldots, m_n)} is derivable in F7.

**Proof:** We show that the hypothesis of the individual cases are strong enough to imply the premises required to type-check the desugared version of the match construct. In our proof, we will first show that the individual match statements and tuple splits of the desugared version can be type-checked. Then we will give an inductive argument that shows that the logical formulas we accumulate in the else branches can be reconstructed using weakening [Lemma 4].

We denote the hypothesis of \textbf{Exp Match-Split}^{\alpha}_{(m_1, \ldots, m_n)} as follows:

(a) \( E \vdash \diamond \)
(b) \( E \vdash M: T \)
(c) \( \forall 0 < i \leq n, \ h_i : (H_i, T) \)
(d) \( \forall 0 < i \leq n, \ H_i = x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i \cdot T_{m_i}^i \)
(e) \( \forall 0 < i \leq n, \ E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : T_i \)

We type-check the \( i \)-th the desugared code to gather the obligations which we need to prove. We do not consider the else case here; we will argue about it in our inductive argument.

\[
\begin{align*}
\textbf{Exp Split}^{\alpha} : \\
E, x^i : H_i, \{h \ x^i = M\} \vdash \diamond \\
E, x^i : H_i, \{h \ x^i = M\} \vdash x^i : T_1^i \cdots T_{m_i}^i \\
E, x^i : H_i, \{h \ x^i = M\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{(x_1^i, \ldots, x_{m_i}^i) = x^i\} \vdash A : T_i \\
\hline
E \vdash M : T \\
h_i : (H_i, T) \\
\{x_1^i, \ldots, x_{m_i}^i\} \cap \text{fv}(T_i) = \emptyset \\
\hline
\textbf{Exp Match} \\
E, x^i : H_i, \{h \ x^i = M\} \vdash \text{let } (x_1^i, \ldots, x_{m_i}^i) = x^i \text{ in } A_i : T_i
\end{align*}
\]

(1) \( E \vdash M : T \)
(2) \( h_i : (H_i, T) \)
(3) \( E, x^i : H_i, \{h \ x^i = M\} \vdash \diamond \)
(4) \( E, x^i : H_i, \{h \ x^i = M\} \vdash x^i : T_1^i \cdots T_{m_i}^i \)
(5) \( E, x^i : H_i, \{h \ x^i = M\}, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{(x_1^i, \ldots, x_{m_i}^i) = x^i\} \vdash A : T_i \)
(6) \( \{x_1^i, \ldots, x_{m_i}^i\} \cap \text{fv}(T_i) = \emptyset \)

We prove that the hypothesis of rule \textbf{Exp Match-Split}^{\alpha}_{(m_1, \ldots, m_n)} are strong enough to entail the obligations collected above.

(1) \( E \vdash M : T \)
Directly entailed by (5) \((E \vdash M : T)\).

(2): \(h_i : (H_i, T)\)

Immediately, by (c) \((\forall 0 < i \leq n. h_i : (H_i, T))\).

(3): \(E, x^i : H_i, \{ h x^i = M \} \vdash \) …

Follows from (a) \((E \vdash \alpha)\) and because of the convention that the \(x^i\) are fresh and occur nowhere else (the formula does not add a binder to the environment).

(4): \(E, x^i : H_i, \{ h x^i = M \} \vdash x^i : T_1^i \cdots * T_n^i\)

Since \(H_i\) is exactly defined as \(H_i = x_1^i : T_1^i, \ldots, x_{m-1}^i : T_{m-1}^i \ast T_m^i\) by (a) \((\forall 0 < i \leq n. H_i = x_1^i : T_1^i, \ldots, x_{m-1}^i : T_{m-1}^i \ast T_m^i)\), the used typing environment is well-formed (all newly-introduced variables are fresh), and contains \(x^i : H_i\), the statement follows by applying rule VAL VAR.

(5): \(E, x^i : H_i, \{ h x^i = M \}, x_1^i : T_1^i, \ldots, x_{m-1}^i : T_{m-1}^i, \{ (x_1^i, \ldots, x_{m-1}^i) = x^i \} \vdash A : T_i\)

This is the most involved proof step.

Proof steps:                                      Proven statement:

Proven as hypothesis (c) of rule

\[\text{EXP MATCH-SPLIT}^n_{m_1, \ldots, m_n}\]

We apply weakening thrice to add the entries \(x^i : H_i, \{ M = h_i x^i \}, \{ (x_1^i, \ldots, x_{m-1}^i) = x^i \}\) to the typing environment.

The formulas in the environment, in particular \(\{ M = h_i x^i \}\) and \(\{ (x_1^i, \ldots, x_{m-1}^i) = x^i \}\) logically entail the formula \(M = h_i(x_1^i, \ldots, x_{m-1}^i)\). We use strengthening (Lemma 5) to drop the last environment entry \(\{ M = h_i(x_1^i, \ldots, x_{m-1}^i) \}\).

This is the required hypothesis (5).

(6): \(\{ x_1^i, \ldots, x_{m-1}^i \} \cap fv(T_i) = \emptyset\)

Immediately, since all variables \(x^i\) and \(x_j^i\) are fresh.

In the above proof, we have lazily neglected to type-check the else branch of the match statements. We now argue why the else branches will also type-check.

Using a simple inductive argument, we see that the \(i\)-th else branch will type-check with the typing environment \(E, \{ \forall x. h_i x \neq M \}_i, \ldots, \{ \forall x. h_i x \neq M \}_j\). Above, we have proven that every branch of the \(n\) branches will type-check under the typing environment \(E\). Therefore, we can apply weakening \(i\) times and add the entries to accommodate the typing environment of the \(i\)-th else branch.

\[\square\]

**Definition 21** (Recursive Variables \textit{recvar}).

**Definition 22** (Free Names, Free Types).

**Lemma 10** (Type \textit{verkey} is public).

**Lemma 11** (Type \textit{proof} and \textit{formula} are public).

The most basic typing environment that we can use contains all the auxiliary functions with their type, the signing and verification key types and so on. The complete list is given in the following definition:
\[

table\text{ VII: Subtyping rules: } E \vdash T <: U
\]

<table>
<thead>
<tr>
<th>KIND VAR</th>
<th>( E \vdash \odot )</th>
<th>( (\alpha :: \nu) \in E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIND UNIT</td>
<td>( E \vdash \odot )</td>
<td>( E \vdash \text{unit} :: \nu )</td>
</tr>
<tr>
<td>KIND FUN</td>
<td>( E \vdash T :: \nu )</td>
<td>( E \vdash (\Pi x : T . U) :: \nu )</td>
</tr>
<tr>
<td>KIND REC</td>
<td>( E \vdash \alpha :: \nu \vdash T :: \nu )</td>
<td>( E \vdash (\mu a . T) :: \nu )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>KIND REFINED PUBLIC</th>
<th>( E \vdash { x : T \mid \mathcal{F} } )</th>
<th>( E \vdash T :: \text{pub} )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( E \vdash { x : T \mid \mathcal{F} } :: \text{pub} )</td>
<td>( E \vdash { x : T \mid \mathcal{F} } :: \text{tnt} )</td>
</tr>
</tbody>
</table>

Table VIII: Kinding rules: \( E \vdash T :: \nu, \nu \in \{ \text{pub}, \text{tnt} \} \)

**Definition 23 (Basic typing environment).** We call a typing environment \( E \) basic if and only if for all principals \( I \) occurring in the protocol all of the following conditions hold:

- \( E \vdash \odot \)
- \( E \vdash \text{fail}(\alpha) : \text{unit} \rightarrow \alpha \)
- \( E \vdash \text{List.member}^{(i,j)}(\alpha_1, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_j) : (y_1 : \alpha_1) \rightarrow \cdots \rightarrow (y_i : \alpha_i) \rightarrow \left( \ell : \alpha_1 \cdots \alpha_i \beta_{i+1} \cdots \beta_j \text{list} \right) \rightarrow \{ x : \text{bool} \mid x = \text{true} \leftrightarrow \exists y_{i+1}, \ldots, y_j, (y_1, \ldots, y_i) \in \ell \} \)
- \( E \vdash \text{List.get}^{(i,j)}(\alpha_1, \ldots, \alpha_i, \beta_{i+1}, \ldots, \beta_j) : (y_1 : \alpha_1) \rightarrow \cdots \rightarrow (y_i : \alpha_i) \rightarrow (\ell : \alpha_1 \cdots \alpha_i \beta_{i+1} \cdots \beta_j \text{list}) \rightarrow \{ (y_1, \ldots, y_i, y_j) : \alpha_1 \cdots \alpha_i \beta_{i+1} \cdots \beta_j \mid \exists y_{i+1}, \ldots, y_j, (y_1, \ldots, y_i) \in \ell \} \)
- \( E \vdash \text{cmp}_\text{op} : (x : \text{unit}) \rightarrow (y : \text{unit}) \rightarrow \{ z : \text{bool} \mid z = \text{true} \leftrightarrow x \text{ op } y \} \)
- \( E \vdash \text{computeR} : (x : \text{unit}) \rightarrow (r : \text{unit}) \rightarrow \text{unit} \)
- \( E \vdash \text{computeDR} : (vk_{E_A} : \text{unit}) \rightarrow (x : \text{verkey}) \rightarrow (r : \text{unit}) \rightarrow (R : \text{unit}) \rightarrow (s : \text{unit}) \rightarrow \{ \text{idr} : \text{unit} \mid \text{EscrowInfo}(vk_{E_A}, vk, R, s, \text{idr}) \} \)
- \( E \vdash \text{computePsd} : (sk : \text{sigkey}) \rightarrow (s : \text{unit}) \rightarrow \{ x : \text{unit} \mid \exists y, z, sk = (y, z) \land \text{SSP}(z, s, x) \} \)
- \( E \vdash \text{fail}(\alpha) : \text{unit} \rightarrow \alpha \text{ c} \text{ unit} \)
- \( E \vdash \text{rand} : \text{unit} \rightarrow \text{unit} \)
- \( E \vdash \text{sign} : (sk : \text{sigkey}) \rightarrow (m : U^E_{\text{sk}}) \rightarrow \text{unit} \)
- \( E \vdash \text{check}_{\text{sig}} : (y : \text{verkey}) \rightarrow (\text{sig} : \text{unit}) \rightarrow T_{sk}(\text{verkey}/\alpha) \)
- \( E \vdash \text{getOperation} : \text{unit} \rightarrow (\text{unit} \ast \text{unit} \rightarrow \text{bool}) \)
- \( E \vdash \text{commit} : (\text{unit} \ast \text{unit}) \rightarrow \text{unit} \)
- \( E \vdash \text{openCommit} : \text{unit} \rightarrow (\text{unit} \ast \text{unit}) \)
- \( E \vdash \text{commit}_{sk} : (\text{sigkey} \ast \text{unit}) \rightarrow \text{unit} \)
- \( E \vdash \text{openCommit}_{sk} : \text{unit} \rightarrow (\text{sigkey} \ast \text{unit}) \)
In the verification function macros, we use code of the following form

\[
\text{match } M \text{ with } h_1(w, (x, y), h_2 z) \rightarrow A
\]

i.e., we have a match, a split, and a inside the split another split and even another match. This syntactic sugar can be spelled out to

\[
\begin{align*}
\text{match } M & \text{ with } h_1(w, v_2, v_3) \rightarrow \\
& \text{let } (x, y) = v_2 \text{ in} \\
& \text{match } v_3 \text{ with } h_2 z \rightarrow \\
& A \\
& \text{else fail()}()
\end{align*}
\]

**Definition 24** (binders, match code, closers, and variables). Let \( v' := (e_1, \ldots, e_n) : T_1 \cdots T_n \) and let \( v := h \ v' \) be the value that is matched against in a match expression \( A \), where \( e_i \in \{x, (x_1, \ldots, x_k), h' x\} \). We define the data type \( D(e_i) \) of \( e_i \), the type \( T(e_i) \) of \( e_i \), the binders \( B(e_i) \) of \( e_i \), the match code \( M(e_i) \) of \( e_i \), the variable \( V(e_i) \) of \( e_i \), and the closer \( C(e_i) \) of \( e_i \) as follows:

\[
\begin{align*}
M(x, e_i) & := \epsilon \quad \text{if } e_i = y \text{ where } \epsilon \text{ is the empty expression;} \\
M(x, e_i) & := \text{let } (y_1, \ldots, y_n) = x; \quad \text{if } e_i = (y_1, \ldots, y_n), \text{ i.e., a tuple split;} \\
M(x, e_i) & := \text{match } x \text{ with } h' y \rightarrow \quad \text{if } e_i = h' y; \\
C(e_i) & := \text{else fail()}() \quad \text{if } e_i = h' x; \\
C(e_i) & := \epsilon \quad \text{else where } \epsilon \text{ is the empty expression.}
\end{align*}
\]

We require that if \( V \) returns a fresh variable, it returns the same variable if queried for the same entry several times.

**Definition 25.** We use the following syntactic sugar:

\[
\begin{align*}
\text{match } M \text{ with} \\
| h_1 (e_1^1, \ldots, e_{m_1}^1) \rightarrow A_1 \\
| \quad \vdots \\
| h_n (e_1^n, \ldots, e_{m_n}^n) \rightarrow A_n
\end{align*}
\]

where \( e_j^i \in \{x, (x_1, \ldots, x_n), h_j^i x\} \) as abbreviation for

\[
\begin{align*}
\text{match } M \text{ with} \\
| h_1 (x_1^1, \ldots, x_{m_1}^1) \rightarrow M(x_1^1, e_1^1) \cdots M(x_{m_1}^1, e_{m_1}^1) A_1 C(e_1^1) \cdots C(e_{m_1}^1) \\
| \quad \vdots \\
| h_n (x_1^n, \ldots, x_{m_n}^n) \rightarrow M(x_1^n, e_1^n) \cdots M(x_{m_n}^n, e_{m_n}^n) A_n C(e_1^n) \cdots C(e_{m_n}^n)
\end{align*}
\]

with fresh \( x_j^i \).

Note that this syntactic sugar only works for non-dependent tuples \( (e_1^1, \ldots, e_{m_1}^1) \), i.e., tuple types where the type \( T_j^i \) does not use entries \( e_{k<j}^i \); since the definition of dependent pair types \( \sum x : T. U \) demands that there a particular variable \( x \) and that \( x \) can be used in \( U \). In our syntactic sugar, however, there might be a tuple \( (x, y) \) that we cannot use in \( U \).

**Definition 26.**

\[
\begin{align*}
B(x, e_i) & := \epsilon \quad \text{if } e_i = x, \text{ i.e., } e_i \text{ is a single variable;} \\
B(x, e_i) & := y_1 : U_1, \ldots, y_n : U_n \quad \text{for some types } U_k, \text{ if } e_i = (y_1, \ldots, y_n), \text{ i.e., } e_i \text{ is a tuple;} \\
B(x, e_i) & := y : U \quad \text{if } e_i = h' y, \text{ i.e., } e_i \text{ is a match;} \\
B_d(x, e_i) & := B(x, e_i) \quad \text{if } e_i = x, \text{ i.e., } e_i \text{ is a single variable;} \\
B_d(x, e_i) & := B(x, e_i), \{x = (x_1, \ldots, x_n)\} \quad \text{if } e_i = (x_1, \ldots, x_n), \text{ i.e., } e_i \text{ is a tuple;} \\
B_d(x, e_i) & := B(x, e_i), \{x = h' y\} \quad \text{if } e_i = h' x, \text{ i.e., } e_i \text{ is a match;}
\end{align*}
\]

where \( \epsilon \) is the empty environment entry, i.e., the environment is not changed when “adding” \( \epsilon \) entries.
Lemma 12. Let $C$ be sugared match statement as in Definition 25 and let $M$, $A_1, \ldots, A_n$, and $e_j^i$ occurring in $C$ be as in Definition 25 and let $C_d$ be the desugared version of $C$. Let $E$ be a typing environment such that

- $E$ satisfies the premises of the EXP MATCH-SPLIT rule for typing the match statement $C_d$
- $E \vdash \forall 0 < i \leq n, E, B(e_1^i), \ldots, B(e_m^i), \{M = h_i (e_1^i, \ldots, e_m^i)\} \vdash A_i$

Furthermore, if $e_j^i = h_j^i \ x$ for some $x$, then $h_j^i : (U_j^i, T)$.

Then, $E \vdash C_d$.

Proof:

We type-check the desugared code. We apply rule EXP MATCH-SPLIT

\[
\begin{array}{c}
E \vdash \phi \\
E \vdash M : T \\
\forall 0 < i \leq n, E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : U \\
\forall 0 < i \leq n, E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i \vdash \text{match } M \text{ with } h_1(x_1^1, \ldots, x_{m_1}^1) \to A_1 \ldots h_n(x_1^n, \ldots, x_{m_n}^n) \to A_n : U
\end{array}
\]

And we are left with the following proof obligations:

1. $E \vdash \phi$
2. $E \vdash M : T$
3. $\forall 0 < i \leq n, h_i : (H_i, T)$
4. $\forall 0 < i \leq n, H_i = x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i, \{M = h_i(x_1^i, \ldots, x_{m_i}^i)\} \vdash A_i : U$
5. $\forall 0 < i \leq n, E, x_1^i : T_1^i, \ldots, x_{m_i}^i : T_{m_i}^i \vdash \text{match } M \text{ with } h_1(x_1^1, \ldots, x_{m_1}^1) \to A_1 \ldots h_n(x_1^n, \ldots, x_{m_n}^n) \to A_n : U$

We now treat the cases of $\phi$ individually.

$A_i = \mathcal{M}(x_1^i, e_1^i) \ldots \mathcal{M}(x_{m_i}^i, e_{m_i}^i) A'_i \mathcal{C}(e_{m_i}^i) \ldots \mathcal{C}(e_1^i)$

In the following, we let $E' := E, x_1^1 : T_1^1, \ldots, x_{m_1}^1 : T_{m_1}^1, \{M = h_i(x_1^1, \ldots, x_{m_1}^1)\}$.

When showing that the code $A_i$ type-checks, we can analyze the $\mathcal{M}(x_j^i, e_j^i)$ individually since they are independent of each other (cf. note after Definition 25).

Let $E_1 := E', E''$ for some $E''$ such that $E_1 \vdash \phi$. We show that if $E_1, B_d(x_j^i, e_j^i) \vdash B : V$ for some expression $B$ and some type $V$, then $E_1 \vdash \mathcal{M}(x_j^i, e_j^i) B : V$. We distinguish among the three different $e_j^i$.

$e_j^i = x_j^i$: In this case, $B_d(x_j^i, e) = \epsilon$ and $\mathcal{M}(x_j^i, e_j^i) = \epsilon = \mathcal{C}(e_j^i)$. The result is immediate because the hypothesis and the conclusion are equal ($\epsilon$ changes neither the expression nor the typing environment).

$e_j^i = (y_1, \ldots, y_e)$: In this case, $B_d(x_j^i, e) = y_1 : U_1, \ldots, y_e : U_e, \{(y_1, \ldots, y_e) = x_j^i\}$ and $\mathcal{M}(x_j^i, e_j^i) = \text{let } (y_1, \ldots, y_e) = x_j^i$. We apply rule EXP SPLIT$^\epsilon$

\[
\begin{array}{l}
E_1 \vdash \phi \\
E_1, y_1 : U_1, \ldots, y_e : U_e, \{(y_1, \ldots, y_e) = x_j^i\} \vdash B : V \\
E_1 \vdash x_j^i : y_1 : U_1 \ast \cdots \ast y_{e-1} : U_{e-1} \ast U_e \vdash (y_1, \ldots, y_e) \cap \text{fv}(V) = \emptyset
\end{array}
\]

$E_1 \vdash \text{let } (y_1, \ldots, y_e) = x_j^i \text{ in } B : V$

- $E_1 \vdash \phi$ is an hypothesis.
- $E_1, y_1 : U_1, \ldots, y_e : U_e, \{(y_1, \ldots, y_e) = x_j^i\} \vdash B : V$ is an hypothesis.
- $E_1 \vdash x_j^i : y_1 : U_1 \ast \cdots \ast y_{e-1} : U_{e-1} \ast U_e$ is an hypothesis of the lemma.

$e_j^i = h' \ y$: In this case, gives us $h' : (U, T_j^i)$, $B_d(x_j^i, e_j^i) = y : U, \{h' \ y = x_j^i\}$, $\mathcal{M}(x_j^i, e_j^i) = \text{match } x_j^i \text{ with } h' \ y \to$, and $\mathcal{C}(e_j^i) = \text{else fail}()$. We apply rule EXP MATCH

\[
\begin{array}{l}
E_1 \vdash x_j^i : T_j^i \\
h' : (U, T_j^i) \quad E_1, y : U, \{h' \ y = x_j^i\} \vdash B : V \\
E_1, \forall z, h' \ z \neq x_j^i \vdash \text{fail}() : V
\end{array}
\]

$E_1 \vdash \text{match } x_j^i \text{ with } h' \ y \to B : V \text{ else fail}()$
E₁ ⊨ x_j : T_j;

Since E₁ ⊨ ⊥ and E₁ = E', E'' ⊢ x_j : T_j, we apply VAL VAR to prove this claim.

\[
\text{VAL VAR}
\]

\[
\begin{array}{c}
E₁ ⊨ ⊥ \\
(x_j : T_j) ∈ E₁
\end{array}
\]

\[
E₁ ⊨ x_j : T_j
\]

h' : (U, T_j):

Follows from

E₁, y : U, \{h' y \equiv x_j\} ⊨ B : V:

This is an hypothesis.

E₁, \{∀ z. h' z \neq x_j\} ⊨ \text{fail()} () : V:

Since E₁ is basic, E₁ ⊨ \text{fail(α)} : unit → α. Rule EXP APPL combined with VAL UNIT yields that fail() () : α and, in particular fail() () : V.

\[
\text{EXP APPL}
\]

\[
\begin{array}{c}
E₁ ⊨ \text{fail(α)} : \text{unit} → α \\
E₁ ⊨ () : \text{unit}
\end{array}
\]

\[
E₁ ⊨ \text{fail()} () : \text{unit}
\]

We prove that Aₙ by an inductive argument. For all length mₙ of elements, our induction is on the current elements k in the matched tuple. More precisely, using induction, we show that if E', B_d(x_i^1, e_i^1), ..., B_d(x_i^m_i, e_i^m_i) ⊨ A_i : T_i, then E' ⊨ M(x_i^1, e_i^1), ..., M(x_i^m_i, e_i^m_i) A_i C(e_i^m_i), ..., C(E_i^1) : T_i.

Our induction hypothesis is if E', B_d(x_i^1, e_i^1), ..., B_d(x_i^m_i, e_i^m_i) ⊨ M(x_i^k+1, e_i^k+1), ..., M(x_k^m, e_k^m) A_i C(e_k^m), ..., C(E_k^1) : T_i, then E' ⊨ M(x_i^1, e_i^1), ..., M(x_k^m_i, e_k^m_i) A_i C(e_k^m), ..., C(E_k^1) : T_i.

- Our base case is k = 0. This case is trivial because the premise of our induction hypothesis is equal to its conclusion.
- For our induction step, we assume the induction hypothesis for k and show that this implies the hypothesis for k + 1.

The induction hypothesis for k states

E', B_d(x_i^1, e_i^1), ..., B_d(x_i^k, e_i^k) ⊨ M(x_i^{k+1}, e_i^{k+1}), ..., M(x_i^m_i, e_i^m_i) A_i C(e_i^m_i), ..., C(E_i^1) : T_i, then

E' ⊨ M(x_i^1, e_i^1), ..., M(x_i^m_i, e_i^m_i) A_i C(e_i^m_i), ..., C(E_i^1) : T_i.

We apply the result from above stating that if E', E'', B_d(x_i, e_i) ⊨ B : V for some expression B, some type V, and some typing environment E'' such that E', E'' ⊨ ⊥, then E', E'' ⊨ M(x_i, e_i) B C(e_i) : V.

We set

E'' := B_d(x_i^1, e_i^1), ..., B_d(x_i^k, e_i^k),

V := T_i, and

B := M(x_i^{k+1}, e_i^{k+1}) M(x_i^{k+2}, e_i^{k+2}) ... M(x_i^m_i, e_i^m_i) A_i C(e_i^m_i), ..., C(E_i^1) C(e_i^k+2) C(e_i^{k+1})

Combining the induction hypothesis for k with this result yields the induction hypothesis for k + 1. This concludes this inductive argument.

\[\square\]

**Lemma 13** (mkPubld well-typed). *Let E be a basic environment. Then, E ⊨ mkPubld : sigkey → \{y : \text{verkey} \mid \exists z. x = (z, y)\}.*

**Proof:** Let E be basic. The following derivation is supported by the summary depicted in [Table IX] It contains the code together with the applied rules and the typing environment extension. Let C_x denote the code starting at line x.

We start by using rule VAL FUN. It remains to be shown that E₁ := E, x : sigkey ⊨ C_2 : \{y : \text{verkey} \mid \exists z. x = (z, y)\}.

Since E₁ ⊨ x : (y' : unit * T^k_v \\{ \text{verkey}/α \} → unit) * \text{verkey} and x, y \notin fnfv(\{y : \text{verkey} \mid \exists z. x = (z, y)\}), we can use rule EXP SPLIT. We are obliged to prove that E₂ := E₁, z : (y' : unit) * T^k_v \\{ \text{verkey}/α \} → unit, y : \text{verkey}, \{(z, y) : x\} : y : \{y : \text{verkey} \mid \exists z. x = (z, y)\}.

An application of VAL REFINERY leaves us with proving that (1) E₂ ⊨ y : \text{verkey} and (2) E₂ ⊨ (\exists z. x = (z, y)) \{y/y\}.

(1) is derivable by an application of VAL VAR since (y : \text{verkey}) \in E₂ and E₂ ⊨ ⊥ by the fact that E is basic by assumption and all extensions to E only introduce new variables. In order to prove (2) we have to prove using rules for well-formedness of typing environments that E₂ ⊨ ⊥, which holds by the same explanation as above, that \text{fnfv(\exists z. x = (z, y))} ⫋ \text{dom(E₂)}, which
Table IX: The code and typing environment extension for the type derivation of mkPubld.

also holds since both \(x\) and \(y\) are contained in \(E_2\), and that \(\text{forms}(E_2) \vdash \exists z. x = (z, y)\). Since \(\text{forms}(E_2) = \{(z, y) = x\}\) this holds trivially by existential instantiation of the underlying authorization logic.

Before we state the next lemma we stipulate that the type \(\text{predicate}^F\) is the same as the type \(T_{\text{vk}}^o\{\text{verkey}/\alpha\}\).

**Lemma 14** (\(\text{mkSay}_E\) well-typed). Let \(E\) be a basic environment. Then, \(E \vdash \text{mkSay}_E : \text{sigkey} \rightarrow \{y : T_{\text{vk}}^o\{\text{verkey}/\alpha\}\mid \exists w, z, x = (w, z) \land \mathcal{F} = \text{Say}(z, y)\} \rightarrow \text{proof}\).

**Proof:** Let \(E\) be a basic environment. The following derivation is supported by the summarized derivation depicted in **Table X** including the applied rules together with the environment extension starting from \(E\). We denote by \(C_x\) the code starting at line \(x\).

Table X: The code and typing environment extension for the derivation of \(\text{mkSay}_E\).

We type-check \(C_1\) by applying \text{VAL FUN} twice and are to show that \(E_1 := E, x : \text{sigkey}, \{y : T_{\text{vk}}^o\{\text{verkey}/\alpha\}\mid \exists w, z, x = (w, z) \land \mathcal{F} = \text{Say}(z, y)\} \vdash C_2 : \text{proof}\).

Since \(x\) is a dependent pair and \(w, z \notin \text{fnfv}(\text{proof})\), we can use \text{EXP SPLIT} to type-check \(C_2\) and are left with showing that \(E_2 := E_1, w : (y' : \text{unit}) \ast T_{\text{vk}}^o\{\text{verkey}/\alpha\} \rightarrow \text{unit}, z : \text{verkey}, \{(w, z) = x\} \vdash C_3 : \text{proof}\).

We use \text{EXP MATCH} to type-check \(C_3\) and are left with showing that \(E_3 := E_2, y = P^F(\text{Revealed } y_1, \ldots, \text{Revealed } y_n), y_1 : T_1, \ldots, y_n : T_n \vdash C_4 : \text{proof}\).

We apply \text{EXP LET} in combination with \text{VAL INL INR FOLD} and \text{VAL REFINE}. It remains to be shown that \(E_4 := E_3, y'' : T_{\text{vk}}^o\{\text{verkey}/\alpha\} \vdash C_5 : \text{proof}\).

Type-checking \(C_5\) requires the use of \text{EXP LET} in combination with \text{EXP APPL}: \(w\) is a function of type \((y' : \text{unit}) \ast T_{\text{vk}}^o\{\text{verkey}/\alpha\} \rightarrow \text{unit}\) which we can derive from \(E_4\) using \text{VAL VAR} and the fact that \(E_4 \vdash o\). The explanation why this is the case is again the same as in the proof of **Lemma 13**. We pass the pair \((z, y'')\) as argument to \(w\). This pair is compatible with the argument type of \(w: z\) of type \(\text{verkey}\) (note that \(\text{verkey} <: \text{unit}\)), \(y''\) is of type \(T_{\text{vk}}^o\{\text{verkey}/\alpha\}\). Hence the
The application of these rules leaves us with the obligation to show that the premise of \( E \) is fulfilled and we can deduce that \( E_4 \vdash w(z, y) : \text{unit} \). As \( \text{sig} \notin \text{fv(proof)} \) we can apply \( \text{EXP LET} \). This leaves us with the obligation to prove that \( E_5 := E_4, \text{sig} : \text{unit} \vdash C_6 : \text{proof} \).

We apply \( \text{EXP LET} \) in combination with \( \text{EXP APPL} \) \( 2n + 4 \) times in a row. This is possible because \( E_5 \vdash \text{rand} : \text{unit} \rightarrow \text{unit} \), \( E_3 \vdash \text{commit} : \text{(unit} \times \text{unit}) \rightarrow \text{unit} \) and the arguments to both functions are proven of the correct type under \( E_5 \). The application of these rules leaves us with the obligation to show that \( E_5 := E_5, \text{r}_{\text{sig}} : \text{unit}, c_{\text{sig}} : \text{unit}, r_z : \text{unit}, c_z : \text{unit}, r_1 : \text{unit}, c_1 : \text{unit}, \ldots, r_n : \text{unit}, c_n : \text{unit} \vdash C_{10} : \text{proof} \).

We apply \( \text{VAL INL INR FOLD} \): Says \( p \) is a constructor for the type \( \text{proof} \), \( \text{proof} \) is derivable from \( E_9 \) since \( \text{proof} \) is the same as \( \text{unit} \) and \( \text{unit} \) is always derivable, and all arguments have the correct type, each derivable by \( \text{VAL VAR} \).

The functions \( \text{mkREL} \), \( \text{mkLM} \), \( \text{mkLNM} \), \( \text{mkIDRev} \), and \( \text{mk}_v \) are typable using the opponent typability \( [8] \). An opponent is a closed expression that does not contain any occurrence of assert. Furthermore, every occurring type is either \( \text{unit} \) or a subtype thereof, e.g., \( \text{verkey} \), formula, and \( \text{proof} \). The lemma states that in this case, the expression is typable with \( \text{unit} \).

It remains to type-check the functions \( \text{verify}_p \), \( \text{verify} \) and \( \text{mkSSP} \). For verify we have to consider only the case where the proof is an \( \text{SSP}_p \) since for all other cases the opponent typability applies. \( \text{SSP}_p \) is different because it involves the type \( \text{sigkey} \) which is not a subtype of \( \text{unit} \).

**Lemma 15 (verify well-typed).** Let \( E \) be a basic environment. Then, \( E \vdash \text{verify} : \text{unit} \rightarrow \text{unit} \rightarrow \text{bool} \).

**Proof:** Let \( E \) be basic. We have to prove the lemma only for the case where \( p \) matches the constructor \( \text{SSP}_p \) (starting at \( v_1 \)). All other cases are provable by the opponent typability \( [8] \).

The type derivation is supported by the summary in [Table XI]. We show the code together with the applied rule for each line and the typing environment extension. Again, we denote by \( C_x \) the code starting at line \( x \).

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_0 )</td>
<td>( \text{verify } (p : \text{unit})(f : \text{unit}) : \text{bool} = )</td>
</tr>
<tr>
<td>( v_1 )</td>
<td>( \text{match } f \text{ with} )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>( \text{let } (y, r_y) = \text{openCommit}(c_y); )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>( \text{let } (s, r_s) = \text{openCommit}(c_s); )</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>( \text{let } (x, r_x) = \text{openCommit}(c_x); )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>( \text{let } (x, r_x) = \text{openCommit}(c_x); )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>( \text{let } (y, r_y) = \text{openCommit}(c_y); )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>( \text{let } (s, r_s) = \text{openCommit}(c_s); )</td>
</tr>
<tr>
<td>( v_8 )</td>
<td>( \text{let } (x, r_x) = \text{openCommit}(c_x); )</td>
</tr>
<tr>
<td>( v_9 )</td>
<td>( \text{let } (w, r_w) = \text{openCommit}(c_w); )</td>
</tr>
<tr>
<td>( v_{10} )</td>
<td>( \text{let } b_y = \text{checkEq } y' \ y; )</td>
</tr>
<tr>
<td>( v_{11} )</td>
<td>( \text{let } b_s = \text{checkEq } s' \ s; )</td>
</tr>
<tr>
<td>( v_{12} )</td>
<td>( \text{let } b_{psd} = \text{checkEq } \text{psd}' \ \text{psd}; )</td>
</tr>
<tr>
<td>( v_{13} )</td>
<td>( \text{if } b_y = \text{true then} )</td>
</tr>
<tr>
<td>( v_{14} )</td>
<td>( \text{if } b_s = \text{true then} )</td>
</tr>
<tr>
<td>( v_{15} )</td>
<td>( \text{if } b_{psd} = \text{true then} )</td>
</tr>
<tr>
<td>( v_{16} )</td>
<td>( \text{if } w = y )</td>
</tr>
<tr>
<td>( v_{17} )</td>
<td>( \text{true} )</td>
</tr>
<tr>
<td>( v_{18} )</td>
<td>( \text{else} )</td>
</tr>
<tr>
<td>( v_{19} )</td>
<td>( \text{false} )</td>
</tr>
<tr>
<td>( v_{20} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( v_{21} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( v_{22} )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( v_{23} )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

Table XI: The code and typing environment extension for the type derivation of \( \text{verify} \).
It suffices to show that \( E_1 := E, p : \text{unit}, f : \text{unit} \vdash C_{v_1} : \text{bool} \). We apply the rule EXP MATCH twice and are to show that 
\[
E_4 := E_1, c_y : \text{unit}, c_s : \text{unit}, c_{psd} : \text{unit}, c_x : \text{unit}, \{p = \text{SSP}_p((c_y,\_),(c_s,\_),(c_{psd},\_),c_x)), y' : \text{unit}, s' : \text{unit}, psd' : \text{unit}, \{y = \text{SSP}(y',s',psd')\} \vdash C_{v_4} : \text{bool}.
\]

We apply rule EXP SPLIT in combination with rule EXP APPL four times in a row. This is possible since \( E_4 \vdash \text{openCommit} : \text{unit} \rightarrow (\text{unit} \times \text{unit}) \), \( E_4 \vdash \text{openCommit}_{sk} : \text{unit} \rightarrow (\text{sigkey} \times \text{unit}) \), the arguments \( c_y, c_s, c_{psd}, \) and \( c_x \) have type \( \text{unit} \), and all new variables are fresh in \( E_4 \). We are to show that \( E_8 := E_4,y : \text{unit}, r_y : \text{unit}, \{y(r_y) = \text{openCommit}(c_y)\} \), \( s : \text{unit}, r_s : \text{unit}, \{(s,r_s) = \text{openCommit}(c_s)\} \), \( psd : \text{unit}, r_{psd} : \text{unit}, \{(psd,r_{psd}) = \text{openCommit}(c_{psd})\} \), \( x : \text{sigkey}, r_x : \text{unit}, \{(x,r_x) = \text{openCommit}(c_x)\} \) \( C_{v_8} : \text{bool} \).

We apply rule EXP SPLIT since \( x \) is a pair type and are left with showing that \( E_9 := E_8, E_3, w : \text{verkey}, \{(<,w) = x\} \vdash C_{v_9} : \text{bool} \).

We apply rule EXP LET in combination with rule EXP APPL four times in a row. This is possible since \( E_9 \vdash \text{hash} : \alpha \rightarrow \text{unit}, E_9 \vdash \text{checkEq} : \text{unit} \rightarrow \text{unit} \rightarrow \text{bool} \), and the arguments to the functions have the correct type provable by \( E_9 \). It remains to be shown that \( E_{13} := E_9, psd'' : \text{unit}, b_y : \text{bool}, b_s : \text{bool}, b_{psd} : \text{bool} \vdash C_{v_{13}} : \text{bool} \).

We use Lemma 7 and are to show that \( E_{18} := E_{13}, \{b_y = \text{true}\}, \{b_s = \text{true}\}, \{b_{psd} = \text{true}\}, \{psd'' = psd\}, \{w = y\} \vdash true : \text{bool} \). The else cases are all outputting false which is of type \text{bool}.

This goal is trivially fulfilled by \( E_{18} \).

The problem with the type \text{sigkey} also occurs in the creation of the SSP, i.e., in the function mkSSP. Hence, we devote the next lemma to prove its typability.

Lemma 16 (mkSSP well-typed). Let \( E \) be a basic environment. Then, \( E \vdash \text{mkSSP} : \text{sigkey} \rightarrow \text{unit} \rightarrow \text{proof} \).

Proof: Let \( E \) be basic. The following type derivation is supported by the summary depicted in Table XII; the summary contains the code, as well as the used rules with the typing environment extensions. We denote by \( C_x \) the code starting at line \( x \).

| \begin{tabular}{l}
1 \text{mkSSP} (x : \text{sigkey}) (s : \text{bitstring}) : \text{proof} = \\
2 \quad \text{let } y = \text{mkPubId} x; \\
3 \quad \text{let } psd = \text{hash}(x,s); \\
4 \quad \text{let } r_y = \text{rand}(); \text{let } c_y = \text{commit}(y,r_y); \\
5 \quad \text{let } r_s = \text{rand}(); \text{let } c_s = \text{commit}(s,r_s); \\
6 \quad \text{let } r_{psd} = \text{rand}(); \text{let } c_{psd} = \text{commit}(psd,r_{psd}); \\
7 \quad \text{let } r_x = \text{rand}(); \text{let } c_x = \text{commit}(x,r_x); \\
8 \quad \text{SSP}_p ((c_y,(y,r_y)),(c_s,(s,r_s)),(c_{psd},(psd,r_{psd})),c_x)) \\
\end{tabular} | \begin{tabular}{l}
\text{VAL FUN} \\
\text{E}_1 := E, x : \text{sigkey}, s : \text{bitstring} \\
\text{EXP LET, EXP APPL} \\
\text{E}_2 := E_1, \{y : \text{verkey} \mid \exists z. x = (z,y)\} \\
\text{E}_3 := E_2, psd : \text{unit} \\
\text{E}_4 := E_3, r_y : \text{unit}, c_y : \text{unit} \\
\text{E}_5 := E_4, r_s : \text{unit}, c_s : \text{unit} \\
\text{E}_6 := E_5, r_{psd} : \text{unit}, c_{psd} : \text{unit} \\
\text{E}_7 := E_6, r_x : \text{unit}, c_x : \text{unit} \\
\end{tabular} |

Table XII: The code and typing environment extension for the type derivation of \text{mkSSP}.

Using rule VAL FUN twice it remains to be shown that \( E_1 := E, x : \text{sigkey}, s : \text{unit} \vdash C_2 : \text{proof} \).

We use rule EXP LET in combination with rule EXP APPL and are to show that \( E_2 := E_1, \{y : \text{verkey} \mid \exists z. x = (z,y)\} \vdash C_3 : \text{proof} \). We can use these two rules since \( x : \text{sigkey} \) and \text{mkPubId} expects an argument of that type.

We use nine time rule EXP LET in combination with rule EXP APPL. We can do so since \( E_2 \vdash \text{hash} : \alpha \rightarrow \text{unit}, \) its argument \((x,s)\) has the correct type under \( E_2 \), namely, any type is possible, \( E_2 \vdash \text{rand} : \text{unit} \rightarrow \text{unit} \) is always applied with an argument of type \text{unit}, namely, \( () \), \( E_2 \vdash \text{commit} : (\text{unit} \times \text{unit}) \rightarrow \text{unit} \) is also always applied to arguments of the correct type, and \( E_2 \vdash \text{commit}_{sk} : (\text{sigkey} \times \text{unit}) \rightarrow \text{unit} \) and its argument \((x,r_x)\) is \text{sigkey} \times \text{unit}. Hence, we are to show that \( E_7 := E_2, psd : \text{unit}, r_y : \text{unit}, c_y : \text{unit}, r_s : \text{unit}, c_s : \text{unit}, r_{psd} : \text{unit}, c_{psd} : \text{unit}, r_x : \text{unit}, c_x : \text{unit} \vdash C_8 : \text{proof} \).

Since every value used in \( C_8 \) is either of type \text{unit} or a subtype thereof (e.g., \( y : \text{verkey} <\) unit) and SSP_ is a constructor for the type \text{proof}, \( E_7 \vdash C_8 : \text{proof} \).

Before we can type-check verify_E we introduce two new notions which we formalize below: firstly, the judgment \( E \vdash A : T \leadsto E' \) intuitively means that if \( E \) is the typing environment used to type check the process \( A \) then it is extended to \( E' \) when it reaches the end. Secondly, we define an invariant of our mappings \( \phi \) and \( \psi \) together with a formula \( F \), and a typing environment \( E \).

Typing environment after type-checking. Our proof for the verification function will proceed step by step, as hinted by our construction of the code of the verification function. In such a proof, we naturally will need to be able to grasp the typing environment that will be used to type-check the next code macro and prove properties about this typing environment. Typically, type-system do not allow to talk about the typing environment since they make the type-system more complex
without aiding their main goal (i.e., type-checking a program). Therefore, we extend the existing RCF type system to allow us to grasp and concretely prove properties about the typing environment at a particular point in the type-checking phase. Also, we add a notation to be able to type-check contexts. We stress that these modifications are merely notation and do not require reproving of type-system properties, in particular, because the final code will not use the new rule.

Our modification to the type system are depicted in Table XIII. The idea is that the type-checking process returns the typing environment that was returned by extended judgments. Besides the extended rules, we also need a way to be able to type-check contexts. Therefore, we add the following rule that will be used to type-check the code inserted into the context. In the proof of the final theorem, all other rules are return in typing environment they started with since there is no continuation, i.e., they are extended to the type-system:

Table XIII: Excerpt of the modified rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exp Let</strong></td>
<td>( E \vdash A : T \rightarrow E' ) ( x \notin fv(U) ) ( E, x : T \vdash B : U \rightarrow E' )</td>
<td>( E \vdash \text{let } x = A; B : U \rightarrow E' )</td>
</tr>
<tr>
<td><strong>Exp Split</strong></td>
<td>( E, x : T, y : U, {(x, y) = M} \vdash A : V \rightarrow E' )</td>
<td>( E \vdash \text{let } (x, y) = M; A : V \rightarrow E' )</td>
</tr>
<tr>
<td><strong>Exp Match</strong></td>
<td>( E \vdash M : T \rightarrow E'' ) ( h : (H, T) ) ( E, x : H, {h x = M} \vdash A : U \rightarrow E' )</td>
<td>( E \vdash \text{match } M \text{ with } h x \implies A \text{ else } B : U \rightarrow E' )</td>
</tr>
<tr>
<td><strong>Exp Appl</strong></td>
<td>( E \vdash M : (\Pi x : T. U) \rightarrow E' ) ( E \vdash N : T \rightarrow E'' )</td>
<td>( E \vdash M , N : U{N/x} \rightarrow E' )</td>
</tr>
<tr>
<td><strong>Val Refine</strong></td>
<td>( E \vdash M : T \rightarrow E' ) ( E \vdash C{M/x} )</td>
<td>( E \vdash M : x : T \vdash C \rightarrow E' )</td>
</tr>
</tbody>
</table>

Definition 27 (Straight context). We call a context straight if and only if \( C \) is of either of the following forms:

- \( \bullet \);  
- \( C = \text{match } M \text{ with } C' \text{ else } B \) where \( C' \) is a straight context;  
- \( C = \text{let } x = A; C' \) where \( C' \) is a straight context;  
- \( C = \text{let } (x, y) = A; C' \) where \( C' \) is a straight context.

Lemma 17 (Straight context environment extension). Let \( C \) be a straight context and let \( E \) be a typing environment such that \( E \vdash C : \{x : \text{bool} \mid x = \text{true} \implies F\} \rightarrow E' \) for some formula \( F \), and let

\[
\text{Exp Empty}^F\quad E'' \vdash \bullet : \{x : \text{bool} \mid x = \text{true} \implies F\} \rightarrow E''
\]

be the only application of rule \( \text{Exp Empty}^F \). Then \( E'' = E'' \).

Proof: Let \( C, E, \) and \( E' \) be as in the lemma. We show that the whole lies always in the branch whose typing environment is returned by the extended judgments.
The proof is by induction on the structure of straight processes. The base case is trivial since the only possible rule to apply is EXP EMPTY. For the induction, there are only three cases to consider. A glimpse at the extended rules in Table XIII shows that the environment passed down is always in the position where the straight process $C'$ resides (cf. Definition 27) and we can apply the induction hypothesis to obtain the desired result.

**Lemma 18** (Transitivity straight process). Let $C_1$ and $C_2$ be two straight processes. Then $C_1[C_2]$ is a straight process.

**Proof:** The proof is by induction on the length of the straight process $C_1$.

**Lemma 19** (Straight verification code). Let $F$ be a formula and let $C = \llbracket F \rrbracket$. Then $C$ is a straight process.

**Proof:** The claim follows by inspection of the code macros and Lemma 18. In our proof, we will also need to reason about the logical maps $\psi$ and $\phi$ and relate them to the code that was type-checked.

**Definition 28** (Mapping extension). Let $F_e$ be an elementary formula, let $C = \llbracket F_e \rrbracket$, and let $\psi$ and $\phi$ be the logical mappings as defined above. We call $\phi'$ and $\psi'$ extended by $C$ if $\phi'$ is $\phi$ with the modification described in $U^n_0$ to $U^n_1$ in $C$, and $\psi'$ is $\psi$ with the modification described in $U^n_0$ to $U^n_1$ in $C$. We write $\text{ext}(\psi, \phi, C, \phi', \psi')$.

Let $\phi$ and $\psi$ be the empty mappings (i.e., $\phi(i) = \bot$ and $\psi(i) = \emptyset$ for all $i$), and let $C = \llbracket F^\land \rrbracket$ for some formula $F^\land$. Let $\phi'$ and $\psi'$ be such that $\text{ext}(\psi, \phi, C, \phi', \psi')$. We call $\phi'$ and $\psi'$ the mappings induced by $C$ and we write $\phi(C) := \phi'$ and $\psi(C) := \psi'$.

Often, we will write $\phi(\bullet)$ and $\psi(\bullet)$ to denote empty mappings.

Here and throughout the rest of the paper, we use the notation $T^{F^\land}$ to denote $T^{F^\land} := \{ x : \text{bool} \mid x = \text{true} \implies F^\land \}$. The following lemma relates the code induced by $\psi$ and its logical consequence when the code is type-checked.

**Lemma 20** ($\psi=\land$). Let $\psi$ be a logical mapping as described above, $C = \llbracket F^\land \rrbracket$, and $E$ be a typing environment. Then (i) $C$ is a straight context and (ii) if $E, (\psi)=\land A : T$ for some expression $A$ and some type $T$, then $E \vdash C[A] : T$.

**Proof:** The part (i) follows by inspecting the code generated for as defined in Definition 12 (if-statements are syntactic sugar for match-statements) as the whole $\bullet$ is always in the then branch (i.e., the with branch). Part (ii) follows from Lemma 7.

**Well-typedness of** $\text{verify}_{F^\land}$.

Finally, we have amassed all the machinery to prove that our verification method is well-typed. We now state the intuition of appropriateness of typing environments, our central notion in the proof. Then we give a quick road map how the proofs will proceed and then we will prove the well-typedness of the verification function.

The central notion of our well-typedness proof is the appropriateness appropriate of typing environments and the logical maps with respect to a formula $F^\land$. We establish this notion as an invariance throughout all of our following proofs. Intuitively, if appropriate($E, E', \psi, \phi, F^\land$) holds, then $E$ type-checks the verification code for formula $F^\land$. The typing environment $E'$ is the resulting typing environment that is already strong enough to proof the formula $F^\land$. The maps $\psi$ and $\phi$ are needed to establish the invariance if $F^\land$ is extended.

The proofs will proceed as follows.

(i) First, we show that if, for a given formula $F^\land$, typing environments, and mappings, appropriateness holds, then we also establish appropriateness for the extended formula $F^\land \land F_e$. More precisely, we show that appropriateness is an invariant of the verification code.

(ii) Second, we show that starting from a basic typing environment and the verification code for a formula, we can establish the appropriateness after we finished type-checking the code. Formally, this is a proof by induction. The base case corresponds to the empty formula and the induction step corresponds to the proof from step (i).

(iii) Third, we show that the verification function is well-typed. Formally, we conclude that the code for formulas $F^\land$ containing no disjunctions is well-typed and, leveraging this result, show that the code macro for disjunctions is also well-typed.

We start by giving the definition of appropriateness.

**Definition 29** (Appropriate). Let $E$ and $E'$ be typing environments, $\psi$ and $\phi$ be the mappings as described above, and $F^\land$ a formula. Then, the predicate appropriate($E, E', \psi, \phi, F^\land$).
holds if and only if, all of the following conditions are fulfilled: let the context $C = [F^\land]$ and let $(x_1, \ldots, x_n) = \text{Vars}(F^\land)$.

1. $E \vdash C : \{x : \text{bool} \mid x = \text{true} \implies F \} \leadsto E' \text{ and } E' = E''$;
2. $E$ and $E'$ are basic;
3. $\forall i \in I_{\text{app}}(F^\land), E' \vdash \phi(i) : \text{verkey}$;
4. $\forall i, y, (x_i = \text{Revealed } y \implies y \in \psi(E(i))) \land (x_i = \text{Hidden } y \implies \exists x'. x' \in \psi(E(i)))$;
5. $E'', (\psi', \phi', F^\land \land \mathcal{F}) \vdash F^\land$.

We now prove that appropriateness is invariant under type-checking verification macros.

**Lemma 21** (Appropriate invariance). Let $\mathcal{F}$ be an elementary formula, $E$ and $E'$ be typing environments, $\psi$ and $\phi$ be mappings, and $F^\land$ be a formula such that appropriate$(E, E', \psi, \phi, F^\land)$. Furthermore, let $C = [F^\land], C_\mathcal{F} = [\mathcal{F}], \phi'$ and $\psi'$ be mappings, where $\text{ext}(\psi, \phi, C, \psi', \phi')$. Then there exists an $E''$ such that $E' \vdash C_\mathcal{F} : T^{F^\land \land \mathcal{F}} \leadsto E''$ and appropriate$(E, E'', \psi', \phi', F^\land \land \mathcal{F})$ holds.

**Proof:** Let $\mathcal{F}, E, E', \psi, F^\land, \phi', C$, and $C_\mathcal{F}$ be as stated in Lemma 21 Furthermore, let appropriate$(E, E', \psi, \phi, F^\land)$ hold. The proof proceeds by case distinction.

The proof $p$ in every macro is proven to be of type unit by the typing environment $E$, since the input $p$ into the verification function is of type unit and the conjunction and disjunction both yield two sub-proofs of type unit. A similar argument justifies that all formulas $\mathcal{F}$ are of type formula.

In the following, we let $C_t$ denote the code beginning with the $i$-th line of the respective code macro. In the code, we consider the $E$ mapping for $F^\land \land \mathcal{F}$, as Proposition 2 guarantees that $E$ remains unchanged in $F^\land$ when appending $\mathcal{F}$.

We proceed as follows: we distinguish cases for different kinds of elementary formulas $\mathcal{F}$. We prove in every step the different subcases for Definition 29. However, we prove the final step in the end, since the proof is not dependent on the formula to be proven. More precisely, we know by assumption that $E' \vdash F^\land$ and we show for every subcase that the extended environment after the derivation, say $E''$, entails $\mathcal{F}_e$, i.e., $E'' \vdash \mathcal{F}_e$. However, the final step that $E'' \vdash F^\land \land \mathcal{F}_e$ is postponed.

$\mathcal{F}_e = \text{ says}$

We apply EXP MATCH twice in a row and need to prove the statement $E_1 := E', \{p = \text{says}_p(\ldots)\}, c_{\text{sig}} : \text{unit}, c_2 : \text{unit}, c_{\text{arg}} : \text{unit}, \ldots, c_{\text{arg}_n} : \text{unit}, \{f = \text{says}(\ldots)\}, z' : \text{verkey}, \{\text{arg}_1 : T_1, \ldots, \text{arg}_n : T_n + C_{s_5} : T^{F^\land \land \mathcal{F}}\}$.

As $E_1$ proves all the arguments to openCommit of type unit, EXP APPL yields that all applications of the arguments to openCommit are of type unit*unit. Thus, applying EXP SPLIT together with EXP APPL (n+2) times leaves us with the obligation to show $E_5 := E_1, \{\text{sig} : \text{unit}, r_{\text{sig}} : \text{unit}, z : \text{unit}, \ldots, \text{arg}_n : \text{unit}, r_{\text{arg}_n} : \text{unit} + C_{s_5} : T^{F^\land \land \mathcal{F}}\}$.

In order to type-check the lines $s_5$ and $s_6$ (where $s_5$ need not to be existing and $s_6$ may actually be $j$ lines for $0 \leq j \leq n$), we use EXP LET in combination with EXP MATCH. We only show how to type check one of the lines, since the other ones are analogously checkable. Both branches of the match are either typable with unit or with verkey depending on the type of the (the Hidden branch always type-checks, since $E_5 \vdash \text{fail}(\alpha) : \text{unit} \rightarrow \alpha$). Note that $\text{arg}_{\text{sig}}' : \text{verkey}$. After applying EXP LET for all these lines, it suffices to show that $E_6 := E_5, z'^o : \text{verkey}, \{\text{arg}_{\text{sig}} : T_1, \ldots, \text{arg}_{\text{arg}_n} : T_n + C_{s_5} : T^{F^\land \land \mathcal{F}}\}$ where $T_1 \in \{\text{verkey}, \text{unit}\}$.

Notice that the environment $E_7$ not necessarily contains $z'^o$ and all $\text{arg}_{\text{arg}}'$, only those for which the corresponding variable from the formula was not constructed using Hidden.

Since $F^\land \land \mathcal{F}$ is well-formed and $\text{arg}_{\text{arg}}' : \text{verkey}$, either $\text{arg}_{\text{arg}}' \beta$ is revealed (i.e., it is constructed with Revealed and carries a verification key $x$) or it is hidden (i.e., it is constructed with Hidden). In the former case, we set $\phi$ at position $E(\omega)$ to $x$ and return $x$, hence, $z'' = x$. In the latter, we know by Definition 29 (Appropriate) (3) that $\phi(E(\omega))$ returns a value $v$ that is equal to $z$ and $v : \text{verkey}$ (the equality is tracked in $\psi$ and checked at the end). It follows that by applying EXP LET and EXP MATCH it remains to be shown that $E_7 := E_6, z'^o : \text{verkey} + C_{s_7} : T^{F^\land \land \mathcal{F}}$.

We apply Lemma 7 and it remains to be shown that $E_8 := E_7, \{z'' = \text{arg}_{\text{arg}}'\} + C_{s_8} : T^{F^\land \land \mathcal{F}}$.

We apply EXP APPL on $\text{check}_{\text{sig}}$ with $z''$ and $\text{sig}$ as arguments. Using EXP LET, we obtain that $m : T_{s_8} \{\text{verkey}/\alpha, z''/\gamma\}$. In this type, $z''$ is the principal identifier in front of the says modality, i.e., the says modality has the form $z'' \text{ says } \ldots$. We are left to show that $E_9 := E_8, m : T_{s_8} \{\text{verkey}/\alpha, z''/\gamma\} + C_{s_9} : T^{F^\land \land \mathcal{F}}$.

We use EXP MATCH to proceed and we are left with showing that $E_{10} := E_9, \{m = P(z^o) \ldots\}, y'_1 : T_1, \ldots, y'_n : T_n + C_{s_{\text{arg}}}: T^{F^\land \land \mathcal{F}}$, where $T_i \in \{\text{verkey}, \text{unit}\}$ (c.f. Table IV).

Lemma 7 is applicable to the cascade of if-statements in line $s_{10}$ to line $s_{11}$, and we are left to show that $E_{12} := E_{10}, \{\text{arg}_1 = y'_1\}, \ldots, \{\text{arg}_n = y'_n\} : \text{unit} : T^{F^\land \land \mathcal{F}}$. This holds by rule EXP EMPTY$^{F^\land \land \mathcal{F}}$ and we receive $E_{12}$ as extended environment back.

We now argue that appropriate$(E, E_{12}, \psi, \phi, F^\land \land \mathcal{F})$ holds.
We apply [Lemma 19] and [Lemma 17] and using EXP EMPTY $\alpha$ and $\eta$, we conclude that $E \vdash C[C_\xi] : T^{\alpha \land \eta \land \xi} \leadsto E_{12}$.

This shows Definition 29 (Appropriate) (2). Note that $C[C_\xi] = [[F \land \xi]]$.

At this point, we mention that $E_{12} \vdash o$ because $E' \vdash o$ (cf. [Definition 23]) and all names that are newly bound are fresh. The type $T^\alpha_{\xi k}$ is closed by the verification key $z''$ used in line $s_8$. It follows that $E_{12}$ is basic, and, hence, Definition 29 (Appropriate) (6) holds.

Verification keys in $F_\xi$ are carried by $arg_0$ at position $\omega$ and potentially by some $y^i$ at positions $\omega + i$. First we treat $arg_0$ since it is more difficult. We have to distinguish two cases, namely, $E(\omega < \omega = \Delta (F')^\omega$ and $E(\omega) \geq \omega$. In the former case, we get by Definition 29 (Appropriate) (3) that $E' \vdash \phi(E(\omega)) : \text{verkey}$ and, hence, $E_{12} \vdash \phi(E(\omega)) : \text{verkey}$. In the latter case, we know that there is no variable $z''$ in $C$ such that $arg_0 = z''$. In particular, it follows that $arg_0$ cannot be constructed with Hidden, since otherwise, $F \land \xi$ would not be well-formed. In that case, $\phi$ is updated such that $\phi(E(\omega)) = x$ where $arg_0 = \text{Revealed} x$ and $E_{12} \vdash \phi(E(\omega)) : \text{verkey}$.

For all $i$, we update $\phi(E(\omega + i))$ to $y^i$ and we know that $E_{12} \vdash \phi(E(\omega + i)) : T_i$, where $T_i \in \{\text{verkey}, \text{unit}\}$, depending on the formula; we conclude that Definition 29 (Appropriate) (8) holds.

Let $(\nu_1, \ldots, \nu_i) = \text{Vars}(F')$. By assumption, we know that Definition 29 (Appropriate) (9) holds for $F'$, i.e., $\psi(E(i)) = x$ for every variable $y$ at position $i$ where $y = \text{Revealed} x$, or $\psi(E(i)) = x'$ for some $x'$ where $y$ is constructed using Hidden. It suffices to show that this also holds for $f$. Considering line $U_{\nu_i}^n$, we observe that $\psi$ is set at exactly those positions where a variable $arg^i$ ($0 \leq i \leq n$) is considered using the constructor $\psi$ (hence set to $arg^i$), hence the first part of the conjunction of Definition 29 (Appropriate) (4) holds. As we add a variable into $\psi(E(j))$ for each index $j$ in lines $U_{\nu_j}^n$ to $U_{\nu_i}^n$ independent of whether the variable is hidden or not, the second part of the for Definition 29 (Appropriate) (4) holds. Hence, it holds for $F' \land \xi$.

For Definition 29 (Appropriate) (5), we know that $E', (\psi)^\omega \vdash F'$. Furthermore, $E_{12} \vdash m : T^\alpha_{\xi_k}\{\nu_1, \ldots, \nu_i\}$ (cf. line $s_8$), $E_{12} \vdash m = P^n(y_1, \ldots, y_n)$ (cf. line $s_9$), and $E_{12} \vdash arg_j = y_j$ (cf. lines $s_{10}$ to $s_{12}$). Consequently, $E_{12} \vdash z''$ says $P(arg_1, \ldots, arg_n)$.

Next, we show that $E_{12}, (\psi)^\omega \vdash F_\xi$.

We add all committed values $arg_0, \ldots, arg_n$ of the proof into their respective position in $\psi$ in lines $U_{\nu_i}^n$ to $U_{\nu_i}^n$. This establishes the connection between the proofs for $F$ and $f$; the corresponding logical formulas are contained in $(\psi)^\omega$ (cf. [Lemma 20]). Now, for each $i$ such that $arg^i = \text{Revealed} x_i$, for some $x_i$, we can now potentially substitute $x_0$ for $z''$ and $x_i$ for $arg^i$, since $(\psi)^\omega$ proves them equal. For the hidden variables, we use existential instantiation from the underlying authorization logic. Note that if $arg^i = \text{Hidden} k$ and $arg^j = \text{Hidden} k$ for different $i$ and $j$, we instantiate the same $k$. This establishes equality of hidden variables. Hence, $E_{12}, (\psi)^\omega \vdash F_\xi$.

$F_\xi = \text{SSP}$

We apply EXP MATCH twice in a row and need to prove the statement $E_1 := E', (p = \text{SSP}_p(\ldots)), c_z := \text{unit}, c_x := \text{unit}, c_{psd} := \text{unit}, c_{ps}_z := \text{unit}, \{f = \text{SSP}(z', s', psd')\}, z' := \text{verkey}, s' := \text{unit}, psd := \text{unit} \vdash C_{ps} : T^{F' \land \xi}$.

Since $E_1 \vdash \text{openCommit}_{\xi_k} : \text{unit} \rightarrow (\text{sigkey} * \text{unit})$ and $E_1 \vdash c_x : \text{unit}$, we apply EXP APPL and EXP SPLIT; we are left with showing that $E_2 := E_1, x : \text{sigkey}, r_x := \text{unit} \vdash C_{ps} : T^{F' \land \xi}$.

Since $E_2 \vdash \text{openCommit} : \text{unit} \rightarrow (\text{unit} * \text{unit})$ and $E_2$ proves the arguments of the correct type (unit), we apply EXP APPL and EXP SPLIT three times in a row; we are left with showing that $E_5 := E_2, z : \text{unit}, r_z := \text{unit}, s : \text{unit}, r_s := \text{unit}, psd := \text{unit}, r_{psd} := \text{unit} \vdash C_{ps} : T^{F' \land \xi}$.

Since $E_5 \vdash x : \text{sigkey}$ and $\text{sigkey}$ is a dependent pair type, we apply EXP SPLIT and we are obliged to show that $E_6 := E_5, w : \text{verkey} \vdash C_{ps} : T^{F' \land \xi}$.

Because of the same reasoning as in the case for $s$ says, we use EXP LET in combination with EXP MATCH three times in a row; it remains to be shown that $E_9 := E_6, z_0 : \text{verkey}, s_0 := \text{unit}, psd_0 := \text{unit} \vdash C_{ps} : T^{F' \land \xi}$.

An application of EXP LET in combination with EXP APPL (since $E_9 \vdash \text{computePsd} : \text{sigkey} * \text{unit} \rightarrow \text{unit}$, $E_9 \vdash x : \text{sigkey}$, and $E_9 \vdash s : \text{unit}$) leaves us with proving that $E_{10} := E_9, psd_0 := \text{unit} \vdash C_{ps} : T^{F' \land \xi}$.

We proceed by using [Lemma 7] and we left with the obligation to show that $E_{12} := E_{10}, \{psd_0 \equiv psd\}, \{w = \phi(E(\omega))\} \vdash C_{ps} : T^{F' \land \xi}$.

We use EXP LET in combination with EXP ASSUME and it remains to show that $E_{13} := E_{12}, \{\text{asmtn} : \text{unit} \mid \text{SSP}(z, s, psd)\} \cup \cdot \vdash T^{F' \land \xi}$.

An application of EXP EMPTY $\alpha$ and $\eta$ finishes the derivation and we obtain $E_{13}$ as extended environment.

We now argue that appropriate $(E, E_{13}, \psi, \phi, F' \land \xi)$ holds.

We apply [Lemma 19] and [Lemma 17] and using EXP EMPTY $\alpha$ and $\eta$, we conclude that $E \vdash C[C_\xi] : T^{F' \land \xi} \leadsto E_{13}$.

This shows Definition 29 (Appropriate) (1). Note that $C[C_\xi] = [[F \land \xi]]$.
fresh. It follows that $E_{13}$ is basic, and, hence, Definition 29 (Appropriate) (2) holds.
A verification key in $F$ is carried by $z'$ at position $\omega$. Again, we distinguish cases: either, $\mathcal{E}(\omega) < \omega = \Delta(\mathcal{F}^\omega)$, or $\mathcal{E}(\omega) \geq \omega$. In the first case, we know by assumption that $E' \vdash \phi(\mathcal{E}(\omega)) : \mathit{verkey}$, and hence, $E_{13} \vdash \phi(\mathcal{E}(\omega)) : \mathit{verkey}$. The second case cannot happen since otherwise $F' \land F$ would not have been well-formed, a contradiction to the assumption. Hence, Definition 29 (Appropriate) (3) holds.

The lines $U^\omega_1$ add an element to $\psi(\mathcal{E}(\omega + i))$ only if the corresponding variable in the formula, i.e., $z'$, $s'$, or $psd'$ are constructed using Revealed. This proves the left part of the conjunction for Definition 29 (Appropriate) (4). Again, the lines $U^\omega_1$ add an element into $\psi(\mathcal{E}(\omega + i))$ independent of whether the variable at that position is constructed using Hidden or not, or also the right part of the conjunction holds. Hence, Definition 29 (Appropriate) (4) holds.

By assumption, we know that $E'(\psi)^e \vdash F'$. Furthermore, we know that $E_{13} \vdash \mathit{SSP}(z, s, psd)$ because $\{\mathit{asmtn} : \mathit{unit} \mid \mathit{SSP}(z, s, psd)\} \in E_{13}$. Next, we show that $E_{13}, (\psi)^e \vdash F_e$.

We added the committed values contained in $F_e$, i.e., $z$, $s$, and $psd$, into $\psi$ in lines $U^\psi_1$. This establishes the connection between $F'$ and $F_e$ if both share any variable. The formulas of $(\psi)^e$ prove these connections (cf. Lemma 20). As we also add all $x$ such that $x' = \mathit{Revealed}$, $x' = \mathit{Revealed}$, or $psd' = \mathit{Revealed}$, $x$ is sent in $U^\psi_1$. We can substitute these $x$ at those positions since $(\psi)^e$ contains the corresponding formulas proving the equality between the committed value and the variable carried by Revealed. For instance, if $s' = \mathit{Revealed}$, we know that $s^o$ is set, and we can substitute $s^o$ for $s$ in SSP($z$, $s$, $psd$), resulting in SSP($z$, $s'$, $psd$). For hidden values, we use existential instantiation from the underlying authorization logic. Note that equality between hidden variables is tracked via $\psi$, hence, equality is still provided. Hence, $E_{13}, (\psi)^e \vdash F_e$.

$$F_e = \mathit{REL}$$

We apply EXP MATCH twice in a row. It remains to be shown that $E_1 := E', \{p = \mathit{REL}_p(.)\}, c_e : \mathit{unit}, op : \mathit{unit}, c_y : \mathit{unit}, \{f = \mathit{REL}(x', op', y')\}, x' : \mathit{unit}, op' : \mathit{unit}, y' : \mathit{unit} \vdash C_{r_1} : T^{F' / \land F_e}$.

We use EXP SPLIT in combination with EXP APPL twice; we can apply EXP APPL since $E_1 \vdash \mathit{openCommit} : \mathit{unit} \rightarrow (\mathit{unit} \ast \mathit{unit})$, $E_1 \vdash c_{x} : \mathit{unit}$, and $E_1 \vdash c_{y} : \mathit{unit}$. We are left with the obligation to show that $E_3 := E_1, x : \mathit{unit}, r_{x} : \mathit{unit}, y : \mathit{unit}, r_{y} : \mathit{unit} \vdash C_{r_{2}} : T^{F' / \land F_e}$.

We apply EXP LET and twice EXP APPL since $E_4 \vdash \mathit{getOperation} : \mathit{unit} \rightarrow (\mathit{unit} \rightarrow \mathit{unit} \rightarrow \mathit{bool})$ and the arguments have the correct type. It remains to be proven that $E_4 := E_3, b : \mathit{bool} \vdash C_{r_{3}} : T^{F' / \land F_e}$.

Using the same reasoning as in the former cases, after type-checking $r_4$ and $r_5$ using EXP LET and EXP MATCH, we are obliged to show that $E_5 := E_4, x^o : \mathit{unit}, y^o : \mathit{unit} \vdash C_{r_5} : T^{F' / \land F_e}$.

We proceed by using Lemma 7. We are left with showing that $E_8 := E_6, \{b = \mathit{true}\}, \{op = op'\} \vdash C_{r_6} : T^{F' / \land F_e}$. Using EXP LET in combination with EXP ASSUME, we are to show that $E_9 := E_8, \{\mathit{asmtn} : \mathit{unit} \mid \mathit{REL}(x, op, y)\} \vdash \mathit{C} : T^{F' / \land F_e}$.

A final application of EXP EMPTY$^{F' / \land F_e}$ returns the extended environment $E_9$.

We now argue that appropriate($E, E_9, \psi, \phi, F_e$) holds.

We apply Lemma 19 and Lemma 17 and using EXP EMPTY$^{F' / \land F_e}$, we conclude that $E \vdash C[C_{F_e}] : T^{F' / \land F_e} \rightarrow E_9$. This shows Definition 29 (Appropriate) (1). Note that $C[C_{F_e}] = [\mathcal{F}_e \land \mathcal{F}_e]$. At this point, we mention that $E_9 \vdash \mathit{e}$ because $E' \vdash \mathit{e}$ (cf. Definition 23) and all names that are newly bound are fresh. It follows that $E_9$ is basic, and, hence, Definition 29 (Appropriate) (2) holds.

Since $F_e$ does not involve any variable of type $\mathit{verkey}$, Definition 29 (Appropriate) (3) extends immediately to $F' \land F_e$.

By assumption, Definition 29 (Appropriate) (4) holds for $F'$. As the lines $U^\psi_1$ only add a variable to $\psi(\mathcal{E}(\omega + i))$ if the variable at position $\omega + i$ is constructed using Revealed, the left part of the conjunction holds. As the lines $U^\psi_1$ add a variable to $\psi(\mathcal{E}(\omega + i))$ no matter if the variable at position $\omega + i$ is constructed using Hidden or not, or also the right part of the conjunction also holds. Consequently, Definition 29 (Appropriate) (4) holds for $F' \land F_e$.

By assumption, we know that $E'(\psi)^e \vdash F'$. Furthermore, we know that $E_9 \vdash \mathit{REL}(x, op, y)$ since $\{\mathit{asmtn} : \mathit{unit} \mid \mathit{REL}(x, op, y)\} \in E_9$. Next, we show that $E_9, (\psi)^e \vdash F_e$.

We added the committed values that are contained in $F_e$, i.e., $x$ and $y$, into $\psi$ in lines $U^\psi_1$, which establishes the connection between $F'$ and $F_e$, proven by the formulas in $(\psi)^e$ (cf. Lemma 20). We also added $x^o$ and $y^o$ if the corresponding variables i.e., $x'$, $y'$ were constructed using Revealed in lines $U^\psi_1$. Since, if they are added, they are proven equal by the formulas of $(\psi)^e$, we can substitute $x^o$ for $x$ and $y^o$ for $y$ if they are existing. If either or both of $x^o$ and $y^o$ were not existing, we use existential instantiation from the underlying authorization logic and, consequently, $E_9, (\psi)^e \vdash F_e$.

$$F_e = \mathcal{LM}$$
We apply EXP MATCH twice in a row and are left with showing that \( E_1 := E', \{ p = LNM_p(\ldots) \}, c_x : \text{unit}, \ell : \text{list}, \{ f = \text{LM}(x', b', \ell') \}, x' : \text{unit}, b' : \text{unit}, \ell' : \text{list} \vdash C_{m_3} : T^{F^\land \land \land} \).

Since \( E_1 \vdash \text{openCommit} : \text{unit} \rightarrow (\text{unit} \ast \text{unit}) \) and \( c_x \) and \( c_b \) have the correct types, we apply EXP SPLIT in combination with EXP APPL twice and are to show that \( E_2 := E_1, x : \text{unit}, r_x : \text{unit}, b : \text{unit}, r_b : \text{unit} \vdash C_{m_3} : T^{F^\land \land \land} \).

Because of the same reasoning as in the former cases, we are to show, after applying EXP LET and EXP MATCH twice, that \( E_3 := E_2, x' : \text{unit}, b' : \text{unit} \vdash C_{m_3} : T^{F^\land \land \land} \).

Using EXP LET and EXP APPL, we are left with the obligation to show that \( E_4 := E_3, r : \{ y : \text{bool} \mid y = \text{true} \Leftrightarrow (x, b) \in \ell \} \vdash C_{m_3} : T^{F^\land \land \land} \).

We apply Lemma 7 and it remains to be shown that \( E_5 := E_4, \{ r = \text{true} \} \vdash \cdot : T^{F^\land \land \land} \).

We finish the derivation by applying EXP EMPTY \( F^\land \land \land \) and we receive the extended typing environment \( E_7 \).

We now argue that \text{appropriate}(E, E_7, \psi, \phi, F^\land \land \land \land J) \) holds.

We apply Lemma 19 and Lemma 17 and using EXP EMPTY \( F^\land \land \land \land \psi \), we conclude that \( E \vdash C[C_E] : T^{F^\land \land \land \land} \). This shows Definition 29 (Appropriate) (1). Note that \( C[C_E] = \mathcal{F}[\land \land \land J] \).

At this point, we mention that \( E_2 \vdash \circ \) because \( E' \vdash \circ \) (cf. Definition 23) and all names that are newly bound are fresh. It follows that \( E_7 \) is basic, and, hence, Definition 29 (Appropriate) (2) holds.

Since \( \mathcal{F} \) does not involve any variable of type \( \text{verkey} \), Definition 29 (Appropriate) (3) extends immediately to \( \mathcal{F}^\land \land \land \land \mathcal{F} \).

By assumption, Definition 29 (Appropriate) (4) holds for \( \mathcal{F}^\land \land \land \land \mathcal{F} \). As the lines \( U_1 \) for \( i \in \{ 0, 1 \} \) only add a variable to \( \psi(\mathcal{E}(\omega + i)) \) if the variable at position \( \omega + i \) is constructed using Revealed and \( U_2 \) adds \( \ell' \) (since \( \ell' \) is not supposed to be hideable), the left part of the conjunction holds. As the lines \( U_2 \) add a variable to \( \psi(\mathcal{E}(\omega + i)) \) no matter if the variable at position \( \omega + i \) is constructed using Hidden or not, the right part of the conjunction also holds. Consequently, Definition 29 (Appropriate) (4) holds for \( \mathcal{F}^\land \land \land \land \mathcal{F} \).

By assumption, we know that \( E', (\psi)' = \mathcal{F}^\land \land \land \land \mathcal{F} \). Furthermore, we know that \( E_7 \vdash (x, b) \in \ell \) since \( r : \{ y : \text{bool} \mid y = \text{true} \Leftrightarrow (x, b) \in \ell \} \in E_7, \{ r = \text{true} \} \in E_7 \), and we can substitute true for \( y \), obtaining \( \{ \text{true} = \text{true} \Leftrightarrow (x, b) \in \ell \} \). Next, we show that \( E_7, (\psi)' \vdash \mathcal{F}^\land \land \land \mathcal{F} \).

We added committed values into \( \psi \) at their respective index in lines \( U_1 \), which establishes the connection between \( \mathcal{F}^\land \land \land \land \mathcal{F} \), proven by the formulas in \( (\psi)' \) (cf. Lemma 20). As we also added \( \ell' \) into \( \psi \) in line \( U_2 \) at the same index where we added \( \ell \), and \( (\psi)' \) proves them equal, we can substitute \( \ell \) by \( \ell' \) and obtain that \( E_7, (\psi)' \vdash (x, b) \in \ell' \). Furthermore, we added the variables carried by Revealed if either or both of \( x' \) and \( b' \) were constructed with Revealed in lines \( U_1 \) where \( i \in \{ 0, 1 \} \). In this case, we can also substitute the respective carried variables for \( x \) and \( b \), namely, \( x'' \) and \( b'' \), respectively, because they are proven equal by \( (\psi)' \). If either or both of \( x' \) and \( b' \) are constructed using Hidden, we apply existential instantiation from the underlying authorization logic. Consequently, \( E_7, (\psi)' \vdash \mathcal{F}^\land \land \land \mathcal{F} \).

We now argue that \text{appropriate}(E, E_5, \psi, \phi, F^\land \land \land \land J) \) holds.

We apply Lemma 19 and Lemma 17 and using EXP EMPTY \( F^\land \land \land \land \psi \), we conclude that \( E \vdash C[C_E] : T^{F^\land \land \land \land} \). This shows Definition 29 (Appropriate) (1). Note that \( C[C_E] = \mathcal{F}[\land \land \land J] \).

At this point, we mention that \( E_5 \vdash \circ \) because \( E' \vdash \circ \) (cf. Definition 23) and all names that are newly bound are fresh. It follows that \( E_5 \) is basic, and, hence, Definition 29 (Appropriate) (2) holds.

Since \( \mathcal{F} \) does not involve any variable of type \( \text{verkey} \), Definition 29 (Appropriate) (3) extends immediately to \( \mathcal{F}^\land \land \land \land \mathcal{F} \).

By assumption, Definition 29 (Appropriate) (4) holds for \( \mathcal{F}^\land \land \land \land \mathcal{F} \). As the line \( U_0 \) only adds a variable to \( \psi(\mathcal{E}(\omega)) \) if
By assumption, we know that \( E', (\psi) = \top \wedge \Phi \). Furthermore, we know that \( E_0 \vdash \neg \exists a. \ (x, a) \in \ell \) since \( b : \{ y : \text{bool} \mid y = \text{true} \Leftrightarrow \exists a. \ (x, a) \in \ell \} \in E_0 \), \( \{ b = \text{false} \} \in E_0 \), and we can substitute false for \( y \), obtaining \( \{ \text{false} = \text{true} \Leftrightarrow \exists a. \ (x, a) \in \ell \} \). Next, we show that \( E_5, (\psi) = \top \). We added committed values into \( \psi \), which establishes the connection between \( \Phi \) and \( \Phi \), proven by the formulas in \( (\psi)^\text{m} \) (cf. \( \text{Lemma 20} \)). As we also added \( \ell' \) into \( \psi \) in line \( U_0^\psi \) at the same index where we added \( \ell' \), and \( (\psi)^m \) proves them equal, we can substitute \( \ell \) by \( \ell' \) and obtain that \( E_5, (\psi) = \top \Leftrightarrow \neg \exists a. \ (x, a) \in \ell' \). Note that this statement is equivalent to the statement \( E_5 \vdash (x, a) \notin \ell \). Furthermore, we added the variable carried by \( \text{Revealed} \) if \( x' \) was constructed with \( \text{Revealed} \) in line \( U_0^\psi \). In this case, we can also substitute the respective carried variable for \( x \), namely, \( x' \), because they are proven equal by \( (\psi)^m \). If some of \( x', R', s', \) and \( i'dr' \) were constructed using \( \text{Hidden} \), we apply existential instantiation from the underlying authorization logic. Consequently, \( E_5, (\psi) = \top \).
Let \( E'' \) be the extended environment for the different cases that we obtain after applying the rule \( \text{Exp} \text{ Empty}^{\mathcal{F}^\wedge \wedge \mathcal{F}_e} \).
Finally, we know by assumption that \( E', (\psi)^\wedge \vdash \mathcal{F}^\wedge \) and by the reasoning above that \( E''', (\psi)^\wedge \vdash \mathcal{F}_e \). First of all, \( E'' \) is basic, in particular, \( E'' \vdash \mathcal{F}^\wedge \). Hence, also \( E'' \vdash \mathcal{F}^\wedge \). In order to prove the final step that \( E''', (\psi)^\wedge \vdash \mathcal{F}^\wedge \wedge \mathcal{F}_e \) we use Definition 29 (Appropriate) (4) and Lemma 20: together, they state that variables that are equal in \( \mathcal{F}^\wedge \wedge \mathcal{F}_e \) are proven equal by \( (\psi)^\wedge \). More precisely, whenever two variables in a formula are equal, the function \( \mathcal{E} \) maps the respective indices to the same number. In particular, this is also true for hidden variables; this equality is proven by the second part of Definition 29 (Appropriate) (4) and the variables added in \( U_0^e \) to \( U_0^e \). By that, \((\psi)^\wedge \) establishes the logical connection between hidden and revealed variables in \( \mathcal{F}^\wedge \) and \( \mathcal{F}_e \). We conclude that \( E''', (\psi)^\wedge \vdash \mathcal{F}^\wedge \wedge \mathcal{F}_e \), hence, Definition 29 (Appropriate) (5) holds for \( \mathcal{F}^\wedge \wedge \mathcal{F}_e \) for all possible forms of \( \mathcal{F}_e \).

Lemma 22 (Appropriateness of code). Let \( E \) be basic, \( \mathcal{F}^\wedge \) be a formula without disjunctions, \( C = [\mathcal{F}^\wedge] \). Then there exists an \( E' \) such that \( E \vdash C : \mathcal{T}^{\mathcal{F}^\wedge} \Rightarrow E' \) and appropriate\( (E, E', \psi(C), \phi(C), \mathcal{F}^\wedge) \) holds.

Proof: Let \( E \) be as specified above. The proof is by structural induction on \( \mathcal{F}^\wedge \) and we distinguish between two cases: \( \mathcal{F}^\wedge = \text{true} \) (this case is needed for technical reasons, actual formulas will always be at least an elementary formula) and \( \mathcal{F}^\wedge = \mathcal{F}^\wedge' \wedge \mathcal{F}_e \).

\( \mathcal{F}^\wedge = \text{true} \):
In this case, \([\mathcal{F}^\wedge] = \bullet \). We have to show that appropriate\( (E, E', \psi(\bullet), \phi(\bullet), \text{true}) \) holds for some \( E' \). First of all, by applying rule \( \text{Exp} \text{ Empty}^\text{true} \), we know that \( E \vdash \bullet : \text{true} \Rightarrow E \), hence, \( E' \vdash E \). Thus, Definition 29 (Appropriate) (1) holds. As \( E' = E \), \( E' \) is basic as well, hence, also Definition 29 (Appropriate) (2) holds. Since \( \mathcal{F}^\wedge \) does not contain variables, \( I_{ek} = 0 \) and, hence, Definition 29 (Appropriate) (3) and Definition 29 (Appropriate) (2) hold. For the same reason, \((\psi)^\wedge \) is the empty environment and \( E' \vdash \text{true} \), consequently, Definition 29 (Appropriate) (5) holds, which concludes the base case.

\( \mathcal{F}^\wedge = \mathcal{F}^\wedge' \wedge \mathcal{F}_e \):
Let \( C' = [\mathcal{F}^\wedge'] \) and \( C_{\mathcal{F}_e} = [\mathcal{F}_e] \). By the induction hypothesis, we know that appropriate\( (E, E', \psi(C'), \phi(C'), \mathcal{F}^\wedge') \) holds for some \( E' \). Furthermore, \( \text{ext}(\psi(C'), \phi(C'), C_{\mathcal{F}_e}, \psi(C), \phi(C)) \) holds by construction (cf. Definition 28). We use Lemma 21 to deduce that there exists an \( E'' \) such that \( E \vdash C : \mathcal{T}^{\mathcal{F}^\wedge} \Rightarrow E'' \) and appropriate\( (E, E'', \psi(C), \phi(C), \mathcal{F}^\wedge) \) holds. This concludes the proof.

Theorem 12 (Well-typedness of \( \text{verify}^{\mathcal{F}^\wedge} \)). Let \( E \) be basic and \( \mathcal{F}^\wedge \) be a well-formed formula in disjunctive normal form. Then \( E \vdash \text{verify}^{\mathcal{F}^\wedge} : (p : \text{proof}) \rightarrow \{ y : \text{formula} \mid y = \mathcal{F}^\wedge \} \rightarrow \{ x : \text{bool} \mid x = \text{true} \Rightarrow \mathcal{F}^\wedge \} \).

Proof: Let \( E \) be a basic environment and \( \mathcal{F}^\wedge \) be a well-formed formula. Since \( \mathcal{F}^\wedge \) is well-formed, we know that \( \mathcal{F}^\wedge \) is in disjunctive normal form. Let \( C = [\mathcal{F}^\wedge] \).

The definition of \( \text{verify}^{\mathcal{F}^\wedge} \) is syntactic sugar. It spells out as follows:

\[
\text{let verify}^{\mathcal{F}^\wedge} = \text{fun } p \rightarrow \text{fun } y \rightarrow \ldots : \Pi(p : \text{proof}). \Pi(y : \{ x : \text{formula} \mid x = \mathcal{F}^\wedge \}). \{ x : \text{bool} \mid x = \text{true} \Rightarrow \mathcal{F}^\wedge \}).
\]

We show that the code type-checks with the desired type.

Since \( \text{verify}^{\mathcal{F}^\wedge} \) is a recursive function, we insert \( \text{verify}^{\mathcal{F}^\wedge} \) into the typing environment while type-checking the code of the function. From a technical point of view, we follow the approach by Gunter [76] as suggested by Bengtson et al. [8]. Applying rule \text{Val} \text{ Fun} twice leaves us with the obligation to show that \( E_1 \vdash E, \text{verify}^{\mathcal{F}^\wedge} : \Pi(p : \text{proof}). \Pi(y : \{ x : \text{formula} \mid x = \mathcal{F}^\wedge \}). \{ x : \text{bool} \mid x = \text{true} \Rightarrow \mathcal{F}^\wedge \}), p : \text{proof}, y : \{ x : \text{formula} \mid x = \mathcal{F}^\wedge \} \vdash C[\text{true}] : \{ x : \text{bool} \mid x = \text{true} \Rightarrow \mathcal{F}^\wedge \}.

We now distinguish between a formula \( \mathcal{F}^\wedge \) without disjunctions and a formula \( \mathcal{F}^\wedge \) with a disjunction.

\( F := \mathcal{F}^\wedge \) does not contain disjunctions:
At this point, we note that appropriate\( (E_1, E_1, \psi(\bullet), \phi(\bullet), \text{true}) \) holds using the reasoning from Lemma 22 case \( \mathcal{F}^\wedge = \text{true} \).
We apply Lemma 22 we obtain that there is a typing environment \( E' \) such that appropriate\( (E_1, E', C, \psi(C), \phi(C), F) \). Using Definition 29 (Appropriate) (5) together with Lemma 20 we obtain that \( E_1 \vdash C[\text{true}] : \mathcal{T}^F \Rightarrow E'' \) and \( E'' \vdash F \) (note that \((\psi)^\wedge \) is already contained in \( E'' \)). In the derivation tree,
we now replace the application of rule EXP EMPTY$^\land$ with VAL REFIN as shown below:

$$\text{EXP EMPTY}^{\land} \quad \frac{}{E'' \vdash \bullet : \{x : \text{bool} \mid x = \text{true} \implies F\} \rightsquigarrow E'''}$$

$$\text{VAL REFIN} \quad \frac{E'' \vdash \text{true} : \text{bool} \quad E'' \vdash \text{true} = \text{true} \implies F}{E'' \vdash \text{true} : \{x : \text{bool} \mid x = \text{true} \implies F\} \rightsquigarrow E'''''}$$

Since we know $E'' \vdash F^{\land}$ and $E'' \vdash \text{true} : \text{bool}$, we can successfully type-check the code $E_1 \vdash C[, \text{true}] : \{x : \text{bool} \mid x = \text{true} \implies F\}$, proving our claim.

$F := F'^{\lor}$ contains disjunctions:
Let us recall the code $C := [F]$ if $F = F_1 \lor F_2$ is a disjunction:

1. match $p$ with $\text{Or}_p(p_1, p_2) \implies$
2. match $f$ with $\text{Or}(f_1, f_2) \implies$
3. let $t_1 = \text{verify}_{E_2} p_1 f_1$
4. if $t_1 = \text{true}$ then
5. $t_1$
6. else
7. $\text{verify}_{E_2} p_2 f_2$
8. $1 \implies \text{false}$
9. $1 \implies \text{false}$

We type-check this code under $E_1$. In the following, we use $C_i$ do denote the code beginning at line $i$ and we let $T^F := \{x : \text{bool} \mid x = \text{true} \implies F\}$. We use rule EXP MATCH twice and are left to show that $E_2 := E_1, \{p = \text{Or}_p(p_1, p_2)\}, \{f = \text{Or}(f_1, f_2)\} \vdash C_3 : T^F$. Since $E_1$ and consequently $E_2$ maps $\text{verify}_{E}$ to the desired type, we can use EXP APPL twice followed by EXP LET and are left with the obligation to show that $E_3 := E_2, t_1 : T^{F_1} \vdash C_4$. Here, we note that if either $F_1$ or $F_2$ hold, then $F$ holds since $F = F_1 \lor F_2$.

We apply EXP IF and are left to show the two obligations $E_4 := E_3, \{t_1 = \text{true}\} \vdash t_1 : T^F$ and $E'_4 := E_3 \vdash C_7 : T^F$.

In the former case, we are finished immediately since $t_1 : T^{F_1}$ and $T^{F_1} <: T^F$ since $F_1 \implies F$.

In the latter case, we apply rule EXP APPL twice and obtain that the result of the application is of type $T^{F_2}$. As above, we know that $T^{F_2} <: T^F$ since $F_2 \implies F$. This concludes this case.

This concludes the proof of $E \vdash \text{verify}_{E'} : (p : \text{proof}) \to \{y : \text{formula} \mid y = F'^{\lor}\} \to \{x : \text{bool} \mid x = \text{true} \implies F'^{\lor}\}$. ■

E. ML Implementation of the API

Intuitively, the wrapper function for the verification should look as follows:

$$\text{verify}^F : (p : \text{proof}) \to \{y : \text{formula} \mid y = F'^{\lor}\} \to \{x : \text{bool} \mid x = \text{true} \implies F'^{\lor}\}.$$  

It calls the refined verification function for formula $F$ and stores the return value in $r$. If $r$ is true, then we place the assertion for the input formula $F$, otherwise, we return $r$. A complete implementation, however, has to take into account the ML encoding of formulas. We state the full implementation below in Appendix C.

Our ultimate goal is to create an API that is secure by construction. In particular, programmers should be able to use this API in their programs without having to take further steps to verify the security such as type-checking their program with the RCF type system. In the previous section, we have successfully verified the well-typedness of the API methods. The type-checked RCF API methods, however, still contains refinement types. As a consequence, every program employing this API must be type-checked with the RCF type system to ensure that it respects the implications of the refined types. In this section, we show how we can circumvent the RCF type-checking of programs that deploy the API.
We accomplish this goal by encapsulating private keys and by wrapping all API methods that take as input values of a non-unit type. The resulting API uses only the RCF type unit and is guaranteed to type-check when used in an ML program. We detail the encapsulation and the wrapping, then we argue that ML programs that use the API do not need to be type-checked with the RCF type system.

Encapsulating private keys. Instead of giving programmers immediate access to private signing keys, we encapsulate the signing key and only allow access via a handle. In particular, the API cannot be used to extract the signing key. Intuitively, this handle is similar to a reference: it is not the signing key itself but only refers to the signing key. This encapsulation has several advantages: first and foremost, it is impossible for programmers to accidentally leak the signing key. Secondly, from a type-checking point of view, it removes the non-public non-tainted type sigkey and replaces it with the type unit.

Of course, using techniques such as reflection, it is still possible to obtain access to the signing key directly, i.e., we only protect against accidental leakage. This, however, is not surprising since the signing key is stored locally on the user’s computer and can always be extracted.

Wrapping of API methods with non-unit types.

that characterizes values without any security-relevant import. Since programs written in ML can be typed by replacing all ML types with the RCF type unit and the API also only introduces the type unit, the resulting program is well-typed by the opponent typability lemma (cf. Lemma 34 [8]).

In a nutshell, the idea is the following: we expose to programmers only functions that use the RCF type unit. Internally, these functions will call the refined API methods and conduct the necessary steps to ensure that the refined API methods will receive as input values with the correct type (for this endeavor, the role of the PKI function is paramount). The core insight is that programs that only use the RCF type unit will always pass the RCF type system. Since the API methods only expose the RCF type unit and all types occurring in an ML program correspond to the RCF type unit, we do not need to type-check the program with the RCF type system.

Regarding assumptions and assertions, we have proven that if a zero-knowledge proof verification succeeds and returns true, then the passed formula (in encoded form) holds. Thus, there is no need for a programmer to explicitly state the assertion since it will always succeed by construction; the assumptions have already been internalized in the RCF API.

The result is an API that yields programs that are secure by construction. In particular, a programmer can neither accidentally nor intentionally expose secret key material.

---

Table XIV: ML API methods and auxiliary functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand</td>
<td>( \text{unit} \rightarrow \text{random} )</td>
</tr>
<tr>
<td>mkPubId</td>
<td>( \text{uid} \rightarrow \text{uid}_{\text{pub}} )</td>
</tr>
<tr>
<td>mkSays</td>
<td>( \text{uid} \rightarrow \text{predicate}^{F} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mkSSP</td>
<td>( \text{uid} \rightarrow \text{string} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mkREL</td>
<td>( \text{formula} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mkLM</td>
<td>( \text{pseudo} \rightarrow \text{bitstring} \rightarrow \text{list} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mkLNM</td>
<td>( \text{pseudo} \rightarrow \text{list} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mkIDRev</td>
<td>( \text{proof} \rightarrow \text{string} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mk_{\land}</td>
<td>( (\text{proof} \ast \text{proof}) \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>mk_{\lor}</td>
<td>( \text{proof} \rightarrow \text{formula} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>verify</td>
<td>( \text{proof} \rightarrow \text{formula} \rightarrow \text{bool} )</td>
</tr>
<tr>
<td>hide</td>
<td>( \text{proof} \rightarrow \text{formula} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>rerand</td>
<td>( \text{proof} \rightarrow \text{formula} \rightarrow \text{proof} )</td>
</tr>
<tr>
<td>PKI</td>
<td>( \text{uid}_{\text{pub}} \rightarrow \text{PID} )</td>
</tr>
</tbody>
</table>

---

3 Although unit-typed values do not add any security-relevant import to the type-checking process, values of type unit are still important for the cryptographic implementation. For instance, the randomness used in the zero-knowledge proof (cf. Section C) is of type unit.