Privacy-Preserving Contact Discovery with Applications to End-to-End Encrypted Messaging and Mobile-First Cryptocurrencies

Nicolas Mohnblatt[1]

Supervised by Dr. Philipp Jovanovic

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MSc in Information Security
at
University College London.

September 7, 2020

[1]**Disclaimer:** this report is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Acknowledgements

Contents

Chapter 1

Introduction

Privacy-oriented services such as end-to-end encrypted messaging are increasingly popular [15]. While they provide strong cryptographic guarantees for the confidentiality of message contents, many still leak or gather user-related data. This is particularly the case during a setup stage known as *contact discovery*. As a result, some of these applications gain access to their users' address books and therefore their mobile social graph [19, 20]. In this project, we are interested in performing *contact discovery* in a privacy-preserving manner while remaining practical for mobile applications with billions of users.

1.1 What is contact discovery?

Contact discovery (alternatively *contact matching*) simply refers to the process by which users of a service are able to find people they know on the said service. The applied method is largely determined by the amount of information users choose to make public. In the case of networks such as Facebook or LinkedIn, users are encouraged to register with — and publish — their legal names and can therefore be found through a simple search. In the cases we study, users are registered using pre-existing human-readable identifiers such as their phone numbers or email addresses. This information is kept private by the service such that only users with prior knowledge of each other's identifier can communicate.

As a user signs up to such a service, she will already hold an *address book* – a register that links people (often referred to as *contacts*) to their identifier. However, phone numbers and email addresses are identifiers generated by other services and there is no guarantee that all her *contacts* are using the new service. Thus in this context, *contact discovery* is more precisely defined as the process by which a user can discover whether or not her *contacts* are using a specific service. Notice that such a process is not only a necessary initialisation step; it must also be regularly refreshed to ensure users keep an up-to-date view of the contacts they can address.

1.2 The privacy challenge

The simplest way to perform contact discovery is arguably to send one's address book to the service operator, allowing them to compute the intersection between the address book and the list of registered users. This is in fact how the popular messaging services WhatsApp and Telegram perform their contact matching [19, 20]. Although efficient, this approach reveals large amounts of private information about users and their contacts, including those that are not register for the service. The service operator is able to construct a social graph of its users and their first connections, allowing it to check for individual connections at will or under government pressure. Such information may discourage whistleblowers from ever speaking up, in fear that their identity may be revealed if they are linked to journalists.

Naive hashing – A naive approach using only cryptographic hash functions will also fail to meet our goal [7, 8]. Alice could upload hashes of her contacts' identifiers for the service operator to compare against hashes of the registered users' identifiers. While this approach is efficient and yields the desired result, it will still leak Alice's address book.

Indeed, although the cryptographic hash function is pre-image resistant, the set of possible pre-images is small enough that hashes can be precomputed into a dictionary and used to find the identifiers that underly the uploaded hashes [8]. Salting these hashes to avoid offline computations renders the system unusable since the service operator would be required to hash the set of registered identifiers using a different salt for each attempt at contact discovery [7].

Advanced approaches and Efficiency – In light of the above, more advanced approaches have been developed to perform privacy-preserving contact discovery. We cover these in greater detail in chapter 2. The issue with such approaches is that they introduce additional complexity through computations, communication requirements, storage requirements or a combination thereof.

In the context of the services we study, contact discovery needs to be performed on mobile devices on a regular basis. These devices are less powerful than modern desktop computers and rely on rechargeable batteries. A computation-intensive process ran regularly on such a device could quickly drain its battery. Furthermore we must allow the process to scale elegantly with the number of registered users, and assume that it can grow to the order of billions.

Efficiency therefore constitutes a priority in the design of such contact discovery schemes. It will also provide a benchmark to evaluate systems against each other, provided that they guarantee a satisfactory level of privacy.

1.3 A peer-to-peer approach

In this report, we present a peer-to-peer approach that makes use of pairing-based cryptography. Users interact with a distributed service to obtain private cryptographic keys. Using these keys and their contacts' identifiers, users can locally derive shared secrets to establish an online meeting point, thus completing the discovery process.

In using this architecture, we reduce the service operator's role to a minimum and provide clients with the tools to compute shared secret keys with their contacts. Computations on the client side are of linear order with respect to the size of their address book. Furthermore, clients are only expected to communicate with the service during setup and are only required to store short cryptographic material.

1.4 Structure

Chapter 2

Related Work

In this chapter we provide an overview of state-of-the-art methods for privacy-preserving contact discovery, as well as academic attempts at solving a similar problem. These methods can be divided according to their underlying approach: the first aims at computing the intersection between a list of registered users and an address book, the second aims at providing users with the necessary cryptographic material needed to authenticate and establish shared secrets between each other.

In section 2.1, we cover Signal's approach which is to simply process each user's address book without storing her contacts [11]. To convince users that they are trustworthy, Signal publish their code and allow users to verify what code is being run by their servers. In section 2.2, we investigate cryptographic ways to perform a set intersection between two parties without either party learning the other's data. This is known as a private set intersection (PSI). The subsequent attempts fall under the second approach described above. Thus section 2.3 focuses on public key infrastructure and section 2.4 on identity-based key exchanges.

2.1 Public source code and remote attestation

2.1.1 Signal and Intel SGX

Signal's approach is arguably the simplest: request a user's address book, process it obliviously against the list of registered users and clear the servers from any knowledge linked to it [11]. While this process may seem trivial, it creates new challenges in terms of security and user trust. First, Signal must guarantee that no knowledge of the address book remains on the server, be it obtained through regular or side channels. Secondly, Signal needs to earn the trust of its users. Not only do they need to convince users that their process is completely oblivious, they must also provide constant evidence that their servers are running that particular process rather than any other.

They meet both challenges by publishing their server-side code and performing all their processing within “secure enclaves” on their servers.

2.2 Private set intersection (PSI)

Kales // Basic Bloom Filter

2.3 Public key infrastructure (PKI)

CONIKs // DNS

2.4 Identity-based key exchange (IBKE)

Boneh Waters //

Chapter 3

Background

Before we introduce our system, we recall some definitions of lesser known cryptographic primitives and assumptions. Our aim is to provide the necessary technical background to then discuss our system's architecture. Alternatively, readers may proceed to chapter 4 and refer back to this section when needed.

3.1 Bilinear pairings

The following definition for a *pairing* is that provided by Boneh and Shoup in *A Graduate Course in Applied Cryptography* [3]. To remain consistent with the source text, group operations are represented multiplicatively.

Definition 3.1 (Pairing [3])**.** *Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be three cyclic groups of prime order q where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. A **pairing** is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ satisfying the following properties:*

1. *bilinear: for all $u, u' \in \mathbb{G}_0$ and $v, v' \in \mathbb{G}_1$ we have*

$$e(u \cdot u', v) = e(u, v) \cdot e(u', v) \qquad \text{and} \qquad e(u, v \cdot v') = e(u, v) \cdot e(u', v) \tag{3.1}$$

2. *non-degenerate: $e(g_0, g_1)$ is a generator of $\mathbb{G}_T$*

*When $\mathbb{G}_0 = \mathbb{G}_1$, we say that the pairing is a **symmetric pairing**. When $\mathbb{G}_0 \neq \mathbb{G}_1$, we say that the pairing is an **asymmetric pairing**. We refer to $\mathbb{G}_0$ and $\mathbb{G}_1$ as the **pairing groups**, or source groups, and refer to $\mathbb{G}_T$ as the **target group**.*

From the bilinear property, we can derive the following equality which is central to our scheme:

$$\forall \alpha, \beta \in \mathbb{Z}_q,\ e({g_0}^{\alpha}, {g_1}^{\beta}) = e(g_0, g_1)^{\alpha\beta} = e({g_0}^{\beta}, {g_1}^{\alpha}) \tag{3.2}$$

Hard Problems in Pairing Groups – The existence of pairings has direct consequences on the discrete logarithm, the decisional Diffie-Hellman (DDH) and the computational Diffie-Hellman (CDH) assumptions. Most notably, the existence of a symmetric pairing on $\mathbb{G}_0$ provides a simple solution to the decisional Diffie Hellman problem. We summarise the effect of pairings on tradition cryptographic assumptions in Table 3.1 below.

	Symmetric Pairing $e : \mathbb{G}_0 \times \mathbb{G}_0 \to \mathbb{G}_T$	**Asymmetric Pairing** $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$
Discrete Logarithm	No harder in $\mathbb{G}_0$ than in $\mathbb{G}_T$	No harder in $\mathbb{G}_0$ or $\mathbb{G}_1$ than in $\mathbb{G}_T$
Decisional DH	Easy to solve in $\mathbb{G}_0$, assumed to hold in $\mathbb{G}_T$	Assumed to be hard in $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$
Computational DH	Assumed to be hard in $\mathbb{G}_0$ and $\mathbb{G}_T$	Assumed to be hard in $\mathbb{G}_0$, $\mathbb{G}_1$ and $\mathbb{G}_T$

Table 3.1: Summary table of classic cryptographic problems under pairings

There exist variants of the DDH and CDH assumptions that take into account the pairing operation: the decisional variant is known as the *decision Bilinear Diffie-Hellman* (DBDH) assumption and the computational variant is known as the *co-Computational Diffie-Hellman* (co-CDH) assumption. We provide formal definitions for both of the assumptions in Appendix A.

Implementation – Pairings have been implemented in practice on certain pairing-friendly elliptic curves. While the underlying constructions are outside of the scope of this project, we wish to emphasise a few of their features. In asymmetric pairings, the group $\mathbb{G}_0$ is usually built upon a finite field, while groups $\mathbb{G}_1$ and $\mathbb{G}_T$ are built on extensions of that field [3]. This implies that elements in $\mathbb{G}_0$ have a shorter representation than those in $\mathbb{G}_1$ or $\mathbb{G}_T$. Furthermore, operations in $\mathbb{G}_0$ are less computationally intensive.

Finally, a pairing operation is much more computationally intensive than exponentiation in any of the three groups [3].

3.2 BLS signatures

One application for pairings is to create deterministic and homomorphic signature schemes such as the one introduced by Boneh, Lynn and Shacham [2] – named BLS after all three of the authors. In this scheme, signatures are elements of one source group and public keys are elements of the other. Although we will make use of both variants, we only present the variant in which signatures are elements of $\mathbb{G}_0$ and public keys are elements of $\mathbb{G}_1$. Once again we write group operations multiplicatively to remain consistent with the source material.

Definition 3.2 (BLS Signatures [2])**.** *A BLS signature scheme* $\mathcal{S}_{BLS}$ *is composed of three efficient algorithms* $\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify}$*. Let* $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ *be three cyclic groups of prime order* q*, with security parameter* λ*, such that there exists a pairing* $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$*.* $g_0 \in \mathbb{G}_0$ *and* $g_1 \in \mathbb{G}_1$ *are generators. Let* H_0 *a cryptographic hash function defined as* $H_0 : \{0,1\}^* \to \mathbb{G}_0$*, and "*$\leftarrow_{\$}$*" denote the "choose uniformly at random" operator, we define the three algorithms as:*

> KeyGen : *Choose uniformly at random* $x \leftarrow_{\$} \mathbb{Z}_q^*$ *and set the secret key* $\mathsf{sk} \leftarrow x$ *and the public key* $\mathsf{pk} \leftarrow {g_1}^x$*. Output* sk *to the message signer and* pk *to the receiver.*
>
> $\mathsf{Sign}(\mathsf{sk}, m)$*: Output the signature* $\sigma = H_0(m)^{\mathsf{sk}}$*.*
>
> $\mathsf{Verify}(\sigma, m, \mathsf{pk})$*: If* $e(\sigma, g_1) = e(H_0(m), \mathsf{pk})$ *accept the signature. Otherwise reject.*

Theorem 3.1 (EUF-CMA Security of BLS Signatures [3])**.** *Let* $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ *be a pairing, let* $\mathcal{M}$ *be the message space and let* $H : \mathcal{M} \to \mathbb{G}_0$ *be a hash function. Then the derived BLS signature scheme is* ***existentially unforgeable under chosen message attacks*** *(EUF-CMA) assuming the co-Computational Diffie-Hellman assumption*[1] *holds for* e*, and* H *is modelled as a random oracle.*

Blind and/or threshold variants of this scheme exist. The former allows to hide the original message from the signer, while the latter allows to hide the complete signature from

[1] see Appendix A

any individual (non-colluding) signer. We define a blind (t,n)-threshold BLS signature scheme below. Once again, we only present the variant in which signatures are elements of $\mathbb{G}_0$ and public keys are elements of $\mathbb{G}_1$.

Definition 3.3 (Blind (t,n)-threshold BLS Signature)**.** *A blind (t,n)-threshold BLS signature scheme $\mathcal{S}_{BTBLS}$ is composed of six algorithms* KeyGen*,* Blind*,* Sign*,* $\mathsf{Combine}$*,* $\mathsf{Unblind}$*,* Verify*. Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be three cyclic groups of prime order q such that there exists a pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$. $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. Let H_0 a cryptographic hash function defined as $H_0 : \{0,1\}^* \to \mathbb{G}_0$, we define the six algorithms as:*

> $\mathsf{KeyGen}(\lambda)$: *n participants $P_1, P_2, \dots, P_n$ jointly execute a (t,n)-distributed key generation algorithm with security parameter λ to compute secret key shares $\mathsf{sk}_1, \mathsf{sk}_2, \dots, \mathsf{sk}_n$ and public key pk. Output sk_i and pk to P_i and pk to the message receiver.*
>
> $\mathsf{Blind}(m)$*: Choose uniformly at random $\alpha \leftarrow_{\$} \mathbb{Z}_q$. Output $\sigma_\alpha \leftarrow H_0(m)^\alpha$ and α.*
>
> $\mathsf{Sign}(\mathsf{sk}_i, \sigma_\alpha)$*: Output the signature $\widehat{\sigma_i} \leftarrow {\sigma_\alpha}^{\mathsf{sk}_i}$.*
>
> $\mathsf{Combine}(\widehat{\sigma_{j_1}}, \widehat{\sigma_{j_2}}, \dots, \widehat{\sigma_{j_t}})$*: Use Lagrange interpolation on a subset of t blinded partialn signatures $\widehat{\sigma_{j_1}}, \widehat{\sigma_{j_2}}, \dots, \widehat{\sigma_{j_t}}$ to recover the full blinded signature $\widehat{\sigma}$.*
>
> $\mathsf{Unblind}(\widehat{\sigma}, \alpha)$*: Output $\sigma \leftarrow \widehat{\sigma}^{(\alpha^{-1})}$ as the full unblinded signature.*
>
> $\mathsf{Verify}(\sigma, m, \mathsf{pk})$*: If $e(\sigma, g_1) = e(H_0(m), \mathsf{pk})$ accept the signature. Otherwise reject.*

3.3 Left/Right constrained pseudorandom functions

Left/right constrained pseudorandom functions were first introduced by Boneh and Waters [4]. These pseudorandom functions (PRFs) are evaluated over a pair of inputs x, y with a random key k – we denote the output value as $F(k, (x, y))$ or $F_k(x, y)$. These functions can then be "constrained" to their left or their right input using *constraining keys*: knowing the left constraining key for a specific value w allows to compute $F(k, (w, \cdot))$ at all points $(w, \cdot)$ with no knowledge of k. Similarly, the right constraining key for a value w allows to compute $F(k, (\cdot, w))$ at all points $(\cdot, w)$ with no knowledge of k. Left/right constrained PRFs are formally defined in [4] as:

Definition 3.4 (Left/right constrained PRF [4])**.** *Let $F : \mathcal{K} \times \mathcal{X}^2 \to \mathcal{Y}$ be a PRF with security parameter λ. For all $w \in \mathcal{X}$ we wish to support constrained keys $k_{w,\mathrm{LEFT}}$ that enable the evaluation of $F(k, (x, y))$ at all points $(w, y) \in \mathcal{X}^2$, that is, at all points in which* *the left side is fixed to w. In addition, we want constrained keys $k_{w,\mathrm{RIGHT}}$ that fix the right hand side of (x, y) to w. More precisely, for an element $w \in \mathcal{X}$ define the two predicates $p_w^{(L)}, p_w^{(R)} : \mathcal{X}^2 \to \{0, 1\}$ as*

$$p_w^{(L)}(x, y) = 1 \iff x = w \quad \text{and} \quad p_w^{(R)}(x, y) = 1 \iff y = w$$

We say that F supports left/right fixing if it is constrained with respect to the set of predicates

$$P_{LR} = \left\{p_w^{(L)}, p_w^{(R)} : w \in \mathcal{X}\right\}$$

Security – We now provide the definition of a secure left/right constrained PRF by adapting a more general definition provided in [4].

Attack Game 3.1 ([4]). *Let $F : \mathcal{K} \times \mathcal{X}^2 \to \mathcal{Y}$ be a left-right constrained PRF with respect to a set system $\mathcal{S} \subseteq 2^{\mathcal{X}^2}$ and security parameter λ. We define constrained security using the following two experiments denoted* EXP(0) *and* EXP(1) *with an adversary $\mathcal{A}$. For $b = 0, 1$ experiment* EXP(b) *proceeds as follows:*

A random key $k \in \mathcal{K}$ is selected and two helper sets $C, V \subseteq \mathcal{X}^2$ are initialised to $\emptyset$. The set $V \subseteq \mathcal{X}^2$ will keep track of all the points at which the adversary can evaluate $F(k, (\cdot, \cdot))$. The set $C \subseteq \mathcal{X}^2$ will keep track of all the points where the adversary has challenged. The sets C and V will ensure that the adversary cannot trivially decide whether challenge values are random or pseudorandom. In particular, the experiments maintain the invariant that $C \cap V = \emptyset$.

The adversary is then presented with three oracles as follows:

- $F.\mathsf{eval}(x, y)$*: given $(x, y) \in \mathcal{X}^2$ from $\mathcal{A}$ if $(x, y) \notin C$ the oracle returns $F(k, (x, y))$ and otherwise returns $\perp$. The set V is updated as $V \leftarrow V \cup \{(x, y)\}$.*
- $F.\mathsf{constrain}(w, d)$*: given a coordinate $w \in \mathcal{X}$ and a direction $d \in \{LEFT, RIGHT\}$ from $\mathcal{A}$ we define S as the set of all points p such that $p = (w, \cdot)$ if $d = LEFT$ or $p = (\cdot, w)$ if $d = RIGHT$. If $S \cap C = \emptyset$ the oracle returns the constraining key $k_{w,d}$ and the set V is updated $V \leftarrow V \cup S$. Otherwise, the oracle returns $\perp$.*
- $\mathsf{Challenge}(x, y)$*: given $(x, y) \in \mathcal{X}^2$ where $(x, y) \notin V$, if $b = 0$ the adversary is given $F(k, (x, y)))$; otherwise the adversary is given a random (consistent) $z \in \mathcal{Y}$. The set C is updated $C \leftarrow C \cup \{(x, y)\}$.*

Once the adversary is done interrogating the oracles, it outputs $b' \in \{0,1\}$.

For $b = 0, 1$ let W_b be the event that $b' = 1$ in $\mathrm{EXP}(b)$*. We define the adversary's advantage as:*

$$\mathrm{AdvPRF}_{\mathcal{A},\mathrm{F}}(\lambda) = |\Pr[W_0] - \Pr[W_1]| \tag{3.3}$$

When experiences EXP(0) and EXP(1) are performed equally many times, an equivalent definition for the adversary's advantage is $\mathrm{AdvPRF}_{\mathcal{A},\mathrm{F}}(\lambda) = \left|\frac{1}{2} - \Pr[b' = b]\right|$. Indeed, using the law of total probability:

$$\begin{aligned}
\mathrm{AdvPRF}_{\mathcal{A},\mathrm{F}}(\lambda) &= |\Pr[W_0] - \Pr[W_1]| && (3.4)\\
&= |\Pr[b' = 1 \wedge b = 0] - \Pr[b' = 1 \wedge b = 1]| && (3.5)\\
&= |\Pr[b' = 1 \wedge b = 0] - (\Pr[b' = b] - \Pr[b' = 0 \wedge b = 0])| && (3.6)\\
&= |\Pr[b' = 1 \wedge b = 0] + \Pr[b' = 0 \wedge b = 0] - \Pr[b' = b]| && (3.7)\\
&= |\Pr[b = 0] - \Pr[b' = b]| && (3.8)\\
&= \left|\frac{1}{2} - \Pr[b' = b]\right| && (3.9)
\end{aligned}$$

Definition 3.5 (Secure left/right constrained PRF [4])**.** *The PRF F is a secure constrained PRF with respect to $\mathcal{S}$ if for all probabilistic polynomial time adversaries $\mathcal{A}$ the function* $\mathrm{AdvPRF}_{\mathcal{A},F}(\lambda)$ *is negligible in λ.*

Implementation – Boneh and Waters [4] present a secure left/right constrained PRF construction under the random oracle model by making use of a symmetric pairing. Here we present a variant that makes use of an asymmetric pairing. Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be three cyclic groups of prime order q such that there exists a pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$. Let $H_0 : \{0,1\}^* \to \mathbb{G}_0$ and $H_1 : \{0,1\}^* \to \mathbb{G}_1$ be two hash functions modelled as random oracles. For a random key k, we define the left/right constrained PRF F as:

$$F(k, (x, y)) = e(H_0(x), H_1(y))^k \tag{3.10}$$

For $w \in \{0,1\}^*$, the constraining keys for the predicates $p_w^{(L)}$ and $p_w^{(R)}$ are:

$$k_{w,\mathrm{LEFT}} = H_0(w)^k \quad \text{and} \quad k_{w,\mathrm{RIGHT}} = H_1(w)^k \tag{3.11}$$

Using the bilinear property of the pairing, we can check that knowing $k_{w,\mathrm{LEFT}}$ allows to evaluate $F(k,(w,y))$ for all $y \in \{0,1\}^*$:

$$e(k_{w,\text{LEFT}}, H_1(y)) = e(H_0(w)^k, H_1(y)) = e(H_0(w), H_1(y))^w = F(k,(w,y)) \tag{3.12}$$

A similar equality can be written to check that $k_{w,\text{RIGHT}}$ allows to evaluate $F(k,(x,w))$ for all $x \in \{0,1\}^*$ by computing $e\left(H_0(x), k_{w,\text{RIGHT}}\right)$.

Notice that left/right constrained PRFs and BLS signatures are closely related. Indeed they both make use of the same underlying pairing construction. Furthermore, BLS signatures take the same form as a constraining key, namely a group element raised to an unknown power.

Chapter 4

Pairing-Based Contact Discovery

In this chapter we present the architecture for our contact discovery scheme (section 4.2). We then provide outlines of security proofs (section 4.3), theoretical performance evaluations (section 4.4) and show how our system maps onto real-world applications such as end-to-end encrypted messaging and mobile-first cryptocurrencies (section 4.5).

4.1 Formal problem statement

First, we provide a formal definition for the problem of contact discovery. User A is registered to a third-party application from which she receives an opaque account identifier $\mathbf{acc}_A$, an address $\mathbf{addr}_A$ and a secret/public key pair $(\mathsf{sk}_A, \mathsf{pk}_A)$. User A also holds a human-readable discovery identifier $\mathtt{id}_A$ (mobile phone number or an email-address) and a list of contacts. We represent A's address book as a set of discovery identifiers $\mathcal{C}_A$. We assume that users exchanged discovery identifiers through out-of-bound communication but are unable to exchange cryptographic material, including their public keys and addresses. Thus for all users B such that $\mathtt{id}_B \in \mathcal{C}_A$ and $\mathtt{id}_A \in \mathcal{C}_B$, A wishes to learn the tuple $(\mathbf{addr}_B, \mathsf{pk}_B)$.

4.2 Service architecture

The foundational design principle for our contact discovery scheme is to provide users with the means to perform contact discovery locally. As we have seen in chapter 2, sending a client the full list of registered users in a probabilistic data structures such as Bloom and Cuckoo filters requires the client to download and store large amounts of data. Instead, we follow an approach similar to the IBKE protocols and is closely related to the NI-IBKE described in [4]. Our scheme runs in three phases which we will investigate individually:

1. **Setup:** a one-time step for each user. During the setup phase, a user interacts with the contact discovery service to obtain her unique cryptographic material.

2. **Key derivation:** using this cryptographic material, the user is able to compute shared secret keys with any of her contacts knowing only their discovery identifier.

3. **Discovery:** using their shared secret key, a pair of users can establish a secure meeting point on an untrusted online cache, thus allowing for asynchronous contact discovery.

Figure 4.1 shows a diagram of the process described above.

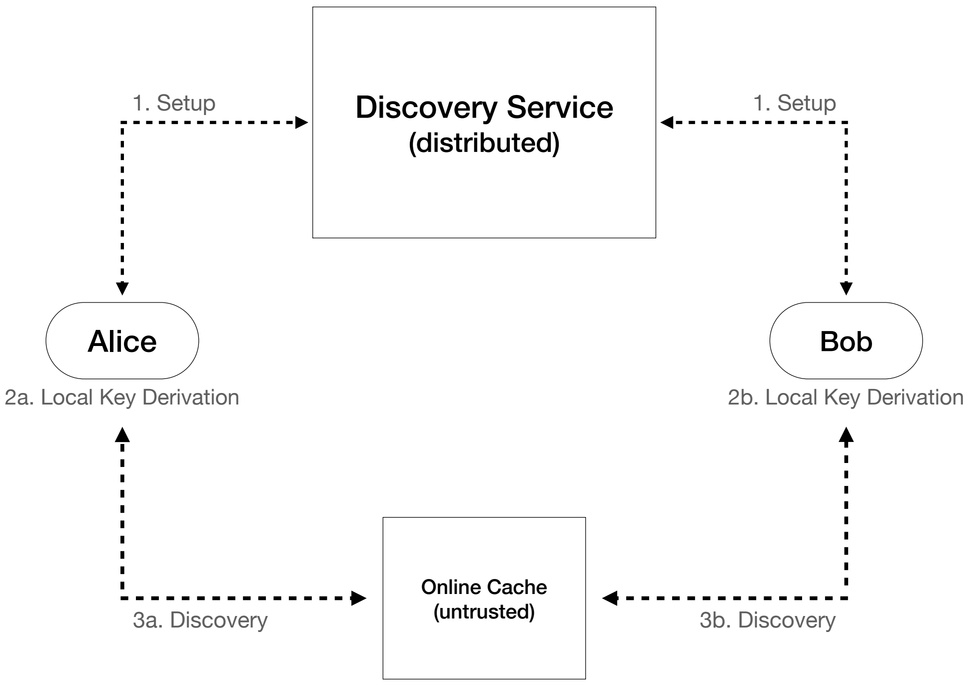

Figure 4.1: Contact discovery between a pair of users Alice and Bob, including setup. Numbers indicate the order of execution

4.2.1 Actors, assets and notation

We make a brief aside to clarify the actors and assets present in our scheme:

- **Users:** each user A holds an opaque account identifier $\mathbf{acc}_A$, an address $\mathbf{addr}_A$, a key pair $(\mathsf{sk}_A, \mathsf{pk}_A)$, a discovery identifier $\mathtt{id}_A$ and an address book $\mathcal{C}_A$ (see section 4.1). We denote $\mathcal{ID}$ the set of all existing discovery identifiers.

- **Discovery Service:** the discovery service is a distributed entity. We denote the set of all servers as $\mathcal{S}$ and the i-th server as S_i. All n servers have jointly executed a (t, n)-distributed key generation algorithm such as to hold shares s_i of an unknown master secret key, which we denote s. Furthermore, each server holds a list of tuples $(\mathbf{acc}, \mathsf{pk})$ for all registered users.

- **Online Cache:** the online cache may be operated by the discovery scheme or by a third party and is assumed to be untrusted. Its role is to manage key-value pairs. One possible implementation of such a cache is to follow the DNS-based approach of Papadopoulos *et al.* [14].

Next we define the cryptographic setting for our scheme. For a security parameter λ:

- $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ are three cyclic groups of prime order $q > 2^\lambda$ such that there exists a pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$.

- $H_0 : \mathcal{ID} \to \mathbb{G}_0$ and $H_1 : \mathcal{ID} \to \mathbb{G}_1$ are two public hash functions modelled as random oracles.

- $F : \mathbb{Z}_q \times \mathcal{ID}^2 \to \mathbb{G}_T$ is a left/right constrained PRF defined as:

$$F(k, (\mathtt{id}_A, \mathtt{id}_B)) = F_k(\mathtt{id}_A, \mathtt{id}_B) = e\left(H_0(\mathtt{id}_A), H_1(\mathtt{id}_B)\right)^k \tag{4.1}$$

- **KDF** is a public, deterministic key derivation function.

- BTBLS is a blind (t, n)-threshold BLS signature scheme (see definition 3.3). We denote this scheme's algorithms as $\mathsf{BTBLS.KeyGen}$, $\mathsf{BTBLS.Sign}$, *etc...*

- DSA is a strong existentially unforgeable signature scheme which makes use of the third-party provided user keys $(\mathsf{sk}_A, \mathsf{pk}_A)$ and is composed of algorithms $\mathsf{DSA.Sign}$ and $\mathsf{DSA.Verify}$.

- The master secret key is set to an integer $s \in \mathbb{Z}_q$ chosen uniformly at random. We define two corresponding master public keys ${g_0}^s$ and ${g_1}^s$, for which there exists i public shares denoted as ${g_0}^{s_i}$ and ${g_1}^{s_i}$ respectively.

- Let n the number of servers ($n = |\mathcal{S}|$)) and t a fixed threshold such that $1 \leq t \leq n$, we assume that the master secret key is shared according to a secure t-out-of-n secret sharing scheme and that no single entity holds the master secret key.

4.2.2 Key derivation

We first introduce the essential key derivation step. In doing so, we provide the reader with the necessary material to understand the security constraints under which the initial setup phase operates.

For all users B such that $\mathtt{id}_B \in \mathcal{C}_A$, user A can compute shared key material with B by evaluating $F_s(\mathtt{id}_A, \mathtt{id}_B)$ and $F_s(\mathtt{id}_B, \mathtt{id}_A)$. From the definition of left/right constrained PRFs, A can do so with the constraining keys $k_{\mathtt{id}_A,\mathrm{LEFT}}$ and $k_{\mathtt{id}_A,\mathrm{RIGHT}}$:

$$f_{AB} = F_s(\mathtt{id}_A, \mathtt{id}_B) = e\left(k_{\mathtt{id}_A,\mathrm{LEFT}}, H_1(\mathtt{id}_B)\right) \tag{4.2}$$

$$f_{BA} = F_s(\mathtt{id}_B, \mathtt{id}_A) = e\left(H_0(\mathtt{id}_B), k_{\mathtt{id}_A,\mathrm{RIGHT}}\right) \tag{4.3}$$

Similarly, B can evaluate F at the same points using the constraining keys $k_{\mathtt{id}_B,\mathrm{LEFT}}$ and $k_{\mathtt{id}_B,\mathrm{RIGHT}}$:

$$f_{AB} = F_s(\mathtt{id}_A, \mathtt{id}_B) = e\left(H_0(\mathtt{id}_A), k_{\mathtt{id}_B,\mathrm{RIGHT}}\right) \tag{4.4}$$

$$f_{BA} = F_s(\mathtt{id}_B, \mathtt{id}_A) = e\left(k_{\mathtt{id}_B,\mathrm{LEFT}}, H_1(\mathtt{id}_A)\right) \tag{4.5}$$

Using this key material, A and B can establish a symmetric secret key using a standardised key derivation function:

$$k_{AB} = k_{BA} = \mathbf{KDF}\left(f_{AB} \oplus f_{BA}\right) = \mathbf{KDF}(f_{BA} \oplus f_{AB}) \tag{4.6}$$

A note on security – The constraining keys $k_{\mathtt{id}_A,\mathrm{LEFT}}$ and $k_{\mathtt{id}_A,\mathrm{RIGHT}}$ allow to compute every symmetric key that A may establish with her contacts. As such, those **constraining keys must remain private** to A. The consequences of a leak range from impersonation to a total leak of A's address book and are further detailed in section 4.3.

4.2.3 Discovery

Using their shared key material (k_{AB}, f_{AB}, f_{BA}), users A and B can determine secret memory locations on the online cache to leave an encrypted message for each other. Let Enc, Dec be a secure symmetric encryption scheme and H a hash function modelled as a random oracle, we define two cache operations **Write** and **Read**:

- **Write**(f_{AB}): store the key-value pair $(H(f_{AB}), \mathsf{Enc}_{k_{AB}}(\mathsf{pk}_A || \mathbf{addr}_A))$ on the online cache.

- **Read**(f_{BA}): retrieve the key-value pair $(H(f_{BA}), c_{BA})$. If B has already run the discovery phase of our scheme then $c_{BA} = \mathsf{Enc}_{k_{BA}}(\mathsf{pk}_B || \mathbf{addr}_B)$. Decrypt c_{BA} using the key $k_{AB} = k_{BA}$.

Using these two operations, A is able to leave a message for B to find (**Write**) and check whether B has previously completed the contact matching process (**Read** at the address $H(f_{BA})$). Both users regularly check the relevant memory locations for a message. Once both users have completed the contact discovery process, they will hold each other's public keys and address, allowing them to communicate securely.

4.2.4 Setup

The setup stage serves to provide user A with the constraining keys $k_{\mathtt{id}_A,\mathrm{LEFT}}$ and $k_{\mathtt{id}_A,\mathrm{RIGHT}}$. Consequently, the setup is a security-critical task. As we have shown in Equation 3.11, under our construction of F the constraining keys can be expressed as:

$$k_{\mathtt{id}_A,\mathrm{LEFT}} = H_0(\mathtt{id}_A)^s \quad \text{and} \quad k_{\mathtt{id}_A,\mathrm{RIGHT}} = H_1(\mathtt{id}_A)^s \tag{4.7}$$

These constraining keys are equivalent to BLS signatures on $\mathtt{id}_A$ by at least t out of n servers of the discovery service. Notice that the service needs to produce signatures under both variants of the BLS scheme: one with signatures in $\mathbb{G}_0$ and one with signatures in $\mathbb{G}_1$.

The setup protocol between user A and a server S_i is described as follows:

1. S_i issues a challenge c

2. A chooses a random blinding factor $\alpha \leftarrow_{\$} \mathbb{Z}_q$ and sends $\mathbf{acc}_A$, $\mathbf{sig}_A \leftarrow \mathsf{DSA.Sign}(\mathsf{sk}_A, \mathbf{acc}_A || c)$, $\sigma_{\alpha,0} \leftarrow H_0(\mathtt{id}_A)^\alpha$, $\sigma_{\alpha,1} \leftarrow H_1(\mathtt{id}_A)^\alpha$ to S_i.

3. Upon reception of A's request, S_i retrieves the associated public key and checks that the signature $\mathbf{sig_A}$ is valid:

$$\mathsf{DSA.Verify}(\mathsf{pk}_A, \mathbf{acc}_A || c, \mathbf{sig}_A) = 1 \tag{4.8}$$

4. If the check succeeds, S_i sends $\widehat{\sigma}_{i,0} \leftarrow {\sigma_{\alpha,0}}^{s_i}$ and $\widehat{\sigma}_{i,1} \leftarrow {\sigma_{\alpha,1}}^{s_i}$ to A.

5. Using S_i's public keys $({g_0}^{s_i}, {g_1}^{s_i})$, A checks the following equalities:

$$e\left(\widehat{\sigma}_{i,0}, g_0\right) = e\left(H_0(\mathtt{id}_A)^\alpha, {g_0}^{s_i}\right) \tag{4.9}$$

$$e\left(g_1, \widehat{\sigma}_{i,1}\right) = e\left({g_1}^{s_i}, H_1(\mathtt{id}_A)^\alpha\right) \tag{4.10}$$

6. If the checks succeed (in other words, if A receives valid signatures from the service), A removes the blinding factor α to obtain $H_0(\mathtt{id}_A)^{s_i}$ and $H_1(\mathtt{id}_A)^{s_i}$.

A repeats the above procedure with at least t servers. Using the obtained signature shares, A can recover the full signatures $H_0(\mathtt{id}_A)^s$ and $H_1(\mathtt{id}_A)^s$ using $\mathsf{BTBLS.Combine}$.

This completes our description of the contact discovery scheme. We have seen how the setup process allows users to obtain their private constraining keys. Using those keys, users can locally and asynchronously derive shared key material with their contacts by evaluating a left/right constrained PRF at specific points. Finally, using the shared key material, users can leave and read messages from an untrusted online cache, thus completing the contact discovery process.

4.3 Privacy

We will now evaluate the privacy guarantees of our scheme when there are strictly less than t malicious servers. Our scheme hides the links between users as long as the decisional bilinear Diffie-Hellman assumption holds for the pairing e, the master secret key s does not leak and both constraining keys $k_{X,\text{LEFT}}, k_{X,\text{RIGHT}}$ are known only to user X. We present the threat model, potential attacks, outlines of security proofs as well as the consequences of a security breach.

4.3.1 Threat model

An adversary $\mathcal{T}$ wishing to break our scheme's privacy property aims to gain information about the contents of any user's address book. This goal is equivalent to determining whether $\texttt{id}_B \in \mathcal{C}_A$, for any user A and any identifier $\texttt{id}_B$ that is not owned by $\mathcal{T}$. $\mathcal{T}$ is characterised as:

- having access to all public information.
- having access to the present and past states of the online cache.
- may eavesdrop on any communication between the users, servers and online cache.
- may spawn any number of users for which $\mathcal{T}$ owns the discovery identifier.
- may control up to $t-1$ servers in the discovery service.

Notice that we are working under the assumption that discovery identifiers are correctly linked to the users who own them. We discuss ways in which this assumption can be upheld in practice in section 4.3.2, under "**Impersonating a user**"

4.3.2 Proof outline

To guide our analysis, we provide an attack tree[1] against the privacy property of our scheme in Figure 4.2. The root node represents the attacker's goal and each child node represents an option to solve the problem indicated in the parent node. Consequently, leaf nodes represent the attacker's entry points: breaking the security of F, obtaining the master secret key s, forging BLS signatures on another user's discovery identifier, impersonating a user or computing the shared key material (k_{AB}, f_{AB}, f_{BA}). We will therefore consider each leaf node and show that our scheme is resistant against these attacks.

[1]as defined by Schneier [16]

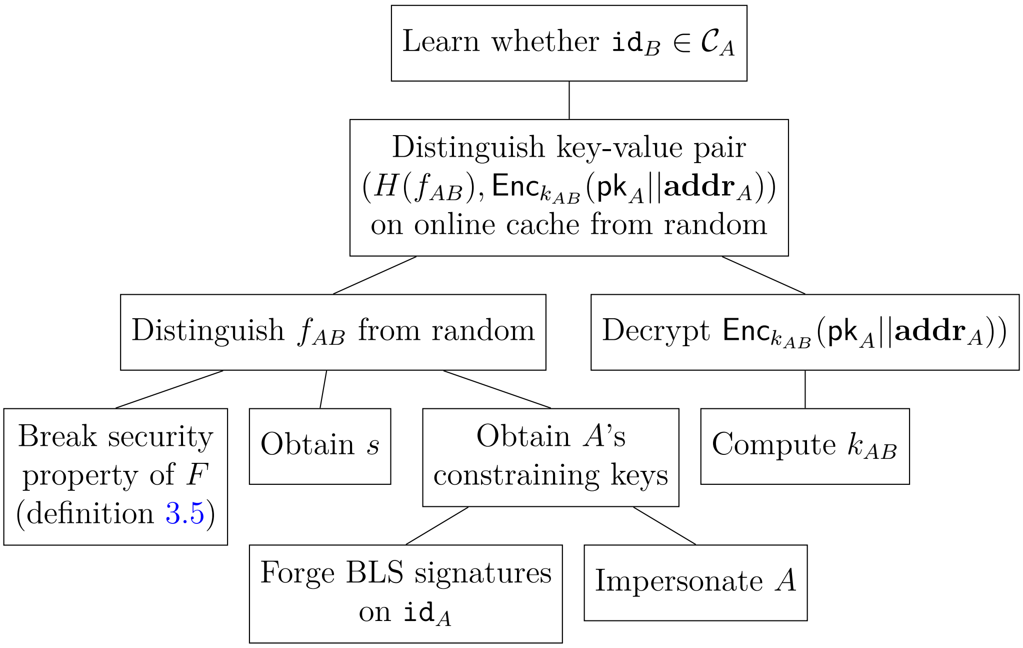

Figure 4.2: Attack tree against our discovery scheme. Branches represent "OR" statements

Security of our PRF construction

We will first show that our construction for the left/right constrained PRF using an asymmetric pairing is secure as per definition 3.5. Our construction is closely related to the one presented by Boneh and Waters [4]. As such our proof sketch makes use of very similar ideas.

Theorem 4.1. *The PRF F defined as $F(k,(x,y)) = e\left(H_0(x), H_1(y)\right)^k$ is a secure constrained PRF with respect to its constraining keys assuming the decisional bilinear Diffie-Hellman assumption holds for e and the functions H_0 and H_1 are modelled as random oracles.*

Proof sketch. We assume for contradiction the existence of a probabilistic polynomial-time adversary $\mathcal{A}$ that distinguishes F from random as in definition 3.5, however $\mathcal{A}$ is limited to a single $\mathsf{Challenge}$ query. We can then construct an adversary $\mathcal{B}$ that breaks the decisional bilinear Diffie-Hellman (DBDH) assumption.

Given $(g_0, g_1, u_0 \leftarrow {g_0}^{\alpha}, u_1 \leftarrow {g_1}^{\alpha}, v_0 \leftarrow {g_0}^{\beta}, w_1 \leftarrow {g_1}^{\gamma}, z^{(b)})$, $\mathcal{B}$'s goal is to determine whether $z^{(b)} = z^{(0)} = {g_0}^{\alpha\beta\gamma}$ or $z^{(b)} = z^{(1)} = {g_0}^{\delta}$, where $\delta \xleftarrow{\$} \mathbb{Z}_q$ (see Attack Game A.2 in

Appendix A). Using the pairing operation, this game can be viewed as distinguishing the output of F from a random element of $\mathbb{G}_T$. Indeed let $b, c \in \mathcal{X}$ such that $H_0(b) = {g_0}^{\beta}$ and $H_1(c) = {g_1}^{\gamma}$, then:

$$e\left(z^{(b)}, g_1\right) = \begin{cases} e\left(g_0, g_1\right)^{\alpha\beta\gamma} = e\left({g_0}^{\beta}, {g_1}^{\gamma}\right)^{\alpha} = F(\alpha, (b, c)), & \text{if } b = 0 \\ e\left(g_0, g_1\right)^{\delta} = {g_T}^{\delta}, & \text{if } b = 1 \end{cases} \tag{4.11}$$

Thus, $\mathcal{B}$ will run $\mathcal{A}$ as a sub-routine and must therefore emulate its oracles, namely $F.\mathsf{eval}$, $F.\mathsf{constrain}$, $\mathsf{Challenge}$ and oracles for the hash functions H_0, H_1.

When $\mathcal{A}$ issues a query to $H_0(x)$, $\mathcal{B}$ chooses a consistent random $\widehat{x} \leftarrow_{\$} \mathbb{Z}_q$ and sets $H_0(x) \leftarrow {g_0}^{\widehat{x}}$. To one of $\mathcal{A}$'s queries to H_0 which we denote x^*, $\mathcal{B}$ responds with $H_0(x^*) \leftarrow v_0$. Queries to H_1 are answered in a similar fashion where one query is responded to with $H_1(y^*) \leftarrow w_1$. Using these values, queries to $F.\mathsf{constrain}(x, LEFT)$ where $x \neq x^*$ are answered with $k_{x,LEFT} \leftarrow {u_0}^{\widehat{x}}$. Notice that as required

$${u_0}^{\widehat{x}} = ({g_0}^{\alpha})^{\widehat{x}} = {g_0}^{\alpha\widehat{x}} = ({g_0}^{\widehat{x}})^{\alpha} = H_0(x)^{\alpha}$$

Similarly, queries to $F.\mathsf{constrain}(y, RIGHT)$ where $y \neq y^*$ are answered with $k_{y,RIGHT} \leftarrow {u_1}^{\widehat{y}}$. Queries to $F.\mathsf{eval}(x, y)$ are answered for $x \neq x^*$ or $y \neq y^*$ by building the constraining keys as it is done for the $F.\mathsf{constrain}$ oracle. Notice that $\mathcal{B}$ does not hold the values β, γ and is therefore unable to answer queries to the $F.\mathsf{constrain}$ oracle for $(x^*, LEFT)$ and $(y^*, RIGHT)$, nor can it answer the $F.\mathsf{eval}$ query for (x^*, y^*). This is in fact equivalent to starting Attack Game 3.1 with the set C initialised to $\{(x^*, y^*)\}$.

After n queries to the H_0 oracle and m queries to the H_1 oracle, $\mathcal{A}$ will hold at most $n \times m$ pairs on which it could challenge. Some of these pairs may have been added to the set V due to queries to $F.\mathsf{constrain}$ and $F.\mathsf{eval}$, and thus become ineligible for challenging. However, since the experiment started with $C = \{(x^*, y^*)\}$, we can be sure that (x^*, y^*) is an eligible pair (remember that the attack game maintains the invariant $C \cap V = \emptyset$). Therefore, $\mathcal{A}$ will challenge the pair (x^*, y^*) with probability $p \geq \frac{1}{n \times m}$, to which $\mathcal{B}$ answers with $z^{(b)}$.

If $b = 0$, then $z^{(b)} = F(\gamma, (x^*, y^*))$ and $\mathcal{A}$ will answer as in experiment EXP(0). On the other hand if $b = 1$, then $z^{(b)} = {g_T}^{\delta}$ and $\mathcal{A}$ will answer as in experiment EXP(1). Let $b'_{\mathcal{A}}$

be the output of $\mathcal{A}$, we define as $b'_{\mathcal{B}} \leftarrow b'_{\mathcal{A}}$ the return value of $\mathcal{B}$. Thus

$$\Pr[b'_{\mathcal{B}} = b] \geq \frac{1}{n \times m} \times \Pr[b'_{\mathcal{A}} = b] \tag{4.12}$$

Given that $\mathcal{A}$ is a probabilistic polynomial-time adversary, $n \times m$ must necessarily be polynomial in λ. Therefore, if $\mathcal{A}$'s advantage is non-negligible then so is $\mathcal{B}$'s, thus breaking the DBDH assumption and yielding a contradiction. □

Computing A and B's shared key material

To compute A and B's shared key material, an attacker needs to compute f_{AB} and f_{BA}. This is in fact a harder problem than the decisional problem investigated above. We have already shown that no probabilistic polynomial-time adversary can break the security of F under the DBDH assumption. Similarly, no probabilistic polynomial-time adversary will be able to compute either f_{AB} or f_{BA} under the DBDH assumption without access to the relevant constraining keys or the master secret key.

Obtaining the master secret key

Next, we consider the option for an attacker to obtain the master secret key s. It is part of our assumption that there are strictly less than t malicious servers. Therefore, they do not meet the threshold required to construct the master secret key. An attacker aiming to break our scheme through this attack will need to steal at least one of the key shares.

Forging a BLS signature

As we have seen in subsection 4.2.4, the service generates constraining keys by signing a user's discovery identifier. The signing algorithm is a blind (t, n)-threshold BLS algorithm. As per Theorem 3.1, the BLS signature is existentially unforgeable against chosen message attacks under the co-Computational Diffie Hellman assumption. This assumption is in fact a weaker than the DBDH assumption which is required for the left/right constrained PRF security.

Impersonating a user

Impersonation attacks are the most threatening to our scheme and lead to open questions. We first describe the issue and offer two solutions, neither of which are fully satisfying. The attack is performed by running the setup process maliciously from the user side: upon receiving a challenge c, $\mathcal{T}$ can send the tuple ($\mathbf{acc}_{\mathcal{T}}$, $\mathsf{DSA.Sign}(\mathsf{sk}_{\mathcal{T}}, \mathbf{acc}_{\mathcal{T}}||c)$, $H_0(\mathtt{id}_A)^\alpha$, $H_1(\mathtt{id}_A)^\alpha$). The server S_i receiving this tuple will find that the signature $\mathsf{DSA.Sign}(\mathsf{sk}_{\mathcal{T}}, \mathbf{acc}_{\mathcal{T}}||c)$ does verify for the specified account and challenge. Furthermore, S_i will be unable to distinguish the blinded values $H_0(\mathtt{id}_A)^\alpha, H_1(\mathtt{id}_A)^\alpha$ from random elements in $\mathbb{G}_0$ and $\mathbb{G}_1$ respectively. As such S_i will issue partial constraining keys for $\mathtt{id}_A$ to $\mathcal{T}$. Repeating this process with t servers, $\mathcal{T}$ will obtain the full constraining keys for user A.

The first solution is for A to transmit her discovery identifier in clear to S_i. The server can then use out-of-bound communication to verify that A indeed owns $\mathtt{id}_A$ (possible techniques include sending a one-time code via text message or email). If A proves that she owns $\mathtt{id}_A$, S_i provides the partial signatures for that discovery identifier. Notice that under this approach, A cannot blind the hash of her discovery identifier. Consequently, the communication between A and S_i must be encrypted to prevent an eavesdropping adversary from learning the signature share on $\mathtt{id}_A$. Furthermore, this identification method allows the servers to build a list of identifiers for the users registered to the mobile application. In some cases, this may be a breach of privacy.

The second solution makes use of identification tokens to delegate the task of linking a user to their discovery identifier. Suppose an entity V (centralised or distributed) is trusted to verify whether a user owns a discovery identifier. Using a secret key $v \leftarrow_{\$} \mathbb{Z}_q$ and the corresponding public keys ${g_0}^v$ and ${g_1}^v$, V could issue ownership tokens in the form of BLS signatures $t_{0,A} = H_0(\mathtt{id}_A)^v, t_{1,A} = H_1(\mathtt{id}_A)^v$. These tokens can then be blinded and verified against a blinded discovery identifier $H_0(\mathtt{id}_A)^\alpha, H_1(\mathtt{id}_A)^\alpha$:

$$e\left({t_{0,A}}^\alpha, g_1\right) = e\left(H_0(\mathtt{id}_A)^\alpha, {g_1}^v\right) \iff t_{0,A} = H_0(\mathtt{id}_A)^v \tag{4.13}$$

$$e\left(g_0, {t_{1,A}}^\alpha\right) = e\left({g_0}^v, H_1(\mathtt{id}_A)^\alpha\right) \iff t_{1,A} = H_1(\mathtt{id}_A)^v \tag{4.14}$$

Users can therefore send ${t_{0,A}}^\alpha$, ${t_{1,A}}^\alpha$ along with the setup tuple ($\mathbf{acc}_A$, $\mathbf{sig}_A$, $H_0(\mathtt{id}_A)^\alpha$, $H_1(\mathtt{id}_A)^\alpha$), to allow each server to perform the checks in Equation 4.13 and Equation 4.14.

While this method allows identification without revealing the discovery identifier to the contact discovery servers, it relies on a trusted verification entity V. In fact, this entity faces the same problem we were trying to avoid: it must output a BLS signature on an identifier only if the request was made by the identifier's owner. This raises the question of trust within our system. The contact discovery scheme can be made secure and oblivious to which user owns which discovery identifier. However, for that to happen, we need another entity to perform that check and gather private information about the users.

4.3.3 Consequences of a breach

To conclude our investigation of the scheme's privacy properties, we evaluate the consequences of various breaches of the protocol. Let us first consider the scenario in which a pair of constraining keys $k_{\mathtt{id}_A,\mathrm{LEFT}}, k_{\mathtt{id}_A,\mathrm{RIGHT}}$ is leaked. Using these keys, an attacker will be able to compute the shared key material (k_{AX}, f_{AX}, f_{BX}) between A and any other user X. The attacker is then able to:

(a) check whether X has written to the cache in location $H(f_{XA})$, thus uncovering whether $\mathtt{id}_A \in \mathcal{C}_X$. Iterating over all $\mathtt{id}_X \in \mathcal{ID}$ allows to determine which users hold $\mathtt{id}_A$ in their contacts.

(b) decrypt the message (if any) left by X at location $H(f_{XA})$ using the key k_{AX}, thus linking $\mathtt{id}_X$ to an address and public key.

(c) check whether A has written to the cache in location $H(f_{AX})$, thus uncovering whether $\mathtt{id}_X \in \mathcal{C}_A$. Iterating over all $\mathtt{id}_X \in \mathcal{ID}$ allows to recover all of A's contacts.

(d) overwrite the value that A wrote in location $H(f_{AX})$ using the key k_{AX}. This allows the attacker to send any address and public key to X, thus hijacking any channel that A and X were wishing to establish.

Should only one of the constraining keys leak, the attacker will only be able to perform a subset of the actions above. Indeed, obtaining a *LEFT* key only allows to compute f_{AX}. The attacker will therefore only be able to perform action (c). Similarly, obtaining only a *RIGHT* key limits the attacker's possibilities to (a). In either case, these breaches represent a complete loss of privacy. Finally, obtaining the master secret key allows to compute any constraining key, thus allowing to perform the above operations for any pair of users.

4.4 Theoretical performance evaluation

In this section we present a brief evaluation of the scheme's efficiency. Importantly, we show that all computational costs grow linearly with respect to the input size. We estimate the computation time associated with each phase of our discovery scheme using benchmark timings from the mobile-friendly elliptic curve pairing library MCL [13]. These benchmark tests were performed on an iPhone 7 running iOS 11.2.1, executing operations over three Barreto-Naehrig (BN) elliptic curves with varying security parameters [12]. The results are summarised in Table 4.1. Notice that we are now working with elliptic curves and therefore adopt an additive notation for the group operation.

Operation	Notation	BN254	BN381_1	BN462
Pairing	$\mathbf{p}$	3.9	11.752	22.578
Addition in $\mathbb{G}_0$	$\mathbf{add}_{\mathbb{G}_0}$	0.006	0.015	0.018
Point doubling in $\mathbb{G}_0$	$\mathbf{dbl}_{\mathbb{G}_0}$	0.005	0.01	0.019
Multiplication in $\mathbb{G}_0$	$\mathbf{mul}_{\mathbb{G}_0}$	0.843	2.615	5.339
Addition in $\mathbb{G}_1$	$\mathbf{add}_{\mathbb{G}_1}$	0.015	0.03	0.048
Point doubling in $\mathbb{G}_1$	$\mathbf{dbl}_{\mathbb{G}_1}$	0.011	0.022	0.034
Multiplication in $\mathbb{G}_1$	$\mathbf{mul}_{\mathbb{G}_1}$	1.596	4.581	9.077
Hash to $\mathbb{G}_0$	$\mathbf{hash}_{\mathbb{G}_0}$	0.212	0.507	1.201
Hash to $\mathbb{G}_1$	$\mathbf{hash}_{\mathbb{G}_1}$	3.486	9.93	21.817

Table 4.1: Timing benchmarks for elliptic curve operations over three pairing-friendly curves (BN254, BN381_1 and BN462) executed on an iPhone 7 running the MCL library [13] on iOS 11.2.1. All timings are given in milliseconds. [12]

Computational cost: Setup with identification tokens

First, let us inspect the setup phase with identification tokens. This phase only needs to be performed once per user and per server. As such we will evaluate the computational cost of a single user-server interaction, then consider the scaling with respect to the total number of users N and the threshold of servers t. We assume that users already hold a hash of their own discovery identifier. By inspection of the protocol, the setup phase requires:

User: 2 multiplications in $\mathbb{G}_0$, 2 multiplications in $\mathbb{G}_1$ (blind and unblind identifier), 4 pairing operations (Equation 4.9, Equation 4.10), one standard digital signature. Identification tokens introduce one additional multiplication in $\mathbb{G}_0$, and one multiplication in $\mathbb{G}_1$ (blind tokens). After repeating these operations with enough servers to meet the system's threshold, a user is required to recombine t partial BLS signatures in both $\mathbb{G}_0$ and $\mathbb{G}_1$. Using Lagrange interpolation, each of these recombinations require t multiplications and t additions in their respective source group. We can now express the time spent by each user on computations for the setup phase as a function of t:

$$\text{comp}_{\text{setup,user}}(t) = t(4(\mathbf{mul}_{\mathbb{G}_0} + \mathbf{mul}_{\mathbb{G}_1} + \mathbf{p}) + \mathbf{add}_{\mathbb{G}_0} + \mathbf{add}_{\mathbb{G}_0} + \mathbf{DSA}_S) \quad (4.15)$$

where $\mathbf{DSA}_S$ is the time to execute the digital signature algorithm.

Server: 1 multiplication in $\mathbb{G}_0$, 1 multiplication in $\mathbb{G}_1$ (BLS signature in each source group) and one standard digital signature verification. Identification tokens introduce 4 additional pairing operations (Equation 4.13, Equation 4.14). We express the time spent by each server on computations for the setup phase as a function of N:

$$\text{comp}_{\text{setup,server}}(N) = N(4\mathbf{p} + \mathbf{mul}_{\mathbb{G}_0} + \mathbf{mul}_{\mathbb{G}_1} + \mathbf{DSA}_V) \quad (4.16)$$

where $\mathbf{DSA}_V$ is the time to verify the digital signature.

Assuming we are using curve BN381_1 and using realistic values for $\mathbf{DSA}_S$ and $\mathbf{DSA}_V$ on a mobile device (respectively 0.82 ms and 3.02 ms [21]) yields:

$$\text{comp}_{\text{setup,user}}(t) = 76.66\text{ms} \quad \text{and} \quad \text{comp}_{\text{setup,server}}(N) = 57.22\text{ms} \quad (4.17)$$

For a threshold of servers set to 6, a single user can complete the setup phase while spending less than 0.5 seconds performing local computations. Similarly, servers can setup a new user within tens of milliseconds. Launching our discovery service for an application with 10 million users can be done with slightly more than one day of serial computations. However, with no optimisations, an application with one billion users would require a roll out period of almost two years.

In order to render the service practical for widely used applications, we must investigate methods to optimise the server-side setup process. The most costly operation performed by the server are the verifications of the identity tokens. These are in fact verification of BLS signatures, which can be aggregated to reduce the number of pairing operations needed [1]. We can establish optimal batching strategies by placing assumptions on the rate of false signatures; however, such strategies are outside of the scope of this report. Promising approaches to establishing an optimal strategy may come from other fields of research [17].

Computational cost: Shared key derivation

We now consider the local key derivation phase on a per-contact basis, before investigating how computations scale with the number of contacts N_c. A user must perform two evaluations of a left/right constrained PRF. Under our construction, this amounts to performing one hash to each source group and two pairing operations. Thus:

$$\text{comp}_{\text{key,user}}(N_c) = N_c(2\mathbf{p} + \mathbf{hash}_{\mathbb{G}_0} + \mathbf{hash}_{\mathbb{G}_1}) \tag{4.18}$$

Using the values from curve BN381_1 in Table 4.1 yields a very fast 34ms per address book entry. Thus an initial computation over an address book of one thousand entries would only require 34 seconds of computations.

Concluding remarks

The discovery phase makes use of standard cryptographic operations and is of lesser interest when estimating the computational costs of our scheme. However, a similar analysis of the scheme can be performed for communication costs. These are largely dependent on the type of mobile network that is being used as well as the availability of the servers and the online cache. We do not perform this analysis in our report, however we wish to emphasise that our scheme requires very few communications rounds to complete all three phases.

Overall, we have seen that all computations grow linearly with respect to their inputs. Using realistic numbers of registered users, we have seen that our service may be immediately applicable to relatively popular applications (in the range of tens of millions of users). For more popular applications such as WhatsApp, our system will require optimisations of the server-side setup process. Under our construction of the left/right constrained PRF, the most promising approach is to batch token verifications to reduce the number of pairing operations required.

4.5 Applications

To conclude this chapter on our system's architecture, we provide a brief overview of its possible integrations into existing mobile applications. We first focus on Signal[2], an end-to-end encrypted messaging service. We then consider the integration with a decentralised payment system such as Celo[3].

It is important to note that the same infrastructure can be used for multiple applications simultaneously. Indeed, each application can be represented by a different master secret key. Let s_X and s_Y be the master secret keys for applications X and Y respectively. Users A and B can interact with the discovery service under both keys to derive the shared secrets $F_{s_X}(\mathtt{id}_A, \mathtt{id}_B), F_{s_X}(\mathtt{id}_B, \mathtt{id}_A)$ to perform contact discovery for application X and the shared secrets $F_{s_Y}(\mathtt{id}_A, \mathtt{id}_B), F_{s_Y}(\mathtt{id}_B, \mathtt{id}_A)$ to perform contact discovery for application Y. However, this once again raises the question of optimisation of the server-side setup process.

4.5.1 End-to-end encrypted messaging

The Signal Protocol uses the "X3DH" (Extended Triple Diffie-Hellman) protocol to "[establish] a shared secret between two parties that mutually authenticate each other" [10]. Users A and B first exchange a set of public keys to then derive a shared secret. We can modify our discovery scheme such that the initial ciphertext left on the online cache contains the necessary public key information to perform the X3DH protocol. As the meeting point and encryption key are known only to A and B, our scheme provides mutual authentication. Following this initial key establishment, the Signal Protocol can be followed as expected normally. This allows features such as the "Double Ratchet" algorithm to be used to provide secrecy to past and future messages in the event of a key leak [9]. Such features allow to isolate the end-to-end encrypted channel from faults in the discovery scheme.

Signal also performs a phone number registration step which makes use of out-of-bound verification [18]. This step is less documented, however its existence implies that Signal is able to issue identification tokens to its users, thus protecting our scheme from impersonation attacks.

[2] `https://signal.org`
[3] `https://celo.org`

4.5.2 Mobile-first cryptocurrencies

Celo is a decentralised payment system based on a blockchain architecture. Users can therefore send and receive transactions using account addresses - a 30+ character hexadecimal string. As Celo aims to bring fast payment systems to "anyone with a smartphone" [5], they provide a service which allows users to ignore these complex addresses and use phone numbers instead. This setting naturally maps to our contact discovery scheme (phone number as a discovery identifier linked to the Celo-provided secret key, public key and address).

Through its blockchain and the validators that run it during a given epoch, Celo offers a decentralised phone number attestation service [6]. Once a user proves they own a phone number, a salted hash of the number and the associated Celo account are committed to the blockchain. Users can discover each other by issuing oblivious, rate-limited requests for a user's salt to then check the on-chain phone number attestation records.

Instead, Celo could provide stronger privacy guarantees by issuing private identification tokens through its attestation system and delegating the contact discovery process. Our contact discovery service could then use the public keys associated to the combined secret key of the validators of a given epoch to verify these tokens. As this authentication step prevents impersonation attacks, our scheme can then run as described in this chapter. Doing so allows to securely link phone numbers to account addresses without committing this data to a public blockchain.

Chapter 5

Proof-of-Concept Implementation

In this chapter we describe a proof-of-concept implementation of the contact discovery service written in Go. At the time of writing, this proof-of-concept performs setup locally by emulating the behaviour of the distributed discovery service. Key derivation is performed locally as expected. Finally, a meeting point is established via the InterPlanetary FileSystem (IPFS)[1]. It is important to highlight that the IPFS is a content-addressed system: rather than storing key-value pairs, the IPFS derives a key as a function of the value. This behaviour does not match our requirements for the online cache, but allows us to establish a meeting point and perform contact discovery nonetheless.

5.1 Local server emulation

To emulate the behaviour of our distributed discovery service, we need to create a `server` object, perform a distributed key generation (DKG) algorithm and implement BLS signatures in both source groups of an asymmetric pairing. We make use of the `kyber` library[2] to provide most of the cryptographic backend.

Server representation – We are performing a local emulation and therefore choose to abstract from networking properties such as a server's address. We however include an `ID` field that represents any such identifying information. Consequently, our model for a server is as simple as possible: it includes an identifier, a secret key share for the BLS signature scheme in $\mathbb{G}_0$ and a secret key share for the BLS signature scheme in $\mathbb{G}_1$ (see Figure 5.1).

[1] https://ipfs.io
[2] https://github.com/dedis/kyber

```
type multiServer struct {
  ID  int
  sk1 *share.PriShare
  sk2 *share.PriShare
}
```

Figure 5.1: Implementation: definition of a server

Distributed Key Generation – Rather than performing a distributed key generation algorithm, we assume the existence of a trusted dealer and perform key distribution by sharing a random secret (see Figure 5.2). As DKG algorithms are not the primary focus of our report, this assumption allows for a simple setup for our proof-of-concept implementation. We perform secret sharing using `kyber`'s `share` package.

```
func setupThresholdServers(suite pairing.Suite, secret kyber.Scalar, n, t
    int) ([]*multiServer, *share.PubPoly, *share.PubPoly) {
  serverList := make([]*multiServer, n)
  if secret == nil {
    secret = suite.GT().Scalar().Pick(random.New())
  }

  priPoly1 := share.NewPriPoly(suite.G2(), t, secret, random.New())
  pubPoly1 := priPoly1.Commit(suite.G2().Point().Base())
  serverPrivateKeys1 := priPoly1.Shares(n)

  priPoly2 := share.NewPriPoly(suite.G1(), t, secret, random.New())
  pubPoly2 := priPoly2.Commit(suite.G1().Point().Base())
  serverPrivateKeys2 := priPoly2.Shares(n)

  for i := 0; i < n; i++ {
    serverList[i] = newMultiServer(i, serverPrivateKeys1[i],
        serverPrivateKeys2[i])
  }

  return serverList, pubPoly1, pubPoly2
}
```

Figure 5.2: Implementation: Key distribution using a trusted dealer

Blind (t, n)-threshold BLS – Finally, we implement blind (t, n)-threshold BLS signature schemes in both variants (with signatures in $\mathbb{G}_0$ and in $\mathbb{G}_1$). The `kyber` library only allows signatures in $\mathbb{G}_0$ and takes messages as inputs. As such, we are unable to manipulate hashes of those messages; more specifically we are unable to blind and unblind our messages. We therefore implement a slight variant of the existing library to allow for blinding and introduce the necessary functions to performs BLS signatures on elements of $\mathbb{G}_1$. We do not however implement a secure hash-to-$\mathbb{G}_1$ as should be the case in a production-grade service.

Using the above setup, clients are able to send their blinded discovery identifiers to any of the n emulated servers (see Appendix C, section C.5). The servers respond by providing a BLS signature using their private key shares. Users therefore receive their constraining keys as expected. We do not however implement many of the identity checks that are required to provide a secure setup.

5.2 User-facing client application

Users – We consider that each user will run an instance of our code. Users are therefore prompted to enter their discovery identifier upon first launch. This identifier is then hashed to both source groups to produce public keys `pk1` and `pk2`. Once the user completes the setup process, she will receive her left and right constraining keys. We call these the user's secret keys `sk1` ans `sk2` to emphasise the fact that both keys must remain private at all times. Users are therefore represented using the data structure shown in Figure 5.3.

```
type user struct {
  name                string
  phoneNumber         string
  pk1, pk2, sk1, sk2 kyber.Point
}
```

Figure 5.3: Implementation: definition of a user

User setup – Upon launching the application, users receive a list of available servers and the setup threshold t. The client application performs the setup process by interacting with t servers of its choice. Each interaction consists of blinding the user's public keys, verifying the received signature and unblinding it to store shares of the constraining keys. When enough shares are gathered, the client application runs the Combine algorithms from each of the two threshold BLS schemes.

Key derivation – Using a user's constraining keys and a contact's discovery identifier, the client application can evaluate the left/right constrained PRF by performing two pairing operations (see Figure 5.4).

```go
// Derive shared keys between users A and B:
// shared12 = e(H1(idA)**s, H2(idB)) = e(H1(idA), H2(idB))**s
// shared21 = e(H1(idB), H2(idA)**s) = e(H1(idB), H2(idA))**s
func deriveSharedKeys(alice *user, contactNumber string) (kyber.Point,
    kyber.Point) {
  bobPk1, bobPk2 := derivePublicKeys(contactNumber)
  shared12 := suite.Pair(alice.sk1, bobPk2)
  shared21 := suite.Pair(bobPk1, alice.sk2)

  return shared12, shared21
}
```

Figure 5.4: Implementation: local key derivation

5.3 Online meeting point via IPFS

The final step required to successfully perform contact discovery is to establish an online meeting point. As mentioned above, the IPFS is not originally a key-value store. We therefore a develop another approach to the discovery phase which slightly differs from that presented in chapter 4.

The IPFS is a content-addressed storage system where the location of an object is its hash. Therefore, we modify the discovery phase such that both parties A and B, can compute two pieces of unique, secret content c_{AB} and c_{BA}. These are in fact ciphertexts under the symmetric key $k_{AB} = k_{BA}$ for standardised plaintexts such that both users can locally compute them. To check whether B is registered to an application, A can check whether c_{BA} is available on the IPFS. Similarly, B can check for the presence of c_{AB}. Notice however that we cannot encrypt information that is not shared between A and B. Indeed, doing so would mean that one of the two parties is unable to compute the hash — and therefore the IPFS address — of one of the ciphertexts. As a result, this simplified method does not allow to transfer information during the contact discovery phase. Users may only receive and send binary information by uploading or withholding their ciphertexts.

The IPFS provides simple command-line tools to upload and access files from its peer-to-peer network. Using these tools, A uploads c_{AB} and tries to retrieve c_{BA}. If the file is available, A knows B is a registered user. Otherwise, the IPFS instruction will time out and A will learn that B is not registered.

This process implies that c_{AB} and c_{BA} must remain available on the IPFS network regardless of either users' connection status. Fortunately, the IPFS implements a "pinning" mechanism to ensure that files are stored by more than one node.

5.4 Results

Chapter 6

Conclusion

Bibliography

[1] Boneh, D., Gentry, C., Lynn, B., , Shacham, H.: Aggregate and verifiably encrypted signatures from bilinear maps. Advances in Cryptology - EUROCRYPT 2003 volume 2656 of LNCS, pp 416–432 (2003)

[2] Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. Journal of Cryptography 17(4), pp 297–319 (2004)

[3] Boneh, D., Shoup, V.: A Graduate Course in Applied Cryptography (v0.5). Published online at https://toc.cryptobook.us (January 2020)

[4] Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. Cryptology ePrint Archive, Report 2013/352 (2013), https://eprint.iacr.org/2013/352

[5] Celo: Documentation. https://docs.celo.org/v/master/, retrieved online on 6 September 2020

[6] Celo: Phone number privacy. Celo Documnetation, https://docs.celo.org/celo-codebase/protocol/identity/phone-number-privacy, retrieved online 7 September 2020

[7] Kales, D., Rechberger, C., Schneider, T., Senker, M., Weinert, C.: Mobile private contact discovery at scale. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1447–1464. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/kales

[8] Marlinspike, M.: The difficulty of private contact discovery. https://signal.org/blog/contact-discovery/ (January 2014)

[9] Marlinspike, M.: The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/ (November 2016), revision 1 - Retrieved online on 6 September 2020

[10] Marlinspike, M.: The X3DH key agreement protocol. https://signal.org/docs/specifications/x3dh/ (November 2016), revision 1 - Retrieved online on 6 September 2020

[11] Marlinspike, M.: Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/ (September 2017)

[12] Mitsunari, S.: MCL benchmarks. https://github.com/herumi/mcl/blob/master/bench.txt (November 2018), last visited on 5 Sept. 2020

[13] Mitsunari, S.: MCL: a portable and fast pairing-based cryptography library. https://github.com/herumi/mcl (June 2020), last visited on 5 Sept. 2020

[14] Papadopoulos, P., Chariton, A., Athanasopoulos, E., Markatos, E.: Where's wally?: How to privately discover your friends on the internet. pp. 425–430 (05 2018)

[15] Reuters: Whatsapp users cross 2 billion, second only to facebook. Online https://www.reuters.com/article/us-whatsapp-users-idUSKBN20626L (February 2020), retrieved on 28th August 2020

[16] Schneier, B.: Attack trees. *Dr. Dobb's Journal* (December 1999), retrieved online from https://www.schneier.com/academic/archives/1999/12/attack_trees.html

[17] Shental, N., Levy, S., Wuvshet, V., Skorniakov, S., Shalem, B., Ottolenghi, A., Greenshpan, Y., Steinberg, R., Edri, A., Gillis, R., Goldhirsh, M., Moscovici, K., Sachren, S., Friedman, L.M., Nesher, L., Shemer-Avni, Y., Porgador, A., Hertz, T.: Efficient high-throughput sars-cov-2 testing to detect asymptomatic carriers. Science Advances (2020), https://advances.sciencemag.org/content/early/2020/08/20/sciadv.abc5961

[18] Signal: Register a phone number. https://support.signal.org/hc/en-us/articles/360007318691-Register-a-phone-number, retrieved online on 6 September 2020

[19] Telegram: Telegram privacy policy. https://telegram.org/privacy, retrieved on 18th August 2020

[20] WhatsApp Inc.: Terms of Service. https://www.whatsapp.com/legal#terms-of-service, retrieved on 17th August 2020

[21] WolfSSL: Reference benchmarks. https://www.wolfssl.com/docs/benchmarks/, retrieved online on 5 Sept. 2020

Appendix A

Bilinear variants of the CDH and DDH problems

A.1 The co-computational Diffie-Hellman (co-CDH) Problem and Assumption

The co-Computational Diffie-Hellman (co-CDH) assumption is a variant of the Computational Diffie-Hellman assumption that applies for asymmetric pairings. Let us recall the definition for the co-Computational Diffie-Hellman assumption given in [3], using a multiplicative notation for the group operation as in the source text..

Attack Game A.1 (co-CDH [3]). *Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be three cyclic groups of prime order q such that there exists a pairing $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$. For a given adversary $\mathcal{A}$, the attack game runs as follows:*

- *The challenger picks at random $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_q$ and computes*

$$u_0 \leftarrow {g_0}^{\alpha}, \qquad u_1 \leftarrow {g_1}^{\alpha}, \quad v_0 \leftarrow {g_0}^{\beta}, \quad z_0 \leftarrow {g_0}^{\alpha\beta}$$

- *The adversary $\mathcal{A}$ receives the tuple (u_0, u_1, v_0) and outputs $\hat{z}_0 \in \mathbb{G}_0$*

We define the advantage of $\mathcal{A}$ in solving the co-CDH problem for e as:

$$\mathrm{coCDHadv}[\mathcal{A}, e] := \Pr(\hat{z}_0 = z_0) \tag{A.1}$$

Notice that for symmetric pairings, $\mathbb{G}_0 = \mathbb{G}_1$ therefore $g_0 = g_1$, $u_0 = u_1$ and attack game A.1 is identical to the Computational Diffie-Hellman attack game.

Definition A.1 (co-CDH Assumption [3])**.** *We say that the co-CDH assumption holds for the pairing e if for all efficient adversaries $\mathcal{A}$ the quantity* $\mathrm{coCDHadv}[\mathcal{A}, e]$ *is negligible.*

A.2 The decision bilinear Diffie-Hellman (DBDH) Problem and Assumption

The decisional variant is relatively straight-forward having already defined the co-CDH assumption. The attack setting is closely related, however the adversary is expected to distinguish an element from random (rather than required to computed it). Once again, the definition is adapted from [3] and uses a multiplicative notation for group operations.

Attack Game A.2 (Decision bilinear Diffie-Hellman [3])**.** *Let $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ be a pairing where $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ are cyclic groups of prime order q with generators $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$. For a given adversary $\mathcal{A}$, we define the following experiment:*

- *The challenger picks at random $\alpha, \beta, \gamma, \delta \xleftarrow{\$} \mathbb{Z}_q$, computes*

$$u_0 \leftarrow {g_0}^{\alpha}, \quad u_1 \leftarrow {g_1}^{\alpha}, \quad v_0 \leftarrow {g_0}^{\beta}, \quad w_1 \leftarrow {g_1}^{\gamma}, \quad z^{(0)} \leftarrow {g_0}^{\alpha\beta\gamma}, \quad z^{(1)} \leftarrow {g_0}^{\delta}$$

  and flips a bit $b \xleftarrow{\$} \{0, 1\}$. Using the result of the bit flip, the challenger sends $(u_0, u_1, v_0, w_1, z^{(b)})$ to $\mathcal{A}$.

- *$\mathcal{A}$ receives $(u_0, u_1, v_0, w_1, z^{(b)})$ and outputs a bit $\hat{b} \in \{0, 1\}$*

We define the advantage of $\mathcal{A}$ in solving the DBDH problem for e as:

$$\mathrm{DBDHavd}[\mathcal{A}, e] := \left| \frac{1}{2} - \Pr(\hat{b} = b) \right| \tag{A.2}$$

Definition A.2 (Decision BDH assumption [3])**.** *We say that the decision bilinear Diffie-Hellman assumption holds for the pairing e if for all efficient adversaries $\mathcal{A}$ the quantity* $\mathrm{DBDHavd}[\mathcal{A}, e]$ *is negligible.*

Appendix B

Calculations: performance evaluation

Appendix C

Proof-of-Concept: full code listing

C.1 Package hash

Listing C.1: hash/hashing.go

```
package hash

import (
  "errors"
  "log"

  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/pairing"
  "go.dedis.ch/kyber/v3/xof/blake2xb"
)

type hashablePoint interface {
  Hash([]byte) kyber.Point
}

// HashtoG1 securely hashes a message into a point on G1
func HashtoG1(suite pairing.Suite, msg []byte) kyber.Point {
  hashable, ok := suite.G1().Point().(hashablePoint)
  if !ok {
    log.Printf("Point cannot be hashed")
  }
  hashed := hashable.Hash(msg)
  return hashed
}
```

```go

// InsecureHashtoG2 hashes a message to a point in G2 by using the
    message as a seed for the Pick method
// !!! Unsure whether this is collision resistant !!!
// To be replaced by a secure version that follows https://tools.ietf.org
    /html/draft-irtf-cfrg-hash-to-curve-07
func InsecureHashtoG2(suite pairing.Suite, msg []byte) kyber.Point {
  seed := blake2xb.New(msg)
  hashed := suite.G2().Point().Pick(seed)

  return hashed
}

// Hash hashes a nsg to a point on the requested curve
func Hash(suite pairing.Suite, group kyber.Group, msg []byte) (kyber.
    Point, error) {
  if group.String() == "bn256.G1" {
    return HashtoG1(suite, msg), nil
  } else if group.String() == "bn256.G2" {
    return InsecureHashtoG2(suite, msg), nil
  } else {
    return nil, errors.New("hash: group not recognised")
  }
}
```

Listing C.2: hash/hashing_test.go

```go
package hash

import (
  "testing"

  "go.dedis.ch/kyber/v3/pairing/bn256"
)

func TestHashToG2(t *testing.T) {
  suite := bn256.NewSuite()
  testMsg := "this is a test message"
  hash1 := InsecureHashtoG2(suite, []byte(testMsg))
```

```
  hash2 := InsecureHashtoG2(suite, []byte(testMsg))

  if !hash1.Equal(hash2) {
    t.Errorf("Hashing the same message yield different points")
  }
}
```

C.2 Package morebls

Listing C.3: morebls/morebls.go

```
// Package morebls mirrors the kyber/bls package.
// Here, signatures are points on G2 and public keys are points on G1.
//
// WARNING: relies on an insecure hash-to-G2 function !
package morebls

import (
  "crypto/cipher"
  "errors"

  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/pairing"
  "go.dedis.ch/kyber/v3/xof/blake2xb"
)

// Hashes a message to a point in G2 by using the message as a seed for
    the Pick method
// !!! Unsure whether this is collision resistant !!!
// To be replaced by a secure version that follows https://tools.ietf.org
    /html/draft-irtf-cfrg-hash-to-curve-07
func insecureHashtoG2(suite pairing.Suite, msg []byte) kyber.Point {
  seed := blake2xb.New(msg)
  hashed := suite.G2().Point().Pick(seed)

  return hashed
}

```

```go
// NewKeyPair2 creates a new BLS signing key pair. The private key x is a scalar
// and the public key X is a point on curve G1.
func NewKeyPair2(suite pairing.Suite, random cipher.Stream) (kyber.Scalar, kyber.Point) {
  x := suite.G1().Scalar().Pick(random)
  X := suite.G1().Point().Mul(x, nil)
  return x, X
}

// Sign2 creates a BLS signature S = x * H(m) on a message m using the private
// key x. The signature S is a point on curve G2.
func Sign2(suite pairing.Suite, x kyber.Scalar, msg []byte) ([]byte, error) {
  HM := insecureHashtoG2(suite, msg)
  xHM := HM.Mul(x, HM)

  s, err := xHM.MarshalBinary()
  if err != nil {
    return nil, err
  }
  return s, nil
}

// Verify2 checks the given BLS signature S on the message m using the public
// key X by verifying that the equality e(X, H(m)) == e(x*B1, H(m)) ==
// e(B1, x*H(m)) == e(B1, S) holds where e is the pairing operation and B1 is
// the base point from curve G1.
func Verify2(suite pairing.Suite, X kyber.Point, msg, sig []byte) error {
  HM := insecureHashtoG2(suite, msg)
  left := suite.Pair(X, HM)
  s := suite.G2().Point()
  if err := s.UnmarshalBinary(sig); err != nil {
    return err
  }
  right := suite.Pair(suite.G1().Point().Base(), s)
  if !left.Equal(right) {
    return errors.New("bls: invalid signature")
```

```
  }
  return nil
}
```

Listing C.4: morebls/morebls_test.go

```
package morebls

import (
  "testing"

  "go.dedis.ch/kyber/v3/pairing/bn256"
  "go.dedis.ch/kyber/v3/util/random"
)

func TestBLS(t *testing.T) {
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  private, public := NewKeyPair2(suite, random.New())
  sig, err := Sign2(suite, private, msg)
  if err != nil {
    t.Errorf("%s", err)
  }
  err = Verify2(suite, public, msg, sig)
  if err != nil {
    t.Errorf("Signature did not match")
  }
}

func TestBLSFailSig(t *testing.T) {
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  private, public := NewKeyPair2(suite, random.New())
  sig, err := Sign2(suite, private, msg)
  if err != nil {
    t.Errorf("%s", err)
  }
  sig[0]  0x01
  if Verify2(suite, public, msg, sig) == nil {
```

```
    t.Fatal("bls: verification succeeded unexpectedly")
  }
}

func TestBLSFailKey(t *testing.T) {
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  private, _ := NewKeyPair2(suite, random.New())
  sig, err := Sign2(suite, private, msg)
  if err != nil {
    t.Errorf("%s", err)
  }
  _, public := NewKeyPair2(suite, random.New())
  if Verify2(suite, public, msg, sig) == nil {
    t.Fatal("bls: verification succeeded unexpectedly")
  }
}
```

C.3 Package `moretbls`

Listing C.5: moretbls/moretbls.go

```
// Package moretbls mirrors the tbls package from the kyber library.
// It implements a (t,n)-threshold BLS signature scheme.
// Here, signatures are points on G2 and public keys are points on G1
//
// WARNING: relies on morebls package, which makes use of an insecure
   hash-to-G2 function
package moretbls

import (
  "bytes"
  "encoding/binary"

  "github.com/nmohnblatt/cd_client/morebls"
  "go.dedis.ch/kyber/v3/pairing"
  "go.dedis.ch/kyber/v3/share"
  "go.dedis.ch/kyber/v3/sign/tbls"
```

```
)

// Sign2 creates a threshold BLS signature Si = xi * H(m) on the given message m
// using the provided secret key share xi.
func Sign2(suite pairing.Suite, private *share.PriShare, msg []byte) ([]byte, error) {
  buf := new(bytes.Buffer)
  if err := binary.Write(buf, binary.BigEndian, uint16(private.I)); err != nil {
    return nil, err
  }
  s, err := morebls.Sign2(suite, private.V, msg)
  if err != nil {
    return nil, err
  }
  if err := binary.Write(buf, binary.BigEndian, s); err != nil {
    return nil, err
  }
  return buf.Bytes(), nil
}

// Verify2 checks the given threshold BLS signature Si on the message m using
// the public key share Xi that is associated to the secret key share xi. This
// public key share Xi can be computed by evaluating the public sharing
// polynonmial at the share's index i.
func Verify2(suite pairing.Suite, public *share.PubPoly, msg, sig []byte) error {
  s := tbls.SigShare(sig)
  i, err := s.Index()
  if err != nil {
    return err
  }
  return morebls.Verify2(suite, public.Eval(i).V, msg, s.Value())
}

// Recover2 reconstructs the full BLS signature S = x * H(m) from a threshold t
// of signature shares Si using Lagrange interpolation. The full
```

```
signature S
// can be verified through the regular BLS verification routine using the
// shared public key X. The shared public key can be computed by evaluating the
// public sharing polynomial at index 0.
func Recover2(suite pairing.Suite, public *share.PubPoly, msg []byte, sigs [][]byte, t, n int) ([]byte, error) {
  pubShares := make([]*share.PubShare, 0)
  for _, sig := range sigs {
    s := tbls.SigShare(sig)
    i, err := s.Index()
    if err != nil {
      return nil, err
    }
    if err = morebls.Verify2(suite, public.Eval(i).V, msg, s.Value()); err != nil {
      return nil, err
    }
    point := suite.G2().Point()
    if err := point.UnmarshalBinary(s.Value()); err != nil {
      return nil, err
    }
    pubShares = append(pubShares, &share.PubShare{I: i, V: point})
    if len(pubShares) >= t {
      break
    }
  }
  commit, err := share.RecoverCommit(suite.G2(), pubShares, t, n)
  if err != nil {
    return nil, err
  }
  sig, err := commit.MarshalBinary()
  if err != nil {
    return nil, err
  }
  return sig, nil
}
```

Listing C.6: moretbls/moretbls_test.go

```go
package moretbls

import (
  "testing"

  "github.com/nmohnblatt/cd_client/morebls"
  "go.dedis.ch/kyber/v3/pairing/bn256"
  "go.dedis.ch/kyber/v3/share"
)

func TestTBLS(test *testing.T) {
  var err error
  msg := []byte("Hello threshold Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  n := 10
  t := n/2 + 1
  secret := suite.G1().Scalar().Pick(suite.RandomStream())
  priPoly := share.NewPriPoly(suite.G1(), t, secret, suite.RandomStream()
    )
  pubPoly := priPoly.Commit(suite.G1().Point().Base())
  sigShares := make([][]byte, 0)
  for _, x := range priPoly.Shares(n) {
    sig, err := Sign2(suite, x, msg)
    if err != nil {
      test.Errorf("%s", err)
    }
    sigShares = append(sigShares, sig)
  }
  sig, err := Recover2(suite, pubPoly, msg, sigShares, t, n)
  if err != nil {
    test.Errorf("%s", err)
  }
  err = morebls.Verify2(suite, pubPoly.Commit(), msg, sig)
  if err != nil {
    test.Errorf("Signature did not match")
  }
}
```

C.4 Package blindbls

Listing C.7: blindbls/blindbls.go

```go
// Package blindbls implements a blind BLS Signature protocol based on
    the kyber library
package blindbls

import (
  "errors"

  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/pairing"
)

// CheckGroup checks whether point P is from the group G
func CheckGroup(P kyber.Point, G kyber.Group) bool {
  isInGroup := false

  if G.String() == P.String()[:8] {
    isInGroup = true
  }

  return isInGroup
}

// Blind returns a blinded byte representation of an input point
func Blind(group kyber.Group, blindingFactor kyber.Scalar, HM kyber.Point
    ) ([]byte, error) {
  if check := CheckGroup(HM, group); !check {
    err := errors.New("blind: HM and group do not match")
    return nil, err
  }
  aHM := group.Point()
  aHM.Mul(blindingFactor, HM)

  out, err := aHM.MarshalBinary()
  if err != nil {
    return nil, err
  }
  return out, nil
}
```

```

// Sign creates a BLS signature S = x * H(m) on a blinded message (byte
    representation) using the private
// key x. The signature S is a point on the curve defined by the argument
     group.
// Warning: "group" must match the original group of "blindedHash"
func Sign(group kyber.Group, x kyber.Scalar, blindedHash []byte) ([]byte,
     error) {
  aHM := group.Point()
  err := aHM.UnmarshalBinary(blindedHash)
  if err != nil {
    return nil, err
  }
  xaHM := aHM.Mul(x, aHM)

  s, err := xaHM.MarshalBinary()
  if err != nil {
    return nil, err
  }
  return s, nil
}

// Unblind outputs the unblinded point underlying the blinded signature s
func Unblind(group kyber.Group, blindingFactor kyber.Scalar, s []byte) (
    kyber.Point, error) {
  axHM := group.Point()
  err := axHM.UnmarshalBinary(s)
  if err != nil {
    return nil, err
  }

  inv := group.Scalar().Inv(blindingFactor)
  xHM := axHM.Mul(inv, axHM)

  return xHM, nil
}

// Verify checks the given BLS signature S on the message m using the
    public
// key X. If group is G1, it verfies that the equality e(H(m), X) == e(H(
    m), x*B2) ==
```

```
// e(x*H(m), B2) == e(S, B2) holds where e is the pairing operation and B2 is
// the base point from curve G2. If group is G2, it verifies that the equality e(X, H(m)) == e(x*B1, H(m)) ==
// e(B1, x*H(m)) == e(B1, S) holds where e is the pairing operation and B1 is
// the base point from curve G1.
func Verify(suite pairing.Suite, group kyber.Group, X kyber.Point, HM, xHM kyber.Point) error {

  if group.String() == "bn256.G1" {
    left := suite.Pair(HM, X)

    right := suite.Pair(xHM, suite.G2().Point().Base())
    if !left.Equal(right) {
      return errors.New("bls: invalid signature")
    }
  } else if group.String() == "bn256.G2" {
    left := suite.Pair(X, HM)

    right := suite.Pair(suite.G1().Point().Base(), xHM)
    if !left.Equal(right) {
      return errors.New("bls: invalid signature")
    }
  } else {
    return errors.New("Group not recognised")
  }

  return nil
}
```

Listing C.8: blindbls/blindbls_test.go

```
package blindbls

import (
  "testing"

  "github.com/nmohnblatt/cd_client/hash"
```

```
  "github.com/nmohnblatt/cd_client/morebls"
  "go.dedis.ch/kyber/v3/pairing/bn256"
  "go.dedis.ch/kyber/v3/sign/bls"
  "go.dedis.ch/kyber/v3/util/random"
)

func TestCheckGroup(t *testing.T) {
  suite := bn256.NewSuite()
  p1 := suite.G1().Point()
  p2 := suite.G2().Point()

  if test := CheckGroup(p1, suite.G1()); !test {
    t.Errorf("p1 was not recognised as a G1 point")
  }

  if test := CheckGroup(p2, suite.G2()); !test {
    t.Errorf("p2 was not recognised as a G2 point")
  }

  if test := CheckGroup(p1, suite.G2()); test {
    t.Errorf("p1 was recognised as a G2 point")
  }

  if test := CheckGroup(p2, suite.G1()); test {
    t.Errorf("p2 was recognised as a G1 point")
  }

}

func TestBlindUnblind(t *testing.T) {
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  H1M := hash.HashtoG1(suite, msg)
  BF := suite.G1().Scalar().Pick(random.New())

  aH1M, err := Blind(suite.G1(), BF, H1M)
  if err != nil {
    t.Errorf("%s", err)
  }

  test, err := Unblind(suite.G1(), BF, aH1M)
```

```
	if err != nil {
		t.Errorf("Could not Unblind")
	}

	if !test.Equal(H1M) {
		t.Errorf("Point was not recovered")
	}
}

func TestBlindBLSG1(t *testing.T) {
	msg := []byte("Hello Boneh-Lynn-Shacham")
	suite := bn256.NewSuite()
	H1M := hash.HashtoG1(suite, msg)
	BF := suite.G1().Scalar().Pick(random.New())
	aH1M, err := Blind(suite.G1(), BF, H1M)
	if err != nil {
		t.Errorf("Could not Blind point")
	}
	blindedPoint := suite.G1().Point()
	if err := blindedPoint.UnmarshalBinary(aH1M); err != nil {
		t.Errorf("%s", err)
	}
	if blindedPoint.Equal(H1M) {
		t.Errorf("No blinding occurred, point is still the same")
	}
	private, public := bls.NewKeyPair(suite, random.New())
	sig, err := Sign(suite.G1(), private, aH1M)
	if err != nil {
		t.Errorf("%s", err)
	}
	xH1M, err := Unblind(suite.G1(), BF, sig)
	if err != nil {
		t.Errorf("%s", err)
	}
	err = Verify(suite, suite.G1(), public, H1M, xH1M)
	if err != nil {
		t.Errorf("Signature did not match")
	}
}

func TestBlindBLSG2(t *testing.T) {
```

```
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  H2M := hash.InsecureHashtoG2(suite, msg)
  BF := suite.G2().Scalar().Pick(random.New())
  aH2M, err := Blind(suite.G2(), BF, H2M)
  if err != nil {
    t.Errorf("Could not Blind point")
  }
  blindedPoint := suite.G2().Point()
  if err := blindedPoint.UnmarshalBinary(aH2M); err != nil {
    t.Errorf("%s", err)
  }
  if blindedPoint.Equal(H2M) {
    t.Errorf("No blinding occurred, point is still the same")
  }
  private, public := morebls.NewKeyPair2(suite, random.New())
  sig, err := Sign(suite.G2(), private, aH2M)
  if err != nil {
    t.Errorf("%s", err)
  }
  xH1M, err := Unblind(suite.G2(), BF, sig)
  if err != nil {
    t.Errorf("%s", err)
  }
  err = Verify(suite, suite.G2(), public, H2M, xH1M)
  if err != nil {
    t.Errorf("Signature did not match")
  }
}

func TestBlindBLSFailSig(t *testing.T) {
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  H1M := hash.HashtoG1(suite, msg)
  BF := suite.G1().Scalar().Pick(random.New())
  aH1M, err := Blind(suite.G1(), BF, H1M)
  if err != nil {
    t.Errorf("Could not Blind point")
  }
  blindedPoint := suite.G1().Point()
  if err := blindedPoint.UnmarshalBinary(aH1M); err != nil {
```

```
    t.Errorf("%s", err)
  }
  if blindedPoint.Equal(H1M) {
    t.Errorf("No blinding occurred, point is still the same")
  }
  private, public := bls.NewKeyPair(suite, random.New())

  msg2 := []byte("Goodbye Boneh-Lynn-Shacham")
  sig2, err := bls.Sign(suite, private, msg2)

  xH1M, err := Unblind(suite.G1(), BF, sig2)
  if err != nil {
    t.Errorf("%s", err)
  }
  err = Verify(suite, suite.G1(), public, H1M, xH1M)
  if err == nil {
    t.Errorf("Verification succeeded on the wrong signature")
  }
}

func TestBlindBLSFailKey(t *testing.T) {
  msg := []byte("Hello Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  H1M := hash.HashtoG1(suite, msg)
  BF := suite.G1().Scalar().Pick(random.New())
  aH1M, err := Blind(suite.G1(), BF, H1M)
  if err != nil {
    t.Errorf("Could not Blind point")
  }
  blindedPoint := suite.G1().Point()
  if err := blindedPoint.UnmarshalBinary(aH1M); err != nil {
    t.Errorf("%s", err)
  }
  if blindedPoint.Equal(H1M) {
    t.Errorf("No blinding occurred, point is still the same")
  }
  private, public := bls.NewKeyPair(suite, random.New())
  sig, err := Sign(suite.G1(), private, aH1M)
  if err != nil {
    t.Errorf("%s", err)
  }
```

```go
	xH1M, err := Unblind(suite.G1(), BF, sig)
	if err != nil {
		t.Errorf("%s", err)
	}

	_, public = bls.NewKeyPair(suite, random.New())
	err = Verify(suite, suite.G1(), public, H1M, xH1M)
	if err == nil {
		t.Errorf("Verification succeeded using the wrong key")
	}
}
```

C.5 Package `blindtbls`

Listing C.9: blindtbls/adapter.go

```go
package blindtbls

import (
	"bytes"
	"encoding/binary"

	"go.dedis.ch/kyber/v3"
	"go.dedis.ch/kyber/v3/share"
	"go.dedis.ch/kyber/v3/sign/tbls"
)

// SigSharetoPubShare converts a SigShare (byte representation) to a
	PubShare (complex representation)
func SigSharetoPubShare(group kyber.Group, sig tbls.SigShare) (*share.
	PubShare, error) {
	i, err := sig.Index()
	if err != nil {
		return &share.PubShare{I: -1, V: nil}, err
	}

	point := group.Point()
	if err := point.UnmarshalBinary(sig.Value()); err != nil {
```

```
    return &share.PubShare{I: -1, V: nil}, err
  }

  return &share.PubShare{I: i, V: point}, nil

}

// PubSharetoSigShare converts a PubShare (complex representation) to a
   SigShare (byte representation)
func PubSharetoSigShare(sig *share.PubShare) (tbls.SigShare, error) {
  buf := new(bytes.Buffer)
  if err := binary.Write(buf, binary.BigEndian, uint16(sig.I)); err !=
     nil {
    return nil, err
  }
  point, _ := sig.V.MarshalBinary()
  if err := binary.Write(buf, binary.BigEndian, point); err != nil {
    return nil, err
  }
  return buf.Bytes(), nil
}
```

Listing C.10: blindtbls/adapter_test.go

```
package blindtbls

import (
  "testing"

  "go.dedis.ch/kyber/v3/pairing/bn256"
  "go.dedis.ch/kyber/v3/share"
  "go.dedis.ch/kyber/v3/util/random"
)

func TestConvert(t *testing.T) {
  suite := bn256.NewSuite()
  integer := 1
  point := suite.G1().Point().Pick(random.New())

```

```
  A := &share.PubShare{I: integer, V: point}

  B, err := PubSharetoSigShare(A)
  if err != nil {
    t.Error(err)
  }

  BI, err := B.Index()
  if err != nil {
    t.Error(err)
  }

  if BI != integer {
    t.Errorf("Wrong index")
  }

  testPoint := suite.G1().Point()
  if err := testPoint.UnmarshalBinary(B.Value()); err != nil {
    t.Error(err)
  }

  if !testPoint.Equal(point) {
    t.Errorf("wrong value")
  }
}
```

Listing C.11: blindtbls/blindtbls.go

```
package blindtbls

import (
  "bytes"
  "encoding/binary"

  "github.com/nmohnblatt/cd_client/blindbls"
  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/pairing"
  "go.dedis.ch/kyber/v3/share"
  "go.dedis.ch/kyber/v3/sign/tbls"
```

```
)

// Blind returns a blinded byte representation of an input point
func Blind(group kyber.Group, blindingFactor kyber.Scalar, HM kyber.Point
    ) ([]byte, error) {
  return blindbls.Blind(group, blindingFactor, HM)
}

// Sign creates a threshold BLS signature Si = xi * H(m) on the given
    message m
// using the provided secret key share xi.
func Sign(suite pairing.Suite, group kyber.Group, private *share.PriShare
    , blindedHash []byte) ([]byte, error) {
  buf := new(bytes.Buffer)
  if err := binary.Write(buf, binary.BigEndian, uint16(private.I)); err
      != nil {
    return nil, err
  }
  s, err := blindbls.Sign(group, private.V, blindedHash)
  if err != nil {
    return nil, err
  }
  if err := binary.Write(buf, binary.BigEndian, s); err != nil {
    return nil, err
  }
  return buf.Bytes(), nil
}

// UnblindShare outputs the unblinded point underlying the blinded
    signature s
func UnblindShare(group kyber.Group, blindingFactor kyber.Scalar, s []
    byte) (*share.PubShare, error) {
  Si := tbls.SigShare(s)
  i, err := Si.Index()
  if err != nil {
    return &share.PubShare{I: -1, V: nil}, err
  }

  axHM := group.Point()
  err = axHM.UnmarshalBinary(Si.Value())
  if err != nil {
```

```go
		return &share.PubShare{I: -1, V: nil}, err
	}

	inv := group.Scalar().Inv(blindingFactor)
	xHM := axHM.Mul(inv, axHM)

	return &share.PubShare{I: i, V: xHM}, nil
}

// Verify checks the given threshold BLS signature Si on the message m using
// the public key share Xi that is associated to the secret key share xi. This
// public key share Xi can be computed by evaluating the public sharing
// polynonmial at the share's index i.
func Verify(suite pairing.Suite, group kyber.Group, public *share.PubPoly, HM kyber.Point, s *share.PubShare) error {
	return blindbls.Verify(suite, group, public.Eval(s.I).V, HM, s.V)
}

// Recover reconstructs the full BLS signature S = x * H(m) from a threshold t
// of signature shares Si using Lagrange interpolation. The full signature S
// can be verified through the regular BLS verification routine using the
// shared public key X. The shared public key can be computed by evaluating the
// public sharing polynomial at index 0.
func Recover(suite pairing.Suite, group kyber.Group, public *share.PubPoly, HM kyber.Point, sigs []*share.PubShare, t, n int) ([]byte, error) {
	for _, sig := range sigs {
		if err := Verify(suite, group, public, HM, sig); err != nil {
			return nil, err
		}
	}

	commit, err := share.RecoverCommit(group, sigs, t, n)
	if err != nil {
		return nil, err
	}
```

```
	sig, err := commit.MarshalBinary()
	if err != nil {
		return nil, err
	}
	return sig, nil
}
```

Listing C.12: blindtbls/blindtbls_test.go

```
package blindtbls

import (
	"testing"

	"github.com/nmohnblatt/cd_client/blindbls"
	"github.com/nmohnblatt/cd_client/hash"
	"go.dedis.ch/kyber/v3/pairing/bn256"
	"go.dedis.ch/kyber/v3/share"
	"go.dedis.ch/kyber/v3/sign/tbls"
	"go.dedis.ch/kyber/v3/util/random"
)

func TestUnblindShare(test *testing.T) {
	// SETUP PHASE
	msg := []byte("Hello threshold Boneh-Lynn-Shacham")
	suite := bn256.NewSuite()
	signGroup := suite.G1()
	keyGroup := suite.G2()
	HM, err := hash.Hash(suite, signGroup, msg)
	HMBytes, err := HM.MarshalBinary()
	if err != nil {
		test.Error(err)
	}
	BF := signGroup.Scalar().Pick(random.New())
	if err != nil {
		test.Error(err)
	}
	n := 6
	t := n/2 + 1
```

```
secret := signGroup.Scalar().Pick(suite.RandomStream())
priPoly := share.NewPriPoly(keyGroup, t, secret, suite.RandomStream())

// BLIND
aHM, err := Blind(signGroup, BF, HM)

// SIGN CLEAR
clearSigShares := make([][]byte, 0)
for _, x := range priPoly.Shares(n) {
  sig, err := Sign(suite, signGroup, x, HMBytes)
  if err != nil {
    test.Error(err)
  }
  clearSigShares = append(clearSigShares, sig)
}

// SIGN BLIND
blindSigShares := make([][]byte, 0)
for _, x := range priPoly.Shares(n) {
  sig, err := Sign(suite, signGroup, x, aHM)
  if err != nil {
    test.Error(err)
  }
  blindSigShares = append(blindSigShares, sig)
}

// UNBLIND
testSigShares := make([]*share.PubShare, len(blindSigShares))
for i := 0; i < len(blindSigShares); i++ {
  buf, err := UnblindShare(signGroup, BF, blindSigShares[i])
  if err != nil {
    test.Error(err)
  }
  testSigShares[i] = buf
}

// CHECKS
for i := 0; i < len(testSigShares); i++ {
  want, err := SigSharetoPubShare(signGroup, tbls.SigShare(clearSigShares[i]))
  if err != nil {
```

```
      test.Error(err)
    }
    if testSigShares[i].I != want.I {
      test.Errorf("unblindshares: indexes do not match")
    }
    if !testSigShares[i].V.Equal(want.V) {
      test.Errorf("unblindshares: index %d values do not match", want.I)
      // test.Logf("want %s \n actual %s", want.V.String(), testSigShares
         [i].V.String())
    } else if testSigShares[i].V.Equal(want.V) {
      test.Logf("unblindshares: index %d OK", want.I)
    }
  }
}

func TestBlindTBLSRecoverThenUnblind(test *testing.T) {
  // SETUP PHASE
  msg := []byte("Hello threshold Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  signGroup := suite.G1()
  keyGroup := suite.G2()
  HM, err := hash.Hash(suite, signGroup, msg)
  if err != nil {
    test.Error(err)
  }
  BF := signGroup.Scalar().Pick(random.New())
  if err != nil {
    test.Error(err)
  }
  n := 10
  t := n/2 + 1
  secret := signGroup.Scalar().Pick(suite.RandomStream())
  priPoly := share.NewPriPoly(keyGroup, t, secret, suite.RandomStream())
  pubPoly := priPoly.Commit(keyGroup.Point().Base())

  // BLIND
  aHM, err := Blind(signGroup, BF, HM)

  // SIGN
  blindSigShares := make([][]byte, 0)
  for _, x := range priPoly.Shares(n) {
```

```
    sig, err := Sign(suite, signGroup, x, aHM)
    if err != nil {
      test.Error(err)
    }
    blindSigShares = append(blindSigShares, sig)
  }

  // RECOVER
  aHMPoint := signGroup.Point()
  if err := aHMPoint.UnmarshalBinary(aHM); err != nil {
    test.Error(err)
  }
  blindSigSharesFormat := make([]*share.PubShare, len(blindSigShares))
  for i, sig := range blindSigShares {
    blindSigSharesFormat[i], _ = SigSharetoPubShare(signGroup, tbls.
       SigShare(sig))
  }
  sig, err := Recover(suite, signGroup, pubPoly, aHMPoint,
     blindSigSharesFormat[:t], t, n)

  // UNBLIND
  final, _ := blindbls.Unblind(signGroup, BF, sig)

  // CHECKS
  want := signGroup.Point().Mul(secret, HM)
  if !final.Equal(want) {
    test.Errorf("Computed signature does not match expected signature")
  }
  err = blindbls.Verify(suite, signGroup, pubPoly.Commit(), HM, final)
  if err != nil {
    test.Errorf("Signature did not verify")
  }
}

func TestBlindTBLSUnblindThenRecover(test *testing.T) {
  // SETUP PHASE
  msg := []byte("Hello threshold Boneh-Lynn-Shacham")
  suite := bn256.NewSuite()
  signGroup := suite.G1()
  keyGroup := suite.G2()
  HM, err := hash.Hash(suite, signGroup, msg)
```

```
BF := signGroup.Scalar().Pick(random.New())
if err != nil {
  test.Error(err)
}
n := 10
t := n/2 + 1
secret := signGroup.Scalar().Pick(suite.RandomStream())
priPoly := share.NewPriPoly(keyGroup, t, secret, suite.RandomStream())
pubPoly := priPoly.Commit(keyGroup.Point().Base())

// BLIND
aHM, err := Blind(signGroup, BF, HM)

// SIGN
blindSigShares := make([][]byte, 0)
for _, x := range priPoly.Shares(n) {
  sig, err := Sign(suite, signGroup, x, aHM)
  if err != nil {
    test.Error(err)
  }
  blindSigShares = append(blindSigShares, sig)
}

//UNBLIND
sigShares := make([]*share.PubShare, 0)
for _, Si := range blindSigShares {
  buf, err := UnblindShare(signGroup, BF, Si)
  if err != nil {
    test.Error(err)
  }
  sigShares = append(sigShares, buf)
}

// RECOVER
sig, err := Recover(suite, signGroup, pubPoly, HM, sigShares[:t], t, n)
if err != nil {
  test.Error(err)
}

// CHECKS
testPoint := signGroup.Point()
```

```
	if err = testPoint.UnmarshalBinary(sig); err != nil {
		test.Error(err)
	}
	want := signGroup.Point().Mul(secret, HM)
	if !testPoint.Equal(want) {
		test.Errorf("Computed signature does not match expected signature")
	}

	err = blindbls.Verify(suite, signGroup, pubPoly.Commit(), HM, testPoint
		)
	if err != nil {
		test.Errorf("Signature did not match")
	}
}
```

C.6 Package `main`

Listing C.13: go.mod

```
module github.com/nmohnblatt/cd_client

go 1.14

require (
	go.dedis.ch/kyber/v3 v3.0.12
	golang.org/x/crypto v0.0.0-20190611184440-5c40567a22f8 // indirect
)
```

Listing C.14: main.go

```
package main

import (
	"fmt"
	"io/ioutil"
```

```
    "go.dedis.ch/kyber/v3"
    "go.dedis.ch/kyber/v3/pairing/bn256"
    "go.dedis.ch/kyber/v3/xof/blake2xb"
)

var suite = bn256.NewSuite()

const prompt string = "> "

// Create a simple UI
// User will be able to enter their details and contact lists.
// Program should find existing rendez-vous points and create new ones
    where needed.
func main() {
    // Setup Phase:
    n := 10
    t := n/2 + 1

    rng := blake2xb.New(nil) // A pseudo RNG which makes this code
        repeatable for testing.

    masterSecret := suite.GT().Scalar().Pick(rng)
    serverList, pubPoly1, pubPoly2 := setupThresholdServers(suite,
        masterSecret, n, t)

    // Initialise the service's user
    u1 := initialiseUser()

    // Communicate with servers to obtain the user's private keys
    fmt.Printf(prompt+"Fetching private keys from %d out of %d servers... \
        n", t, n)
    u1.obtainPrivateKeysBlindThreshold(suite, serverList[0:t], pubPoly1,
        pubPoly2, t, n)
    fmt.Println(prompt + "Keys successfully received.")

    // Compute shared key material with a manually entered contact number
    sharedAB, sharedBA := processSingleContactManualInput(u1)
    // fmt.Println(prompt + "Derived the following keys:\n" + sharedAB.
        String() + "\n" + sharedBA.String())

```

```
  meetingPoint := createMeetingPoint(u1, sharedAB, sharedBA)
  output := append([]byte("Meeting point "), meetingPoint...)
  if err := ioutil.WriteFile("mp.txt", output, 0644); err != nil {
    panic(fmt.Errorf("Could not generate file"))
  }
}

// A function that promts the user for their name and number.
// The function returns a pointer to a new user created with the name and number provided.
// Public keys are automatically computed. Private keys will need to be fetched from server
func initialiseUser() *user {
  fmt.Println(prompt + "Initialising. Please enter your name:")
  var Name string
  fmt.Scanf("%s", &Name)
  fmt.Printf(prompt+"Thank you %s. Please enter your phone number:\n", Name)
  var Number string
  fmt.Scanf("%s", &Number)
  u1 := newUser(Name, Number)
  fmt.Println(prompt + "You have been registered as a user.")

  return u1
}

// A function that prompts the user for their contact's phone number.
// The function computes the contact's corresponding public key and derives shared keys
func processSingleContactManualInput(u *user) (kyber.Point, kyber.Point) {
  fmt.Println(prompt + "Enter your contact's phone number:")
  var contactNumber string
  fmt.Scanf("%s", &contactNumber)

  sharedAB, sharedBA := deriveSharedKeys(u, contactNumber)

  return sharedAB, sharedBA
}
```

Listing C.15: main_test.go

```go
package main

import (
  "testing"

  "github.com/nmohnblatt/cd_client/moretbls"
  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/pairing/bn256"
  "go.dedis.ch/kyber/v3/share"
  "go.dedis.ch/kyber/v3/sign/tbls"
  "go.dedis.ch/kyber/v3/util/random"
)

func TestKeyDerivationLocal(t *testing.T) {
  s1 := newDummyServer(1)
  // setup three users: Alice, Bob and Charlie
  alice := newUser("Alice", "07111111111")
  bob := newUser("Bob", "07222222222")
  charlie := newUser("Charlie", "07333333333")

  alice.obtainPrivateKeys(s1)
  bob.obtainPrivateKeys(s1)
  charlie.obtainPrivateKeys(s1)

  // Alice and Bob compute shared keys. Charlie tries to use his key material to find A and B's shared keys
  // Format xSharedxy = e(H(x)s, H(y)) i.e. the shared point in GT with x in G1 and y in G2 computed using x's private key
  aSharedab, aSharedba := deriveSharedKeys(alice, bob.phoneNumber)
  bSharedba, bSharedab := deriveSharedKeys(bob, alice.phoneNumber)
  cSharedca, cSharedac := deriveSharedKeys(charlie, alice.phoneNumber)
  cSharedcb, cSharedbc := deriveSharedKeys(charlie, bob.phoneNumber)

  // Check that Alice and Bob's computatins match
  if !aSharedab.Equal(bSharedab) {
    t.Errorf("Keys don't match: Alice AB does not match with Bob's")
  }
  if !aSharedba.Equal(bSharedba) {
    t.Errorf("Keys don't match: Alice BA does not match with Bob")
  }
```

```

  // Check that Charlie's computations are different from those of Alice
     and Bob
  aliceBobKeys := [4]kyber.Point{aSharedab, aSharedba, bSharedab,
     bSharedba}
  charlieKeys := [4]kyber.Point{cSharedac, cSharedca, cSharedcb,
     cSharedbc}
  for i := 0; i < 4; i++ {
    for j := 0; j < 4; j++ {
      if charlieKeys[i].Equal(aliceBobKeys[j]) {
        t.Errorf("Charlie computed one of Alice and Bob's keys")
      }
    }
  }
}

func TestKeyDerivationMultiLocal(t *testing.T) {
  // Vary the number of servers
  n := 10
  thr := n/2 + 1
  suite := bn256.NewSuite()

  // Create a master secret and deal shares
  secret := suite.GT().Scalar().Pick(random.New())
  serverList, pubPoly1, pubPoly2 := setupThresholdServers(suite, secret,
     n, thr)

  alice := newUser("Alice", "07111111111")
  bob := newUser("Bob", "07222222222")
  charlie := newUser("Charlie", "07333333333")

  alice.obtainPrivateKeysBlindThreshold(suite, serverList, pubPoly1,
     pubPoly2, thr, n)
  bob.obtainPrivateKeysBlindThreshold(suite, serverList, pubPoly1,
     pubPoly2, thr, n)
  charlie.obtainPrivateKeysBlindThreshold(suite, serverList, pubPoly1,
     pubPoly2, thr, n)

  // Alice and Bob compute shared keys. Charlie tries to use his key
     material to find A and B's shared keys
  // Format xSharedxy = e(H(x)s, H(y)) i.e. the shared point in GT with x
```

```
      in G1 and y in G2 computed using x's private key
  aSharedab, aSharedba := deriveSharedKeys(alice, bob.phoneNumber)
  bSharedba, bSharedab := deriveSharedKeys(bob, alice.phoneNumber)
  cSharedca, cSharedac := deriveSharedKeys(charlie, alice.phoneNumber)
  cSharedcb, cSharedbc := deriveSharedKeys(charlie, bob.phoneNumber)

  // Check that Alice and Bob's computatins match
  if !aSharedab.Equal(bSharedab) {
    t.Errorf("Keys don't match: Alice AB does not match with Bob's")
  }
  if !aSharedba.Equal(bSharedba) {
    t.Errorf("Keys don't match: Alice BA does not match with Bob")
  }

  // Check that Charlie's computations are different from those of Alice
     and Bob
  aliceBobKeys := [4]kyber.Point{aSharedab, aSharedba, bSharedab,
     bSharedba}
  charlieKeys := [4]kyber.Point{cSharedac, cSharedca, cSharedcb,
     cSharedbc}
  for i := 0; i < 4; i++ {
    for j := 0; j < 4; j++ {
      if charlieKeys[i].Equal(aliceBobKeys[j]) {
        t.Errorf("Charlie computed one of Alice and Bob's keys")
      }
    }
  }
}

func TestThresholdG1(t *testing.T) {
  // Initialise client
  alice := newUser("Alice", "07111111111")
  msg := []byte(alice.phoneNumber)

  // Set number of servers and threshold
  n := 10
  thr := n/2 + 1

  // Create a master secret
  secret := suite.GT().Scalar().Pick(random.New())

```

```
  // Set-up the sharing scheme and give one share to each server
  priPoly := share.NewPriPoly(suite.G2(), thr, secret, random.New())
  pubPoly := priPoly.Commit(suite.G2().Point().Base())
  serverKeys := priPoly.Shares(n)

  // Use the first thr keys to sign alice's number
  var alicePartialKeys [][]byte
  for _, key := range serverKeys[:thr] {
    sig, err := tbls.Sign(suite, key, msg)
    if err != nil {
      t.Errorf("Error whilst signing")
    }
    alicePartialKeys = append(alicePartialKeys, sig)
  }

  // Compute Alice's key in G1 using her partial keys
  fullKey, err := tbls.Recover(suite, pubPoly, msg, alicePartialKeys, thr
    , n)
  if err != nil {
    t.Errorf("Error whilst recovering")
  }
  test := suite.G1().Point()
  err = test.UnmarshalBinary(fullKey)
  if err != nil {
    t.Errorf("could not unmarshall point")
  }

  // Compute the expected value for Alice's private key in G1
  want := suite.G1().Point().Mul(secret, alice.pk1)

  // Compare Alice's computation with the expected value
  if !test.Equal(want) {
    t.Errorf("value is not as expected")
  }

}

func TestThresholdG2(t *testing.T) {
  // Initialise client
  alice := newUser("Alice", "07111111111")
  msg := []byte(alice.phoneNumber)
```

```

	// Set number of servers and threshold
	n := 10
	thr := n/2 + 1

	// Create a master secret
	secret := suite.GT().Scalar().Pick(random.New())

	// Set-up the sharing scheme and give one share to each server
	priPoly := share.NewPriPoly(suite.G1(), thr, secret, random.New())
	pubPoly := priPoly.Commit(suite.G1().Point().Base())
	serverKeys := priPoly.Shares(n)

	// Use the first thr keys to sign alice's number
	var alicePartialKeys [][]byte
	for _, key := range serverKeys[0:thr] {
		sig, err := moretbls.Sign2(suite, key, msg)
		if err != nil {
			t.Errorf("Error whilst signing")
		}
		alicePartialKeys = append(alicePartialKeys, sig)
	}

	// Compute Alice's key in G2 using her partial keys
	fullKey, err := moretbls.Recover2(suite, pubPoly, msg, alicePartialKeys, thr, n)
	if err != nil {
		t.Errorf("Error whilst recovering")
	}
	test := suite.G2().Point()
	err = test.UnmarshalBinary(fullKey)
	if err != nil {
		t.Errorf("could not unmarshall point")
	}

	// Compute the expected value for Alice's private key in G2
	want := suite.G2().Point().Mul(secret, alice.pk2)

	// Compare Alice's computation with the expected value
	if !test.Equal(want) {
		t.Errorf("value is not as expected")
```

```
  }

}

func TestThresholdUserKeys(t *testing.T) {
  // Initialise client
  alice := newUser("Alice", "07111111111")

  // Set number of servers and threshold
  n := 10
  thr := n/2 + 1

  // Create a master secret and deal shares
  secret := suite.GT().Scalar().Pick(random.New())
  serverList, pubPoly1, pubPoly2 := setupThresholdServers(suite, secret,
     n, thr)

  // Obtain private key from t servers
  alice.obtainPrivateKeysThreshold(suite, serverList[:thr], pubPoly1,
     pubPoly2, thr, n)

  // Compute the expected values for Alice's private keys
  want1 := suite.G1().Point().Mul(secret, alice.pk1)
  want2 := suite.G2().Point().Mul(secret, alice.pk2)

  // Check the value recovered from servers matches the expected value
  if !alice.sk1.Equal(want1) {
    t.Errorf("Did not compute correct private key 1")
  }
  if !alice.sk2.Equal(want2) {
    t.Errorf("Did not compute correct private key 2")
  }

}

func TestBlindThresholdUserKeys(t *testing.T) {
  // Initialise client
  alice := newUser("Alice", "07111111111")

  // Set number of servers and threshold
  n := 10
```

```go
	thr := n/2 + 1

	// Create a master secret and deal shares
	secret := suite.GT().Scalar().Pick(random.New())
	serverList, pubPoly1, pubPoly2 := setupThresholdServers(suite, secret,
		n, thr)

	// Obtain private key from t servers
	err := alice.obtainPrivateKeysBlindThreshold(suite, serverList[:thr],
		pubPoly1, pubPoly2, thr, n)
	if err != nil {
		t.Error(err)
	}

	// Compute the expected values for Alice's private keys
	want1 := suite.G1().Point().Mul(secret, alice.pk1)
	want2 := suite.G2().Point().Mul(secret, alice.pk2)

	// Check the value recovered from servers matches the expected value
	if !alice.sk1.Equal(want1) {
		t.Errorf("Did not compute correct private key 1")
	} else {
		t.Log("private key 1 OK")
	}
	if !alice.sk2.Equal(want2) {
		t.Errorf("Did not compute correct private key 2")
	}

}
```

Listing C.16: user

```go
package main

import (
	"errors"

	"github.com/nmohnblatt/cd_client/blindbls"
	"github.com/nmohnblatt/cd_client/blindtbls"
```

```
	"github.com/nmohnblatt/cd_client/moretbls"
	"go.dedis.ch/kyber/v3"
	"go.dedis.ch/kyber/v3/pairing"
	"go.dedis.ch/kyber/v3/share"
	"go.dedis.ch/kyber/v3/sign/tbls"
	"go.dedis.ch/kyber/v3/util/random"
	"go.dedis.ch/kyber/v3/xof/blake2xb"
)

type user struct {
	name                 string
	phoneNumber          string
	pk1, pk2, sk1, sk2 kyber.Point
}

// Creates a new user with the name and phone number specified.
// Automatically derive public keys. (Private keys need to be provided by server)
func newUser(Name, Number string) *user {
	var u user

	u.name = Name
	u.phoneNumber = Number

	u.pk1, u.pk2 = derivePublicKeys(u.phoneNumber)

	return &u
}

/*
// Request private key from a TCP server
func (u *user) requestKeysTCP(server string) {
	conn, err := net.Dial("tcp", server)
	if err != nil {
		panic(err)
	}

	// send to socket
	fmt.Fprintf(conn, u.pk1.String()+u.pk2.String()+"\n")

	// listen for reply
```

```go
  message, _ := bufio.NewReader(conn).ReadString('\n')
  fmt.Print("Message from server: " + message)

}
*/

// Request private key from a dummy server (i.e. one that runs locally)
func dummyRequestKeys(u *user, serverID string) (kyber.Point, kyber.Point
    ) {
  // Use a fixed server key for testing purposes
  seed := blake2xb.New([]byte("this is a seed" + serverID))
  serverKey := suite.GT().Scalar().Pick(seed)

  sk1 := suite.G1().Point().Mul(serverKey, u.pk1)
  sk2 := suite.G2().Point().Mul(serverKey, u.pk2)

  return sk1, sk2
}

func (u *user) obtainPrivateKeys(servers ...server) {
  buf1 := suite.G1().Point()
  buf2 := suite.G2().Point()
  for _, s := range servers {
    partial1, partial2 := s.sign(u.phoneNumber)
    buf1.Add(buf1, partial1)
    buf2.Add(buf2, partial2)
  }

  u.sk1 = buf1
  u.sk2 = buf2
}

func (u *user) obtainPrivateKeysThreshold(suite pairing.Suite, servers
    []*multiServer, pubPoly1, pubPoly2 *share.PubPoly, t, n int) error {
  if len(servers) < t {
    return errors.New("Not enough servers to meet thre threshold")
  }

  buf1 := make([][]byte, len(servers))
  buf2 := make([][]byte, len(servers))

```

```go
	for i, s := range servers {
		buf1[i], buf2[i] = s.sign(u.phoneNumber)
	}

	key1, _ := tbls.Recover(suite, pubPoly1, []byte(u.phoneNumber), buf1, t
		, n)
	key2, _ := moretbls.Recover2(suite, pubPoly2, []byte(u.phoneNumber),
		buf2, t, n)

	u.sk1 = suite.G1().Point()
	err := u.sk1.UnmarshalBinary(key1)
	if err != nil {
		return err
	}
	u.sk2 = suite.G2().Point()
	err = u.sk2.UnmarshalBinary(key2)
	if err != nil {
		return err
	}

	return nil
}

func (u *user) obtainPrivateKeysBlindThreshold(suite pairing.Suite,
	servers []*multiServer, pubPoly1, pubPoly2 *share.PubPoly, t, n int)
	error {
	if len(servers) < t {
		return errors.New("Not enough servers to meet the threshold")
	}

	// Choose blinding factor
	BF := [2]kyber.Scalar{suite.G1().Scalar().Pick(random.New()), suite.G2
		().Scalar().Pick(random.New())}

	// Blind
	aH1M, err := blindtbls.Blind(suite.G1(), BF[0], u.pk1)
	if err != nil {
		return err
	}
	aH2M, err := blindtbls.Blind(suite.G2(), BF[1], u.pk2)
	if err != nil {
```

```
    return err
  }

  // Sign
  buf1 := make([][]byte, len(servers))
  buf2 := make([][]byte, len(servers))

  for i, s := range servers {
    buf1[i], buf2[i], err = s.blindsign(aH1M, aH2M)
    if err != nil {
      return err
    }
  }

  // Recover
  aH1MPoint := suite.G1().Point()
  if err := aH1MPoint.UnmarshalBinary(aH1M); err != nil {
    return err
  }
  aH2MPoint := suite.G2().Point()
  if err := aH2MPoint.UnmarshalBinary(aH2M); err != nil {
    return err
  }

  buf1Formatted := make([]*share.PubShare, len(buf1))
  buf2Formatted := make([]*share.PubShare, len(buf2))
  for i := 0; i < len(buf1); i++ {
    buf1Formatted[i], err = blindtbls.SigSharetoPubShare(suite.G1(), tbls
        .SigShare(buf1[i]))
    if err != nil {
      return err
    }
    buf2Formatted[i], err = blindtbls.SigSharetoPubShare(suite.G2(), tbls
        .SigShare(buf2[i]))
    if err != nil {
      return err
    }
  }
  blindKey1, err := blindtbls.Recover(suite, suite.G1(), pubPoly1,
      aH1MPoint, buf1Formatted[:t], t, n)
  if err != nil {
```

```
    return err
  }
  blindKey2, err := blindtbls.Recover(suite, suite.G2(), pubPoly2,
      aH2MPoint, buf2Formatted[:t], t, n)
  if err != nil {
    return err
  }

  // Unblind
  u.sk1, _ = blindbls.Unblind(suite.G1(), BF[0], blindKey1)
  u.sk2, _ = blindbls.Unblind(suite.G2(), BF[1], blindKey2)

  return nil
}
```

Listing C.17: server

```
package main

import (
  "github.com/nmohnblatt/cd_client/blindtbls"
  "github.com/nmohnblatt/cd_client/moretbls"
  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/pairing"
  "go.dedis.ch/kyber/v3/share"
  "go.dedis.ch/kyber/v3/sign/tbls"
  "go.dedis.ch/kyber/v3/util/random"
  "go.dedis.ch/kyber/v3/xof/blake2xb"
)

type server interface {
  sign(string) (kyber.Point, kyber.Point)
}

// Local server for testing purposes
type dummyServer struct {
  ID int
  sk kyber.Scalar
}
```

```go

type multiServer struct {
  ID  int
  sk1 *share.PriShare
  sk2 *share.PriShare
}

func newDummyServer(id int) *dummyServer {
  return &dummyServer{id, suite.GT().Scalar().Pick(blake2xb.New([]byte("
      this is a seed" + string(id))))}
}

func (s dummyServer) sign(phoneNumber string) (kyber.Point, kyber.Point)
   {
  pk1, pk2 := derivePublicKeys(phoneNumber)
  return suite.G1().Point().Mul(s.sk, pk1), suite.G2().Point().Mul(s.sk,
      pk2)
}

func setupThresholdServers(suite pairing.Suite, secret kyber.Scalar, n, t
     int) ([]*multiServer, *share.PubPoly, *share.PubPoly) {
  serverList := make([]*multiServer, n)
  if secret == nil {
    secret = suite.GT().Scalar().Pick(random.New())
  }

  priPoly1 := share.NewPriPoly(suite.G2(), t, secret, random.New())
  pubPoly1 := priPoly1.Commit(suite.G2().Point().Base())
  serverPrivateKeys1 := priPoly1.Shares(n)

  priPoly2 := share.NewPriPoly(suite.G1(), t, secret, random.New())
  pubPoly2 := priPoly2.Commit(suite.G1().Point().Base())
  serverPrivateKeys2 := priPoly2.Shares(n)

  for i := 0; i < n; i++ {
    serverList[i] = newMultiServer(i, serverPrivateKeys1[i],
        serverPrivateKeys2[i])
  }

  return serverList, pubPoly1, pubPoly2
}
```

```
func newMultiServer(id int, key1, key2 *share.PriShare) *multiServer {
	return &multiServer{
		ID:  id,
		sk1: key1,
		sk2: key2,
	}
}

func (s multiServer) sign(phoneNumber string) ([]byte, []byte) {
	toSign := []byte(phoneNumber)
	buf1, _ := tbls.Sign(suite, s.sk1, toSign)
	buf2, _ := moretbls.Sign2(suite, s.sk2, toSign)

	return buf1, buf2
}

func (s multiServer) blindsign(H1M, H2M []byte) ([]byte, []byte, error) {
	buf1, err := blindtbls.Sign(suite, suite.G1(), s.sk1, H1M)
	if err != nil {
		return nil, nil, err
	}
	buf2, err := blindtbls.Sign(suite, suite.G2(), s.sk2, H2M)
	if err != nil {
		return nil, nil, err
	}

	return buf1, buf2, nil
}

// TCP server to test a networked version of our service
type tcpServer struct {
	ID   int
	addr string
	sk   kyber.Scalar
}

func newTCPServer(id int, addr string) *tcpServer {
	s := tcpServer{id, addr, suite.GT().Scalar().Pick(random.New())}
	return &s
}
```

```

// TODO: implement a "sign" method for TCP server (dial, send public keys
    , perform checks (?), etc)
```

Listing C.18: crypto.go

```
package main

import (
	"errors"

	"go.dedis.ch/kyber/v3"
)

// Derive "Public Keys" pk1 =  H1(id), pk2 = H2(id) by hashing phone
    number to points
func derivePublicKeys(phoneNumber string) (pk1, pk2 kyber.Point) {

	pk1 = hashtoG1([]byte(phoneNumber))
	pk2 = insecureHashtoG2([]byte(phoneNumber))

	return pk1, pk2
}

// Derive shared keys between users A and B:
// shared12 = e(H1(idA)s, H2(idB)) = e(H1(idA), H2(idB))s
// shared21 = e(H1(idB), H2(idA)s) = e(H1(idB), H2(idA))s
func deriveSharedKeys(alice *user, contactNumber string) (kyber.Point,
    kyber.Point) {
	bobPk1, bobPk2 := derivePublicKeys(contactNumber)
	shared12 := suite.Pair(alice.sk1, bobPk2)
	shared21 := suite.Pair(bobPk1, alice.sk2)

	return shared12, shared21
}

// Blind a point in any curve from the suite (G1, G2, GT) using a
    predefined blinding factor
func blind(p kyber.Point, blindingFactor kyber.Scalar) kyber.Point {
```

```go
	blinded := p.Clone()
	blinded.Mul(blindingFactor, p)
	return blinded
}

// Unblind a point in any curve from the suite (G1, G2, GT) using a predefined blinding factor
func unblind(p kyber.Point, blindingFactor kyber.Scalar) kyber.Point {
	unblinded := p.Clone()
	inverse := blindingFactor.Clone()
	unblinded.Mul(inverse.Inv(blindingFactor), p)
	return unblinded
}

// Bytewise XOR opertaion for same-sized slices of bytes
func xorBytes(a, b []byte) ([]byte, error) {
	var c []byte
	if len(a) != len(b) {
		return nil, errors.New("xorBytes: arguments must be of the same length")
	}

	for i := 0; i < len(a); i++ {
		buf := (int(a[i]) + int(b[i])) % 256
		c = append(c, byte(buf))
	}

	return c, nil
}

// Sum of points in G1.
// Note to self: (slices can be passed as arguments but need to be unpacked using the ... operator)
func sumG1Points(Points ...kyber.Point) kyber.Point {
	buf := suite.G1().Point()
	for _, X := range Points {
		buf.Add(buf, X)
	}
	return buf
}

```

```
// Sum of points in G2.
// Note to self: (slices can be passed as arguments but need to be
    unpacked using the ... operator)
func sumG2Points(Points ...kyber.Point) kyber.Point {
  buf := suite.G2().Point()
  for _, X := range Points {
    buf.Add(buf, X)
  }
  return buf
}

// Sum of scalars.
// Note to self: (slices can be passed as arguments but need to be
    unpacked using the ... operator)
func sumScalars(Scalars ...kyber.Scalar) kyber.Scalar {
  buf := suite.G1().Scalar()
  for _, X := range Scalars {
    buf.Add(buf, X)
  }

  return buf
}
```

Listing C.19: crypto_test.go

```
package main

import (
  "bytes"
  "testing"

  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/util/random"
)

func TestBlindUnblindG1(t *testing.T) {
  p := suite.G1().Point().Pick(random.New())
  blindingFactor := suite.G1().Scalar().Pick(random.New())

```

```go
  blindedP := blind(p, blindingFactor)

  if blindedP.Equal(p) {
    t.Errorf("blind: G1 Point was not blinded properly")
  }

  check := unblind(blindedP, blindingFactor)

  if !check.Equal(p) {
    t.Errorf("unblind: G1 Point was not recovered")
  }

}

func TestBlindUnblindG2(t *testing.T) {
  p := suite.G2().Point().Pick(random.New())
  blindingFactor := suite.G2().Scalar().Pick(random.New())

  blindedP := blind(p, blindingFactor)

  if blindedP.Equal(p) {
    t.Errorf("blind: G2 Point was not blinded properly")
  }

  check := unblind(blindedP, blindingFactor)

  if !check.Equal(p) {
    t.Errorf("unblind: G2 Point was not recovered")
  }

}

func TestBlindUnblindGT(t *testing.T) {
  p := suite.GT().Point().Pick(random.New())
  blindingFactor := suite.GT().Scalar().Pick(random.New())

  blindedP := blind(p, blindingFactor)

  if blindedP.Equal(p) {
    t.Errorf("blind: GT Point was not blinded properly")
  }
```

```go

  check := unblind(blindedP, blindingFactor)

  if !check.Equal(p) {
    t.Errorf("unblind: GT Point was not recovered")
  }

}

func TestXorBytes(t *testing.T) {
  // Check for correct error handling
  a := []byte{1, 2}
  b := []byte{0, 0, 0}
  _, err := xorBytes(a, b)
  if err == nil {
    t.Errorf("xor: allowed to XOR arguments of different lengths")
  }

  // Check XOR without modular reduction
  a = []byte{1, 2}
  b = []byte{3, 4}
  want := []byte{4, 6}
  c, err := xorBytes(a, b)
  if err != nil {
    t.Errorf("xor: error arose: %s", err)
  }
  if bytes.Compare(c, want) != 0 {
    t.Errorf("xor: not added properly before modular reduction")
  }

  // Check XOR with modular reduction
  a = []byte{255, 200}
  b = []byte{1, 100}
  want = []byte{0, 44}
  c, err = xorBytes(a, b)
  if err != nil {
    t.Errorf("xor: error arose: %s", err)
  }
  if bytes.Compare(c, want) != 0 {
    t.Errorf("xor: not added properly after modular reduction")
  }
```

```
}

func TestSumG1Points(t *testing.T) {
  n := 2
  var scalars []kyber.Scalar

  for i := 0; i < n; i++ {
    scalars = append(scalars, suite.GT().Scalar().Pick(random.New()))
  }

  scalarSum := sumScalars(scalars...)

  p := suite.G1().Point().Pick(random.New())

  want := suite.G1().Point().Mul(scalarSum, p)

  var points []kyber.Point
  for _, X := range scalars {
    points = append(points, suite.G1().Point().Mul(X, p))
  }

  test := sumG1Points(points...)

  if !test.Equal(want) {
    t.Errorf("sumG1: did not add G1 points properly")
  }

}

func TestSumG2Points(t *testing.T) {
  n := 4
  var scalars []kyber.Scalar

  for i := 0; i < n; i++ {
    scalars = append(scalars, suite.GT().Scalar().Pick(random.New()))
  }

  scalarSum := sumScalars(scalars...)

  p := suite.G2().Point().Pick(random.New())

```

```
  want := suite.G2().Point().Mul(scalarSum, p)

  var points []kyber.Point
  for _, X := range scalars {
    points = append(points, suite.G2().Point().Mul(X, p))
  }

  test := sumG2Points(points...)

  if !test.Equal(want) {
    t.Errorf("sum G2: did not add G2 points properly")
  }

}

func TestSumScalars(t *testing.T) {
  a := suite.GT().Scalar().Pick(random.New())
  b := suite.GT().Scalar().Pick(random.New())
  c := suite.GT().Scalar().Pick(random.New())
  sumAB := suite.GT().Scalar().Add(a, b)
  sumABC := suite.G1().Scalar().Add(sumAB, c)

  if !sumScalars(a, b, c).Equal(sumABC) {
    t.Errorf("sumScalar: did not add scalars correctly")
  }
}
```

Listing C.20: hashing.go

```
package main

import (
  "log"

  "go.dedis.ch/kyber/v3"
  "go.dedis.ch/kyber/v3/xof/blake2xb"
)

type hashablePoint interface {
```

```
  Hash([]byte) kyber.Point
}

func hashtoG1(msg []byte) kyber.Point {
  hashable, ok := suite.G1().Point().(hashablePoint)
  if !ok {
    log.Printf("Point cannot be hashed")
  }
  hashed := hashable.Hash(msg)
  return hashed
}

// Hashes a message to a point in G2 by using the message as a seed for
    the Pick method
// !!! Unsure whether this is collision resistant !!!
// To be replaced by a secure version that follows https://tools.ietf.org
    /html/draft-irtf-cfrg-hash-to-curve-07
func insecureHashtoG2(msg []byte) kyber.Point {
  seed := blake2xb.New(msg)
  hashed := suite.G2().Point().Pick(seed)

  return hashed
}
```

Listing C.21: hashing_test.go

```
package main

import (
  "testing"
)

func TestHashToG2(t *testing.T) {
  testMsg := "this is a test message"
  hash1 := insecureHashtoG2([]byte(testMsg))
  hash2 := insecureHashtoG2([]byte(testMsg))

  if !hash1.Equal(hash2) {
    t.Errorf("Hashing the same message yield different points")
```

```
  }
}
```

Listing C.22: ipfs

```
package main

import (
  "crypto/sha256"
  "fmt"

  "go.dedis.ch/kyber/v3"
)

func createMeetingPoint(u *user, sharedAB, sharedBA kyber.Point) []byte {
  bytesSharedAB, _ := sharedAB.MarshalBinary()
  bytesSharedBA, _ := sharedBA.MarshalBinary()

  keymaterial, err := xorBytes(bytesSharedAB, bytesSharedBA)
  if err != nil {
    panic(fmt.Errorf("Could not xor bytes"))
  }

  h := sha256.New()
  h.Write(keymaterial)

  return h.Sum(nil)
}
```