The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption *

Melissa Chase
Microsoft Research
melissac@microsoft.com

Trevor Perrin
Signal Technology Foundation
trevp@signal.org

Greg Zaverucha
Microsoft Research
gregz@microsoft.com

Draft – November 9, 2020

Abstract

In this paper we present a system for maintaining a membership list of users in a group, designed for use in the Signal Messenger secure messaging app. The goal is to support *private groups* where membership information is readily available to all group members but hidden from the service provider or anyone outside the group. In the proposed solution, a central server stores the group membership in the form of encrypted entries. Members of the group authenticate to the server in a way that reveals only that they correspond to some encrypted entry, then read and write the encrypted entries.

Authentication in our design uses a primitive called a keyed-verification anonymous credential (KVAC), and we construct a new KVAC scheme based on an algebraic MAC, instantiated in a group $\mathbb{G}$ of prime order. The benefit of the new KVAC is that attributes may be elements in $\mathbb{G}$, whereas previous schemes could only support attributes that were integers modulo the order of $\mathbb{G}$. This enables us to encrypt group data using an efficient Elgamal-like encryption scheme, and to prove in zero-knowledge that the encrypted data is certified by a credential. Because encryption, authentication, and the associated proofs of knowledge are all instantiated in $\mathbb{G}$ the system is efficient, even for large groups.

1 Introduction

Secure messaging applications enable a user to send encrypted messages to one or more recipients. A notion of *groups* is often supported: messages sent to a group will be delivered to all users who are current members of the group. Typically a group is created by a user to contain an initial set of members. These members (and the group creator) are given

*An extended abstract of this paper appeared at CCS'2020, this is the full version.

privileges to add and remove other members and grant them privileges, and so on. The result is that group membership is managed by the members.

The standard approach is to store the membership list, in plaintext, in a database on a server. The downside to this approach is that the server has a stored repository of associations between its users, and can easily insert malicious users into groups to receive messages. These are serious threats for an encrypted messaging system.

The Signal messaging app [Sig19] previously introduced a *private group* approach where the membership list is hidden from the server. In Signal's system the group membership list is maintained in a distributed fashion by each user [Mar14]. To change the membership of a group, a user updates their local copy of the membership list, then sends this new list to every other member via encrypted 1-to-1 messages. If some messages are lost (e.g. the sender loses connectivity before sending all messages), or clients attempt simultaneous updates, then members will end up with inconsistent views of membership. In an attempt to reduce the duration of this inconsistency, Signal clients will process group updates from users outside the membership list if the message contains a group-specific secret, but this weakens access-control [RMS17].

To address these problems with distributed private groups, we introduce a new approach. In our new approach, group members encrypt the membership list using a shared key and store the encrypted entries on a server. This means clients can acquire an up-to-date view of group membership by simply querying the server, and the server can apply access-control rules to all group updates. Using encryption in this manner introduces new requirements:

- *Anonymous authentication*: When a group member wishes to add or remove another user from the group, or fetch the membership list, the existing member must first authenticate to the server so that the server can determine whether the member is allowed to perform this operation. This is also true for the standard "plaintext on server" approach, but in our system the group entry contains an encrypted user identity (UID) rather than a plaintext UID. The group member must anonymously authenticate by proving ownership of the encrypted UID, without the server learning the UID.

- *Deterministic encryption*: It is important that each plaintext UID in a group corresponds to a single encrypted UID in that group, and that an entry must not decrypt successfully unless it is the unique deterministic encryption of the underlying UID. If this requirement is not met, a single UID could be added to the group using different ciphertexts. This would complicate access control and operations such as deletion. Additionally, deterministic encryption means users can calculate the encrypted user entry they are authenticating against without having to retrieve it from the server.

- *Decryption and authentication consistency*: Because encrypted entries are used in two ways (decrypted by users to learn the group membership, and used for authentication

by the server), it is important for entries to decrypt successfully if and only if they can be used for authentication.

We satisfy the anonymous authentication requirement using server-issued *anonymous credentials*. In particular, we introduce a new form of *keyed-verification anonymous credentials*, extending the construction from [CMZ14] to support efficient zero-knowledge proofs compatible with *verifiable encryption*.

Given this credential scheme, the server will issue users time-limited *auth credentials* for their UID. Users can then provide the server a zero-knowledge proof that they have a valid auth credential matching an encrypted entry. Because of the zero-knowledge property, the server receives assurance that the user possesses such an auth credential without learning the UID certified by the credential.

We satisfy the requirements for deterministic encryption, and decryption and authentication consistency, in two ways. As part of authentication, users prove to the server that their encrypted entry is a correct deterministic encryption of some UID. As part of decryption, users check that a ciphertext is a deterministic encryption of the decrypted UID.

Profile keys With the above building blocks we have a rudimentary private group system. We then build a more sophisticated system that additionally stores an encrypted *profile key* for each group member. Profile keys are used in Signal to encrypt *profile data* such as avatar images and profile names that provide a more user-friendly view of a user's identity [Lun17]. Encrypted profile data is stored on the server, but is not decryptable by the server. Users will share their profile key (and thus their profile data) with other users whom they trust.

To improve the Signal group experience we will store encrypted profile keys in the group membership list alongside encrypted UIDs so that group members will see a profile-enhanced view of the membership list, rather than simply a list of UIDs.

Storing encrypted profile keys introduces a new requirement for *UID and profile key consistency*: it is important that the server only stores a pair (UID ciphertext, profile key ciphertext) if this pair correctly decrypt to a UID and its associated profile key, even if these ciphertexts are created by a malicious group member.

We satisfy this requirement with an additional server-issued anonymous credential. Unlike the *auth credentials* discussed previously which are issued to the owners of UIDs, these *profile key credentials* are issued to any user who knows another user's profile key. Users will register a *profile key commitment* with the server. This enables other users to perform a *blinded credential issuance* with the server where the user proves knowledge of the profile key matching the profile key commitment, and the server issues them a profile key credential that certifies both a UID and profile key (this is a blinded issuance since the server is issuing a credential for a profile key it does not know).

After a user (Alice) acquires a profile key credential for another user (Bob), she can add Bob to groups by providing UID and profile key ciphertexts for Bob along with a zero-knowledge proof that these ciphertexts encrypt values which are certified by a profile key credential.

1.1 System Overview

We can summarize the main objects in the Signal Private Group System from the perspective of two users, Alice and Bob. Fig. 1 shows a typical interaction sequence in the system, and more details on these objects are presented in Section 5.

- Bob generates a *ProfileKey* and registers his *ProfileKeyCommitment* with the server.
- Bob trusts Alice to view his profile data and so shares his *ProfileKey* with Alice by sending her an encrypted message.[1]
- Alice contacts the server, without identifying herself, and uses Bob's *ProfileKey* to fetch a *ProfileKeyCredential* for Bob's UID and *ProfileKey*.
- Alice and Bob contact the server periodically to fetch *AuthCredentials* for their UID.
- Alice creates a new group containing her and Bob by generating a random *GroupMasterKey* and deriving *GroupPublicParams* from it, then registering the *GroupPublicParams* with the server. Alice also uploads pairs of (*UidCiphertext*, *ProfileKeyCiphertext*) for herself and Bob. Alice proves these ciphertexts are correct by proving that she has an *AuthCredential* for her *UidCiphertext*, and by proving she has a *ProfileKeyCredential* for each pair of ciphertexts.
- Alice sends Bob the *GroupMasterKey* via an encrypted message. Bob can now authenticate with his *AuthCredential* to download the other group entries and decrypt them using the *GroupMasterKey*. If Bob's entry is authorized to add or delete members of the group, Bob can also authenticate to the server and request it to perform these operations.

Cryptography For efficiency and simplicity, our solution is designed to work using cryptography instantiated in a group $\mathbb{G}$ of prime order q. Our encryption scheme is symmetric-key, deterministic, CCA-secure, and has a property we call *unique ciphertexts*, meaning that it is intractable to find two valid encryptions of the same plaintext, even with knowledge of the key. Since it is a variant of Elgamal encryption in $\mathbb{G}$, it has small ciphertexts with efficient encryption and decryption. Moreover, it is compatible with the efficient

[1] This is an end-to-end encrypted message which we assume the secure messaging platform provides. For details of E2E encryption in Signal, see [Sig19].

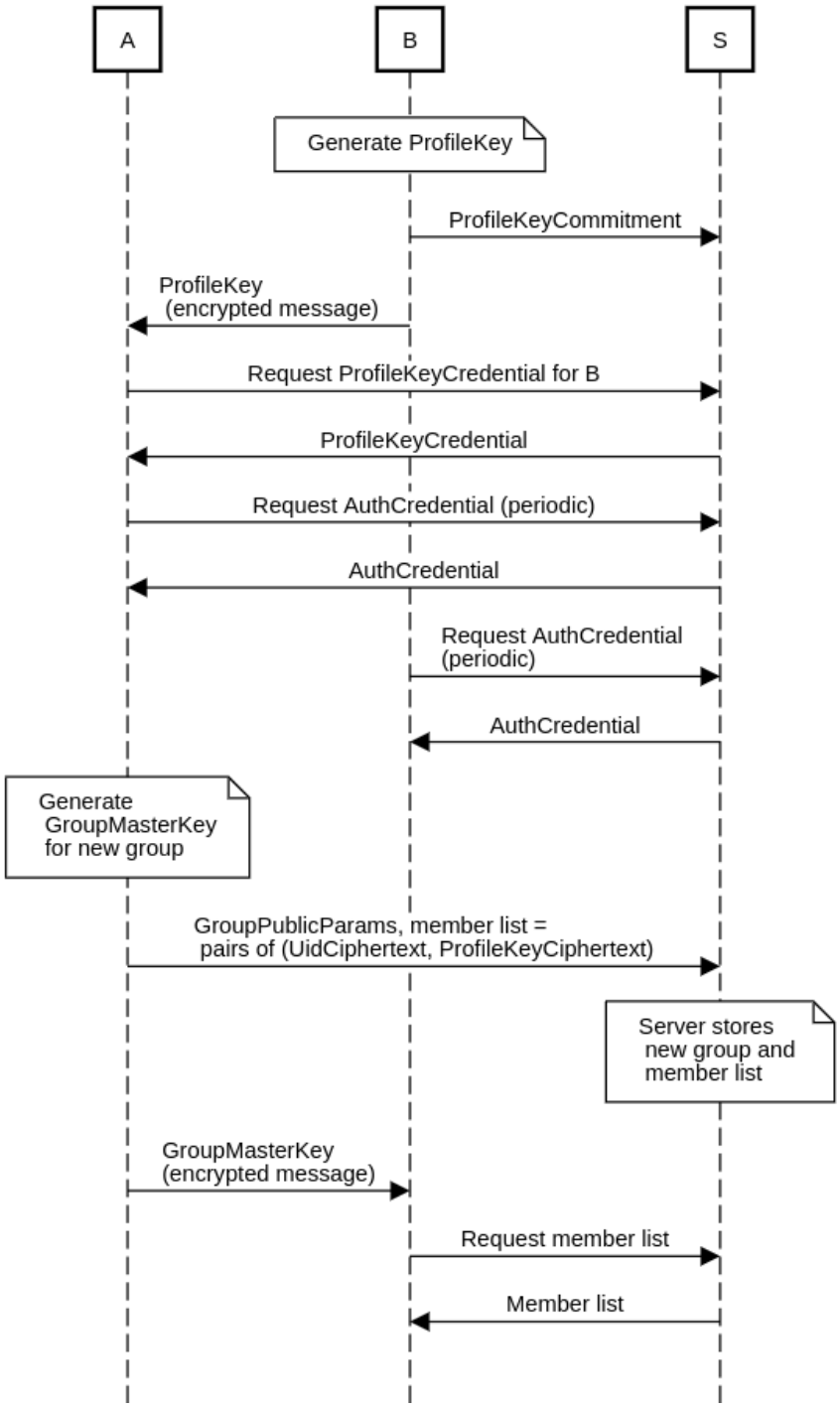

Figure 1: Overview of the main operations. In this sequence user A creates a group and adds user B to it, then user B fetches the group state.

zero-knowledge proof system we use for credential presentation, allowing us to prove that ciphertexts are well-formed with respect to a public commitment of the key, and that the plaintext is an attribute from a credential.

For the latter part of the proof, we need a credential system that supports attributes that are group elements. Previously known anonymous credentials and KVAC schemes only support attributes from $\mathbb{Z}_q$. Following the approach to constructing a KVAC scheme from [CMZ14], we first design an algebraic MAC where messages may be elements of $\mathbb{G}$ or $\mathbb{Z}_q$. We prove our new MAC is secure in the random oracle model, assuming i) that DDH in $\mathbb{G}$ is hard, and ii) that a simpler MAC, called $\mathsf{MAC}_{\mathsf{GGM}}$, from [CMZ14] is secure. Our security analysis of our encryption scheme first defines the new properties required for the private group system, then we prove the scheme is secure under the DDH assumption.

We then give protocols for credential (blind) issuance and presentation, to construct a complete KVAC system satisfying the security properties defined in [CMZ14]. The resulting credential scheme and proof protocols are efficient, and can be instantiated using well-known non-interactive generalized Schnorr proofs of knowledge.

Security Properties The server in our system can neither decrypt group entries, nor forge new entries. These are our main security goals.

However, the server can observe a small amount of information regarding group state, and could perform limited modifications to this state:

- The server could observe when a particular encrypted entry performs some action, such as fetching the membership list, or adding or deleting other encrypted entries. Making updates not leak anything would be very expensive, e.g. clients would have to re-encrypt and rewrite a maximum-size group state with each operation and include a proof of correctness over the entire state.

- The server could delete ciphertexts or reinstate old ciphertexts. We believe this system could be extended to add end-to-end integrity-protection by having clients sign each new group state (along with an incrementing "version" number), however we have not tackled that in the initial system.

A malicious server could corrupt the group state by writing invalid ciphertexts, or inconsistent UID and profile key ciphertexts. This shouldn't provide the server any capability beyond interrupting service to users, which the server could do more easily by simply not responding to client queries.

A user who has acquired a group's *GroupMasterKey* and then leaves the group (or is deleted) retains the ability to collude with the server to encrypt and decrypt group entries. In the current system a new group would have to be created, excluding the removed user(s). An automated or more sophisticated re-keying strategy could also be added as a future extension.

Assuming an honest server, it should not be possible for malicious users to forge authentications, violate the server's access control rules, or violate the consistency requirements between decryption and authentication of UID ciphertexts, or between UID ciphertexts and profile key ciphertexts.

2 Preliminaries and Related Work

Notation We use capital letters to denote group elements, and lower case letters to denote integers modulo the group order. The notation $x \in_R X$ means that x is chosen uniformly at random from the set X.

2.1 Group Description and Hardness Assumptions

The new cryptographic primitives in this paper are designed to work in a cyclic group, denoted $\mathbb{G}$, of prime order q. We require that $\mathbb{G}$ has three associated functions.

1. A function $\mathsf{HashTo}\mathbb{G} : \{0,1\}^* \to \mathbb{G}$ that hashes strings to group elements. This should be based on a cryptographic hash function; we will model it as a random oracle.
2. A function $\mathsf{HashTo}\mathbb{Z}_q : \{0,1\}^* \to \mathbb{Z}_q$, also based on a cryptographic hash function.
3. A function $\mathsf{EncodeTo}\mathbb{G} : \{0,1\}^\ell \to \mathbb{G}$, that maps ℓ-bit strings to elements of $\mathbb{G}$ in a reversible way. The parameter ℓ depends on the size of $\mathbb{G}$ and the encoding.

For our security analysis, we will assume that the decisional Diffie-Hellman problem (DDH) is hard in $\mathbb{G}$, i.e., given (G^a, G^b, C) decide if $C = G^{ab}$. This implies that the discrete logarithm problem (DLP) is also hard in $\mathbb{G}$, i.e., given $Y = G^x$ it is hard to find x. We also require that $\mathsf{MAC}_{\mathsf{GGM}}$ is $\mathsf{uf\text{-}cma}$-secure for the security of our MAC, and the only known proofs are in the generic group model, so we inherit this assumption as well.

Unlike some credential systems, we don't require pairings or the strong RSA assumption. The CPU cost of a pairing in BLS12-381 is about 40x the time required for a scalar multiplication in the Ristretto group we use (as described in §6). Similarly, the gap is roughly 50x between scalar multiplication in Ristretto and RSA-3072 exponentiations. Using prime order groups also means the credential system can use the same elliptic curve as is used for key agreement and signatures.

2.2 Keyed-Verification Anonymous Credentials (KVAC)

An anonymous credential system [Cha85, CV02, PZ13] is a set of cryptographic protocols: A *credential issuance* protocol provides users with credentials that "certify" some set of attributes. A *credential presentation* protocol enables the user to prove that they possess a credential whose attributes satisfy some predicate without revealing the credential or

anything else about it (a zero-knowledge proof). There is a vast literature on anonymous credentials, a good starting point on the subject is [RCE15].

Traditional anonymous credentials designs are based on public key signatures: the credential Alice holds is a special type of signature on the attributes. When she presents the credential to Bob, she proves (in zero-knowledge) that her credential is a valid signature with respect to the credential issuer's public key. The benefit of signature-based credentials is that Alice may present her credential to anyone in possession of the issuer's public key, but the drawback is that known constructions are relatively expensive, being based on the strong RSA assumption [CL03] or groups with a pairing [CL04], or if the credentials are efficient (using prime order groups [PZ13, BL13]) they do not support *multi-show unlinkability*. This means that if Alice presents her credential to Bob twice, he can link these presentations.

With *keyed-verification anonymous credentials* (KVAC) [CMZ14], the issuer and verifier are the same party (or share a key), and so the design can use a MAC in place of a signature scheme. It is then possible to have an efficient credential system constructed in a group of prime order, with multi-show unlinkability. In the present scenario the issuer and verifier are the same party, so a KVAC system is a natural fit.

2.3 MACs and Algebraic MACs

Many popular MAC algorithms are constructed using symmetric-key primitives like hash functions (e.g., HMAC [KBC97]) and block ciphers (e.g., Poly1305-AES [Ber05]). Unlike algebraic MACs, these MACs do not have efficient zero-knowledge proofs associated to them. We use the term algebraic MAC to mean a MAC constructed using group operations. Dodis et al. [DKPW12] study many algebraic MACs, and Chase et al. [CMZ14] show that certain algebraic MACs can be used to construct an efficient type of anonymous credential.

We describe a particular algebraic MAC, called $\mathsf{MAC}_{\mathsf{GGM}}$, that we use as a building block in our new MAC scheme.

Definition 1. *The* $\mathsf{MAC}_{\mathsf{GGM}}$ *construction [DKPW12, CMZ14] is an algebraic MAC constructed in a group* $\mathbb{G}$ *of prime order* q*, with the following algorithms.*

KeyGen*: choose random* $(x_0, x_1) \in {\mathbb{Z}_q}^2$*, output* $sk = (x_0, x_1)$*.*

$\mathsf{MAC}(sk, m)$*: choose random* $U \in G$*, output* $\sigma = (U, U^{x_0 + x_1 m})$*.*

$\mathsf{Verify}(sk, (U, U'), m)$*: recompute* $U'' = U^{x_0 + x_1 m}$*, output "valid" if* $U'' = U'$*, and "invalid" otherwise.*

In [DKPW12] it is shown that $\mathsf{MAC}_{\mathsf{GGM}}$ satisfies a weak notion of MAC security called selective security (where the adversary must specify the message that will be forged in advance), assuming DDH. In [CMZ14, ABS16], it is shown that $\mathsf{MAC}_{\mathsf{GGM}}$ is $\mathsf{uf\text{-}cmva}$ secure in the generic group model.

The security notions for algebraic MACs are the same as for traditional MACs.

uf-cma: unforgeability under chosen message attacks

suf-cma: strong unforgeability under chosen message attacks

suf-cmva: strong unforgeability under chosen message and verification attacks

Definition 2. *For a MAC with algorithms* (KeyGen, MAC, Verify)*, consider the following security game between a challenger* $\mathcal{C}$ *and an attacker* $\mathcal{A}$*.*

1. $\mathcal{C}$ *uses* KeyGen *to generate* sk*. If the MAC has public parameters,* $\mathcal{C}$ *gives them to* $\mathcal{A}$*.*
2. $\mathcal{A}$ *makes queries to* $\mathcal{C}$*.*
   - *MAC query:* $\mathcal{A}$ *submits* m *and* $\mathcal{C}$ *outputs* $\sigma = \mathsf{MAC}(sk, m)$*.* $\mathcal{C}$ *stores* (m, σ) *in a set* M*.*
   - *Verify query:* $\mathcal{A}$ *submits* (σ, m) *and* $\mathcal{C}$ *outputs* $\mathsf{Verify}(sk, \sigma, m)$
3. $\mathcal{A}$ *outputs* (σ^*, m^*)

We say that $\mathcal{A}$ *wins the* uf-cma *game if no Verify queries are made, and* m^* *is not in* M*. We say that* $\mathcal{A}$ *wins the* suf-cma *security game if no Verify queries are made, and* $(m^*, \sigma^*) \notin M$*. We say that* $\mathcal{A}$ *wins the* suf-cmva *game if* $(m^*, \sigma^*) \notin M$*. The MAC is* uf-cma*-secure if no polynomial-time* $\mathcal{A}$ *wins the* uf-cma *game with probability that is non-negligible in* κ *(and* suf-cma*,* suf-cmva *security are defined analogously).*

A proof of the following lemma is in [BS20, Theorem 6.1]. Basically it says that for a strongly unforgeable MAC, verification queries don't help an attacker (when looking only at asymptotic security).

Lemma 3. *Let* $\mathcal{M}$ *be a MAC scheme. The security notions* suf-cma *and* suf-cmva *are equivalent. If* $\mathcal{M}$ *is* suf-cma *secure, then it is also* suf-cmva *secure (and vice-versa).*

2.4 Zero-Knowledge Proofs

In multiple places our constructions use zero-knowledge (ZK) proofs to prove knowledge of discrete logarithms and of representations of elements in $\mathbb{G}$. We use the notation introduced by Camenisch and Stadler [CS97]. A non-interactive proof of knowledge π is described by:

$$\pi = \mathrm{PK}\{(x, y, \ldots) : \text{Predicates using } x,\ y \text{ and public values}\}$$

which means that the prover is proving knowledge of $(x, y, \ldots)$ (all elements of $\mathbb{Z}_q$), such that the predicates are satisfied. Predicates we will use in this paper are knowledge of a discrete logarithm, e.g., $\mathrm{PK}\{(x) : Y = G^x\}$ for public Y and G, and knowledge of a representation using two or more bases, e.g., $\mathrm{PK}\{(x_1, \ldots, x_n) : Y = \prod_{i=1}^{n} G_i^{x_i}\}$. We also use multiple predicates, and require that they all be true, e.g., $\mathrm{PK}\{(x, y) : Y =$

$G^x \ \wedge \ Z = G^y H^x\}$. Given two proofs we can combine them by merging the list of secrets and predicates, e.g., proofs $\pi_1 = \text{PK}\{(x) : Y = G^x\}$ and $\pi_2 = \text{PK}\{(x, y) : Z = G^x H^y\}$ combine to give $\pi_3 = \text{PK}\{(x, y) : Y = G^x \ \wedge \ Z = G^x H^y\}$.

There are multiple ways to instantiate the proofs of knowledge we need. The Signal implementation uses the "generic linear" generalization of Schnorr's protocol described in [BS20, Ch.19], made noninteractive with the Fiat-Shamir transform [FS87].

2.5 Secure Messaging and Signal

In a secure messaging application such as Signal, users send each other encrypted messages with the aid of a server. For the purposes of this document, most details of the Signal Protocol [Sig19] can be abstracted away, leaving a few points which are crucial for understanding the Signal Private Group System in Section 5.

Users can contact the Signal server over a mutually-authenticated secure channel, or over a secure channel that only authenticates the server. For simplicity, we'll describe the former case as an *authenticated channel*, and the latter case as an *unauthenticated channel*. Unauthenticated channels are used when the user wishes to interact with the server without revealing their identity, and thus will be used extensively in the protocols described here.

When users in a Signal group send encrypted messages to the group, they encrypt and send the message to each group member, individually, with end-to-end encryption. The server is given no explicit indication of the difference between group and non-group encrypted messages, apart from traffic analysis.

Users are identified by a *UID*. Users send their profile key attached to encrypted text messages if the recipient is trusted, which we interpret to mean either: the recipient is in the sender's address book; or the sender initiated the conversation; or the sender opted in to sharing profile data with the recipient. Given a user's UID and profile key, one can fetch and decrypt profile data they have uploaded for themselves.

3 A New KVAC and Protocols

In this section we define our new keyed-verification anonymous credential system. We start with the new algebraic MAC that the scheme is based on, then describe protocols for credential issuance and presentation. Security analysis of these new primitives is given in Section 7.

3.1 A New Algebraic MAC

Our new MAC is constructed in a group $\mathbb{G}$ of prime order q. A new feature that is important for our use case is that the list of attributes may contain a mix of elements of $\mathbb{G}$ (*group attributes*), or integers in $\mathbb{Z}_q$ (*scalar attributes*), while in previous work attributes were restricted to being chosen from $\mathbb{Z}_q$. When using generalized Schnorr proofs in a

cyclic group (the most common ZK proof system), the types of statements that can be proven about attributes in $\mathbb{G}$ are limited, but we will be able to prove statements about the plaintext encrypted by an Elgamal-like ciphertext.

Parameters Let κ be a security parameter. The number of attributes in the message space is denoted n. We write $\vec{x}$ to denote a list of values. The scheme requires the following fixed set of group elements:

$$G, G_w, G_{w'}, G_{x_0}, G_{x_1}, G_{y_1}, \ldots, G_{y_n}, G_{m_1}, \ldots, G_{m_n}, G_V$$

generated so that the relative discrete logarithms are unknown, e.g., $G_{m_1} = \mathsf{HashTo}\mathbb{G}(\text{“}\mathtt{m1}\text{”})$.

$\mathsf{KeyGen}(params)$ The secret key is $sk := (w, w', x_0, x_1, (y_1, \ldots, y_n))$, all randomly-chosen elements of $\mathbb{Z}_q$. We will write $W := {G_w}^w$, and W is considered part of sk. Optionally, compute the *issuer parameters* $iparams$ (C_W, I) as follows:

$$C_W = {G_w}^w {G_{w'}}^{w'}, \;\; I = \frac{G_V}{{G_{x_0}}^{x_0} {G_{x_1}}^{x_1} {G_{y_1}}^{y_1} \ldots {G_{y_n}}^{y_n}}$$

The *iparams* are optional for basic use of the MAC, but are required when the MAC is used in the protocols we consider, therefore we assume *iparams* is always present.

$\mathsf{MAC}(sk, \vec{M})$ The MAC is calculated over a collection of group attributes and scalar attributes. Each attribute is represented by a group element $M_i \in \mathbb{G}$. For a given MAC key, each of the n attribute positions is fixed to always contain a group attribute in $\mathbb{G}$, or always encode a scalar attribute in $\mathbb{Z}_q$. If M_i encodes a scalar attribute then $M_i = {G_{m_i}}^{m_i}$.

Choose random $t \in \mathbb{Z}_q$, $U \in \mathbb{G}$, and compute

$$V = W U^{x_0 + x_1 t} \left(\prod_{i=1}^{n} M_i^{y_i} \right)$$

Output (t, U, V) as the MAC on $\vec{M}$.

$\mathsf{Verify}(sk, \vec{M}, (t, U, V))$ Recompute V as in MAC (denote it V') and accept if $V \stackrel{?}{=} V'$.

Security Intuitively, for security, the component $U^{x_0 + x_1 t}$ is a $\mathsf{MAC}_{\mathsf{GGM}}$ tag on t, and uses distinct random values (U, t), to prevent multiple MACs with different t from being combined in a forgery. The terms using y_i prevent manipulation of individual attributes. Note also that the term W is necessary, since without it, given a MAC (t, U, V) on $(M_1, \ldots, M_n)$, then (t, U^c, V^c) is a valid MAC on $(M_1^c, \ldots, M_n^c)$ for any $c \in \mathbb{Z}_q$. In Section 7.3 we prove this MAC is $\mathsf{suf\text{-}cmva}$ secure, assuming DDH is hard in $\mathbb{G}$ and $\mathsf{MAC}_{\mathsf{GGM}}$ is a $\mathsf{uf\text{-}cma}$ secure MAC.

Optimizations We note that the construction can be derandomized by setting t as the hash of the attributes, and U as a hash of t. The resulting tags would be a single group element long.

In the context of a credential system requiring many scalar attributes, it may be more efficient to first commit to many scalar attributes as a group attribute, e.g., $C = {G_1}^{m_1} \cdots {G_\ell}^{m_\ell} H^r$, then compute the MAC over C. Then during credential presentation, the prover can avoid having to create and send a commitment for each scalar attribute separately (one of the performance drawbacks of [CMZ14]). In this work credentials have at most one scalar attribute, so we did not investigate this optimization further.

3.2 Credential Issuance and Presentation

Here we describe how credentials are issued and presented. We describe issuance when there are no blind attributes (i.e. attributes not known to the issuer), and describe blind issuance in Section 5.10.

Credential Issuance A credential is a MAC (t, U, V) from Section 3.1 on the attributes M_i. The issuer proves knowledge of the secret key, and that (t, U, V) is correct relative to $iparams = (C_W, I)$, with the following proof of knowledge.

$$\begin{aligned}\pi_I = \mathrm{PK}\{(w, w', x_0, x_1, y_1, \ldots, y_n) : \\ C_W &= {G_w}^w {G_{w'}}^{w'} \wedge \\ I &= \frac{G_V}{{G_{x_0}}^{x_0} {G_{x_1}}^{x_1} {G_{y_1}}^{y_1} \ldots {G_{y_n}}^{y_n}} \wedge \\ V &= {G_w}^w (U^{x_0})(U^t)^{x_1} \left(\prod_{i=1}^{n} {M_i}^{y_i} \right) \}\end{aligned}$$

Credential Presentation To present the credential (t, U, V) on attributes $\vec{M}$, a user creates the following proof. This proves that the user holds a valid credential, and has knowledge of the hidden scalar attributes (for proving that hidden group attributes match some ciphertext we will need additional predicates; see the next section).

1. Choose $z \in_R \mathbb{Z}_q$ and compute (i ranges from 1 to n):

$$\begin{aligned} Z &= I^z \\ C_{x_0} &= {G_{x_0}}^z U \\ C_{x_1} &= {G_{x_1}}^z U^t \\ C_{y_i} &= \begin{cases} {G_{y_i}}^z M_i \text{ if } i \text{ is a hidden group attribute} \\ {G_{y_i}}^z {G_{m_i}}^{m_i} \text{ if } i \text{ is a hidden scalar attribute} \\ {G_{y_i}}^z \text{ if } i \text{ is a revealed attribute} \end{cases} \\ C_V &= {G_V}^z V \end{aligned}$$

   along with the value $z_0 = -zt \pmod{q}$. Let $\mathcal{H}_s$ denote the set of hidden scalar attributes.

2. Compute the following proof of knowledge:

$$\begin{aligned} \pi = \mathrm{PK}\{ & (z, z_0, \{m_i\}_{i \in \mathcal{H}_s}, t) : \\ & Z = I^z \ \wedge \\ & C_{x_1} = {C_{x_0}}^t {G_{x_0}}^{z_0} {G_{x_1}}^z \ \wedge \\ & C_{y_i} = \begin{cases} {G_{y_i}}^z {G_{m_i}}^{m_i} \text{ if } i \text{ is a hidden scalar attribute} \\ {G_{y_i}}^z \text{ if } i \text{ is a revealed attribute} \end{cases} \\ & \} \end{aligned}$$

3. Output $(C_{x_0}, C_{x_1}, C_{y_1}, \ldots, C_{y_n}, C_V, \pi)$

4. Let $\mathcal{H}$ denote the set of all hidden attributes. The verifier computes

$$Z = \frac{C_V}{(W {C_{x_0}}^{x_0} {C_{x_1}}^{x_1} \prod_{i \in \mathcal{H}} {C_{y_i}}^{y_i} \prod_{i \notin \mathcal{H}} (C_{y_i} M_i)^{y_i})}$$

   using the secret key $(W, x_0, x_1, y_1, \ldots, y_n)$ and revealed attributes, and then verifies π.

Security The security of our new KVAC construction is analyzed in Section 7.4. We show that the scheme has the security properties defined in [CMZ14], namely, correctness, unforgeability, anonymity, blind issuance and key-parameter consistency.

4 Verifiable Encryption

Since our credential system supports attributes that are group elements, we can use the Elgamal encryption scheme to create an efficient verifiable encryption scheme [CD00]. By *verifiable*, we mean that we can prove properties about the plaintext in zero-knowledge.

Suppose we have a credential certifying a group attribute M_1, and let $Y = G^y$ be an Elgamal public key. The encryption of M_1 with Y is $(E_1, E_2) = (G^r, Y^r M_1)$. To prove that the plaintext is certified, we add two predicates to the credential presentation proof:

$$E_1 = G^r \wedge C_{y_1}/E_2 = {G_{y_1}}^z/Y^r \ .$$

Previous verifiable encryption schemes did not allow us to efficiently encrypt group elements, and thus required more expensive techniques, such as a variant of Paillier's encryption scheme [CS03], or groups with bilinear maps [CHK+11]. We caveat that the above basic Elgamal scheme is not CCA secure, and we have not carefully analyzed its security. Since it will be sufficient for our application, we focus on *symmetric-key* verifiable encryption that is CCA secure.

Symmetric-key verifiable encryption with unique ciphertexts Informally, we will need a symmetric-key encryption scheme that (i) has *unique ciphertexts*, meaning that for every plaintext there is at most one ciphertext that will decrypt correctly, (ii) has public verifiability, meaning that we can prove that a ciphertext encrypts a certified plaintext with a key that is consistent with some public parameters, and (iii) is correct under adversarially chosen keys, meaning that it is hard to find a key and message that cause decryption to fail. In Section 7.2 we define these properties formally and prove our construction meets them in the random oracle model assuming the DDH problem is hard in $\mathbb{G}$.

4.1 Construction

System parameters A cyclic group $\mathbb{G}$ of prime order q. Recall that $\mathsf{EncodeTo}\mathbb{G}$ is a function that encodes strings as group elements, that $\mathsf{HashTo}\mathbb{G}$ and $\mathsf{HashTo}\mathbb{Z}_q$ are cryptographic hash functions that hash strings to elements of $\mathbb{G}$ and $\mathbb{Z}_q$ (respectively). We define a fourth function $\mathsf{Derive} : \{0,1\}^{2\kappa} \to (\mathbb{Z}_q)^2$, used to derive two keys from a master key. Derive should also be a cryptographic hash function (and our analysis in Theorem 10 models it as a random oracle). Two group elements G_{a_1} and G_{a_2} are chosen such that the relative discrete logs are unknown.

$\mathsf{KeyGen}(1^\kappa)$ Choose the secret key k_0 at random from $\{0,1\}^{2\kappa}$, and derive $k = \mathsf{Derive}(k_0) = (a_1, a_2) \in (\mathbb{Z}_q)^2$. We assume that honest parties will not use k that was not derived from a k_0 in this way. Compute the public parameters $pk := {G_{a_1}}^{a_1} {G_{a_2}}^{a_2}$.

$\mathsf{Enc}(k, m)$ Compute $M_1 = \mathsf{HashTo}\mathbb{G}(m)$ and $M_2 = \mathsf{EncodeTo}\mathbb{G}(m)$, then

$$E_1 = {M_1}^{a_1}$$
$$E_2 = (E_1)^{a_2} M_2.$$

$\mathsf{Dec}(k, E_1, E_2)$ First compute $m' = \mathsf{DecodeFrom}\mathbb{G}(E_2/{E_1}^{a_2})$ and $M_1' = \mathsf{HashTo}\mathbb{G}(m')$. Then perform the following checks and return m' if they succeed, and $\perp$ otherwise.

$$E_1 \stackrel{?}{=} (M_1')^{a_1}$$
$$E_1 \stackrel{?}{\neq} 1$$

$\mathsf{Prove}(k, pk, E_1, E_2, \vec{C})$ To prove that (E_1, E_2) encrypts the plaintext committed in the list of commitments $\vec{C}$:

$$C_{y_1} := {G_{y_1}}^z M_1, \text{ and } C_{y_2} := {G_{y_2}}^z M_2,$$

first compute the scalar $z_1 = -za_1$, then create the proof

$$\begin{aligned}\pi_{\mathsf{Enc}} = \mathrm{PK}\{(a_2, a_2, z, z_1) : \\ & pk = {G_{a_1}}^{a_1} {G_{a_2}}^{a_2} \ \wedge \\ & C_{y_2}/E_2 = {G_{y_2}}^z / {E_1}^{a_2} \ \wedge \quad \text{//plaintext is } M_2 \\ & E_1 = {C_{y_1}}^{a_1} {G_{y_1}}^{z_1}\} \qquad \text{//} E_1 \text{ is well-formed}\end{aligned}$$

$\mathsf{Verify}(pk, \pi_{\mathsf{Enc}}, \vec{C})$ Accept if π_{Enc} verifies, otherwise reject.

Discussion When we use the Prove function, it will be combined with the credential presentation proof, which creates the pair of commitments $\vec{C}$ (and this is why they use the same z value). E_1 plays a role similar to the *synthetic initialization vector* of [RS06] and can be seen as an authentication tag on m, assuming DDH, as we prove in Lemma 11.

We also note that a more generic way to achieve CCA security is possible with an encrypt-then-MAC construction. Since the algebraic MAC of Section 3.1 can authenticate group elements, we can MAC the pair (E_1, E_2) and append the tag. This is less performant, but has the advantage that decryption can immediately reject ciphertexts without a valid MAC.

The Derive function in key generation serves two purposes. First, when sharing group keys amongst themselves, group members can share a short master key, saving bandwidth. Second, it ensures that (a_1, a_2) are not a degenerate value (such as all zero), that might be used to break the correctness under the adversarially chosen keys property (Definition 9).

5 The Signal Private Group System

In Section 1.1 we provided an overview of the system. Here we review the main data objects; spell out the high-level operations; and then describe encryption and credential operations and objects in detail.

5.1 Data Objects

The user encrypts *UIDs* and *ProfileKeys* into *UidCiphertexts* and *ProfileKeyCiphertexts*, using *GroupSecretParams* which are shared between users. For efficiency, and to avoid maliciously chosen keys (as described in Section 4.1) users actually share a smaller *GroupMasterKey*, and use this to derive *GroupSecretParams*. The *GroupPublicParams* for each group are registered with the server.

Users prove correctness of these ciphertexts to the server using *AuthCredentials* and *ProfileKeyCredentials*. An *AuthCredential* certifies a *UID* and a *redemption date* during which the credential will be valid. A *ProfileKeyCredential* certifies a *UID* and *ProfileKey*.

There are five data objects used to acquire and present credentials which all contain zero-knowledge proofs of knowledge: The server issues credentials using an *AuthCredentialResponse* or *ProfileKeyCredentialResponse*. The *ProfileKeyCredentialResponse* is sent in response to a *ProfileKeyCredentialRequest*. Users present these credentials and prove they correspond to ciphertexts via *AuthCredentialPresentation* and *ProfileKeyCredentialPresentation* objects, which contain both the ciphertexts and proofs of knowledge.

These objects are described in more detail below.

5.2 General Data Object

UID: A 16-byte UUID (universally unique identifier) representing a user.

ServerSecretParams: A set of secret values the server uses to issue and verify credentials.

ServerPublicParams: A set of public values which are derived from *ServerSecretParams* and which are known to all users.

5.3 Data Objects for Authentication

AuthCredential: A credential with attributes based on the *UID* and a redemption date which specifies the day on which this credential is valid.

AuthCredentialResponse: A message sent from the server to a user containing an *AuthCredential* and a proof that this credential was constructed correctly. Since the corresponding request is trivial, we omit it.

AuthCredentialPresentation: A message sent from a user to the server containing a *UidCiphertext*, a redemption date, and the credential presentation proof π_A from Section 5.12.

5.4 Data Objects for Profile Keys

The data objects in this section are all related to profile keys, commitment and credentials.

ProfileKey: A 32-byte key used for symmetric-key encryption of profile data. A user shares their *ProfileKey* with users they trust, but not with the server. At any point in time a UID is associated with a single profile key. The uses of profile data are outside the scope of this document, but two examples are a user's screen name and profile picture.

ProfileKeyCommitment: A deterministic commitment to a *ProfileKey*.

ProfileKeyVersion: An identifier derived from a *ProfileKey*.

ProfileKeyCredential: A credential with attributes based on a *UID* and *ProfileKey*. Note that an *AuthCredential* for a UID is issued only to the user who owns that UID, whereas *ProfileKeyCredentials* are issued to anyone who knows the profile key for a UID.

ProfileKeyCredentialRequest: A message sent from a user to the server to request a *ProfileKeyCredential*. The message contains a $ProfileKeyVersion$, a proof of knowledge of the corresponding *ProfileKey*, and some data to help the server perform a blinded credential issuance.

ProfileKeyCredentialResponse: A message sent from the server to a user containing a *ProfileKeyCredential* and the blind issuance proof π_{BI} from Section 5.10.

ProfileKeyCredentialPresentation: A message sent from a user to the server containing a *UidCiphertext*, a *ProfileKeyCiphertext*, and a zero-knowledge proof of knowledge of some *ProfileKeyCredential* issued over the *UidCiphertext* and *ProfileKey*.

5.5 Data Objects for Groups

The data objects in this section exist for a specific group.

GroupMasterKey: A random value which the *GroupSecretParams* are derived from. When a new user is added or invited to a group, the user adding them will send the new member the group's *GroupMasterKey* via an encrypted message, so the new member can derive the *GroupSecretParams*. Each encrypted message sent within the group will also contain a copy of the *GroupMasterKey*, in case the initial message fails to arrive. Note that a user who has acquired a group's *GroupMasterKey* and then leaves the group (or is deleted) retains the ability to collude with a malicious server to encrypt and decrypt group

entries. We deem this risk acceptable for now due to the complexities in rapid and reliable rekey of the *GroupMasterKey*.

GroupSecretParams: A set of secret values which group members use to encrypt and decrypt *UidCiphertexts* and *ProfileKeyCiphertexts*, as well as construct zero-knowledge proofs about these ciphertexts. Derived from the *GroupMasterKey*.

GroupPublicParams: A set of public values corresponding to the *GroupSecretParams*. The *GroupPublicParams* are stored on the server to represent a group.

UidCiphertext: A deterministic encryption of a *UID* using *GroupSecretParams*.

ProfileKeyCiphertext: A deterministic encryption of a *ProfileKey* using *GroupSecretParams*.

Role: A value specifying what access privileges a user has to modify the group. For example, a user with an administrator role may have more privileges than other users. Discussion of specific roles is out of scope of this document. Roles are enforced by the server, not by a cryptographic mechanism.

5.6 Operations for Credentials

GetAuthCredential

1. The user contacts the server over an authenticated channel and requests an *AuthCredential* for some redemption date.
2. If the date is in the allowed range (e.g., within next few days) the server returns an *AuthCredentialResponse* for the date.
3. The user verifies the proof of knowledge in the *AuthCredentialResponse* and stores an *AuthCredential*.

CommitToProfileKey

1. The user generates a random *ProfileKey*, and derives a *ProfileKeyVersion* and *ProfileKeyCommitment* from it.
2. The user sends the (*ProfileKeyVersion*, *ProfileKeyCommitment*) pair over the authenticated channel to the server.
3. The server stores the *ProfileKeyCommitment* associated with the authenticated user's *UID* and the *ProfileKeyVersion*.

GetProfileKeyCredential
This operation provisions a user with a *ProfileKeyCredential* for some (*UID*, *ProfileKey*) if and only if the user knows a *ProfileKey* matching the target user's *ProfileKeyCommitment*.

1. The user derives a $ProfileKeyVersion$ from the *ProfileKey*, and computes a *ProfileKeyCredentialRequest* from the *ProfileKey*.
2. The user sends the (*UID*, $ProfileKeyVersion$, *ProfileKeyCredentialRequest*) over an unauthenticated channel to the server.
3. If the server has a stored *ProfileKeyCommitment* for the specified *UID* and *ProfileKeyVersion*, the server verifies the proof of knowledge in the *ProfileKeyCredentialRequest*.
4. If verification succeeds the server generates a *ProfileKeyCredentialResponse*.
5. The user verifies the proof of knowledge in the *ProfileKeyCredentialResponse*, and if verification succeeds the user stores a *ProfileKeyCredential* for the target *UID*.

5.7 Operations for Group Management

AuthAsGroupMember
This operation uses an unauthenticated channel so that the server does not learn the user's *UID*. Upon completion, the channel is authenticated to a particular *UidCiphertext* within a group. This operation is used by the subsequent operations.

1. The user recomputes their *UidCiphertext* for the group and creates an *AuthCredentialPresentation* to prove knowledge of an $AuthCredential$ matching the *UidCiphertext*.
2. The user contacts the server over an unauthenticated channel and sends the *GroupPublicParams* and *AuthCredentialPresentation*.
3. The server verifies the proof of knowledge in the *AuthCredentialPresentation* and that the *GroupPublicParams* and *UidCiphertext* correspond to some user in the specified group.

AddGroupMember

1. The user and server execute $AuthAsGroupMember$.
2. The user encrypts the new user's (*UID*, *ProfileKey*) into (*UidCiphertext*, *ProfileKeyCiphertext*) using the *GroupSecretParams*, then creates a *ProfileKeyCredentialPresentation* for these ciphertexts and sends it to the server, along with a *Role* for the new user.
3. The server verifies the *ProfileKeyCredentialPresentation* and checks that:

(a) The authenticated user's *Role* allows them to add users.

(b) The *UidCiphertext* does not already exist in the group as a full member. If the *UidCiphertext* exists in the group as an invited member (i.e., missing *ProfileKeyCiphertext*; see *AddInvitedMember* below), then this operation adds the user as a full member.

If these checks succeed, the server stores the tuple (*UidCiphertext*, *ProfileKeyCiphertext*, *Role*) in the group. Otherwise the server returns an error.

CreateGroup

1. The user generates a $GroupMasterKey$, and uses it to derive $GroupSecretParams$ and $GroupPublicParams$.
2. The user contacts the server over an unauthenticated channel and sends the $GroupPublicParams$ for the new group.
3. The user performs a variant of $AddGroupMember$ which initializes the group with one member (the *UidCiphertext* from $AuthAsGroupMember$) and skips the *Role* check.

FetchGroupMembers

1. The user and server execute $AuthAsGroupMember$.
2. All of the (*UidCiphertext*, *ProfileKeyCiphertext*) pairs are returned for full members as well as invited members.

DeleteGroupMember

1. The user and server execute $AuthAsGroupMember$.
2. The user sends the *UidCiphertext* of another user or themselves (the target user).
3. The server checks whether the authenticated user's *Role* allows them to delete the target user.
4. If so, the entry in the group membership is deleted.

AddInvitedGroupMember and **UpdateProfileKey**
These operations are variants of *AddGroupMember*. *AddInvitedGroupMember* is used when a group member would like to add a target user to the group but doesn't know the target's *ProfileKey*. In this case, the target user is added without a *ProfileKeyCiphertext* or *ProfileKeyCredentialPresentation*. An invited user only becomes a full group member once their *ProfileKeyCiphertext* is populated via *AddGroupMember* or *UpdateProfileKey*.

UpdateProfileKey is the same as *AddGroupMember* except that users are only allowed to replace their own *ProfileKeyCiphertext* (to prevent the possibility of rollback attacks to older versions of profile data).

5.8 System Parameters and Server Parameters

AuthCredentials and *ProfileKeyCredentials* use the same system parameters, but they each use a separate issuer key and *iparams*.

System Parameters In addition to the parameters of the MAC scheme, the group elements $(G_{a_1}, G_{a_2}, G_{b_1}, G_{b_2}, G_{j_1}, G_{j_2}, G_{j_3})$ are generated so that the relative discrete logarithms are unknown.

ServerSecretParams and ServerPublicParams The *ServerSecretParams* contains two secret keys for the MAC scheme. The *ServerPublicParams* contains the corresponding issuer parameters, denoted $iparams_A$ (for auth credentials) and $iparams_P$ (for profile key credentials).

5.9 Auth Credentials

An *AuthCredential* has three attributes:

1. $M_1 := \mathsf{HashTo}\mathbb{G}(UID)$,
2. $M_2 := \mathsf{EncodeTo}\mathbb{G}(UID)$, a reversible encoding of *UID*,
3. $M_3 := {G_{m_3}}^{m_3}$, where $m_3 \in \mathbb{Z}_q$, is a "redemption date".

An *AuthCredentialResponse* contains an algebraic MAC for the credential, and also the proof of issuance π_I. The user verifies this proof in the *GetAuthCredential* operation, using attribute values (M_1, M_2) which the user derives from their own UID.

5.10 Profile Key Credentials

A *ProfileKeyCredential* has four attributes. The first two credential attributes encode the UID and are the same as the *AuthCredential*, and the last two encode the *ProfileKey*:

1. $M_1 = \mathsf{HashTo}\mathbb{G}(UID)$,
2. $M_2 = \mathsf{EncodeTo}\mathbb{G}(UID)$,
3. $M_3 = \mathsf{HashTo}\mathbb{G}(ProfileKey, UID)$, a hash of the profile key and UID,
4. $M_4 = \mathsf{EncodeTo}\mathbb{G}(ProfileKey)$, an encoding of the profile key.

Blind Issuance Issuance of *ProfileKeyCredentials* differs from *AuthCredentials* because the *ProfileKeyCredential* attribute values are not all known to the server.

Instead, the user and server will perform a blind issuance protocol, based on the same idea as in [CMZ14]. The *ProfileKeyCredentialRequest* will contain an Elgamal encryption of the blinded attributes (M_3, M_4) and a proof that these values match a *ProfileKeyCommitment*.

A *ProfileKeyCommitment* commits to the values M_3 and M_4. Since M_3 and M_4 are group elements and not scalars, we can't simply use Pedersen's commitment scheme. Instead, a *ProfileKeyCommitment* is the triple of values $(J_1, J_2, J_3) = ({G_{j_1}}^{j_3} M_3, {G_{j_2}}^{j_3} M_4, {G_{j_3}}^{j_3})$ where $j_3 = \mathsf{HashTo}\mathbb{Z}_q(\textit{ProfileKey}, \textit{UID})$. Note that this commitment scheme is not perfectly hiding, but since *ProfileKeys* are assumed to have high min-entropy, this is sufficient. Further, the commitment is deterministic since (M_3, M_4, j_3) are derived from the *ProfileKey* and *UID*, thus any user with these values can reconstruct the *ProfileKeyCommitment*.

Upon receiving a *ProfileKeyCredentialRequest* that matches the user's *ProfileKeyCommitment* the server will use the homomorphic properties of Elgamal encryption to create an encrypted MAC and return it to the user, along with a proof of correctness, in a *ProfileKeyCredentialResponse*. The user will verify the proof and then decrypt the MAC to recover their *ProfileKeyCredential*.

To generate the *ProfileKeyCredentialRequest* the user generates an Elgamal key pair $(y, Y = G^y)$, where G is a generator of $\mathbb{G}$. The blind attributes (M_3, M_4) are encrypted as

$$(D_1, D_2) = (G^{r_1}, Y^{r_1} M_3)$$
$$(E_1, E_2) = (G^{r_2}, Y^{r_2} M_4)$$

for random r_1 and r_2. The *ProfileKeyCredentialRequest* contains the ciphertexts, the public key Y, and a proof that the encrypted values match the commitment $(J_1, J_2, J_3) = ({G_{j_1}}^{j_3} M_3, {G_{j_2}}^{j_3} M_4, {G_{j_3}}^{j_3})$:

$$\begin{aligned}\pi_{BR} = \mathrm{PK}\{(y, r_1, r_2, j_3) : \\ & Y = G^y \wedge D_1 = G^{r_1} \wedge E_1 = G^{r_2} \wedge J_3 = {G_{j_3}}^{j_3} \wedge \\ & D_2 / J_1 = Y^{r_1} / {G_{j_1}}^{j_3} \wedge \\ & E_2 / J_2 = Y^{r_2} / {G_{j_2}}^{j_3}\}\end{aligned}$$

To create a *ProfileKeyCredentialResponse* after verifying the *ProfileKeyCredentialRequest* the server will create a partial credential (t, U, V') that covers the unblinded attributes, and encrypt V' with the user's public key Y to get $(R_1, R_2) = (G^{r'}, Y^{r'} V')$ for a random r'. Then the server will compute

$$(S_1, S_2) = ({D_1}^{y_3} {E_1}^{y_4} R_1, {D_2}^{y_3} {E_2}^{y_4} R_2) \ .$$

Because Elgamal encryption is homomorphic, the ciphertext (S_1, S_2) is an encryption of V for a credential (t, U, V) which covers both blinded and revealed attributes. With the attributes $(M_1, \ldots, M_4)$ as described above, (S_1, S_2) will be:

$$\begin{aligned} S_1 &= G^{y_3 r_1 + y_4 r_2 + r'}, \\ S_2 &= Y^{y_3 r_1 + y_4 r_2 + r'} W U^{x_0 + t x_1} \prod_{i=1}^{4} M_i^{y_i} \\ &= Y^{y_3 r_1 + y_4 r_2 + r'} V \end{aligned}$$

The server can prove that (S_1, S_2) were calculated correctly by modifying the issuance proof to be the following proof π_{BI}:

$$\begin{aligned} \pi_{BI} = \mathrm{PK}\{(w, w', y_1, \ldots, y_4, x_0, x_1, r') : \\ C_W &= {G_w}^w {G_{w'}}^{w'} \ \wedge \\ I &= \frac{G_V}{{G_{x_0}}^{x_0} {G_{x_1}}^{x_1} {G_{y_1}}^{y_1} \ldots {G_{y_4}}^{y_4}} \ \wedge \\ S_1 &= {D_1}^{y_3} {E_1}^{y_4} G^{r'} \ \wedge \\ S_2 &= {D_2}^{y_3} {E_2}^{y_4} Y^{r'} {G_w}^w (U^{x_0})(U^t)^{x_1} {M_1}^{y_1} {M_2}^{y_2} \} \end{aligned}$$

The server sends $(S_1, S_2, t, U, \pi_{BI})$ to the user, and if π_{BI} is valid, the user decrypts $V = S_2 / {S_1}^y$ and outputs the credential (t, U, V) with attributes $(M_1, \ldots, M_4)$.

5.11 Verifiable Encryption of UIDs and Profile Keys

Encryption of UIDs and *ProfileKeys* is done with the symmetric-key scheme from Section 4.1. Both encryption and decryption use the *GroupSecretParams*. The *GroupSecretParams* are $(a_1, a_2, b_1, b_2) \in {\mathbb{Z}_q}^4$ derived from a randomly-chosen *GroupMasterKey*. The *GroupPublicParams* are (A, B) where $A = {G_{a_1}}^{a_1} {G_{a_2}}^{a_2}$ and $B = {G_{b_1}}^{b_1} {G_{b_2}}^{b_2}$.

Encryption of UIDs Recall that $M_1 = \mathsf{HashTo}\mathbb{G}(\mathit{UID})$ and $M_2 = \mathsf{EncodeTo}\mathbb{G}(\mathit{UID})$. To encrypt a *UID* to a *UidCiphertext* (E_{A_1}, E_{A_2}) calculate:

$$\begin{aligned} E_{A_1} &= {M_1}^{a_1} \\ E_{A_2} &= {E_{A_1}}^{a_2} M_2 \end{aligned}$$

To decrypt the *UidCiphertext* first compute:

$$M_2' = E_{A_2} / {E_{A_1}}^{a_2}$$

then decode M_2' to get UID', and compute $M_1' = \mathsf{HashTo}\mathbb{G}(UID')$. Then perform the following checks and return UID' if they succeed, $\perp$ otherwise:

$$E_{A_1} \stackrel{?}{\neq} 1$$
$$E_{A_1} \stackrel{?}{=} (M_1')^{a_1}$$

Encryption of ProfileKeys Recall that $M_3 = \mathsf{HashTo}\mathbb{G}(ProfileKey, UID)$ and $M_4 = \mathsf{EncodeTo}\mathbb{G}(ProfileKey)$. To encrypt a *ProfileKey* as a *ProfileKeyCiphertext* (E_{B_1}, E_{B_2}) calculate:

$$E_{B_1} = {M_3}^{b_1}$$
$$E_{B_2} = {E_{B_1}}^{b_2} M_4$$

To decrypt the *ProfileKeyCiphertext* first compute:

$$M_4' = E_{B_2} / {E_{B_1}}^{b_2}$$

then decode M_4' to get $ProfileKey'$, compute $M_3' = \mathsf{HashTo}\mathbb{G}(ProfileKey', UID)$. Then perform the following checks and return $ProfileKey'$ if they succeed, $\perp$ otherwise:

$$E_{B_1} \stackrel{?}{\neq} 1$$
$$E_{B_1} \stackrel{?}{=} (M_3')^{b_1}$$

5.12 Presenting an AuthCredential

An *AuthCredentialPresentation* contains a *UidCiphertext*, a redemption date, and a proof of knowledge calculated as follows:

1. Recompute (E_{A_1}, E_{A_2}) from UID and (a_1, a_2) as described in Section 5.11.
2. Choose $z \in_R \mathbb{Z}_q$ and compute

$$C_{y_1} = {G_{y_1}}^z M_1 \qquad C_{x_0} = {G_{x_0}}^z U$$
$$C_{y_2} = {G_{y_2}}^z M_2 \qquad C_{x_1} = {G_{x_1}}^z U^t$$
$$C_{y_3} = {G_{y_3}}^z \qquad C_V = {G_V}^z V$$

   along with two values in $\mathbb{Z}_q$: $z_0 = -zt$ and $z_1 = -za_1$.

3. Compute the following proof of knowledge:

$$\begin{aligned}\pi_A = \mathrm{PK}\{(z, sk, z_0, z_1, t) : \\ & Z = I^z \ \wedge \\ & C_{x_1} = {C_{x_0}}^t {G_{x_0}}^{z_0} {G_{x_1}}^z \ \wedge \\ & A = {G_{a_1}}^{a_1} {G_{a_2}}^{a_2} \ \wedge \\ & C_{y_2}/E_{A_2} = {G_{y_2}}^z / {E_{A_1}}^{a_2} \ \wedge \quad \text{//plaintext is } M_2 \\ & E_{A_1} = {C_{y_1}}^{a_1} {G_{y_1}}^{z_1} \ \wedge \quad \text{//} E_{A_1} \text{ is well-formed} \\ & C_{y_3} = {G_{y_3}}^z \ \}\end{aligned}$$

4. Output $(C_{x_0}, C_{x_1}, C_{y_1}, \ldots, C_{y_3}, C_V, E_{A_1}, E_{A_2}, \pi_A)$

5. The server computes

$$Z = C_V / (W {C_{x_0}}^{x_0} {C_{x_1}}^{x_1} {C_{y_1}}^{y_1} {C_{y_2}}^{y_2} (C_{y_3} {G_{m_3}}^{m_3})^{y_3})$$

using the current date m_3 and the secret key $(W, x_0, x_1, y_1, \ldots, y_3)$, and then verifies π_A.

5.13 Presenting a ProfileKeyCredential

A *ProfileKeyCredentialPresentation* contains a *UidCiphertext*, a *ProfileKeyCiphertext*, and a proof of knowledge calculated as follows:

1. Choose random z, r then compute

$$\begin{aligned} & C_{y_i} = {G_{y_i}}^z M_i \text{ for } i = 1, \ldots, 4 && C_{x_0} = {G_{x_0}}^z U \\ & C_V = {G_V}^z V && C_{x_1} = {G_{x_1}}^z U^t \end{aligned}$$

along with three values in Z_q: $z_0 = -zt$, $z_1 = -za_1$, and $z_2 = -zb_1$.

2. Then compute the proof of knowledge

$$\begin{aligned}\pi_P = \mathrm{PK}\{(sk, z, z_0, z_1, z_2, t) : \\ & Z = I^z \wedge \\ & C_{x_1} = {C_{x_0}}^t {G_{x_0}}^{z_0} {G_{x_1}}^z \wedge \\ & A = {G_{a_1}}^{a_1} {G_{a_2}}^{a_2} \ \wedge \\ & B = {G_{b_1}}^{b_1} {G_{b_2}}^{b_2} \ \wedge \\ & C_{y_2}/E_{A_2} = {G_{y_2}}^z / {E_{A_1}}^{a_2} \ \wedge \quad \text{//plaintext is } M_2 \\ & E_{A_1} = {C_{y_1}}^{a_1} {G_{y_1}}^{z_1} \ \wedge \quad \text{//} E_{A_1} \text{ is well-formed} \\ & C_{y_4}/E_{B_2} = {G_{y_4}}^z / {E_{B_1}}^{b_2} \ \wedge \quad \text{//plaintext is } M_4 \\ & E_{B_1} = {C_{y_3}}^{b_1} {G_{y_3}}^{z_2} \quad \text{//} E_{B_1} \text{ is well-formed} \\ & \}\end{aligned}$$

and output $(\{C_{y_i}\}_{i=1}^4, C_{x_0}, C_{x_1}, C_V, \pi_P)$.

3. The server computes

$$Z = C_V/(W{C_{x_0}}^{x_0}{C_{x_1}}^{x_1}\prod_{i=1}^{4}{C_{y_i}}^{y_i})$$

using the secret key $(W, x_0, x_1, y_1, \ldots, y_4)$, and then verifies π_P.

6 Implementation

This system has been implemented and is undergoing testing before being deployed to Signal users. This implementation instantiates the cryptography as follows:

- $\mathbb{G}$: For the group $\mathbb{G}$ we use `ristretto255` [HdVLA19], a prime-order group built on top of the (non-prime-order) `Curve25519` elliptic curve [Ber06].

- $\mathsf{HashTo}\mathbb{G}$: For most $\mathsf{HashTo}\mathbb{G}$ operations we use the `HashToGroup` operation defined for Ristretto in [HdVLA19]. This operation take a 64-byte hash output and converts each of the 32-byte halves to a *field element* (an integer module $2^{255}-19$). These field elements are converted to group elements using the Ristretto version of the Elligator2 map, and then these groups elements are added together to ensure this operation is a surjection onto the entire Ristretto group. $\mathsf{HashTo}\mathbb{G}$ for M_4 is handled differently for performance reasons; see next bullet.

- $\mathsf{EncodeTo}\mathbb{G}$: For $\mathsf{EncodeTo}\mathbb{G}$ we have to contend with the fact that the "inverse" of the Elligator map on Ristretto will return from one to eight field elements, only one of which is the value that was originally encoded. Since our encryption scheme MACs the plaintext before encoding (to compute the E_1 value), decryption tests the candidates and returns the correct value.

  Encoding a 256-bit profile key into a single Ristretto group element is not possible, as the Ristretto group order is less than 2^{256}. Thus, we can't decode the E_{B_2} element of a profile-key ciphertext directly into a single profile key; rather we decode it into 64 different candidates, and then test whether ${E_{B_1}}^{1/b_1} == \mathsf{HashTo}\mathbb{G}(p)$ for each candidate profile key p. For efficiency, we define this $\mathsf{HashTo}\mathbb{G}$ operation to comprise only a single Elligator map on 32 bytes, since having it cover the entire Ristretto group is unnecessary.

- *Zero-knowledge proofs of knowledge*: The proofs of knowledge are implemented using the "generic linear" generalization of Schnorr's protocol described in [BS20, Ch. 19], made noninteractive with the Fiat-Shamir transform [FS87].

- *Hash functions*: For hashing and key-derivation within the proofs of knowledge and elsewhere we use `HMAC-SHA256` within a new "stateful hash object" construction which provides labels for domain-separation, extensible output, and other convenient features.

The implementation is available online under an open source license (GPL), in the form of three libraries written in Rust:

- `zkgroup`: This is the main library, providing a high-level API and bindings so the library can be used in other programming languages.
- `poksho`: This library implements what we call the `POKSHO` construction for zero-knowledge proofs of knowledge; and the `SHO/HMAC-SHA256` construction for hashing.
- `curve25519-dalek`: We modified the `curve25519-dalek` library from Henry de Valence and Isis Lovecruft to add support for Elligator inverse and decoding byte strings from Ristretto elements, as discussed above.

Table 1 lists the main objects the `zkgroup` library deals with, and gives their sizes and the amount of time taken to produce (i.e., issue a credential, encrypt a ciphertext) or consume (i.e., verify, decrypt) them on a ThinkPad X1 Carbon laptop with an Intel Core i7-8650U processor.

The listed times cover encryption and decryption for the ciphertext objects, and additionally cover creating and verifying proofs of knowledge for the other objects. The presentation objects contain ciphertext objects, so the ciphertext costs in space and time are included for *AuthCredentialPresentation* and *ProfileKeyCredentialPresentation*. For comparison, each Ristretto group element or scalar is stored in 32 bytes. A variable-base scalar multiplication on this computer takes around 60 microseconds, while decoding or encoding a group element to 32 bytes, or applying the Elligator map, takes around 4 microseconds. This implementation is well-optimized for object sizes but could be further optimized for time.

The most expensive operations are *FetchGroupMembers* when the number of users in the group is large, or adding a large number of users to a group. For a group with n users, *FetchGroupMembers* requires users download about $128n$ bytes and spend about $n(0.86 + 0.18)$ ms to decrypt. Adding members scales linearly with the number of members added, with 713 bytes and 2.87 ms required to produce each *ProfileKeyCredentialPresentation*.

The small ciphertexts are a notable feature of this system. We only need a 64-byte ciphertext to encrypt a 16-byte UID or a 32-byte profile key. We had prototyped an alternative using the MAC from [CMZ14] with exponential Elgamal encryption where plaintexts are MAC'd and encrypted in 16-bit chunks (so that the discrete log computations required for decryption are practical). The ciphertext (including a MAC) for a 16 byte UID was 600+ bytes.

Object *name*	**Size** *bytes*	**Time to produce** *milliseconds*	**Time to consume** *milliseconds*
UidCiphertext	64	0.13	0.18
ProfileKeyCiphertext	64	0.13	0.86
AuthCredentialResponse	361	1.95	0.98
AuthCredentialPresentation	493	2.16	1.17
ProfileKeyCredentialRequest	329	1.48	-
ProfileKeyCredentialResponse	457	2.70	0.97
ProfileKeyCredentialPresentation	713	2.87	1.53

Table 1: Benchmarks of the main operations in the Signal Private Group System. The sizes are in bytes and the times are in milliseconds, measured on an Intel Core i7-8650U processor. Producing a *ProfileKeyCredentialResponse* involves verifying a *ProfileKeyCredentialRequest*, so the times are combined.

7 Security Analysis

In this section we analyze the security of: the encryption scheme defined in Section 4.1, our new algebraic MAC from Section 3.1, and the security of the keyed-verification anonymous credential system built on top of the MAC.

We then discuss security of the system as a whole in Section 7.5 with respect to the ideal functionality in Appendix A. In [CPZ19, Appendix B], we also sketch a simulation-based security proof for the system.

7.1 Weak PRFs and f_k

Below we give a definition of weak pseudorandom functions (wPRF) [NR95], tailored to our setting, and define a specific wPRF which we will use in analyzing encryption and credential security.

Definition 4. *Let $\mathbb{G}$ be a group of prime order q. A function $f_k : \mathbb{G} \to \mathbb{G}$ with key $k \in \mathbb{Z}_q$ is said to be a weak pseudorandom function (wPRF), if the following two sequences (of length polynomial in κ) are indistinguishable*

$$(x_1, f_k(x_1)), (x_2, f_k(x_2)), \ldots$$

and

$$(x_1, r_1), (x_2, r_2), \ldots$$

where x_i and r_i are sampled from the uniform distribution on $\mathbb{G}$.

Weak PRFs are useful because they are PRFs when the inputs are chosen at random. The specific wPRF we use in our security analysis is the function $f_k : \mathbb{G} \to \mathbb{G}$ defined as

$f_k(x) = x^k$. The fact that f_k is a wPRF is known in the literature, and for completeness we include a proof (similar to the one in [NPR99]).

Theorem 5. *Let $\mathbb{G}$ be a group of prime order q, with generator g. Then the family of functions $f_k : \mathbb{G} \to \mathbb{G}$ defined as $f(x) = x^k$ is a wPRF with key $k \in \mathbb{Z}_q$ if the DDH problem is hard in $\mathbb{G}$.*

Proof. Suppose $\mathcal{A}$ is an adversary that can distinguish the sequences in Definition 4 with probability ϵ. We construct an algorithm $\mathcal{B}$ for the DDH problem that uses $\mathcal{A}$ as a subroutine. Suppose that DDH is ϵ_{ddh}-hard in $\mathbb{G}$, i.e., no polynomial time algorithm exists for DDH that succeeds with probability better than ϵ_{ddh}.

The DDH instance (A, B, C) is provided as input to $\mathcal{B}$, who must decide whether this triple has the form (G^a, G^b, G^{ab}) or (G^a, G^b, G^r) for a random r (here $G \in \mathbb{G}$ is part of the group description). $\mathcal{B}$ constructs a wPRF instance for $\mathcal{A}$ as follows:

$$(B, C), (G^{r_1}, A^{r_1}), (G^{r_2}, A^{r_2}), \ldots$$

where r_i are chosen uniformly at random in $\mathbb{Z}_q$. When $\mathcal{B}$ is given a DDH triple, the sequence is

$$(G^b, G^{ab}), (G^{r_1}, G^{ar_1}), (G^{r_2}, G^{ar_2}), \ldots$$

which is a valid sequence of random group elements, and their images under f_a. When $\mathcal{B}$ is given a non-DDH triple, the sequence is

$$(G^b, G^r), (G^{r_1}, G^{ar_1}), (G^{r_2}, G^{ar_2}), \ldots,$$

By a hybrid argument, $\mathcal{B}$ can replace the first pair in the sequence with a random one, and $\mathcal{A}$'s success probability will differ by at most ϵ_{ddh}. We can argue similarly that the remaining elements in the sequence are indistinguishable from random. □

7.2 Security of Encryption

In this section we define the security notions for deterministic symmetric-key encryption with public verifiability. Unique ciphertexts (Definition 8) and correctness under adversarially chosen keys (Definition 9) are new to this work, while CPA, CCA, plaintext integrity (PI) and ciphertext integrity (CI) are direct adaptations of definitions from [BS20] to deterministic encryption.

We show that the encryption scheme of §4.1 is CPA secure and has ciphertext integrity, properties that together imply CCA security and authenticated encryption [BS20, Definition 9.2]. We prove CPA-security directly. We prove CI by proving unique ciphertexts and PI (which together imply CI).

Deterministic Encryption We start by showing that the encryption scheme of §4.1 is CPA secure. Our definition uses a real-or-random experiment [BDJR97], and to model the deterministic property, the encryption oracle can only be queried once per plaintext. By adding the same restriction on encryption queries, we can adapt the common CCA security definition (see, e.g., [BS20, Definition 9.5]) to the deterministic case.

Definition 6. *For a deterministic symmetric key cipher with public verifiability* ($\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}$)*, we define CPA security by the following security game.*

- *The challenger generates* $(k, pk) \leftarrow \mathsf{KeyGen}(1^\kappa)$*, and a random bit* b*.*
- *The attacker is given* pk *and an oracle* $\mathcal{O}_k(\cdot)$ *that outputs* $\mathsf{Enc}_k(\cdot)$ *when* $b = 0$ *and* $\mathsf{Enc}_k(r)$ *for uniformly random* r *(of the same length) when* $b = 1$*.* $\mathcal{O}_k$ *outputs* $\perp$ *if the input was previously queried.*
- $\mathcal{A}$ *outputs a guess bit* b' *and wins if* $b = b'$*.*

We say the encryption scheme is CPA-secure if $\mathcal{A}$ wins with probability non-negligibly different from $1/2$.

In the following proof and throughout this section, we use the shorthand H to denote $\mathsf{HashTo}\mathbb{G}$, and N to denote $\mathsf{EncodeTo}\mathbb{G}$.

Theorem 7. *The encryption scheme of Section 4.1 is CPA secure for deterministic encryption, in the random oracle model, assuming the DDH problem is hard in* $\mathbb{G}$*.*

Proof. Let $\mathcal{A}$ be an attacker in the CPA game. We construct a DDH distinguisher $\mathcal{B}$ that uses $\mathcal{A}$ as a subroutine. We proceed with a hybrid argument. Let G_i be the probability that $\mathcal{A}$ outputs 1 in Game i.

Game 0 This is the real CPA game, where $\mathcal{B}$ is the challenger, and H is modeled as a random oracle. The probability that $\mathcal{A}$ breaks CPA security of the scheme is G_0.

Game 1 is the same as Game 0, but $\mathcal{B}$ replaces pk with a random value. We claim that $\mathsf{G}_1 - \mathsf{G}_0 \leq \epsilon_{\text{ddh}}$. Let $\mathcal{B}$ have a DDH triple as input, $(A, B, C) = (G^{a_1}, G^b, G^{a_1 b}$ or $G^z)$ for a random $z \in \mathbb{Z}_q$. $\mathcal{B}$ chooses a_2 at random and creates pk, by first programming H so that $G_{a_1} = B = G^b$ (this is possible since G_{a_1} is derived using H). Then $\mathcal{B}$ computes $pk = C{G_{a_2}}^{a_2}$. On hash queries $H(m)$, $\mathcal{B}$ outputs G^r for random r and stores (m, r). To answer $\mathsf{Enc}(m)$ queries, $\mathcal{B}$ programs H (or has already) so that $H(m) = G^r$, then $A^r = G^{a_1 r} = H(m)^{a_1}$. $\mathcal{B}$ outputs $(E_1, E_2) = (A^r, (E_1)^{a_2} N(m))$.

When the DDH triple is real, games G_0 and G_1 are identical, and when the triple is random, pk is uniformly distributed in $\mathbb{G}$. The output of Enc queries is always the same in both games because it doesn't depend on C. Therefore, $\mathsf{G}_1 - \mathsf{G}_0 \leq \epsilon_{\text{ddh}}$.

Game 2 is the same as Game 1, except $\mathcal{B}$ replaces E_1 with a random value when $\mathcal{A}$ makes an Enc query. $\mathcal{B}$ chooses a_2 at random, and acts as a wPRF attacker, for an instance where a_1 is the secret. When $\mathcal{A}$ queries $H(m)$, $\mathcal{B}$ queries the wPRF oracle to get (U, U'). $\mathcal{B}$ programs $H(m) = U$, then outputs $(E_1, E_2) = (U', (E_1)^{a_2} N(m))$. Note that since m never repeats, U is a fresh random group element for every Enc query with overwhelming probability. When the wPRF oracle outputs real pairs then $\mathcal{B}$ outputs $E_1 = U' = U^{a_1} = H(m)^{a_1}$, and $\mathsf{G}_2 = \mathsf{G}_1$. When the wPRF output is random, then E_1 is uniformly distributed in $\mathbb{G}$. Therefore $\mathcal{B}$ is a distinguisher for the wPRF game (and hence DDH) with probability $\mathsf{G}_2 - \mathsf{G}_1 \leq \epsilon_{\text{ddh}}$.

Game 3 is the same as 2 but now E_2 is replaced with a random value. $\mathcal{B}$ does not use a_1, and again plays the wPRF game, this time for an instance with secret a_2. When $\mathcal{A}$ makes an $\mathsf{Enc}(m)$ query, $\mathcal{B}$ queries the wPRF oracle to get (U, U'), and $\mathcal{B}$ outputs $(E_1, E_2) = (U, U'N(m))$. E_1 is uniformly distributed in both games 2 and 3. For E_2, when the wPRF output is real, we have $E_2 = (E_1)^{a_2} N(m)$, exactly as in Game 2, and when the wPRF output is random, E_2 is uniformly distributed. Therefore $\mathsf{G}_3 - \mathsf{G}_2 \leq \epsilon_{\text{ddh}}$.

In Game 3 $\mathcal{B}$ no longer uses m. By a union bound

$$\Pr[\mathcal{A} \text{ wins the CPA game }] \leq 3\epsilon_{\text{ddh}}$$

□

Now we formally define the unique ciphertexts property.

Definition 8. *We say a symmetric-key encryption scheme* (KeyGen, Enc, Dec) *has* unique ciphertexts *if for all polynomial-time $\mathcal{A}$,*

$$\Pr[(k, c_1, c_2) \leftarrow \mathcal{A}(1^\kappa) : c_1 \neq c_2 \ \wedge \ \mathsf{Dec}_k(c_1) = \mathsf{Dec}_k(c_2) \neq \perp]$$

is negligible in κ.

Next we define correctness under adversarially chosen keys. Our definition refers to the Derive hash function from our construction, used to derive the secret key from a seed.

Definition 9. *We say a symmetric-key encryption scheme* (KeyGen, Enc, Dec) *is* correct under adversarially chosen keys *if for all polynomial-time $\mathcal{A}$,*

$$\Pr[(k_0, m) \leftarrow \mathcal{A}(1^\kappa) : sk = \mathsf{Derive}(k_0) \ \wedge \ \mathsf{Dec}_{sk}(\mathsf{Enc}_{sk}(m)) \neq m]$$

is negligible in κ.

Now we prove that our encryption scheme has unique ciphertexts (Definition 8) and is correct under adversarially chosen keys(Definition 9).

Theorem 10. *The encryption scheme of Section 4.1 has unique ciphertexts, and is correct under adversarially chosen keys assuming* $\mathsf{HashTo}\mathbb{G}$ *and* Derive *are random functions.*

Proof. First we prove that the scheme has unique ciphertexts. Because decryption recomputes E_1' from m, and this step is deterministic, for each m there is exactly one value $E_1 = E_1'$ such that decryption will succeed. For a pair of ciphertexts (E_1, E_2) and (E_1^*, E_2^*) that decrypt successfully to the same m, then $m = \mathsf{DecodeFrom}\mathbb{G}(E_2/{E_1}^{a_2}) = \mathsf{DecodeFrom}\mathbb{G}(E_2^*/(E_1^*)^{a_2})$. Provided $\mathsf{DecodeFrom}\mathbb{G}$ is invertible then $E_2/{E_1}^{a_2} = E_2^*/(E_1^*)^{a_2}$, and since $E_1 = E_1^*$ then $E_2 = E_2^*$.

Note that this proof continues to hold if $\mathsf{DecodeFrom}\mathbb{G}$ returns a set of group elements, only one of which is correct, provided that the image of $\mathsf{DecodeFrom}\mathbb{G}$ consists of disjoint sets. In this case there will still be a single valid E_1 for each m, a single M_2' which decodes to a set containing m, and thus a single valid E_2.

Now for correctness under adversarially chosen keys. We argue that no efficient $\mathcal{A}$ can output (k_0, m) such that $k = \mathsf{Derive}(k_0)$ and $\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m' \neq m$. By definition Enc_k succeeds for any k and m in range, therefore Enc always outputs some (E_1, E_2), where $E_2 = E_1^{a_2} N(m)$. Dec computes $N(m) = E_2/E_1^{a_2}$, so $N(m)$ must be the same. When $N(m)$ is the same, Dec can only fail if $E_1 = H(m)^{a_1} = 1$, which occurs with negligible probability since H and Derive are random functions. □

The next lemma establishes that $E_1 = H(m)^{a_1}$, is a MAC on m, since our plaintext integrity proof will depend on this fact.

Lemma 11. *The function* $F : \mathbb{Z}_q \times \{0,1\}^* \to \mathbb{G}$*, defined* $F_{a_1}(m) := H(m)^{a_1}$ *is an* $\mathsf{suf\text{-}cma}$*-secure MAC (with key* a_1*) if* H *is a random oracle, and the DDH problem is hard in* G*. The public parameters for the MAC include* $pk = {G_{a_1}}^{a_1} {G_{a_2}}^{a_2}$*.*

Proof. We show that $F_{a_1}(m) := H(m)^{a_1}$ is a PRF, assuming H is a random oracle and that DDH is hard in $\mathbb{G}$. Then since $F_{a_1}(m)$ is a deterministic PRF, it is an $\mathsf{suf\text{-}cmva}$-secure MAC (where verification just recomputes the MAC from the key and message and compares it to the received MAC). Let $\mathcal{A}$ be a PRF adversary for $F_{a_1}(m)$, we define an algorithm $\mathcal{B}$ that is a DDH distinguisher using $\mathcal{A}$ as a subroutine. Let Game 0 be the real PRF security game, where $\mathcal{B}$ acts as challenger to $\mathcal{A}$. Let Game 1 be identical to Game 0, but pk is replaced with a random value. This transition is the same as Game 0 to 1 in the CPA security proof (§7); by the same argument, Games 0 and 1 are indistinguishable if DDH is hard in $\mathbb{G}$.

In Game 2, $\mathcal{B}$ acts as an adversary for the wPRF instance f_{a_1}, and no longer chooses a_1. When $\mathcal{A}$ makes a hash query $H(m)$, $\mathcal{B}$ queries the wPRF oracle, to get the pair U, U', and sets $H(m) = U$, and stores $\sigma_m := U'$ as the MAC on m. When $\mathcal{A}$ makes a PRF query on a message m, $\mathcal{B}$ queries $H(m)$ if m has not already been queried, then returns σ_m to $\mathcal{A}$. When $\mathcal{A}$ outputs 0 or 1 (indicating real or random), $\mathcal{B}$ outputs the same for the wPRF instance.

When the wPRF instance is real, $\mathcal{B}$'s output is $\sigma_m = U' = U^{a_1} = H(m)^{a_1}$ (since $H(m)$ was defined as U), identical to Game 0. But when the wPRF instance is random, $\mathcal{B}$ outputs a random value. Therefore, $\mathcal{A}$'s success probability in Game 2 is bounded by ϵ_{ddh}. □

As with the definition of CPA security, we specialize the definitions of ciphertext integrity and plaintext integrity to our setting, since these will be used in our analysis of CCA security (Theorem 13).

Definition 12. *(PI and CI security) For a deterministic symmetric key cipher with public verifiability* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$*, define the following game.*

- *The challenger generates* $(k, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$*, and sends* pk *to the adversary* $\mathcal{A}$*.*
- $\mathcal{A}$ *makes several encryption queries, submitting distinct messages* m_i*. The challenger computes* $c_i = \mathsf{Enc}(k, m_i)$ *and returns* c_i *to* $\mathcal{A}$*.*
- *Eventually* $\mathcal{A}$ *outputs a candidate ciphertext* c^**.*

We say that $\mathcal{A}$ *wins the ciphertext integrity game if* $c^* \neq c_i$ *for any* i*, and* $\mathsf{Dec}(k, c_i) \neq \perp$*. We say that* $\mathcal{A}$ *wins the plaintext integrity game if* $\mathsf{Dec}(k, c^*) = m^*$ *and* $m^* \neq m_i$ *for any* i*. The encryption scheme is CI secure (or PI secure) if* $\mathcal{A}$ *wins the ciphertext integrity (or plaintext integrity) game with negligible probability.*

Theorem 13. *The encryption scheme of Section 4.1 is CCA secure in the random oracle model, assuming the DDH problem is hard in* $\mathbb{G}$*.*

Proof. First note that the unique ciphertexts property combined with PI implies CI. Then, if an encryption scheme is both CI and CPA secure, then it is also CCA secure [BS20, §9.2.3]. We've proven CPA security in Theorem 7 and unique ciphertexts in Theorem 10, what remains is to prove that the scheme has PI.

Our proof will use the fact that E_1 is a MAC on the plaintext, namely that $H(m)^{a_1}$ is a MAC on m with key a_1 (as shown in Lemma 11). Since E_1 is recomputed during decryption, a successful PI attacker must compute a valid MAC on the plaintext.

Now consider the PI security game, where $\mathcal{B}$ is the challenger and $\mathcal{A}$ is an adversary. $\mathcal{B}$ will choose a and act as an attacker in the $\mathsf{uf\text{-}cma}$ security game for the MAC $H(m)^{a_1}$. To compute the public key, $\mathcal{B}$ gets the MAC σ_t from its MAC oracle, where t is the string used to derive G_{a_1}. Then $\mathcal{B}$ computes $pk = {G_{a_2}}^{a_2}\sigma_t$.

$\mathcal{B}$ initializes $\mathcal{A}$ with pk and must answer $\mathcal{A}$'s hash and encrypt queries. $\mathcal{B}$ forwards $\mathcal{A}$'s hash queries to it's own hash oracle, and relays the output to $\mathcal{A}$. When $\mathcal{A}$ makes an encrypt query for plaintext m, $\mathcal{B}$ queries the MAC oracle to get $\sigma_m := H(m)^{a_1}$, then computes $E_1 = \sigma_m$ and $E_2 = (E_1)^{a_2}m$, and outputs (E_1, E_2) to $\mathcal{A}$. When $\mathcal{A}$ outputs a ciphertext (E_1^*, E_2^*), then $\mathcal{B}$ partially decrypts $m^* = E_2^*/(E_1^*)^{a_2}$ and outputs E_1^* as a MAC on m^*.

We now argue that if $\mathcal{A}$ wins the PI game, $\mathcal{B}$ is a successful forger. If $\mathcal{A}$ wins the PI game, then the winning conditions of the game ensure that

1. $\mathcal{B}$ never queried m^* to its MAC oracle (since $\mathcal{A}$ did not query m^* in an encryption query), and

2. (E_1^*, E_2^*) decrypts correctly.

Since decryption re-computes E_1^* from m^* and the secret key, it must be that $E_1^* = H(m^*)^{a_1}$, and so $\mathcal{B}$'s output is a valid MAC on m^*. □

7.3 Security of our New MAC

In this section we analyze the security of our new MAC.

Theorem 14. *The MAC defined in Section 3.1 is* suf-cmva *secure in the random oracle model, assuming the DDH problem is hard in* $\mathbb{G}$*, and that the* $\mathsf{MAC}_{\mathsf{GGM}}$ *construction is* uf-cma *secure.*

Proof. Using Lemma 3, we can ignore verification queries and prove that the MAC is suf-cma secure.

We consider three possible types of forgeries, and show that each can occur with at most negligible probability. Recall that the forged MAC on message M^* consists of three values (t^*, U^*, V^*), and let M_i and (t_i, U_i, V_i) be the message used and MACs resulting from the adversary's MAC oracle queries. In Type 1 forgeries, $t^* \neq t_i$ for any i. In Type 2 forgeries, there exists a previous query i such that $t^* = t_i$, but $M^* \neq M_i$. (Note that since t is chosen freshly at random for each MAC produced by the oracle, there will be at most one such i.) Finally, in Type 3 forgeries, there exists a previous query i such that $t^* = t_i$ and $M^* = M_i$, but $U^* \neq U_i$. Note that since V^* is fully defined by M^*, U^*, t^*, this covers all possible forgeries in the suf-cma game. Let $\mathcal{A}$ be an attacker who plays the suf-cma security game.

Type 1 (t^* was not output by the MAC oracle) Suppose that there is an attacker $\mathcal{A}$ that can produce a Type 1 forgery in the suf-cma game with non-negligible probability. In this case, $\mathcal{A}$'s forgery uses a new tag value t^*, i.e., one that has not been output by in response to a MAC query from $\mathcal{A}$.

We construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ as a subroutine. $\mathcal{B}$ will be a uf-cma forger for $\mathsf{MAC}_{\mathsf{GGM}}$. $\mathcal{B}$ plays the uf-cma security game for $\mathsf{MAC}_{\mathsf{GGM}}$ with a challenger, who provides a MAC oracle. First $\mathcal{B}$ generates part of the MAC key and the issuer parameters. $\mathcal{B}$ chooses $(w, y_1, \ldots, y_n)$ at random and computes C_W. Then $\mathcal{B}$ chooses a random z and queries $\mathsf{MAC}_{\mathsf{GGM}}(z)$ to get a MAC (U, U'). Since $U' = U^{x_0 + x_1 z} = U^{x_0} U^{x_1 z}$, when the random oracle used to generate parameters is programmed to output $G_{x_0} = U$ and $G_{x_1} = U^z$, $\mathcal{B}$ can create $I = G_V / (G_{y_1}{}^{y_1} \ldots G_{y_n}{}^{y_n} G_{x_0}{}^{x_0} G_{x_1}{}^{x_1})$ as $I = G_V / (G_{y_1}{}^{y_1} \ldots G_{y_n}{}^{y_n} U')$. Now $\mathcal{B}$ initializes $\mathcal{A}$ with the issuer parameters.

For MAC queries, $\mathcal{B}$ computes $\tilde{M} = \prod_{j=1}^{n} M_j{}^{y_j}$. $\mathcal{B}$ chooses a random t, queries $\mathsf{MAC}_{\mathsf{GGM}}(t)$ to get (U, U') and computes the MAC as $(t, U, V = WU'\tilde{M})$. Since $U' = U^{x_0 + x_1 t}$, this MAC is computed correctly.

When $\mathcal{A}$ outputs a forgery, $\mathcal{B}$ can compute $(U^*, V^*/\tilde{M}W)$, and output this as a $\mathsf{MAC}_{\mathsf{GGM}}$ forgery on the message t^*. If t^* was never output in a MAC created by $\mathcal{B}$ it was never queried to the $\mathsf{MAC}_{\mathsf{GGM}}$ oracle, and is therefore a valid $\mathsf{MAC}_{\mathsf{GGM}}$ forgery, if σ^* is a valid MAC and a Type 1 forgery.

Type 2 or Type 3 (t^* was output by the MAC oracle) Suppose that there is an attacker $\mathcal{A}$ that can produce a Type 2 or Type 3 forgery in the suf-cma game with non-negligible probability. We argue that this will allow us to construct a reduction $\mathcal{B}$ to break DDH. We proceed via a series of games:

Game 0 This is the real suf-cma game, with the modification that the adversary wins if the forgery is valid and is of Type 2 or Type 3. By assumption $\mathcal{A}$ produces a Type 2 or Type 3 forgery with non-negligible probability ϵ.

Game 1 This game proceeds as the suf-cma game with the following exception: the game first chooses a random $i^* \in 1 \ldots Q$, where Q is the maximum number of queries that $\mathcal{A}$ can make, and the adversary wins only if $t^* = t_{i^*}$. The adversary will win this game with probability at least ϵ/Q.

Game 2 This game proceeds as in Game 1 with the following exceptions: First, the issuer parameters are chosen at random. Then, on the i^*th query that $\mathcal{A}$ makes to its MAC oracle, the game will respond by running the MAC algorithm. For all other MAC oracle queries, the game will return three random values (t, U, V) in the appropriate groups. The adversary wins if the MAC verifies and $t^* = t_{i^*}$ but either $\vec{M}$ or U is new. Suppose that the adversary's success probability in Game 2 is non-negligibly lower than in Game 1. In this case we build an algorithm $\mathcal{B}$ that breaks DDH.

Let (R, X_1, Z) be a DDH instance in $\mathbb{G}$, that $\mathcal{B}$ will use $\mathcal{A}$ to answer. We use the notation (G^r, G^{x_1}, G^{rx_1}) for a real DDH triple, and replace G^{rx_1} with G^z when Z is random. The base $G \in \mathbb{G}$ is assumed to be different from the parameters used by the MAC scheme.

$\mathcal{B}$ no longer chooses x_0, x_1 in the secret key. Instead $\mathcal{B}$ chooses random $d, t_{i^*} \in Z_q$. The value of x_1 is fixed by X_1 in the DDH instance, and the value x_0 used by $\mathcal{B}$ when creating MACs will be implicitly defined as $x_0 = d - x_1 t^*$.

To create I in the issuer parameters without (x_0, x_1), $\mathcal{B}$ first programs the random oracle so that G_{x_1} is derived as $R^a G^b$ for a random a, b. Then $\mathcal{B}$ computes the term ${G_{x_1}}^{x_1}$ as $Z^a X_1^b$. Similarly, for the term ${G_{x_0}}^{x_0}$, $\mathcal{B}$ programs the random oracle to derive $G_{x_0} = R^{a'} G^{b'}$ for a random a', b'. Then $\mathcal{B}$ computes ${G_{x_0}}^{x_0} = ((R^{a'} G^{b'})^d (Z^{a'} X_1^{b'})^{-t^*}$. Finally $\mathcal{B}$ chooses

C_W at random; since this is a perfectly hiding commitment the distribution is identical to that in the real parameters.

For MAC query i^*, $\mathcal{B}$ chooses random a_{i^*}, b_{i^*} and outputs the MAC

$$(t^*, R^{a_{i^*}} G^{b_{i^*}}, W(R^{a_{i^*}} G^{b_{i^*}})^d \tilde{M}) \tag{1}$$

and for the other query $i \neq i^*$ $\mathcal{B}$ chooses random a_i, b_i, t and outputs

$$(t, R^{a_i} G^{b_i}, W(R^{a_i} G^{b_i})^d (Z^{a_i} X_1^{b_i})^{t-t^*} \tilde{M}) \tag{2}$$

Note that (1) is a special case of (2), since when $t = t^*$ part of (2) cancels out.

By the definition of d, the MAC in (1) is valid and distributed identically to the output of the MAC algorithm. When $Z = G^{rx_1}$, it can be checked that (2) is also valid and distributed identically to the output of the MAC algorithm, as are the issuer parameters. When Z is random, $Z^a X_1^b$ is random and independent of $R^a G^b$, so (2) consists of 3 independent random values. Similarly in this case I is random as well.

When A outputs a forgery (M^*, t^*, U^*, V^*), by assumption it will use a tag output by $\mathcal{B}$, and if the tag is not t^* output in query i^*, $\mathcal{B}$ aborts. If the tag is t^*, B computes $\tilde{M}$ from M^*, uses $\vec{y_i}$ and checks whether $V^*/(U^*)^d = \tilde{M}W$. If the comparison fails, the forgery is invalid, and $\mathcal{B}$ outputs "random" to the DDH instance, if it succeeds, $\mathcal{B}$ outputs "DDH tuple". So if $\mathcal{A}$'s forgery probability changes between games 0 and 1, $\mathcal{B}$'s DDH advantage changes by the same amount. Thus games 1 and 2 are indistinguishable assuming DDH is hard in $\mathbb{G}$.

Success probability in Game 2 Now we argue that $\mathcal{A}$'s forgery probability in Game 2 is negligible. First consider a Type 2 forgery. Note that a forgery on a new message $\vec{M'}$ must have $\tilde{M'} = \prod_{j=0}^{n} M_j'^{y_j}$. If $\tilde{M'} = G^{\vec{m'}}$, the logarithm of $\tilde{M'}W$ to the base G is $y_1 m_1' + \ldots + y_n m_n' + w \pmod q$. Note that this is a pairwise independent function [BS20] of $m_1', \ldots, m_n'$. Since $\mathcal{A}$ has only received one value using $\vec{y}$ and w (in the response to the i^*th MAC query), the adversary can produce this value with probability at most $1/q$.

Next, we consider a Type 3 forgery. Let $d = x_0 + x_1 t^*$. Then the one MAC that $\mathcal{B}$ has output using the secret key has $V = \tilde{M} G^{w+ud}$ and the forgery has $V^* = \tilde{M} G^{w+u^* d}$, where u, u^* are the discrete logs of U and U^* from the i^*th query and the forgery respectively. Again, note that this is a pairwise independent function of u, and since $\mathcal{A}$ only has one MAC using w, d, the adversary has only negligible probability of producing the right value for u^*.

□

7.4 Credential Security

Referring to the definition in [CMZ14], a keyed-verification anonymous credential scheme has the following protocols: CredKeygen, BlindIssue, BlindObtain, Show, ShowVerify. The

key generation, (non-blind) issuance, and show protocols are described in Section 3, and the blind issuance protocol is described in Section 5.10

The following security properties are formally defined in [CMZ14]. In this section we review them briefly and argue that a similar analysis applies here.

Correctness The first part of correctness (that credentials always verify) follows from correctness of the MAC. Correctness of the second part (that Show always succeeds for valid credentials), follows from the correctness of the zero-knowledge proof system, and the equation $Z = I^z$. Using a 4-attribute example where (M_1, M_2, M_3) are hidden and M_4 is revealed, when Z is computed honestly, we have

$$\begin{aligned} Z &= \frac{C_V}{{C_{y_1}}^{y_1}{C_{y_2}}^{y_2}{C_{y_3}}^{y_3}(M_4 C_{y_4})^{y_4} W {C_{x_0}}^{x_0}{C_{x_1}}^{x_1}} \\ &= \frac{V {G_V}^z}{(M_1 {G_{y_1}}^z)^{y_1} \cdots (M_4 {G_{y_4}}^z)^{y_4} W (U {G_{x_0}}^z)^{x_0} (U^t {G_{x_1}}^z)^{x_1}} \\ &= \frac{V {G_V}^z}{({M_1}^{y_1} \cdots {M_4}^{y_4} W U^{x_0 + t x_1}) {G_1}^{z y_1} \cdots {G_4}^{z y_4} {G_{x_1}}^{z x_1} {G_{x_0}}^{z x_0}} \\ &= \frac{{G_V}^z}{{G_{y_1}}^{z y_1} \cdots {G_{y_4}}^{z y_4} {G_{x_0}}^{z x_0} {G_{x_1}}^{z x_1}} = I^z \end{aligned}$$

and it can be checked similarly that this also holds with more than four attributes.

Unforgeability Intuitively, credential unforgeability means that an adversary cannot create a valid proof for a statement not satisfied by the credentials they have been issued. This follows from the unforgeability of the MAC (proven in Section 7.3), and the extractability of the proof system.

If the adversary outputs a proof based on a MAC with attributes that were not output by Issue, then we can extract a forgery for the MAC scheme.

For example, referring the to the proof of knowledge used for authentication in Section 5.12, note that from a successful prover we can extract (z, sk), then use these to compute (t, U, V), which is a valid MAC on the attributes (M_1, M_2, m_3) since it satisfies the verification equation (assured by the proof statement $Z = I^z$). If the MAC was created by the issuer, authentication should succeed. If not, and the MAC is new, it is a forgery and the MAC scheme is broken.

Anonymity This requires that the proofs output when presenting a credential reveal only the statement being proven. Below we sketch a proof that the authentication proof is zero-knowledge, and this proof includes the statements common to any credential presentation.

To show that the proof of Section 5.12 is zero-knowledge, we first need to show that the commitments are hiding (which is nontrivial since they all share the same random value z). Note that in the random oracle model, the bases G_{y_i} are a random set, that are then

input to the wPRF f_z, so ${G_{y_i}}^z$ is a PRF output under the DDH assumption. Therefore, the commitments $(C_{y_1}, \ldots, C_{y_4}, C_{x_0}, C_{x_1}, C_V)$ are hiding, and they can be simulated with random group elements. Since the ciphertext (E_{A_1}, E_{A_2}) is CPA secure, it can also be simulated with random values, and since the proof π_A is zero-knowledge, a simulator exists to simulate it.

Blind Issuance This property requires that blind issuance be a secure two-party protocol, between the user, who has the blind attributes as private input, and the issuer, who has the issuer secret key as private input. Our blind issuance protocol based on homomorphic Elgamal encryption is unchanged from [CMZ14], and security follows from CPA security of Elgamal (implied by DDH) and the privacy and extractability of the zero-knowledge proof system. Note that non-blind issuance is the special case where no attributes are hidden.

Key-parameter consistency This property ensures that an issuer cannot use different secret keys with different users, in order to link an instance of BlindIssue with an instance of Show.

We consider two cases, starting with the key consistency of (w, w'). From an issuer that creates two proofs π_I with different (w, w'), we can extract two openings to the Pedersen commitment $C_W = {G_W}^w {G_{W'}}^{w'}$, breaking the binding property. Given such a malicious issuer, we can construct an algorithm for the DLP in $\mathbb{G}$. Given a DLP instance $Y = G^x$, set $G_W = Y$ and $G_{W'} = G^{r_1}$. Then given two distinct openings of the commitment C_W, and knowledge of r_1, we can solve for x.

Now consider the secrets used in the I value of *iparams*. Similarly, the product ${G_{y_1}}^{y_1} \ldots {G_{y_n}}^{y_n} {G_{x_0}}^{x_0} {G_{x_1}}^{x_1}$ is a binding commitment under the DLP assumption in $\mathbb{G}$, and by the same argument no malicious issuer can prove knowledge of distinct openings if the DLP is hard in $\mathbb{G}$.

7.5 System Security

When considering the security and privacy properties of the system as a whole, there are no existing definitions in the literature we can leverage. In Appendix A, we give a definition of a secure private group system as an ideal functionality. The functionality, denoted $\mathcal{F}$, is a trusted party that implements the system, and interacts with users and the server. Specifying $\mathcal{F}$ in this way concisely defines the behavior our protocol aims to achieve, including any leakage or misbehaviors that may be possible when the server or users are malicious.

In terms of security, when the server is honest we must ensure the privacy of honest users against attacks by malicious users, for example, groups of honest users should have privacy from malicious users.

When the server is malicious, denoted S^*, there are two cases, depending on whether a group contains a malicious user. If so, then between the malicious user and S^* they know

all secrets (*ServerSecretParams* and *GroupSecretParams*) for that group, and can learn all group members and modify the group arbitrarily; no security is possible. The more interesting case is when the malicious server manages the state for a group where all users are honest. Here S^* can deviate from the protocol in many ways (e.g., delete members from a group, reject requests to add a new user, etc.) but none of these deviations should violate privacy. Informally: the group should remain confidential if all members are honest. Some amount of integrity is possible as well: since S^* does not know the group key, it cannot add arbitrary users to a group.

Appendix A then provides a sketch of a simulation-based security proof, to provide some assurance that our design implements $\mathcal{F}$ securely, and to motivate some of the security properties we require from our new encryption and credentials.

8 Additional Related Work

Structure preserving signatures [AFG+16] are similar to our new MAC in the sense that it's possible to sign group elements. However, known constructions require a group with a pairing, making them significantly more expensive than our MAC.

Bellare et al. [BPR14] also study deterministic symmetric-key encryption with unique ciphertexts, in the context of preventing malicious implementations of ciphers from leaking information in the context of internet protocols. Our definition of unique ciphertexts matches theirs, but our constructions are algebraic, and are therefore quite different.

There are alternative constructions of algebraic MACs[ZYHW16, BBDT16, CDDH19], inspired by pairing-based signatures, with BB-MAC being most common (derived from Boneh-Boyen signatures). BB-MAC also works in a cyclic group $\mathbb{G}$, and requires that the strong Diffie-Hellman problem be hard in $\mathbb{G}$. While BB-MAC (and SDH-based variants) may be more efficient in terms of the number of group operations, the comparison does account for the larger groups required to provide concrete security [BG04, Che06, JY09].

For example, Zhang et al. [ZYHW16] design a system using BB-MAC for anonymous password-based authentication in the context of TLS. While the implementation claims to target 128-bit security, is uses the elliptic curve `nistp256r1` [oST13], which does not provide 128 bits of security for the SDH assumption [BG04, Che06, JY09]. The target 128-bit security could be achieved by moving up to the `nistp384r1` curve, which has 1.5x larger group elements and significantly higher computational costs (e.g., the nistp256r1 implementation in OpenSSL 1.1.1 is about 18x faster than the nistp384r1 implementation). The choice of concrete parameters in [CDDH19] has similar issues. Since using SDH-based MACs would require using a different, larger curve than what is currently deployed in Signal, our design consciously avoids SDH-based MACs.

Barki et al. observe that if BB-MAC is instantiated in a group with a pairing, then the MAC can become a public-key signature scheme if the verifier uses parings. Pointcheval and Sanders [PS16] do the same for $\mathsf{MAC}_{\mathsf{GGM}}$. The drawback of enabling public verification

is that groups with a pairing are generally less efficient than the fastest cyclic groups from elliptic curves.

Concrete performance comparisons of KVAC systems based on these MACs to the new system we present here are difficult, since they do not allow group elements as attributes, and so do not support our encryption scheme. We did experiment with $\mathsf{MAC}_{\mathsf{GGM}}$ from [CMZ14], where we had to divide plaintexts into small scalars (e.g., 16 bits), in order to encrypt them using an exponential Elgamal variant, that required solving small discrete logs during decryption. For example, instead of encoding a 16-byte UUID into a single 32-byte group element, giving a 64-byte ciphertext, our prototype had eight group elements, and 288-byte ciphertexts. Decryption, credential issuance and presentation were also significantly slower.

Group signatures [Cv91] and ring signatures [RST01] allow users to form a group, such that any member can anonymously create a signature that verifies with a group public key. Using group signatures would allow a user to authenticate as a group member, without revealing which ciphertext encrypts their UID (though this would prevent the server from easily implementing access control). However, group signatures are based on relatively expensive public-key signatures, and are often more complex than our proposal. Further, we do not require the notion of a group manager, who can de-anonymize signers.

There is a growing list of systems that use credentials based on algebraic MACs for authentication. There are designs for privacy-preserving federated identity management [IHD16], anonymous payment channels [GM17], electronic voting [ABBT16], private e-cash [BBD+16], censorship resistance [LdV17] and smart card authentication [CDDH19]. The NEXTLEAP project [Hal17] considers building a fully decentralized PKI for use with the Signal protocol, while our approach increases centralization while maintaining privacy.

9 Future Work

The private group system has been implemented, and deployment is underway. Here we discuss possible improvements to the security analysis, or extensions.

Some aspects of our work could benefit from formal methods. For instance, the encryption and MAC schemes may be within reach of automated analysis in the generic group model, as in [ABS16].

In the security analysis of our new MAC, we assume that $\mathsf{MAC}_{\mathsf{GGM}}$ is $\mathsf{uf\text{-}cma}$ secure, which is only known to be true in the generic group model. Ongoing work is developing a new proof of Theorem 14 assuming instead that DDH is hard in $\mathbb{G}$ (also in the random oracle model). To make the proof work, each MAC must have one of the attributes be a random group element. Thus for a small decrease in performance, the GGM assumption can be replaced with the weaker DDH assumption.

While we define security of the system as a whole in a formal way with an ideal functionality, our security proof is informal, being only a sketch. Making this proof rigorous is

another possible improvement.

Further analysis on encoding data into group elements, and analysis of our encryption scheme in the deterministic authenticated encryption framework of [RS06], might be productive.

Our security analysis that our system satisfies the private groups functionality defined by the ideal functionality in the case of malicious servers also requires the generic group model assumption, in order to allow a type of selective opening property of the encryption. Future work could investigate this issue with the goal of removing this assumption.

In Section 1.1 we noted some security properties that might be strengthened.

Acknowledgements We extend our thanks to Dan Boneh, Isis Lovecruft, Henry de Valence, Bas Westerbaan, Bram Westerbaan, Jan Camenisch, and the Signal team for helpful discussions. We also thank Cathie Yun for help with the sequence diagram.

References

[ABBT16] Roberto Araújo, Amira Barki, Solenn Brunet, and Jacques Traoré. Remote electronic voting can be efficient, verifiable and coercion-resistant. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 224–232. Springer, Heidelberg, February 2016.

[ABS16] Miguel Ambrona, Gilles Barthe, and Benedikt Schmidt. Automated unbounded analysis of cryptographic constructions in the generic group model. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 822–851. Springer, Heidelberg, May 2016.

[AFG+16] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-preserving signatures and commitments to group elements. *Journal of Cryptology*, 29(2):363–421, April 2016.

[BBD+16] Amira Barki, Solenn Brunet, Nicolas Desmoulins, Sébastien Gambs, Saïd Gharout, and Jacques Traoré. Private eCash in practice (short paper). In Jens Grossklags and Bart Preneel, editors, *FC 2016*, volume 9603 of *LNCS*, pages 99–109. Springer, Heidelberg, February 2016.

[BBDT16] Amira Barki, Solenn Brunet, Nicolas Desmoulins, and Jacques Traoré. Improved algebraic MACs and practical keyed-verification anonymous credentials. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 360–380. Springer, Heidelberg, August 2016.

[BDJR97] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403. IEEE Computer Society Press, October 1997.

[Ber05] Daniel J. Bernstein. The poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *FSE 2005*, volume 3557 of *LNCS*, pages 32–49. Springer, Heidelberg, February 2005.

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006.

[BG04] Daniel R. L. Brown and Robert P. Gallant. The static Diffie-Hellman problem. Cryptology ePrint Archive, Report 2004/306, 2004. `http://eprint.iacr.org/2004/306`.

[BL13] Foteini Baldimtsi and Anna Lysyanskaya. Anonymous credentials light. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 1087–1098. ACM Press, November 2013.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.

[BS20] Dan Boneh and Victor Shoup. A graduate course in applied cryptography, 2020. Available online `https://crypto.stanford.edu/~dabo/cryptobook/`.

[CD00] Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to separable group signatures and signature sharing schemes. In Tatsuaki Okamoto, editor, *ASIACRYPT 2000*, volume 1976 of *LNCS*, pages 331–345. Springer, Heidelberg, December 2000.

[CDDH19] Jan Camenisch, Manu Drijvers, Petr Dzurenda, and Jan Hajny. Fast keyed-verification anonymous credentials on standard smart cards. Proceedings of ICT Systems Security and Privacy Protection, 34th IFIP International Conference, 2019.

[Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[Che06] Jung Hee Cheon. Security analysis of the strong Diffie-Hellman problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 1–11. Springer, Heidelberg, May / June 2006.

[CHK+11] Jan Camenisch, Kristiyan Haralambiev, Markulf Kohlweiss, Jorn Lapon, and Vincent Naessens. Structure preserving CCA secure encryption and applications. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 89–106. Springer, Heidelberg, December 2011.

[CL03] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 268–289. Springer, Heidelberg, September 2003.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 56–72. Springer, Heidelberg, August 2004.

[CMZ14] Melissa Chase, Sarah Meiklejohn, and Greg Zaverucha. Algebraic MACs and keyed-verification anonymous credentials. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 1205–1216. ACM Press, November 2014.

[CPZ19] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The Signal private group system and anonymous credentials supporting efficient verifiable encryption. Cryptology ePrint Archive, Report 2019/1416, 2019. `https://eprint.iacr.org/2019/1416`.

[CS97] J. Camenisch and M. Stadler. Proof systems for general statements about discrete logarithms. Technical Report TR 260, Institute for Theoretical Computer Science, ETH Zürich, 1997.

[CS03] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 126–144. Springer, Heidelberg, August 2003.

[Cv91] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 257–265. Springer, Heidelberg, April 1991.

[CV02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 21–30. ACM Press, November 2002.

[DKPW12] Yevgeniy Dodis, Eike Kiltz, Krzysztof Pietrzak, and Daniel Wichs. Message authentication, revisited. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 355–374. Springer, Heidelberg, April 2012.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[GM17] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 473–489. ACM Press, October / November 2017.

[Hal17] Harry Halpin. NEXTLEAP: decentralizing identity with privacy for secure messaging. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES'17),*, pages 92:1–92:10. ACM, 2017.

[HdVLA19] Mike Hamburg, Henry de Valence, Isis Lovecruft, and Tony Arcieri. The Ristretto group, 2019. `https://ristretto.group/`.

[IHD16] Marios Isaakidis, Harry Halpin, and George Danezis. Unlimitid: Privacy-preserving federated identity management using algebraic macs. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, WPES 16, page 139142, 2016.

[JY09] David Jao and Kayo Yoshida. Boneh-Boyen signatures and the strong Diffie-Hellman problem. In Hovav Shacham and Brent Waters, editors, *PAIRING 2009*, volume 5671 of *LNCS*, pages 1–16. Springer, Heidelberg, August 2009.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. IETF Internet Request for Comments 2104, February 1997.

[LdV17] Isis Agora Lovecruft and Henry de Valence. HYPHAE: Social secret sharing, 2017. `https://patternsinthevoid.net/hyphae/hyphae.pdf`.

[Lun17] Joshua Lund. Encrypted profiles for Signal now in public beta, September 2017. `https://signal.org/blog/signal-profiles-beta/`.

[Mar14] Moxie Marlinspike. Private group messaging, May 2014. `https://signal.org/blog/private-groups/`.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.

[NR95] Moni Naor and Omer Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. In *36th FOCS*, pages 170–181. IEEE Computer Society Press, October 1995.

[oST13] National Institute of Standards and Technology. Federal information processing standards publication: Digital signature standard (DSS), July 2013. FIPS PUB 186-4.

[PS16] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126. Springer, Heidelberg, February / March 2016.

[PZ13] C. Paquin and G. Zaverucha. U-prove cryptographic specification v1.1 (revision 2), 2013. Available online: `www.microsoft.com/uprove`.

[RCE15] Kai Rannenberg, Jan Camenisch, and Ahmad Sabouri (Editors). *Attribute-based credentials for trust, identity in the information society*. Springer, 2015. `https://doi.org/10.1007/978-3-319-14439-9`.

[RMS17] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: How group chats weaken the security of instant messengers signal, WhatsApp, and threema. Cryptology ePrint Archive, Report 2017/713, 2017. `http://eprint.iacr.org/2017/713`.

[RS06] Phillip Rogaway and Thomas Shrimpton. Deterministic authenticated-encryption: A provable-security treatment of the key-wrap problem. Cryptology ePrint Archive, Report 2006/221, 2006. `http://eprint.iacr.org/2006/221`.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 552–565. Springer, Heidelberg, December 2001.

[Sig19] Signal. Technical information (specifications and software libraries), 2019. `https://www.signal.org/docs/`.

[ZYHW16] Zhenfeng Zhang, Kang Yang, Xuexian Hu, and Yuchen Wang. Practical anonymous password authentication and TLS with anonymous client authentication. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1179–1191. ACM Press, October 2016.

A Ideal Functionality for a Private Group System

In this section we give a definition of a secure private group system as an ideal functionality. The functionality, denoted $\mathcal{F}$, is a trusted party that implements the system, and interacts with both honest and malicious users (the attacker) and either an honest or malicious server.

Note that all (honest) protocols are initiated by a user, and their UID (denoted *uid*) is sent to $\mathcal{F}$. The ideal functionality then shares information with a (potentially) malicious server S^*, and allows S^* to decide whether to deviate from the protocol, e.g., S^* may reject valid requests, or return incorrect information.

The definition of $\mathcal{F}$ has two cases, depending on whether the server is honest. In Fig. 2 we describe $\mathcal{F}$ when the server, denoted S, is honest. In this case we know that the honest server will try to correctly manage the groups, and our main goal is to ensure that malicious users cannot cause the server to perform an unauthorized group operation, corrupt the state of honest groups (those which have no corrupt members), or learn about the operations and membership of honest groups.

When the server is malicious, denoted S^*, we describe $\mathcal{F}$ in Fig. 3. Here there are again two cases, depending on whether the relevant group contains a malicious user. If so, then between the malicious user and S^* they know all secrets (credential issuer secret keys and group secret key) for that group, and can learn all group members and their profile keys (since the group was created), and modify the group arbitrarily. This case is rather trivial from the perspective of a security definition, since no security is possible. The more interesting case is when S^* manages the state for a group where all users are honest. Here S^* can deviate from the protocol in many ways (e.g., delete members from a group, reject requests to add a new user, etc.) but none of these deviations should allow, e.g., S^* to learn members of the group. Informally, the group should remain confidential if all users in it are honest. Some amount of integrity is possible as well, since S^* does not know the group key, it cannot add arbitrary users to a group.

By comparing Figs. 2 and 3 we can see how S^* can deviate from the honest server behavior, depending on whether the group is corrupted. S^* can also abort a protocol at any time, and we do not explicitly include this in our description. For groups that are not corrupted, at any point S^* can re-add a previous invitation in the group, or re-add a previously removed user to the group, by ignoring the delete step in DeleteGroupMember. When FetchGroupMembers is called for an honest group, S^* can choose to return or omit any of the members that have ever been in the group regardless of whether they have since been removed with any profile key they have used with that group. S^* can also commit to an arbitrary profile key for any user, since profile key commitments are stored on the server, and could be replaced with commitments to arbitrary profile keys. This operation is modeled with the special function CommitToAdvProfileKey, only used by a malicious server.

Setup and notation State in $\mathcal{F}$ is maintained in hash tables. Setup initializes tables T_U and T_G. Table T_U is a table of users in the system, a row $T_U[uid]$ has $T_U[uid].\mathit{Times}$ a set of times at which *uid* can authenticate, and $T_U[uid].\mathit{ProfileKeys}$, a set of profile keys currently associated with *uid*, and $T_U[uid].\mathit{corrupt}$, a flag that indicates whether *uid* is a corrupt user. T_G is a table of groups in the system, a row $T_G[gid]$ contains a set of members indexed by UID $\{T_G[gid][uid_1], \ldots, T_G[gid][uid_n]\}$, each with a profile key

$T_G[gid][uid].ProfileKey$ that is currently in use, and a list of all profile keys they have used in this group, $T_G[gid][uid].ProfileKeyHistory$. Groups also have a list of all current and past members, $T_G[gid].UIDHistory$. The flag $T_G[gid].corrupt$ indicates whether the group contains a corrupted user. The *corrupt* flags default to 1 (corrupted), since S^* may create groups without interacting with $\mathcal{F}$. The function index() returns the position of an element in a list, or $\perp$ if the element is not in the list. Authentication credentials are valid for a time period, denoted t, and when requesting credentials users may request a set of times, denoted T.

Comments on the honest server case The function AuthAsMember is used as a subroutine by $\mathcal{F}$ in other functions, and not exposed to users. In AuthAsMember, a user with UID *uid* authenticates as a member of the group with GID *gid*. We also define AuthAsInvitedMember as the same except the check that the profile key is not $\perp$ is omitted. The lines **ensure** AuthAsMember(...) return an error if AuthAsMember fails (similarly for AuthAsInvitedMember).

Honest users call CommitToProfileKey to register a new random profile key with $\mathcal{F}$. This models the fact that honest users choose random profile keys. Malicious users may commit to (or update) an arbitrary profile key using CommitToAdvProfileKey. The function UpdateProfileKey allows a user to update their own profile key in a specific group, or allows an invited member to set their profile key.

A user *uid*, who is a member of the group *gid*, can add another user uid' with AddGroupMember. Note that this will overwrite an invitation, should one exist, for uid'.

Comments on the malicious server case Note that we do not allow S^* to add arbitrary K', to model model issuing bad profile credentials, since S^* can maintain it's own data structure, and behave as if $K' \in ProfileKeys$.

When the user is honest, S^* only learns that GetProfileCredential was called, and can decide whether to issue the credential.

GetAuthCred(T) from uid

if $T_U[uid]$ is not defined
  return Error: invalid user
Query S to ensure T valid for uid
if T is invalid for uid
  return Error: invalid time(s)
Append T to $T_U[uid].Times$
return 1

CommitToProfileKey(t) from uid

if $t \notin T_U[uid].Times$
  return Error: Authentication failure
Create new random K
Add K to $T_U[uid].ProfileKeys$
return K

GetProfileCredential(uid, K)

if $K \in T_U[uid].ProfileKeys$
  return 1
return 0

AuthAsMember(gid, t) from uid

if $T_G[gid]$ not defined
  return Error: gid doesn't exist
if $T_G[gid][uid].ProfileKey = \perp$
  return Error: user invited, not added
if $T_G[gid][uid]$ not defined
  return Error: User not in group
if $t \notin T_U[uid].Times$
  return Error: Invalid timestamp
return 1

FetchGroupMembers(gid, t) from uid

ensure AuthAsMember(gid, t, uid)
return $T_G[gid]$
// Here $T_G[gid]$ is the list of all users and
// their profile keys in gid, without history

DeleteGroupMember(gid, uid', t) from uid

ensure AuthAsMember(gid, t, uid)
if $T_G[gid][uid']$ not defined
  return Error: no such user
Delete $T_G[gid][uid']$
return 1

DeleteGroup(gid, t) from uid

ensure AuthAsMember(gid, t, uid)
Delete $T_G[gid]$
return 1

CreateGroup(gid, K, t) from uid

if $T_G[gid]$ is defined
  return Error: group exists
if $K \notin T_U[uid].ProfileKeys$
  return Error: invalid profile key
if $t \notin T_U[uid].Times$
  return Error: expired credential
Set $T_G[gid][uid].ProfileKey = K$
Append K to $T_G[gid][uid].ProfileKeyHistory$
Append uid to $T_G[gid].UIDHistory$
if $T_U[uid].corrupt$
  Set $T_G[gid].corrupt$

InviteToGroup(uid', gid, t) from uid

ensure AuthAsMember(gid, t, uid)
if $uid' \in T_G[gid]$
  return Error: user already in gid
Append $(uid', \perp)$ to $T_G[gid]$
Append uid' to $T_G[gid].UIDHistory$
if $T_U[uid'].corrupt$
  Set $T_G[gid].corrupt$
return 1

AddGroupMember(gid, uid', K, t) from uid

ensure AuthAsMember(gid, t, uid)
if $T_G[gid][uid'].ProfileKey \neq \perp$
  return Error: user exists in gid
if $K \notin T_U[uid'].ProfileKeys$
  return Error: Invalid profile key
Set $T_G[gid][uid].ProfileKey = K$
Append K to $T_G[gid][uid].ProfileKeyHistory$
if $T_U[uid'].corrupt$
  Set $T_G[gid].corrupt$
return 1

UpdateProfileKey(gid, K, t) from uid

ensure AuthAsMember(gid, t, uid) **or**
  AuthAsInvitedMember(gid, t, uid)
if $K \notin T_U[uid].ProfileKeys$
  return Error: Invalid profile key
$T_G[gid][uid].ProfileKey = K$
return 1

Figure 2: Ideal functionality for a private group system, for the **case when the server S is honest**. The function CommitToAdvProfileKey is defined in Fig. 3.

GetAuthCred(T) from uid

Send (uid, T) to S^*

CommitToProfileKey() from uid

if $T_U[uid]$ not defined
  return Error: Invalid user
Create new random K
Add K to $T_U[uid].ProfileKeys$
return K

UpdateProfileKey(gid, K, t) from uid

if $T_G[gid].corrupt$
  Send (gid, t, uid, K) to S^*
else
Send $(gid, t, \text{index}(T_G[gid][uid])$ to S^*

GetProfileCredential(uid, K)

if $T_U[uid].corrupt$
  Ignore this request
else
  Send index of K to S^*

AuthAsMember(gid, t) from uid

if $T_G[gid].corrupt$
  Send (gid, t, uid) to S^*
else
  Send $gid, t, \text{index}(T_G[gid][uid])$ to S^*

FetchGroupMembers(gid, t) from uid

if $T_G[gid].corrupt$
  Send (gid, t, uid) to S^*
else
  S^* can return "no such group"
  Send $(gid, t, \text{index}(T_G[gid][uid])$ to S^*
  S^* specifies a list of indices (i, j), $\mathcal{F}$ only
  sends UIDs at index i from $T_G[gid].UIDHistory$,
  with the corresponding profile keys with
  index j from $T_G[gid][uid].ProfileKeyHistory$

CreateGroup(gid, K, t) from uid

Send (gid, t) to S^*
if $T_U[uid].corrupt$
  Ignore this request
Set $T_G[gid].corrupt = 0$

InviteToGroup(uid', gid, t) from uid

if $T_U[uid'].corrupt$, set $T_G[gid].corrupt = 1$
if $T_G[gid].corrupt$
  Send $(gid, t, uid, uid', T_G[gid])$ to S^*
else
  Send $(gid, t, \text{index}(T_G[gid][uid]))$ to S^*
  Send $\text{index}(T_G[gid][uid'])$ to S^*

AddGroupMember(gid, uid', K, t) from uid

if $T_U[uid'].corrupt$ then set $T_G[gid].corrupt = 1$
if $T_G[gid].corrupt$
  Send $(gid, t, uid, uid', K, T_G[gid])$ to S^*
else
  Send $(gid, t, \text{index}(T_G[gid][uid])$ to S^*
  Send $\text{index}(T_G[gid][uid'])$ to S^*

DeleteGroupMember(gid, uid', t) from uid

if $T_G[gid].corrupt$
  Send $(gid, t, uid, uid', T_G[gid])$ to S^*
else
  Send to S^* :
  $(gid, t, \text{index}(T_G[gid][uid]), \text{index}(T_G[gid][uid']))$

DeleteGroup(gid, t) from uid

if $T_G[gid].corrupt$
  Send (gid, t, uid) to S^*
else
  Send $(gid, t, \text{index}(T_G[gid][uid])$ to S^*
  $T_G[gid]$ is not deleted

CommitToAdvProfileKey(K', uid) from S^*

if $T_U[uid].corrupt$
  Append K' to $T[uid].ProfileKeys$

Figure 3: Ideal functionality for a private group system, for the **case when the server S^* is malicious**. We omit places where S^* may direct $\mathcal{F}$ to abort, or to return arbitrary information to uid. S^* is always notified which type of call is being made.

B Security Argument

In this section we sketch a security argument for the private groups system as a whole. The security argument is in the random oracle model, and has two main cases, depending on whether the server is honest.

B.1 Honest server

We first consider the case when the server is honest, and some of the users are malicious. We'll group the malicious users as a single adversary $\mathcal{A}$. Security should be maintained for the honest users, and the malicious users should learn no more than they would from the same interactions with $\mathcal{F}$.

We describe a simulator $\mathcal{S}$ that interacts with $\mathcal{F}$ and $\mathcal{A}$, simulating the real protocol for $\mathcal{A}$. The system is secure if $\mathcal{A}$'s view when interacting with $\mathcal{S}$ is computationally indistinguishable from his view when executing the real protocol.

We describe each of the server operations, and describe how $\mathcal{S}$ implements them. We write uid_A to denote the user ID of a user controlled by $\mathcal{A}$.

$\mathcal{S}$ setup Generate the *ServerSecretParams* and *ServerPublicParams* for issuing credentials. Initialize storage for profile key commitments, to store a hash and profile commitment for each user, and a list of users created by $\mathcal{A}$. $\mathcal{S}$ also initializes storage for groups, and will store for each group the *gid*, the group public key, and a list of ciphertexts (one for each member). $\mathcal{S}$ chooses a random profile key K_{bad} that will be used as a placeholder, when the adversary uses an invalid profile key commitment. Since profile keys are assumed to be large, and K_{bad} is only used between $\mathcal{S}$ and $\mathcal{F}$, never in the simulated real protocols between $\mathcal{A}$ and $\mathcal{S}$, K_{bad} will only collide with another profile key in the system with negligible probability. $\mathcal{S}$ initializes random oracles for all hash functions in the system.

Create User from uid_A $\mathcal{S}$ sends *ServerPublicParams* to uid_A. For simplicity we assume all UIDs are unique. $\mathcal{S}$ stores uid_A in the list of UIDs created by $\mathcal{A}$.

GetAuthCred(T) from uid_A $\mathcal{S}$ calls GetAuthCred(T) as uid_A to $\mathcal{F}$, if $\mathcal{F}$ responds 0, $\mathcal{S}$ aborts otherwise $\mathcal{S}$ replies with a set of credentials, one for each time in T.

Since $\mathcal{S}$ is behaving exactly as the honest server would in this function, $\mathcal{A}$'s view is identical to the real world.

CommitToAdvProfileKey(c) from uid_A $\mathcal{S}$ starts by checking whether the commitment $c = (J_1, J_2, J_3)$ is well-formed. Note that $\mathcal{S}$ can "decrypt" the commitment by generating the system parameters G_{j_1} and G_{j_2} as ${G_{j_3}}^{s_2}$ and ${G_{j_3}}^{s_3}$ for random (s_2, s_3), since the commitment is an Elgamal encryption. Since all parameters are derived from a random oracle, $\mathcal{S}$ can program it accordingly. Decryption yields the pair (M_3, M_4), and $\mathcal{S}$ can test

whether the commitment is well-formed by computing $\mathit{ProfileKey} = \mathsf{DecodeFrom}\mathbb{G}(M_4)$, and checking whether $M_3 = \mathsf{HashTo}\mathbb{G}(\mathit{ProfileKey}, \mathit{UID})$.

If c is correct, $\mathcal{S}$ sets $K = \mathit{ProfileKey}$, and sets $K = K_{\text{bad}}$ otherwise. Then $\mathcal{S}$ sends to $\mathcal{F}$: CommitToAdvProfileKey(K) from uid_A. $\mathcal{S}$ stores $(\mathit{uid}_A, H(c))$ in its list of commitments.

In subsequent operations, $\mathcal{S}$ will ensure that K_{bad} is never returned to $\mathcal{A}$; when issuing profile key credentials, the invalid (M_3, M_4) from the profile key commitment are used, and when $\mathcal{A}$ fetches group members, the ciphertexts he has uploaded are cached and returned. The value K_{bad} is only used between $\mathcal{S}$ and $\mathcal{F}$.

GetProfileCredential(uid, $\mathit{ProfileKeyVersion}$) from uid_A This message is received from a user uid_A controlled by $\mathcal{A}$.

If $\mathcal{S}$ does not have a commitment recorded for uid_A, or if the profile key version does not match, then $\mathcal{S}$ fails. From the $\mathit{ProfileKeyVersion}$, $\mathcal{S}$ can invert the hash (using the RO query history) to recover the commitment c (as described in CommitToAdvProfileKey), and from c recover the profile key K'. If c is invalid, $\mathcal{S}$ sets $K' = K_{\text{bad}}$.

$\mathcal{S}$ sends to $\mathcal{F}$: GetProfileCredential(uid, K') as uid_A. If $\mathcal{F}$ responds 0, fail, otherwise $\mathcal{S}$ executes BlindIssue with uid_A, and enforces the condition that K' is consistent with c. Even if c is not well-formed, it is a commitment to some pair (M_3, M_4), and security of BlindIssue ensures that the profile credential is issued only on these attributes.

Here $\mathcal{A}$'s interaction with $\mathcal{S}$ is indistinguishable from an honest server, because $\mathcal{A}$ gets a profile credential only on keys that have been previously committed (or invalid keys, if that is the case), and because $\mathcal{S}$ executes blind issuance as the honest server would.

CreateGroup(gid, t) from $\mathcal{A}$ $\mathcal{S}$ receives gid, $\mathit{GroupPublicParams}$, an $\mathit{AuthCredentialPresentation}$ and $\mathit{ProfileKeyCredentialPresentation}$ for $\mathcal{A}$. Since in the real protocol the channel is unauthenticated, $\mathcal{S}$ does *not* get the uid. $\mathcal{S}$ verifies the proofs and fails if any are invalid. From the authentication proof, $\mathcal{S}$ extracts the UID of the user creating the group, uid^*, and their associated profile key K^*, and the group secret key sk. If decryption of the profile key would fail because (M_3, M_4) is invalid, $\mathcal{S}$ sets $K^* = K_{\text{bad}}$. If uid^* was not created by $\mathcal{A}$, then $\mathcal{S}$ fails, denote this event E_1. Then $\mathcal{S}$ sends "CreateGroup(K^*, gid, t) from uid^*" to $\mathcal{F}$, and fails if $\mathcal{F}$ returns zero (denote this event E_2), otherwise $\mathcal{S}$ stores sk and the ciphertexts in its storage for gid.

$\mathcal{S}$'s behavior might diverge from the honest server if E_1 or E_2 occur. If E_1 occurs, $\mathcal{A}$ has created an authentication credential for an honest user, breaking unforgeability of the credential system. Event E_2 occurs if t is not a valid time, gid exists or K^* is invalid for uid^*. The first two are the same between $\mathcal{S}$ and the honest server, and for the third, unforgeability and security of BlindIssue of profile key credentials ensures that any K^* that $\mathcal{S}$ uses will have previously been sent to $\mathcal{F}$ for uid^* in a call to CommitToAdvProfileKey.

AddGroupMember(gid, t) from $\mathcal{A}$ $\mathcal{S}$'s behavior here is similar to CreateGroup. After verifying the proofs, that *gid* exists, and authenticating the caller by comparing the ciphertext (the *entire* $E_{A_1}, E_{A_2}, E_{B_1}, E_{B_2}$) to the list of ciphertexts stored for this group, $\mathcal{S}$ decrypts the ciphertext to recover uid^* and K^*, and sets $K^* = K_{\text{bad}}$ if appropriate. Then $\mathcal{S}$ calls AddGroupMember(gid, uid^*, K^*, t) as uid^*. If $\mathcal{F}$ returns 0, $\mathcal{S}$ fails and otherwise $\mathcal{S}$ stores the ciphertext for the new user in its storage for *gid*.

If the AddGroupMember call to $\mathcal{F}$ fails because the user already exists, $\mathcal{S}$ behaves exactly as the honest server would. It will not fail because t is invalid, by the unforgeability of the auth credential (unless $\mathcal{A}$ is using an expired credential, in which case $\mathcal{S}$ fails as the honest server would). Unforgeability of the profile key credential and security of BlindIssue ensure that the profile key is valid or K_{bad} and is registered with $\mathcal{F}$.

Authentication by comparing the ciphertext to the stored value is sound because the ciphertext is bound to a valid user that $\mathcal{A}$ created by the security of the authentication credentials, and correct because the encryption is deterministic and has unique ciphertexts, and

In the simulation, $\mathcal{S}$ relies on $\mathcal{F}$ to ensure that users are not added twice to the group. In the real world, this is ensured because encryption is deterministic and has unique ciphertexts, by comparing two ciphertexts the server can tell whether they encrypt the same UID. So $\mathcal{S}$ is consistent with the honest server in this regard as well.

FetchGroupMembers(gid, t) from $\mathcal{A}$ $\mathcal{S}$ receives (E_{A_1}, E_{A_2}) and the *AuthCredentialPresentation*. If the proofs are valid, and *gid* exists, and (E_{A_1}, E_{A_2}) is in $\mathcal{S}$'s list of ciphertexts for *gid*, authentication succeeds. $\mathcal{S}$ can use the group secret key to decrypt the UID. Decryption succeeds if the proofs are valid, and because the encryption is correct under adversarially chosen keys. If the UID, denoted uid_A was not created by $\mathcal{A}$, then $\mathcal{S}$ fails, denote this event E_1.

$\mathcal{S}$ makes the call to $\mathcal{F}$: FetchGroupMembers(gid, t) as uid_A. If $\mathcal{F}$ returns 0 then $\mathcal{S}$ fails, denote this event E_2. Otherwise $\mathcal{S}$ receives a list $(uid_1, K_1), \ldots, (uid_n, K_n)$. Now $\mathcal{S}$ must create a list of ciphertexts to return to $\mathcal{A}$. For UIDs that $\mathcal{A}$ created, return the cached ciphertext, and for honest users, create the ciphertext using the group secret key. Note that the cached ciphertexts contain any invalid profile keys $\mathcal{A}$ may have used, and for these $\mathcal{F}$ returns K_{bad} to $\mathcal{S}$, which $\mathcal{S}$ *does not* send to $\mathcal{A}$. By the unique ciphertexts property, the ciphertexts for honest users that $\mathcal{S}$ re-creates are identical to those created in the honest execution of the real system.

UpdateProfileKey(gid, t) from $\mathcal{A}$ As in FetchGroupMembers, $\mathcal{S}$ can recover the UID, uid_A from the ciphertext, and authenticate uid_A as a belonging to the group as the real server would (but ignoring the profile key ciphertext if uid_A is an invited member of the group). By decrypting the profile key ciphertext, $\mathcal{S}$ can determine whether the profile key K for uid_A is valid, and sets $K = K_{\text{bad}}$ if K is invalid. Then $\mathcal{S}$ calls to $\mathcal{F}$:

UpdateProfileKey(gid, K, t) as uid_A.

$\mathcal{F}$ can fail is if K is not registered as a profile key for uid_A. Security of the profile credential ensures that $\mathcal{A}$ only has a credential for a K that was called to GetProfileKeyCredential, which in turn registers a key with $\mathcal{F}$. If K is valid, then it has been sent to $\mathcal{F}$, otherwise $\mathcal{F}$ has K_{bad}, so $\mathcal{S}$'s call to $\mathcal{F}$ will succeed with overwhelming probability.

Other functions The remaining functions are similar to the ones above; InviteToGroup and DeleteGroupMember are similar to AddGroupMember, and DeleteGroup is simple once the caller has authenticated to the group (which is done in, e.g., FetchGroupMembers).

B.2 Malicious server

Our definition implies a form of selective opening security from the underlying encryption. In particular, we need that if the adversary is first given encryptions under a variety of different keys, and then allowed to request some of the decryption keys, the remaining ciphertexts remain secure. This is because the malicious server can first see the ciphertexts stored for each group, and then choose which groups to try to attack (either by corrupting existing members or convincing them to add a malicious user), we require that the other groups remain secure.

We could alternatively define a weaker version of our functionality which would require the adversary to declare when he creates a group whether it will ever have malicious users. In that case the properties defined in section 7.2 would be sufficient. Here though we will focus on the stronger definition.

More formally, the selective opening property we need is as follows:

Definition 15. *For a encryption scheme with algorithms* $(\mathsf{KeyGen}, \mathsf{Enc}\ \mathsf{Dec})$*, consider the following two experiments between a challenger* $\mathcal{C}$*, a simulator* $\mathcal{S}$ *and an attacker* $\mathcal{A}$*.*

Real Experiment:

1. *$\mathcal{A}$ makes queries to $\mathcal{C}$.*
   - *New key query: for query i, $\mathcal{C}$ generates new $(k^{(i)}, \mathsf{pk}^{(i)}) \leftarrow \mathsf{KeyGen}(1^\kappa)$. It sends $\mathsf{pk}^{(i)}$ to $\mathcal{A}$ and stores $k^{(i)}$.*
   - *Encrypt query: $\mathcal{A}$ submits index i and message $m^{(i,j)}$ and $\mathcal{C}$ outputs $c^{(i,j)} = \mathsf{Enc}(k^{(i)}, m^{(i,j)})$.*
   - *Corrupt Key query: $\mathcal{A}$ submits index i and receives $k^{(i)}$.*
2. *$\mathcal{A}$ outputs a bit b.*

Simulated Experiment:

1. *$\mathcal{A}$ makes queries to $\mathcal{C}$.*
   - *New key query: $\mathcal{C}$ calls $\mathcal{S}$ to obtain $(\mathsf{pk}^{(i)}) \leftarrow \mathsf{KeyGen}(1^\kappa)$. It sends $\mathsf{pk}^{(i)}$ to $\mathcal{A}$.*
   - *Encrypt query: $\mathcal{A}$ submits index i an message $m^{(i,j)}$. $\mathcal{C}$ stores $m^{(i,j)}$ and send i to $\mathcal{S}$ to obtain $c^{(i,j)}$, which it returns to $\mathcal{A}$.*
   - *Corrupt Key query: $\mathcal{A}$ submits index i. $\mathcal{C}$ sends all the messages encrypted under this key, i.e. $m^{(i,1)}, m^{(i,n)}$ for some maximum n to $\mathcal{S}$, and receives $k^{(i)}$ which it sends to $\mathcal{A}$.*
2. *$\mathcal{A}$ outputs a bit b.*

We say that the scheme satisfies selective opening if for every $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{A}$'s probability of producing 1 in both games differs by at most a negligible function.

This property is impossible in the standard model by an information theoretic argument: there are more possible sets of plaintexts than there are decryption keys, so there must be a noticeable fraction of plaintexts for which the simulator cannot provide an appropriate decryption key. However, we can show that our encryption scheme satisfies this definition in the generic group model.

Theorem 16. *The encryption scheme defined in Section Section 4.1 satisfies Definition 15 in the generic group model when* $\mathsf{HashTo}\mathbb{G}$ *is modeled as a random oracle.*

Proof. (sketch) In the real experiment the adversary interacts with a group oracle which maintains a table of pairs (d, h), where the first element is the discrete log of an element w.r.t. a fixed base g, and the second is the label by which the adversary refers to this element. The adversary can provide 2 handles h_1, h_2, and request the oracle perform the group operation. The oracle will look up the associated discrete logs and return the handle associated with $d_1 + d_2$ (or create a new one if this does not yet exist). The adversary can also query the random oracle: on a new input, the random oracle chooses a random discrete log d, and returns the associated handle (or a fresh handle if this is a new element).

We then consider the experiment where the discrete logs corresponding to random secrets chosen by the selective opening challenger (i.e. encryption secret keys) are replaced by formal variables, and the oracle maintains a table of (polynomial, handle) pairs, where the first terms are polynomials in those variables. When a group operation is performed, the oracle computes the sum of the associated polynomials, and if that polynomial occurs already in the table it returns the associated handle. Otherwise it generates a new handle to associate with the new polynomial. When the adversary calls the RO on a new element, the oracle picks a new formal variable to assign to the output, and returns a new handle.

This will be identical to the previous experiment except when the random secrets chosen by the challenger cause two polynomials that are not formally identical to evaluate to the same value, or when the random oracle chooses an output that matches to the evaluation of a polynomial. In our scheme, all of our polynomials are of degree at most 3 (the highest degree term is $M_1^{a_1 a_2}$ that occurs in E_2), so the probability of this happening if the adversary makes L group element or RO queries is at most $\binom{L}{2}3/q$, where q is the groups order, which is negligible since the adversary is polynomial time and $q \approx 2^{2\kappa}$.

Finally, we consider the ideal experiment. Our simulator will be have as follows: When the adversary requests new public parameters, instead of choosing the associated formal variables for the associated secret key, the simulator will choose a formal variable pk_i for the parameters and an associated handle which it will return. When the adversary requests a new ciphertext encrypted under parameters p_i, the simulator will choose new formal variables $\mathsf{ctxt}_1^{(i,j)}, \mathsf{ctxt}_2^{(i,j)}$ and an associated handle to return. Finally, when the adversary provides messages $(m_{i,1}, \ldots m_{i,n})$, the simulator will look up the associated random oracle outputs $\mathsf{r}^{(i,1)}, \ldots \mathsf{r}^{(i,n)}$ (or generate them if they do not exist), choose formal variables $a_1^{(i)}, a_2^{(i)}$ associated with sk_i, replace pk_i with $a_1^{(i)} x_1 + a_2^{(i)} x_2$, where x_1, x_2 are the discrete logs

of G_{a_1}, G_{a_2} respectively, and replace each pair $\mathsf{ctxt}_1^{(i,j)}, \mathsf{ctxt}_2^{(i,j)}$ with $\mathsf{r}^{(i,j)}a_1^{(i)}, \mathsf{r}^{(i,j)}a_1^{(i)}a_2^{(i)} + \mathsf{m}^{(i,j)}$, where $\mathsf{m}^{(i,j)}$ is the formal variable associated with the discrete log of $m^{(i,j)}$.

This will produce a different view to the adversary than the previous experiment only if, when each pk_i, ci, j is replaced with the associated value in all polynomials, some of the resulting polynomials become identical. However, note that this cannot happen because the polynomials we use to replace pk_i, ci, j have no monomials in common with one another or with the random oracle responses, which are the only other formal variables.

Thus, the adversary's probability of producing 1 differs by at most $\binom{L}{2}3/q$ between the real and simulated games, where L is the number of oracle queries the adversary makes, and q is the group order. □

Now we are ready to consider the case when the server and some of the users are malicious. We'll group the malicious users and server as a single adversary $\mathcal{A}$. Security should be maintained for the honest users, and the malicious users should learn no more than they would from the same interactions with $\mathcal{F}$.

We describe a simulator $\mathcal{S}$ that interacts with $\mathcal{F}$ and $\mathcal{A}$, simulating the real protocol for $\mathcal{A}$. The system is secure if $\mathcal{A}$'s view when interacting with $\mathcal{S}$ is computationally indistinguishable from his view when executing the real protocol.

We describe each of the server operations, and describe how $\mathcal{S}$ implements them. We write uid_A to denote the user ID of a user controlled by $\mathcal{A}$.

$\mathcal{S}$ setup $\mathcal{S}$ initializes storage for groups, and will store for each group the *gid*, the group secret key, and a list of ciphertexts, in a table L_G. It will also store a list of fake profile keys it has used on behalf of honest users L_{profile}. $\mathcal{S}$ initializes random oracles for all hash functions in the system.

Create User $\mathcal{S}$ sends *ServerPublicParams* to uid_A. For simplicity we assume all UIDs are unique. $\mathcal{S}$ stores uid_A in the list of UIDs created by $\mathcal{A}$.

GetAuthCred The simulator receives (uid, T) from the GetAuthCred interface of $\mathcal{F}$ (i.e. an honest user is requesting an AuthCredential): it will forward (uid, T) to the malicious server as part of an honest request for an authentication token. If the credential issuance succeeds for each time in T, it will send 1 to the ideal functionality.

CommitToProfileKey $\mathcal{S}$ receives *uid* from the CommitToProfileKey interface of $\mathcal{F}$ (i.e. an honest user is committing to a profile key). It will choose a random key K and send the commitment to K and store K in $L_{\mathsf{profile}}[uid]$.

GetProfileCredential If the simulator receives (K, uid) from the GetProfileCredential interface of $\mathcal{F}$ (i.e. an honest user is requesting a profile key credential for a malicious

uid): S will form a profile key credential commitment to K and honestly execute the GetProfileCredential protocol with $\mathcal{A}$.

If the simulator receives (i, uid) from the GetProfileCredential interface of $\mathcal{F}$ (i.e. an honest user is requesting a profile key credential for an honest uid): It will retrieve the ith key from from $L_{\mathsf{profile}}[uid]$. Then it will simulate the blind issuance protocol.

CreateGroup If the simulator receives (gid) from the CreateGroup interface of $\mathcal{F}$ (i.e. an honest user is requesting to create a group): $\mathcal{S}$ will generate a new simulated public key and a simulated encryption under that public key using the selective opening simulator. It will send this ciphertext along with a simulated credential proof. It will then store the ciphertext in $L_G[gid].\mathsf{ctxts}$.

AddGroupMember If the simulator receives $(gid, t, uid, uid', K, T_G[gid])$ from the AddGroupMember interface of $\mathcal{F}$ (i.e. an honest user wants to add a user to a corrupt group): If this is the first such message for this group (i.e. this is the first corrupt user added to the group). $\mathcal{S}$ will call the selective opening simulator to open all of the ciphertexts in $L_G[gid].\mathsf{ctxts}$ to the values revealed by $T_G[gid]$ and store the resulting secret key as $L_G[gid].\mathsf{sk}$. It will then look up the encryptions of uid', K and uid. It will send these ciphertexts along with a simulated zero knowledge proof of authentication credential w.r.t. the encryption of uid and a simulated zero knowledge proof of a profile key credential for uid', K.

Otherwise: $\mathcal{S}$ will lookup $L_G[gid].\mathsf{sk}$, the encryption key for this group, and encrypt uid', K and uid. It will send these ciphertexts along with a simulated zero knowledge proof of authentication credential w.r.t. the encryption of uid and a simulated zero knowledge proof of a profile key credential for uid', K.

If the simulator receives (gid, t, i) from the AddGroupMember interface of $\mathcal{F}$ (i.e. an honest user wants to add a user to a non-corrupt group): $\mathcal{S}$ will look up the ciphertext $L_G[gid].\mathsf{ctxts}[i]$. It will also simulate a profile key and uid encryptions under the key corresponding to gid using the selective opening simulator. It will send these ciphertexts along with a simulated zero knowledge proof of authentication credential w.r.t the first ciphertext and and a zero knowledge proof of profile key credential for the second ciphertext.

FetchGroupMembers If the simulator receives $(gid, T_G[gid], t, uid)$ from the FetchGroupMembers interface of $\mathcal{F}$ (i.e. an honest user wants to retrieve the membership list for a corrupt group): Encrypt uid, and send $\mathcal{A}$ the ciphertext and a simulated AuthCredential proof along with a request for the membership list of gid. When $\mathcal{A}$ returns the list of ciphertexts, decrypt them and send the resulting list of (uid, K) pairs to $\mathcal{F}$.

If the simulator receives (gid, t, i) from the FetchGroupMembers interface of $\mathcal{F}$ (i.e. an honest user wants to retrieve the membership list for an uncorrupted group): lookup the ciphertext $L_G[gid].\mathsf{ctxts}[i]$ and send that to $\mathcal{A}$ along with a simulated auth credential proof

and a request for the membership list of *gid*. When $\mathcal{A}$ returns a list of ciphertexts, look up the associated positions in $L_G[gid].\mathsf{ctxts}$ and send them to $\mathcal{F}$, ignoring any ciphertexts that do not occur in the table.

Other functions The remaining functions are similar to the ones above; InviteToGroup and UpdateProfileKey and DeleteGroupMember are similar to AddGroupMember, and DeleteGroup is simple once the caller has authenticated to the group (which is done in, e.g., FetchGroupMembers).

Showing the real and ideal experiments are indistinguishable. Here we sketch the sequence of experiments we can use to argue security:

1. Real Experiment
2. Replace all presentation proofs with the simulated versions. This is indistinguishable by anonymity of the credential system.
3. Abort if, for an uncorrupted group, the ciphertexts sent by the adversarial server in FetchGroupMembers and decrypted correctly by honest users is not a subset of the ciphertexts uploaded by the group members. The abort happens with negligible probability by CCA security of the encryption.
4. Switch to using the selective opening simulator to generate ciphertexts and public keys, and only generating secret keys when a group is corrupted. This is indistinguishable by our selective opening security.
5. Simulate the blind issuance protocol. This is indistinguishable by security of blind issuance.
6. Switch the K's in the commitments uploaded by honest users in CommitToProfileKey and GetProfileCredential to a new random K' different from what is used in the rest of the experiment. This is indistinguishable by hiding of the commitment
7. Ideal Experiment