The Federated Open Key Service (FOKS)

Maxwell Krohn (max@ne43.com)

March 25, 2025

Abstract

This paper presents FOKS (Federated Open Key System), a decentralized key management system designed to provide secure and flexible key distribution across federated networks. The basic problem addressed is that of of two parties sharing end-to-end encrypted data across the internet, where both parties have several devices. They might rotate devices, form mutable teams with other users, or even teams of teams in an arbitrary graph. They need to share secret key material to facilitate symmetric encryption, and this material must rotate whenever devices are replaced, or team membership changes. This is a very natural problem but one that still lacks an adequate solution. Morever, we believe key management should not lock users into a particular, walled provider, but instead, should allow for federation and independent management of server resouces, as we see in HTTP and SMTP. We describe the system architecture, security model, and implementation details of a system that achieves secure, federated key exchange, and enables useful applications like end-to-end encrypted data sharing and git hosting.

1 Introduction

2 Threat Model

In FOKS, we consider a threat model similar to that of the Keybase [1], SEAMLess [2] or CONIKs [4] systems. The high level north star is end-to-end secrecy and integrity. Only the clients at the edges of the system should be able to decrypt important data, and only those clients can make authorized changes to the data. Of course, multiple devices per user and mutable groups complicate the picture.

We assume that clients are trustworthy, and behave properly. If this assumption is violated, say, if a client is compromised by a rootkit, then we cannot offer any guarantees.

Users might sometimes lose their devices. In an ideal world, hardware protections would prevent whoever recovered the device from accessing the device's private key material. In the case of hardware keys (like YubiKeys), or backup-keys written on paper, the user has less protection during comrpomise. Regardless, once the user revokes the lost device, keys should rotate so that data is secure going forward (this property is known post-compromise security). In some cases, past data might be safe from the attacker (this property is known as forward-secrecy). but the specifics depend on the trustworthiness of the server (see below). Similarly, revoked keys on lost devices lose their signing power, and other devices will not accept their signatures going forward.

The threat model is here is similar but not exactly the same as Signal's and WhatsApp's, because our applications feature persistent (rather than ephemeral) data. If a new user joins an existing group, or if a user adds a new device, they should be able to access old data, which might be required to reassemble the shared resource. For instance, when Alice adds Bob to a git project, Bob should see all past commits in the commit history, otherwise the application will break. Thus,

we can't guarantee foward-secrecy, since lack of forward secrecy is needed for the application to function properly.

In FOKS, clients pick their servers. They might select for servers that are generally aligned with. They can run their own servers, or pick from third party hosting providers. Users should assume that servers are generally trustworthy, but might suffer comrpomises from time to time. For instance, servers might be running on cloud infrastructure, and the underlying storage, network, or computation might be compromised. Insiders or state actors might have privileged access to the underlying infrascture.

If servers behave honestly, the FOKS system works securely as expected. If servers behave maliciously, they can deny access to data through a variety of mechanisms: they can go offline, they can withoold data, or they can subtly current server-resident data to confuse clients. In this last case, the system's security design should prevent the clients from leaking secrets or accepting unauthorized changes to data. But as in the other more obvious cases, the clients will lose access to their data.

When servers are behaving honestly, they can provide clients with forward-secrecy. That is, if honest servers throw away data encrypted with old keys, an attackets with access to private keys cannot rcannot ecover past data. This property is an improvement over that offered by the Keybase system, which assumed the worst in the case of a comrpomise. If we assume on the other hand that an attacker who steals a private key operates in cohoots with the server, then we cannot offer any guarantees about forward secrecy.

Servers do not trust each other. If one server becomes corrupted, it has no bearing on the other servers in the system. In other words, we assume attackers can stand up their own servers, since anyone in the system can do so.

3 Design

FOKS is a classic client-server system. At a hight level, the clients manage private keys, and the server manages public keys, encryptions of, shared secret keys, and encrypted data. Users generally trust their servers to be online, available and not to intentionally sabotage agreed-upon protocols.

3.1 System Architecture

Much like HTTP or STMP, FOKS clients communicate with one or more servers, depending on where users have accounts. They can safely ignore the other servers in the system. Most communication is between client and server, and there is little if any server-to-server or client-to-client communication. This property simplifies protocol upgrades and network configuration.

Each client can speak for many users, as users can have accounts on different servers, or several accounts on the same server. By analogy, an email client can server multiple emails accounts for the same user concurrently, say one for work and one for personal use. Or a web browser might have different personae (with different cookies, preeferences, passwords and history) for the same user.

Each of the users can of course have multiple devices, like a desktop, a laptop, a phone, and a Yubikey. Additionally, users can have "backup devices", which can be writted down on paper and stored in a safe place. The system recommends at least two devices to prevent data loss. That is, these devices have private keys that decrypt data, and the loss of the last key prevents decryption of the data. Obviously there is a trade-off here: the more devices, the more likely the user will lose one, or have one stolen; the fewer devices, the more likely the user will lose all devices and therefore access to data. Some optimal middle ground exists, but varies with the users and their behaviors.

3.2 Key Hierarchy

The FOKS key hierarchy sits at the core of the system. It aims to provide users with a sequence of symmetric keys shared across all of their devices, so that they can store data encrypted with the latest key, and can decrypt (and authenticate) data encrypted with older keys when necessary. Similarly, users in a team should share secret keys that users outside their teams cannot see, allowing them to share encrypted data via untrusted FOKS servers.

3.2.1 Device Keys

When a user sits down at a FOKS client to signup or provision a new device for an existing account, she first creates a new key-pair specifically for that device. The private key never leaves the device. She shares the public key with the FOKS server, who eventually selectively shares it with user users. We detail the exact cryptography in Section 3.8.

Hardware keys that support the PIV protocol (like Yubikey version 5 and later) can also be used as device keys. These devices get randomly-generated private keys in the factory, written to one of 20 possible "slots." FOKS users select a slot to use, and the client sends the corresponding public key to the FOKS server. Signing and decryption operations happen on the device against the chose slot.

3.2.2 Per-User Keys (PUKS)

Every user on the FOKS system has one of more per-user keys, or PUKS. A PUKS is a randomly-generated key-pair whose private key is encrypted for each of the device public keys. This way, all current devices can access the current PUK secret key, and perform decryptions or signatures for the current PUK public key. The client makes a new PUK every time the user revokes a device. The system encrypts the old PUK secret keys for the new PUK secret key. This way, a device that has access to the latest PUK can get access easily to all prior PUKs.

Once the PUK sequence is estblished, the system has a convenient way to encrypt a data for all of the user's device — it simply encrypts the data for the user's latest PUK.

3.2.3 Per-Team Keys (PTKs)

Each team has a sequence of per-team-keys, or PTKs, which are analogous to PUKs for users. Upon creation, a team gets a new random PTK. The client performing the creation sends the public part of the PTK to the server. The private part of the PTK is encrypted for each member's latest PUK, and therefore is available on each of the user's devices.

As with PUKs, data that the team shares is encrypted for the team's latest PTK, and all members can decrypt it. As we will see in Section 3.6, teams can join other teams, but the key hierarchy works just the same. When team A joins team B, the secret part of team B's PTK is encrypted for team A's latest PTK, so that all members of team A can decrypt B's PTK, and therefore, all of B's encrypted data.

3.3 Key Roles

FOKS has a notion of a "role" for device keys, PUKs and PTKs. The roles are: owner, admin, and reader, but reader keys have a "visibility level" that varies between -32768 and 32767. There is a total ordering among key roles, so that owner > admin > reader, and between reader keys, $k_1 > k_2$ iff k_1 has a higher visibility level than k_2.

The important property enforced is that we only encrypt PUK k for device key j if $\mathsf{role}(k) \leq \mathsf{role}(j)$, and similarly, we only encrypt PTK k for PUK j if $\mathsf{role}(k) \leq \mathsf{role}(j)$.

The idea here is that the owners of a group get to see all the keys; the admins can see the admin and reader keys; and the readers can see keys at or below their visiblility level. This configuration allows groups to have lower-privileged members, and for users to have lower-privileged devices. At Keybase, a similar but less-flexible property allows "bots" into teams, so that all the members of the teams can interact with the bots, but the members had channels to communicate that the bots aren't privvy to. For now, all user devices are at the owner role, but we plan to relax this requirement in the future.

3.4 Data Structures

We now have some basic motivation as to what the key system ought to achieve. It ought to allow groups of devices, groups of users, or groups of users and teams to share a secret encryption key. From there, they can share data encrypted (and authenticated) with that key. But the question becomes, how are users formulated from devices, and how are teams formulated from users so that only desired members are in the group, especially if the server behave maliciously?

For instance, a malicious server might fool a user into encrypting secret data for an invalid device, or team administrator into encrypted data for an invalid user.

3.4.1 Signature Chains

FOKS uses the same mechanism as Keybase here — the signature chain (or "sigchain" for short). The sigchain is a series of signed statements that form a cryptographic chain, meaning they can only be replayed in the intended order. Replaying the chain allows a viewer to confirm the chain appears how the author intended and wasn't tampered with, even if the set of signers varies over time. Of course, signers do vary over time as users add and remove devices, or as they add and remove members from teams.

Each user (and team) gets its own sigchain. The sigchain keeps an indellable record of which keys can update the chain, and which PUKs or PTKs are currently active for the user (or team).

Users The first link in a sigchain is called the "eldest" link. For user sigchains, the first device generates this link, generates the first PUK, and then computes a signature over the following data:

- U.1 The hash of the previous link in the chain (nil for the eldest)
- U.2 The current sequence number of the sigchain (which is 1 for the eldest link)
- U.3 A commitment to the next random tree location (see Section 3.4.5)
- U.4 The current Merkle root hash (see Section 3.4.3)
- U.5 The user's ID and the server's host ID (see Section 3.4.6)
- U.6 The user's new PUK public keys
- U.7 The user's new device key
- U.8 A "subchain tree location seed commitment" (see Section 3.4.7)
- U.9 A cryptographic commitment to the user's username (see Section 3.4.8)
- U.10 A cryptographic commitment to the user's device name (picked by the user)
- U.11 The role of the new device (currently always owner).

U.12 For Yubikeys, a public "subkey' (see Section 5.3)

The client computes nested signatures first by the new PUKs introduced in Step U.6, and lastly by the user's device key. (Recall that sometimes several PUKs can be introduced at once due to the different possible device roles). The client uploads the whole package as the user's eldest link.

Subsequent links proceed in largely the same way, with a few minor differences. The previous hash (U.1) is the collision-resistent hash of the package uploaded in the previous step. In some cases, like device addition, new PUK public keys (U.6) do not appear. In these cases, no signatures with PUKs are required.

For any link in the chain, a set of devices is authorized to make further updates to the chain. After the first link, the set contains only the first device (sometimes called the "eldest" device). A link can either add a new device, or revoke an existing device, updating the set of authorized devices accordingly. When clients upload new chainlinks, the server enforces valid signatures by authorized devices. When users replay this chain, they perform the same check. This simple mechanism ensures the server can't introduce a bogus device.

Teams A team chain link contains the following fields, many of which are analagous to user chains:

T.1 The hash of the previous link in the chain (nil for the eldest)

T.2 The current sequence number of the sigchain (starting at 1)

T.3 A commitment to the next random tree location

T.4 The current Merkle root hash

T.5 The team's ID and the server's host ID

T.6 The user (or team) ID, host ID, and PUK (or PTK) of the actor making the change

T.7 New PTK public keys

T.8 A set of membership changes

T.9 A "subchain tree location seed commitment"

T.10 A cryptographic commitment to the team's name (optional if not changing)

T.11 The team's "index range" (see 3.6.2)

Since teams can contain both users and other teams, the actor creating or modifying the team can be either a user or a team. In FOKS, a *party* refefes to someone or something that can be in a team, so either a user or a team. In field T.6, the link contains the unique identifier of the party (which is the user or team ID plus the host ID), and also the key making the change. For users, this key is the user's latest PUK at the owner role. For teams, it's the team's latest PTK at the desired source role. That is, consider a teams T where users a and b are owners of T, c is an admin and d is a reader (at visibility level 0). If T creates a new team U with source role of owner, then only users a and b will have access. If T creates the new team with source role of reader, then all users will have access.

FOKS clients and servers enforce these access controls with the key hierachy. In the case of the owners of T creating U, T's owner PTK appears in field T.6 and performs the signature over the chainlink. As T creates U, it makes new PTKs for U. It encrypts the secret keys of these new PTKs for the owner PTK of T. This way, everyone in the owner group of T can now access U's

PTKs. The second example follows similarly, with the readers of T getting access to U's secret PTKs after team creation.

The membership changes field (T.8) contains the following fields for each member being modified:

M.1 The "destination role": the role the member is to have in the team; for removals, this is the role none.

M.2 The member's party and host IDs (a party ID is a user ID or a team ID).

M.3 The source role: the role the member has in its current party.

M.4 The member's public PTK or PUK from its current party.

M.5 The generation number of that PTK or PUK.

M.6 A commitment to a "team removal key" (see Section 3.7.1)

In the case of team creation, member addition, member role upgrade or downgrade, the role in field M.1 is the new role in the target team that the member has after the change is applied. In the case of removal, the role is none.

Note that in field M.4, the public PUK (or PTK) appears directly in the chain, in addition to the ID of that party. Admins and owners are later allowed to make team modifications, and these are the public keys that will sign these modifications. Team readers in particular might lack the permission to load the chains of these users and teams directly, so it's crucial the keys appear directly in the team chain. See Section 3.9.2 for further details about sigchain visibility.

As alluded to above, owners have ultimate control over the team. They can add and remove members, add other owners, downgrade owners to admins, etc. Admins have more limited control; the have similar control over admins and readers, but cannot: upgrade admins to owners, introduce new members as owners, remove existing owners; or downgrade existing owners. Readers cannot make any team modifications but can of course read team chains, and can access data protected by the reader's PTKs at their level and below.

Teams, unlike users, can include members located on different servers. Above, in item M.2, we include the host ID of the party in the membership change. Remote members cannot be admins or owners, but can be readers. This configuration allows for convenient data sharing across federation boundaries, but simplifies team management relative to an alternative system where remote members can be owners or admins.

Parties making changes to team sign new team chain links much the same way as user devices sign user chain links. First, all new PTKs sign the chain link, and then the acting party's latest PUK or PTK signs the chain link. This PUK or PTK must be the exact key advertised earlier in the chain in the case of link 2 and above. The eldest link is essentially self-signed.

3.4.2 Sigchain Playback

Whenever a client on the time interacts with another party, or even itself, it begins by playing back that party's sigchain. The client connects to the party's home server, downloads any new links it hasn't seen, and ensures the new links play back cleanly on top of links it previously cached. The rules for playback are largely implied by the discussion above, such as:

- All chains start at 1, and have ascending sequence numbers.
- All links should contain the hash of the previous link.

- All links should have valid signatures by devices, PUKs or PTKs that the chain itself authorized. Eldest links are essentially self-signed.
- The role restrictions described in Section 3.4.1 are obeyed in team chains.

Once playback succeeds, the client has a set of devices keys, PUKs and PTKs that can speak on behalf of the party. It also has rosters of devices of members who share that key. From here, useful application work can take over. For instance, in the case of file sharing (whether via Git or via a KV-store interface), a key derived from the most recent PUK or PTK serves as the symmetric key for authenticated encryption. Whenever the PUK or PTK rotates to a new generation, future encryption should use the new key.

3.4.3 Merkle Tree

Section 2 describes the FOKS threat model, and we assume the server can behave maliciously. Sigchains prevent a server from creating new chainlinks out of whole cloth and interleaving those with legitimate links. The protection simply is that the server does not have the private keys the parties use for signing. We assume those never leave the devices they are made on. By these mechanisms, the server cannot extend a chain or replace an inner link. The server can of course create a brand new chain, and might attempt clients into using this chain rather than the one the users intended. This case of "mistaken identity" is one of *naming*, which we cover in Section 3.4.8.

The server can nonetheless *withhold* links at the end of a chain, or show some clients the full chains, and other various subchains, with the hope of forcing them to "fork" the sigchain into 2 incompatible directions. We need a mechanism to force the server's hand into showing a consistent chain tail to all users who request it. This is where the Merkle tree comes in. FOKS servers expose a Merkle Tree that forces the server to commit to a coherent state of the system, across all parties, preventing selective rollback of sigchains.

FOKS uses a Merkle Tree strategy that is a hybrid of Keybase's [1], CONIK's [4] and SEAM-Less's [2], but with a simple novel mechanism introduced to obviate the need for pseudo-random functions. The Merkle tree stores each party's sigchain tail at a leaf in the Merkle tree. The server computes hashes all the way up to the root node. The *root block* contains this tree root hash, a pointer to the previous version of the tree (called the previous "epoch"), a logarithmicly-sized set of pointers to previous roots further back in history, and also the tail of the server's hostchain (see Section 3.4.6). The hash of this root block is now a summary of the entire system. The server attempts to publish root blocks immediately after every sigchain update, but sometimes batches them for efficiency. In general, roots should be produced no more than 15 seconds after a sigchain update to keep clients responsive (since sometimes they need to wait for updates).

3.4.4 Root Chaining

Since many users and teams might be active on a signle FOKS server, the user has no reason to download every root Merkle epoch. Rather, the user will experience long gaps between epochs, especially if going periodically offline. Whenenever a client downloads a root block, say at epoch i, it writes it to local storage. When the client later downloads another root block, call it k, the client enforces that $i < k$. It also sends up the paremeter i, and the server replies with intermediate blocks $j_1, j_2 \ldots j_n$, such that $i < j_1 < \cdots j_n < k$, where block j_a contains a pointer to j_{a-1}, block j_1 contains a pointer to i and block k contains a pointer to j_n. The spacing of the previous pointers in the root block ensures than n is appromixately $\log(k - i)$. Chaining root blocks in this way encourages the server to maintain a consistent, coherent append-only tree. Clients should perform periodic audits of all root blocks to ensure previous-pointer consistency across all epochs.

3.4.5 Location Hiding

For the party p, at sequence number i, the leaf node is a key-value pair, where the key is $H(p, i, t, r_i)$, and the value is the hash of the signature of the last link in the sigchain (the ith). t is an enumerated value that specifies which type of chain for p this is. As we will see in Section 3.4.7, each party has multiple chains. r_i is a random value that was generated when the $(i-1)$ link was published. The $(i-1)$ link contains the value $H(r_i)$ in the field T.3 from Section 3.4.1. This simple mechanism achieves the same end as the PRF in CONIKs and SEAMLess, in that it's unpredictable in the tree where a team's sighcains is stored. This unpredictability prevents data leaks that would otherwise allow the owners of neighboring nodes in the tree to deduce when p's chain advances.

When a client requests a sighchain for party p, the server returns $(n+1)$ paths down from the root of the tree, if the sigchain is n links long. The first n paths point to chain links 1 through n as described above, and the last is a proof that the $(n+1)$ link does not exist. For each path, the server returns neighbors necessary to trace the path back to the tree root. The server also returns the sequence $(r_2, r_3, \ldots r_{n+1})$, so the client can verify they match the commitments in the sigchain. Note the first link location is not randomized with r_1, since there was no 0th link to commit to r_1. However, the eldest link's placements is already essentially random since the party's ID p is generately randomly at the first link (and is predictable thereafter).

3.4.6 Hostchains

Servers maintain hostchains so they can manage and rotate their signig keys, DNS names, and TLS keys. Like team and user chains, hostchains form a cryptographic chain, ensuring they can only be replayed in the intended order, even if modified in transit. Chain links have sequence numbers and contain the cryptographic hashes of previous links. When an administrator creates a new server, they first create a *hostkey*, a signing key-pair. This public key becomes the host's ID. The first chainlin contains this hostkey and several subkeys, one that serves as a TLS CA for the server, and one used to sign *zonefiles* for the server. The *zonefile* contains the DNS names for the server's various services (see Section 5.4). Subsequent chainlinks can change any of these keys or subkeys, as long as they are signed with keys valid up until that point. clients play these links back to map host IDs to DNS names as they establish connections to new servers.

3.4.7 Subchains

User Settings We have already seen two sigchains: the user chain and the team chain. Each of these chains has a subchain. Users have a *user settings subchain* in addition to the user chain. Currently, this chains contains information about the user's passphrase, which can optionally be used to locally encrypt device keys (see Section 3.10.2). Whenever the user changes her passphrase, she writes a new link to the chain, so that the server cannot roll back her passphrase information to an earlier setting. Chain locations for links i_2 and above are computed as in the user chain itself, but with $t = \mathsf{settings}$ rather than $t = \mathsf{user}$. However, the eldest link is randomized, since otherwise, its location in the tree is predictable given the user ID.

Recall that the eldest user chain link contains a "subchain tree location seed commitment"(U.8). When creating the link, the client software generates a random valaue s, and puts the hash $H(s)$ into the chain link. Then $r_1 = H(s, \mathsf{settings})$, and the Merkle Tree stores the first link of u's user settings chain at location $H(u, 1, \mathsf{settings}, r_1)$. The server knows s, and u knows s, so both can compute and verify the random location of the first link of the user settings chain. But other users who view u's sigchain do not know s and therefore cannot infer if u has a user settings chain or not.

Team Membership Users and teams have a subchain to keep track of which team it is a member of. As we discuss in Section 3.7, users need to rotate their teams from time to time based on key rotation events, like device revocation and removals from teams. The general strategy the user follows is to enumerate all teams she is transitively a member of (since recall teams can be members of other teams), and then check each team for staleness, rotating those that are stale. If this membership list were based purely on server trust, a malicious server could withhold team memberships from a user to silent suppress team rotations.[1] Instead of relying on server trust, the client software maintains a list of team memberships as a sigchain, which the server must faithfully synchronize across all of the user's devices. The tree location mechanism here is the same as for user chains, with $t = \mathsf{teamMembership}$. See Section 3.7.1 for more information on team additions and removals, and how the client maintains this subchain.

3.4.8 Naming

The server stores user and team chains at tree locations indexed by user IDs and team IDs respectively. These IDs are large random identitiers, and are not friendly to humans. Users of course need to be able to refer to these parties with convenient names. Morever, parties should have the ability to change names from time-to-time, though not too frequently, as too much churn in naming would complicate the user experience.

Naming in FOKS works as follows. When a user signs up, she first picks a name n, and the software later generates a random user ID u that n will correspond to. The server stores the user ID u at the Merkle Tree location $H(n, 1, \mathsf{name}, nil)$, and the sigchain for u commits to n at U.9. When another user loads u, she looks up the Merkle leaves at location $H(n, 1, \mathsf{name}, nil)$ and $H(n, 2, \mathsf{name}, nil)$, confirming that the first location maps to u, and that the second location is absent from the tree, confirming the name hasn't be remapped. The loader then confirms that the name matches the commitment in u's sigchain.

This system ensures a 1-to-1 mapping between a name n and a user ID u. u can later switch to a new name n' by adding another sigchain link commiting to n' that overrides the earlier commitment to n. The server can likewise later reassign n (though we currently do not allow this) by posting a new user ID u' at tree location $H(n, 2, \mathsf{name}, nil)$.

Team naming works exactly the same way.

3.5 Provisioning

When users sign up for a new FOKS account, they establish a first device. They establish further devices via a process called "provisioning". In provisioning, an existing device signs the device key of the new device, and also sends it the private side of the current PUK. As we saw above, the signatures form a link in the sigchain. The server helps exchanged encrypted PUK keys as "sidecar" data.

3.5.1 Device-to-Device

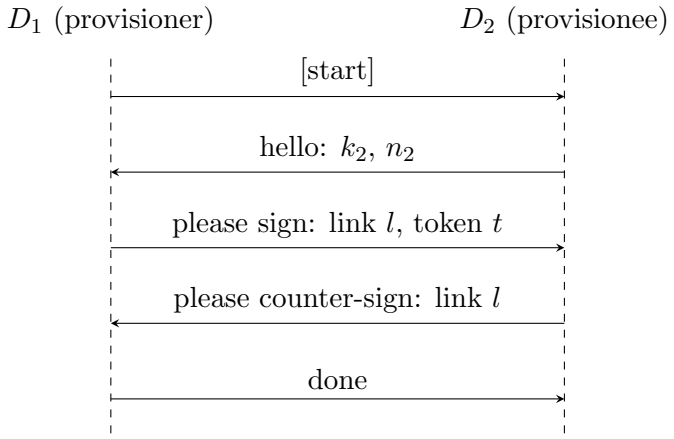
The most complicated case is device-to-device provisioning. A user has an active account on device D_1 and uses it to directly provision device D_2. We call the existing device, D_1, the "provisioner", and the new device, D_2, the "provisionee." Device D_2 generates a new device key k_2 and prompts the user for a device name, which is n_2. Both devices pick random 125-bit session keys, call them

[1] A malicious server could also reject a legitimate team update, but the user's software can alert the user of such a failure.

s_1 and s_2. Both clients show these keys to the user, using a simple encoding scheme we call "high-entropy secret phrase". See Section 3.8.5 for more details. An example might look like:[2]

```
sniff 216 tilt 139 fat 230 patient 228 same 87 burger 13 master
```

To initiate the key exchange protocol, either the user enters s_2 on device D_1 or s_1 on device D_2. In both cases, the message flow is as follows:



The first message, "start", is optional and only needed in the case that s_2 is entered on device D_1. In the "hello" message, the provisionee sends its new key and desired device name over to the provisioner. The provisioner then makes a sigchain link, l, and in the "please sign" message, asks the provisionee to sign it with k_2, the new device key. The provisoner also includes token t so the provisionee can load the user's sigchain from the server. In the "please counter-sign" message, the provisionee returns the signed link and asks the provisioner sign it a second time with k_1, D_1's device key. Finally, the provisioner can post the new signed sigchain link l to the server, including encryption of the latest PUK for k_2 with l. Once the provisionee receives the "done" message, it can load the user's PUKs from the server, and decrypt them with k_2.

Though messages between D_1 and D_2 could use interesting local peer-to-peer protocols, we employ a simpler, more reliable strategy. The FOKS server proxies all of these messages. One device sends a message, while the other polls for the next message in the sequence. Let s^* be the session key that was input on the other device. Both devices encrypt all messages send to the server with the session key s^*, using authenticated encryption.

Some researchers [5] have suggested using a passphrase-authenticated key exchange (PAKE) rather than this simple s_1 / s_2 mechanism. The observation is that the secret phrases, which users have to type until smart-phone apps are built, are quite long and could be much shorter with PAKEs. We considered these constructions but rejected them for FOKS, since we are concerned the PAKEs can be easily DOSed. That is, the provisionee D_2 is not authenticated to the server since the user hasn't logged in on D_2 yet; this login happens naturally as a result of the provisioning process. Thus, there is nothing to stop an attacker from targeting a user and entering bad PAKE codes before the legimiate user can enter the correct code. Such an attack would not break the integrity of the protocol, but could prevent the user from ever successfully provisioning.

[2]The `foks` client running `device assist` one-shotted this secret phrase, and I am pleasantly surprised. When I first used a BBS in 1990, the SYSOP (my friend's older brother) assigned me the handle "Burger Master". I hadn't thought of this in years.

Note that several of these provisioning sessions can happen concurrently on the same the server. D_1 and D_2, with the server's assistance, must separate their messages from other currently active clients. They therefore tag each message in the exchange with the channel identifier $H(s^*)$, where H is a one-way function like SHA2. This channel identifier is unguessable for any malicious clients who do not know s^*, and will clearly be unique across all device pairs.

3.5.2 Yubikey-to-Device and Device-to-Yubikey

Most of the protocol in Section 3.5.1 can be skipped in the case of either provisioning a regular device (a computer or a phone) with a Yubikey, or vice versa. No communication needs to bounce off the server, as the device and the Yubikey can comunicate directly via the local machine. The crux of the protocol, however, remains. For a Yubikey provisioning a new device, the device makes the new link, the device signs the link, and the Yubikey countersigns it. Then the device posts the link (and encrypted PUK keys) to the server. And vice-versa for a device adding a new Yubikey.

3.6 Teams

We covered much of how teams work in Section 3.4.1, largely by analogy to users and devices. There are, however, some important differences, which we cover here in more detail.

3.6.1 Cross-Server Teams

As described in Section 3.4.1, teams are composed of parties that can span multiple federated servers. A team has a home server, which hosts its public keys, encrypted secret keys, sigchain, and Merkle Tree. An importantly, all the admins and owners — those who have the ability to change the team — must be on the same home server. However, team readers can join the team from across the Internet.

Because readers can be local or remote, important team protocols, like addition and removal, must work with more generality, as we cannot assume that the reader and the team have the same home server. Morever, we cannot assume much cooperation among independently administered servers under our thread model (see Section 2).

3.6.2 Cycle Avoidance

One problem in particular the presents itself immediately: potential cycles. Imagine a simple case, with three hosts h_1, h_2 and h_3, and three teams on those host, respectively, t_1, t_2 and t_3. If t_1 adds t_2 as a reader, t_2 adds t_3 as a reader, and t_3 adds t_1 as a reader, there is now a cycle in the team membership graph. That cycles are bad is not immediately obvious, but we must consider what happens in a rekey scenario.

Imagine at the start of the sequence, team t_1 consists of two owners — call then u and v — and no other members. Team t_2 consists of a single owner x, and team t_3 consists a single owner y. Focus on user v, who will eventually be removed. At the start v has access to t_1's reader PTK at generation 1, call it k^1. Next, x adds the readers of t_1 to t_2 as readers. As a result, v gets access via k_1 to t_2's reader PTK, call it k_2. Next, user y adds the readers of t_2 to t_3 as readers, giving v access to t_3's PTK k_3^1 transitively via k_1 and k_2. Finally, u completes the cycle, adding the readers of t_3 to t_1 as readers. Figure 1 shows this initial configuration.

Everything up until now is working as planned, but now consider what happens when u removes v from t_1. What should happen is that all 3 teams should rotate so that v loses access to their most recent reader PTKs. But in practice, this objective cannot be achieved. Playing out the scenario,

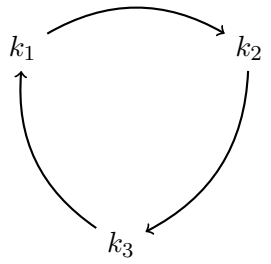


Figure 1: The initial configuration of teams t_1, t_2 and t_3. A link from k to j means that knowing k gives one access to j, i.e., that k is a reader of j.

u rotates t_1's reader PTK from k_1 to k_1'. It encrypts k_1' for k_1 so that the remaining members can still access old data. It also encrypts k_1' for t_2's most recent reader PTK, which remains k_2. See Figure 2 for what this new configuration looks like. One thing to notice immeidately is that any node on the graph is reachable from any other. That is, if user v knows k_1 (which he had access to before u removed him), then he can still decrypt any other key on the graph, including k_1', which he reach via $k_1 \to k_2 \to k_3 \to k_1'$. So in a deep sense, the rotation has failed.

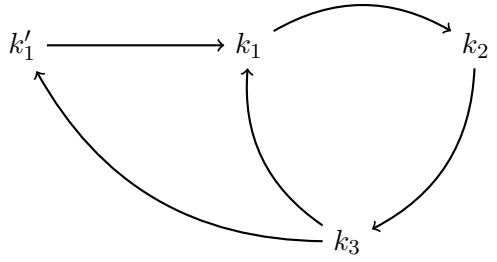


Figure 2: After u removes v from t_1 and rotates t_1's reader PTK to k_1'.

To continue a few more steps, assume t_2 rotates (see Figure 3), and then t_3 rotates (see Figure 4). for the new configuration. The problem remains. Any node in the graph is reachable from any other, and therefore, v can still access the new PTKs for all teams. We can generalize (without proof) that there is no "fixed point" during this decentralized key rotation process, and that no amount of rotation will lock v out of the teams. A different system in which t_1, t_2 and t_3 rotated simultaneously might avoid this problem, but we see no robust way for them to do so if those teams reside on independent servers.

The simplest path forward is to disallow cycles in the global team graph, which seems like a natural constraint. However, enforcing lack of cycles globally is challenging, and seems to require some concept of global locks across the internet. To see why, consider four teams, t_1, t_2, t_3 and t_4, on four different servers. First, t_1 adds t_2 as a reader, and t_3 adds t_4 as a reader. So far so good. If t_2 adds t_3 as a reader, or t_4 adds t_1 as a reader, we still are cycle-free. But if the two additions happen roughly simultaneously, a cycle is formed, and there is no easy way for any of the hosts involves to detect it.

The most blunt and naive solution is for each team to get an "index", which which is simply a natural number. This index cannot change once assigned, and t_1 can only add t_2 iff t_2's index is less than t_1's. This simple rule prevents cycles, but is inflexible and doesn't allow much play in the team graph as the system evolves. FOKS opts for a slightly more flexible solution of a similar flavor. Each team gets a "range" of possible index values, representing bounds on where the team's

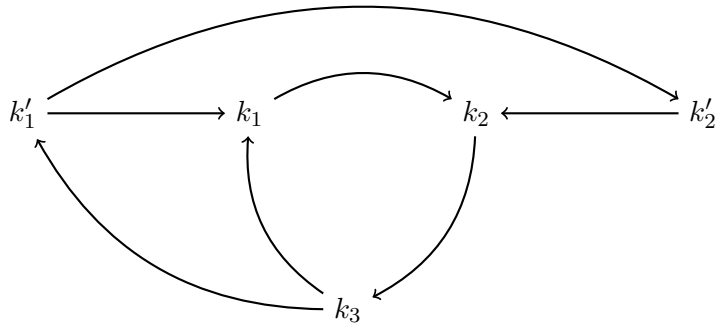


Figure 3: *The Widening Gyre*: After u removes v, t_1 rotates, then t_2 rotates.

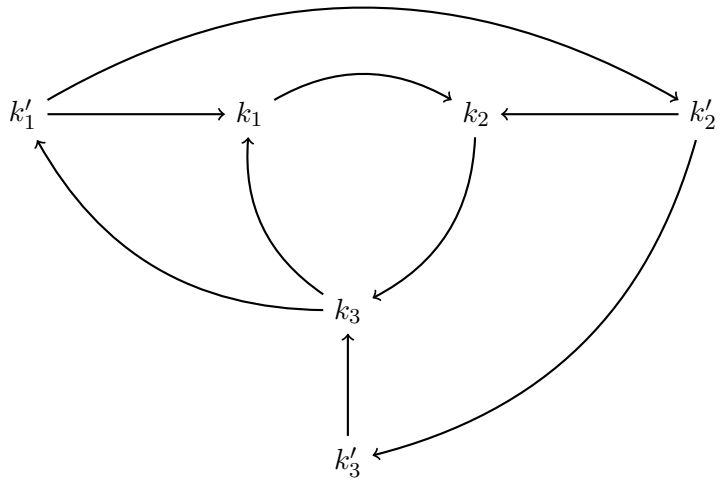


Figure 4: *Things Fall Apart*: After u removes v, t_1 rotates, then t_2 rotates, and finally t_3 rotates.

index will eventually settle. This range in practice is an interval on the positive rational numbers, of the form $[a, b)$, where $a, b \in \mathbb{Q}^+$. b can take on the special value ∞. A team can always restrict its range but can never expand it. The rule for addition then becomes that t_1 can add t_2 iff the lower bound of t_1's range is greater than the upper bound of t_2's range. The rule guarantees that whereever t_1 and t_2's indices eventually settle, t_1's index is greater than t_2's.

All teams get the initial default range of $[1, \infty)$. When a team administrator seeks to add one team to another, she might need to change one or both of the team's ranges. For instance, take teams t_1 and t_2, both at the default range $[1, \infty)$. To add t_1 to t_2 as a reader, the administrator can reduce t_1's range to $[1, 128)$ and team t_2's range to $[128, \infty)$. Now the property above is upheld, since $[1, 128) < [128, \infty)$. Note that it still remains possible to add t_2 to another team or to add another team to t_1 by making appropriate range reductions.

3.6.3 Invitation Sequence

By default, two parties p and q cannot view each other's sigchains, even if on the same server. But p must be able to read q's sigchain in order to compute its current PUKs or PTKs if adding q to a team. So a protocol must be invented to allow p and q to exchange information before the actual join process occurs. There are three general steps: (1) p creates a multi-use invitation token,

which p can share with one or more parties; this invitation exposes the team's name, host and ID to all invitees; (2) q "accepts" this invitation, thereby granting p access to its sigchain; and (3) p "admits" q into the team.

Invite Tokens A team's administrator or owner can create a multi-use invitation token for a team t. First, the administrator creates a team certificate with the fields:

- The team's ID
- The team's Host ID
- The team's current admin PTK
- The current time
- The name of the team
- The team's index range

The adminstrator then signs this certificate payload with both the current PTK, and also the original PTK, whose hash is the team's ID. If the PTK hasn't rotated, then of course only one signature suffices. In totality, the above data and the signatures are quite large, so we introduce one layer of indirection to make them shorter. The administrator posts the signed certificate it's home server, and then formulates an invitattation as: (1) the hash of the above data; and (2) the host's ID. The resulting token can look something like this:

```
YcarI5JTMAVDHG47cb5l1Pp87COqmK6nzf6h12hchVLMMjlplmjC5dky
wwnqlOhuakWkAjzO2zJ6BtxrnOxdZV8NyLby2r16MhCvC
```

Which is small enough to exchange via group text chat.

Accepting Invitations Once a party p receives the invitation to team t (p can be a user or a user acting on behalf of a team), it breaks the token into its two parts, the host ID and the certificate hash. It maps the host ID to a DNS name (see Section 3.11), and then asks the server for the preimage of the hash from the invitation. It checks that the preimage hashes correctly, and then opens the certificate. From here, it checks the signatures described just above. Note that at this point, p is not a member of the team, so cannot read the team's sigchain.

There is therefore an oppoortunity for the team certificate to be partially forged. p can check that that first PTK corresponds to the team ID, so therefore knows that whoever generated this certificate was at one time a team admin. But that administrator might have been removed from the team. Also, the name of the team cannot be checked against the Merkle Tree, so the team's name might be forged. Eventually, once p is admitted into the team, it can check that these fields are correct, and leave the team if not. By accepting the invitation, be it malicious or otherwise, p is primarily giving access to reading its sighcain. It must decide whether or not this is a good idea on the basis of who sent the team invitation in the first place. This problem can recursively be solved in FOKS, by a different team, or by trusted channel on a different platform, like a secure team chat.

If p decides to accept this invitation, there are four subcases to consider:

1. p is a user, and p and t are on the same server. Here, the user p instructs her home server to allow the owners and administrators of t to allow access to her sigchain. The server makes a quick note in the database to honor future accesses. Then p writes a link to its Team Membership Chain (see Section 3.4.7) saying it accepted an invitation to team t.

2. p is a user, and p and t are on different servers. User p contacts its home server and asks that it assue a new "remote view permission token" for the administratos and owners of t. The server sends back this token. p encrypts this token with the PTK it received in the team's certifcate, and then posts this encryption to t's home server. Note that p isn't trusting the server's administrators here, so does not send this token in plaintext. As with the above case, p writes a link to its Team Membership Chain saying it accepted an invitation to t. It also informs t's server that it has accepted the invitation.

3. p is a team, and p and t are on the same server. The actor here is an an administrator or owner of team p. He firsts does a cycle check, checking that the index range of p is less than that of t. The server would reject the addition of p otherwise. The admin/owner now operates analogously to Case 1. First, he instructs the server to allow the admins of t to read p's sigchain. Then, he posts a link to p's Team Membership Chain saying that it has accepted an invitation to t.

4. p is a team, and p and t are on different servers. This case is a hybrid of Cases 2 and 3. As in Case 3, the actor first performs a cycle check. As in Case 2, the actor makes a new review view token for p and posts it to t's home server. Then it writes a link to p's Team Membership Chain as above.

Admitting into Teams For a given team t, the owners and administrators see an inbox of pending accepted invitations. For each of these, the administratos makes a yes/no decision as to whether the team can be admitted into the team, and also what it's role should be in the new team. Recall some restrictions on these roles: (1) remote parties can only be readers; and (2) administrators cannot add new members to teams as owners. In any of the four subcases above, the administrator gets the necessary access to play the accepted party's sigchain; as such, he can put their latest PUK or PTK into the sigchain, and can encrypt t's latest PTK for that PUK or PTK.

Once the party is admitted into the team, it can sign a statement with its PTK or PUK to play the team's sigchain. When it does that for the first time, it updates its Team Memership Chain to show that the accepted invite transitioned to an official admittance.

The initial inspiration for FOKS is the SUNDR project [3], which first originated the idea of a fork-consistent blockchain of edits facilitated by a untrusted server. Like Keybase [1], FOKS applies this basic architecture to the problem of key distribution, rather than the data those keys might secure. Many other projects have riffed on this, from CONIKS [4], to the SEAMless [2] work out of Microsoft Research, to the widespread adoption of Key Transparency Signal, WhatsApp and iMessage. The question of federation has largely been ignored, as these sytems all shared the basic architecture of a single upstream server.

8 Conclusion

References

[1] Keybase. Available at `https://keybase.io`.

[2] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1656, 2019.

[3] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[4] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[5] Keegan Ryan, Thomas Pornin, and Shawn Fitzgerald. Protocol security review: Keybase. NCC Group, 2019.