A Decentralized Secure Social Network

Nate Foss, Matthew Pfeiffer, Kifle Woldu, Arthur Williams

May 2019

Contents

1 Motivation

As you are surely painfully aware, Facebook provides decent service at the extremely high cost of constantly disregarding user privacy, common decency, and laws around the world, and yet they still have billions of users. How can this be possible? They have a monopoly. Monopolies breed complacency, low quality, and high costs. And Facebook has a monopoly over your friend network. You can't leave because everyone is on Facebook. Switching to a new platform means every single one of your friends has to make a new account, download a new app, and you have to rebuild that whole network—if you can convince them at all. It's the same story for any platform (though most don't charge such a high price), and it's unavoidable. Metcalfe's Law states that the value of a network is proportional to the square of the number of its users, so it's no wonder the network effect is one of the most powerful social phenomena.

2 Introduction

We believe the solution is a decentralized social network which is encrypted at rest. When the user has the key to decrypt and modify their own data, they have complete control, and can grant and revoke control from third parties. Everyone's data is just 'out there', many copies floating around in encrypted blobs that anyone can host or download but only friends can decrypt. Decentralization also provides robustness against censorship, internet outages, and would-be social monopolies.

When you're offline, your friends could distribute your profile, and to keep it always online, you could pay for hosting with your data (let them decrypt it) or your money, host it yourself, or even get hosting through your school or workplace. You could use a free open source client, one that shows ads, or one which charges a premium—your choice. If your current client starts misbehaving, you can switch to any other with zero friction. Your friends won't even notice. Perfect competition between hosts and between clients.

The key to this decentralized paradigm is not merely security, which is not too hard with public key cryptography, but *user friendly* security, which lets us have the conveniences we're used to in centralized systems, but keeps the network secure and open to anyone interfacing with it in whatever way they please.

We achieve these features, including confidentiality, metadata hiding, profiles, friend networks, instant messaging, groups, and much more through a carefully constructed profile file tree distributed peer to peer over IPFS. It can be easily updated, distributed via deltas rather than bulk transfers, and hosted without being able to glean any information about that user. Offline or network-partitioned use is natural for our system, and it will handle such network difficulties gracefully—spreading data to what peers it can reach and recovering effortlessly once the partition is resolved. Most importantly of all, it lays the foundation for a secure system that people may actually want to use.

3 Related Work

The idea of alternative social networking services that give users more agency is not new. However many of these social networks have deficient properties. They either provide a compromised form of "decentralization", or are hard to use.

3.1 Mastodon

Mastodon[1] is a popular "decentralized" social network. Unlike Facebook or Twitter, Mastodon is not compromised of a single governing entity. Instead, Mastodon is made up of a collection of independently controlled "federated" servers, called nodes. Users decide which node they want to join based on its policies and other users, and then their Twitter-like community is confined to the participants on that node. However, Mastodon's federated approach to decentralization is, in our opinion, somewhat lacking as nodes are not very interoperable and users still have to choose and trust some 3rd party node.

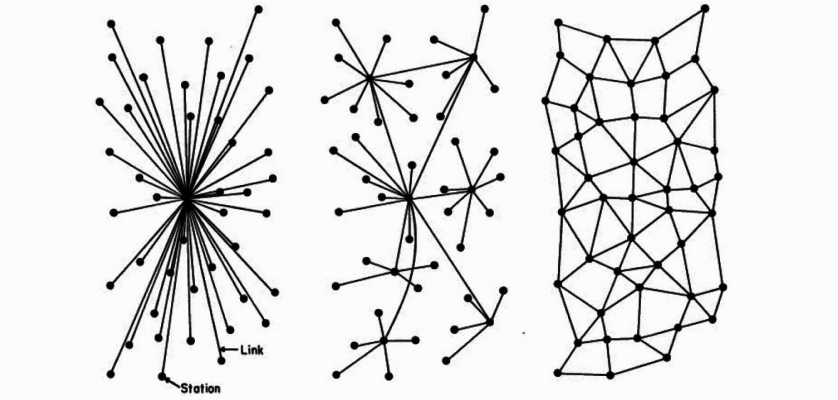

Figure 1: Centralized, Federated, Decentralized

3.2 Secure Scuttlebutt

Secure Scuttlebutt[2] is a protocol for building decentralized applications that work well offline and that no one person can control. The protocol currently offers secure communication between peers, allowing the platform to be leveraged for various applications including that of a social network. Peers all have an identity that's permanent. Peers constantly broadcast UDP packets on their local network advertising their presence and communicate through their peers. Though Scuttlebutt is decentralized and allows for peers to securely communicate, it is not designed for an internet-scale social networking application. Relying on peers to receive updates is very limiting.

In the simple example of two friends, f_1 and f_2, who lack mutual friends. Let's say f_1 posts a funny picture of a cat and all of his friends but f_2 are online to receive the update u. Shortly after, f_1 goes offline. f_2 comes back online and does not receive u because none of f_2's friends have u but f_1. Then, f_2 will not receive said picture until f_1 is online and needs to wait an indeterminate amount of time. This is not a good model with which to build the type of social networking experiences we see today, like instant messaging.

4 Design Goals

4.1 Data Ownership

Today, social networks control your data and are mostly free to use and abuse it, sometimes without users' permission. We desire a system where users have **complete** ownership of their data. Users should have control over who can see what in their profile with fine granularity. For practically every piece of information in your profile, the user can set read permissions. This applies to nicknames, posts, messages, subscribers, who the user subscribes to, etc. But most of all, users should not have to reveal any information to any 3rd party in order to use this social network.

4.2 A Universal Social Profile

There are too many social networks users must maintain independent of each other: Facebook, LinkedIn, OkCupid. We believe this to be a failure of the current centralized paradigm with regards to web applications. Users are forced to take the extra step to grant permissions for applications to use data from other applications in order to stitch their social networks together. The resulting product is unsatisfying and users have trouble finding their friends on every new platform they join. We've designed a platform where third-parties can create custom experiences for any social networking service[3] by providing the basic capabilities for users to construct a profile, articulate a list of subscribers, and view and traverse the connections those users are subscribed to. As an additional benefit, such an open platform would have many interchangeable options for clients people can use to interact with it, leading to much more direct competition between them than we see today. The more competition there is, the higher the quality and lower the costs for users.

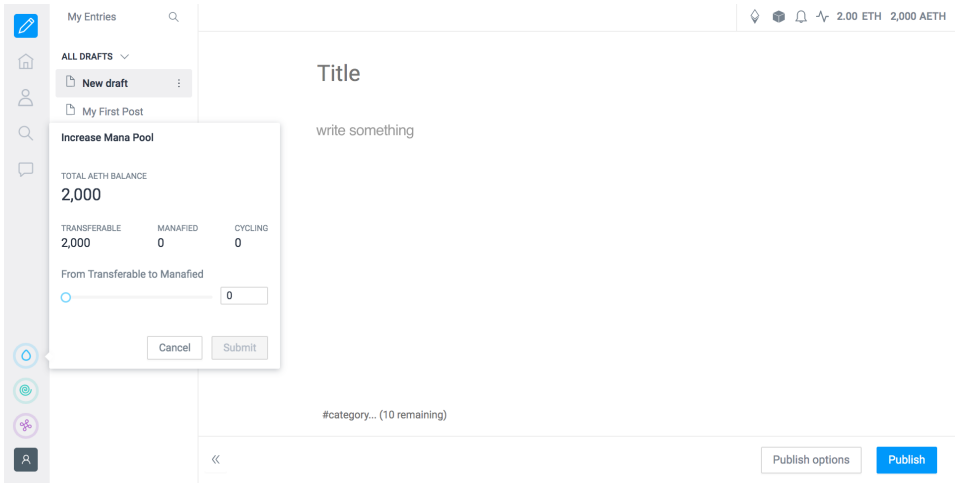

Figure 2: Akasha: Ethereum-Based Social Network. Clean UI, but requires a Chrome extension, Ethereum wallet, and their own cryptocurrency to use it.

4.3 Easy to Use

It does not matter how secure or theoretically satisfying a system is, it will never gain adoption unless people actually want to use it. In order to solve the problem of centralized monopolies, we must ensure users have all of the nice features they are used to from things like Facebook. Many alternative secure or decentralized social networks fail this basic test. One such alternative, for example, requires a chrome extension, Ethereum wallet, and a cryptocurrency payment to make posts (Figure 4.2). Others are not fully decentralized, and require you to choose and at least partially trust some federated third party "node" or "pub" server.

Beyond those basic pitfalls, being decentralized allows us to provide additional features. Our platform does not rely on the online status of you or your peers to interact with the network. We can also provide features allowing users to quickly gather friends to help mitigate network effects that resist transferring from prior social networks. Of course, our system must also include the common suite of features in modern social networking services.

4.4 Reliable

Users should always be able to view their profiles. As users are no longer bound to the performance of Facebook servers but to the performance of computers all across the world on a globally distributed peer-to-peer system, users should be able to view the latest content from their friends all the time.

5 Prerequisites

Our design is built on top of IPFS[4] and relies on many of the abstractions that it provides. We chose to use IPFS because it solves many hard problems for us, including the addressing, networking, and transport layers, a well connected swarm of peers at all times, and (with IPNS) the beginnings of a public key infrastructure with a mutable key-value store.

In this section we review additional details of IPFS, highlight some key abstractions, and make note of any assumptions we have on their capabilities for the sake of our design.

5.1 IPFS

IPFS is a peer-to-peer distributed file system across the entire internet. IPFS enables us to have users interact with their social network without some central entity.

Overall we focus on the following details of IPFS.

5.1.1 Retrieval: Distributed Sloppy Hash Table (DSHT)

IPNS, the naming system explained more at length below, relies on a Distributed Sloppy Hash Table (DSHT) to map public keys to profile hashes. We first review the more basic variant, Distributed Hash Table (DHT). DHTs allow for fast lookup similar to a hash table `(key, value)` in a peer-to-peer network made up of many nodes. This is critical to retrieve files in a p2p file system and is used in applications like BitTorrent. The DHST, most notably, does not store large objects in the DHT but instead stores references to those objects, to save on bandwidth. Also, it provides secure identity generation with a proof-of-work to make Sybil attacks expensive. Below are the data structures used for data generation.

The keys are typically generated via RSA and the `multihash` format looks like the following, where there is a short header specifying the hash function used:

<function code><digest length><digest bytes>

This allows for the system to use different parameter choices and remain compatible.

```
type NodeID Multihash
type Multihash []byte
//self-describing cryptographic hash digest

type PublicKey []byte
type PrivateKey []byte

type Node struct {
    NodeID NodeID
    PubKey PublicKey
    PriKey PrivateKey
}
```

Figure 3: Node Data Structures

5.1.2 Networking

In addition to providing secure node identities and routing via the DHT, IPFS provides a network stack for communicating regularly with nodes across the internet. The stack can provides reliable transport, integrity using a hash checksum, authenticity using HMAC, and the ability to foster high connectivity traversing NATs. To encourage data distribution among the peer-to-peer network and lower latency during DHT lookups, IPFS operates a protocol akin to BitTorrent where data is exchanged by peers in a repeated game.

5.1.3 Immutable Storage: Object Merkle DAG

IPFS stores data in an Object Merkle DAG, a directed acyclic graph where links between objects are cryptographic hashes of the targets embedded in the sources. What this allows for is content addressing, where all content is uniquely identified by its `Hash` , **including links**.

```
type IPFSObject struct{
    links []IPFSLink
    //array of links

    data []byte
}
```

```
type IPFSLink struct{
    Name string

    Hash Multihash

    Size int
    //size of target
}
```

Figure 4: IPFS Object Format

By virtue of being a Merkle DAG, IPFS is also able to verify content using the cryptographic hashes of the object and its links. Another effected of the links is that objects can be referenced with string path lookups, such as /longuglyhash/foo/bar/baz, emulating a filesystem. When performing the lookup, IPFS resolves the first path component to a hash in the object's link table, fetches that second object, and repeats with the next component.

Typically, nodes store objects on their disk space. Nodes who want an object to persist on the network can simply keep that object in its local storage forever, making the object permanent. For objects to be distributed by that node, nodes can simply add the object's hash to the DHT, thereby adding themselves as a peer who can deliver said object, and giving other users the object's path.

Immutability due to the hashing in the Merkle DAG is both a benefit and a drawback. Objects cannot capture mutable state. Thus, clients must be informed of new content out of band (likely a direct peer-to-peer connection) or via the DHT. That is where IPNS, the InterPlanetary Name System comes in.

5.1.4 Mutable State: IPNS

First, let's recall that `NodeId = hash(node.PubKey)` . Now, to publish an object at a consistent address, IPNS provides every node a mutable namespace at `/ipns/NodeId` . Thus, to create mutable state,

1. Nodes can publish an object to this path **Signed** by their private key.
2. When other users retrieve the object, they can check the signature matches the public key and NodeId, verifying the authenticity of the object published by the user.

Publishing this object relies on a similar process as in IPFS. IPNS (1) formats the object as a regular immutable IPFS object, (2) publish its hash on the DHT, keyed by the `NodeID` .

5.2 Key Abstractions and Assumptions

1. **Object Merkle DAG** The Object Merkle DAG governs the way we store data. This of course has major ripple effects on the way we have designed our system. To access an updated profile, users must check for a new content address through IPNS, then load the immutable object associated with it. The bulk of the design is then how we store data so our platform is as secure, performant, and usable as possible.
2. **Cryptographic Hashing** We make use of hashing frequently in our design. We use it for node generation, IPNS verification, content addressing, and metadata hiding (e.g similar to how one might securely store passwords).

   Assumption 1: There exists a family of hash functions providing security (under the Random Oracle Model) from computationally bounded adversaries.
3. **Object Pinning** Nodes can permanently store objects by simply storing data, typically on their node's local storage, to keep it available to the network.

   Assumption 2: IPFS's incentives guarantee reasonably fast retrieval of content so as not to be disruptive to the user experience.

This is quite an assumption as there are no hard incentives (i.e the monetary kind) to incent users to store content of no value to them on IPFS. Currently, users are weakly incentivized by the promise of increasing their reputation on the network and getting more in return. This is still an ongoing research problem and we have identified technologies that we feel introduce better incentives and could include in our system as future work (see section 9).

4. **IPNS** We rely on IPNS as a failsafe (after direct connections) to propagate new content to users. Thus, every time the user produces new content, it must be propagated through IPNS, and it must replace the old entry.

   Assumption 3: IPNS maintains the most recent update and allows for updates that are performant enough. We have found IPNS to generally be somewhat slow, which is why we aim, as part of our system, to have friends who are online at the same time be connected directly to each other. This will both make IPNS lookups faster and allow for push notifications from peers on profile changes.

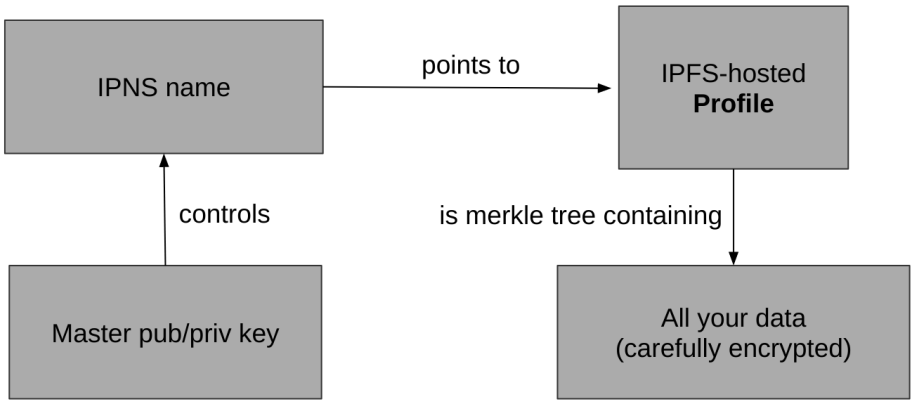

Figure 5: How IPFS and IPNS are used in our system.

6 System Description

6.1 Profile

Each user has a public/private RSA key pair. We will refer to this key pair as a user's master key pair(`rsa-public` , `rsa-private`). This key pair is used for their IPNS identity, which allows them to publish one entry in the DHT. Each user stores the hash of their profile as their entry in the IPNS DHT using this master key pair. This IPNS entry will allow anyone to resolve the user's `rsa-public` to the user's *profile* stored in IPFS.

A user's private and public data is stored in their `profile` , which is an `IPFSObject` and can be thought of as a directory. This object links (via hashes) to other files stored in IPFS and logically include any amount of data in this way. See section 5.1.3 for more.

Figure 6: Profile file hierarchy. *note: pk = public key (usually friend's RSA key), and all other keys and secrets are symmetric keys*

All forms of interaction (aside from optimizations we will mention later) in our system consist of users changing their profile and reading data from other users' profiles. Users may PUSH updates to their profile, and users may PULL updates to others' profiles.

Our choice of IPFS and IPNS have two requirements for maximum availability. First, users must *pin* their data on IPFS in order for it to be readable by others. This can be done on a user's own computers or by an untrusted service (since the data inside the profile is encrypted). Next, users must publish IPNS records linking their public ID (RSA public key) to the hash of their profile, at least every 24 hours to prevent expiry. If a user does not do both of these, their profile will not be resolvable or unavailable to other users.

We decided to use IPNS/IPFS instead of users communicating directly peer-to-peer as in Secure Scuttlebutt because it enables our Always Reliable design goal. We want users to be able to see their friends' data immediately after going online without manually trusting intermediate nodes, and the state kept in the DHT even when a user is offline allows for this.

The profile stores a lot of metadata, much of which is only readable by specific other people. This is all pointed to by IPFS links. See figure 6 for a summary of everything in the profile. We will describe each item in the profile in the following sections.

6.1.1 Access Control

Users may want to encrypt data to only be readable by a set of people in their social network. We accomplish the above by establishing a shared secret for the set of users a user wants to share data with. Then encrypt data readable only to those users with that shared secret

6.2 Following and Friending

The whole point of "Friending" someone is to establish a `shared-secret` (in this case, a symmetric key) with them, both to allow secure communication and so that you can *publicly refer to them in a way that only they can recognize*. It comes down to the problem of having a folder full of encrypted files. How do you indicate to this friend which one is for them, without indicating the same anyone else? With only public key crypto, you could not give it a deterministic file name, since you do not know anything that another person with the friend's public key does not. The only way to get the message to them is to make them test (decrypt) each encrypted file in the folder until they found one that came out nonrandom. With a shared secret only known by the pair, you can indicate which file is for them up front (by hashing the secret) and then they only need to decrypt that one file. As you will see throughout this section, we make extensive use of this convenient feature of the shared secret.

To **follow** a user entails periodically reading that user's profile. **Subscribers** are synonymous with followers in our system. **"Friending"** occurs when two users follow each other. A user stores the list of identities they follow in the private section of their profile (encrypted under the master symmetric key).

Alice and Bob become friends as follows. Alice encrypts "Hello `<Bob-master-pubkey>` :"|| `shared-secret` (symmetric AES key) under Bob's master pubkey and places the result in the `subscribers` section of her profile. Bob does the same, encrypting a different shared secret using Alice's public key and placing it in the `subscribers` section of his profile. This is the only time a subscriber will have to sequentially test decrypt a list of files to see which one was encrypted with their public key. The result is that both of them have established a

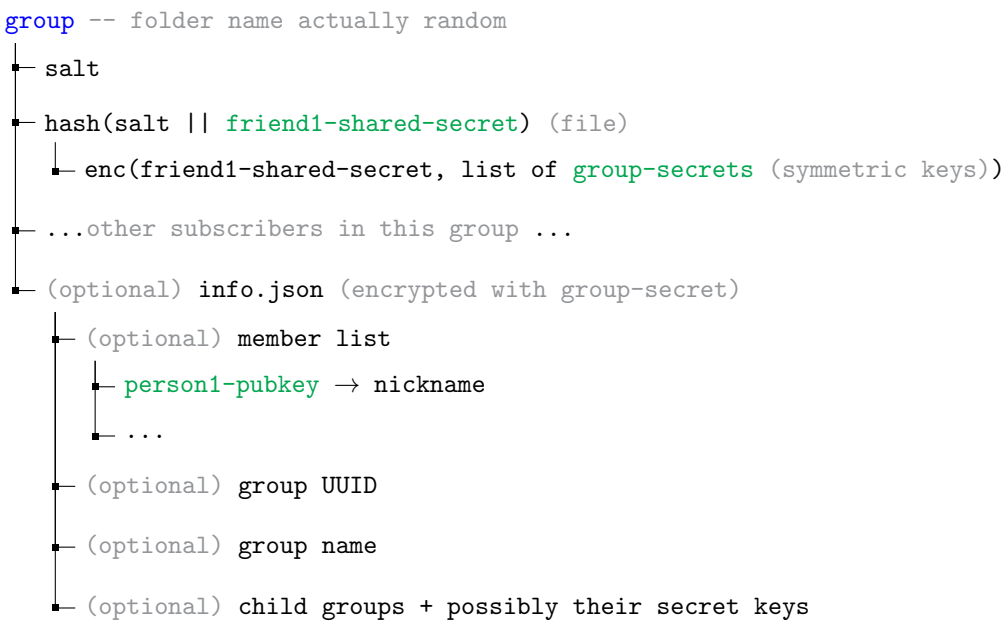

Figure 7: What one group looks like in that list in the profile. The (optional) elements are used for multi-user groups (see section 6.6).

different shared secret with each other. This allows both faster encryption and decryption as well as the type of communication mentioned above, upon which we build groups and everything else (see section 6.3).

6.3 Groups

To truly own their data, users must be able to determine who can read what they store on their profile with fine granularity. It is also desirable to only encrypt each post once, instead of once per person that can see it. That is where the `groups` folder comes in. Each user creates a unique group and key for each audience they wish to publish to. They encrypt all data published to a given group under the most recent `group-key` (symmetric AES key). A user shares group keys with other users by encrypting them with each person's `friend-shared-secret` and storing the result in a list in the appropriate `group` folder, under a file name generated from some `salt` and the `friend-shared-secret` so as to only be recognizable by them (see figure 7). By default, groups are similar to mailing lists—data encrypted with a group key is readable to all recipients who know that group key, but they cannot tell who else is in the group. Note that they can tell how many people are in the group—we actually view this as a feature, because it is then clear whether someone else's post is an important direct message or spam sent to a wide audience.

More on what the optional fields are used for later.

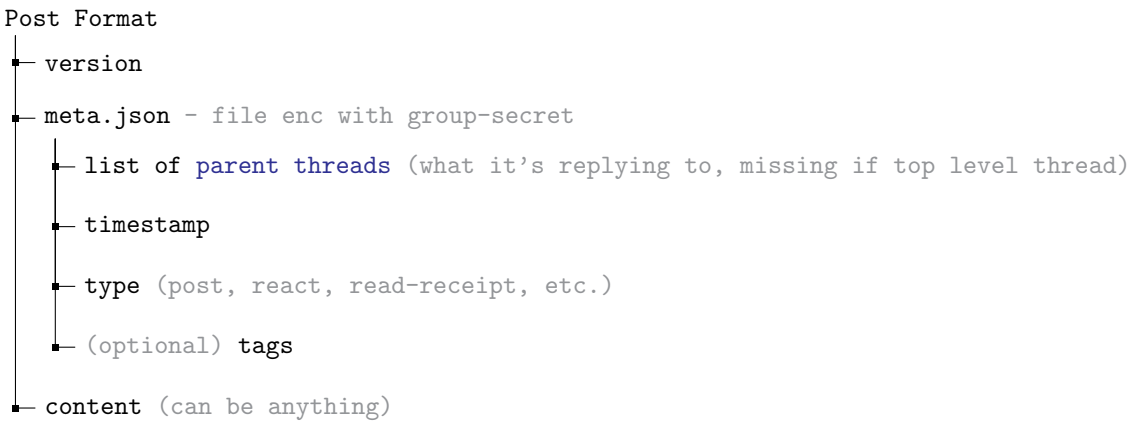

Figure 8: Format of IPFS file storing post contents

6.3.1 Removing Group Members

One may remove someone from a group as follows. First, they remove them from the group membership list. Next, generate a new `group-secret` key and prepend it to the list of group secrets for all remaining members of the group. All future posts to a group must be encrypted with the new secret.

This prevents the removed member from reading any new messages to the group, though it allows them and everyone else in the group to continue to read previous messages without having to re-encrypt them. One may re-encrypt all old data with the new group secret if they wish, but this would require re-upload of all data each time a member is added or removed from a group and likely break any links from other posts.

6.4 Sharing Content

The simplest example of sharing content is the `bio` folder. In this folder is a series of JSON files encrypted with various `group-secrets`. That JSON could include things like your name, profile picture, friends list, etc. that you wish to be visible to the people in the group associated with that key. Remember that group membership is not visible, but an adversary may learn some metadata by correlating which groups are used where and how often. In order to hide this, we use a random `salt` in many places of the profile and then refer to a group as `hash(salt||group-secret)` so that the same group will look different to an adversary in different places, but will always be recognizable by members of that group (and furthermore, findable in $O(1)$ time rather than $O(n)$). This is how we name the encrypted JSONs so that group members can find them easily. Of course, you may also have a public name and profile picture in that folder.

6.5 Messages, Read Receipts, Reacts and Other Posts

The main way of sharing content on a social network is posts and messages, which we treat the same underneath and leave the UI to make a distinction given the context. Like with the `bio`, we hash `group-secrets` with salt to discretely refer to groups, but there are

two distinctions here. First is that there are many subdirectories of the `posts` folder. This is for performance reasons, since there will be many more posts than groups, and this allows users to download only the most recent day or month of posts rather than all of them at once every time. Second is that the post object is not itself encrypted, it is split into two encrypted pieces—the metadata and the content. This is so friends can host all of the necessary metadata to make sense of each others' profiles and posts without having to store large images or videos. That burden is left to the original poster to figure out, but note that they can use a service like Dropbox or Google Drive to host these without trusting them since it's encrypted.

Another feature is threads, so it's clear what is being responded to. Users seem to value this feature in Slack so we have added support for optionally referring to parent posts so that the UI can bundle messages in the same thread.

Read receipts are another common mechanism in social media platforms to let the sender know you have received a message. In our system, each message has a type that lets the UI know how interrupt the message. So as long as all parties agree on what a type means, they can be anything. For read receipts, we could agree upon some unused type, then let the *parent* refer to the message we are marking as read. Similar solutions for reacts, but the *content* could also be used to specify the specific type of reaction.

6.6 Group Chats

Until now, we have discussed groups much like bcc'ed mailing lists, thinking about profile data mostly as "posts" to an audience. However, users often wish to communicate in more of a "group chat" or even Slack workspace type of group. For these we may use the term "messages" instead of posts, but underneath they are the same thing and only the context changes the terminology. This is what the optional group info is for.

A group chat is comprised of multiple profiles with groups that have the same group UUID (universal unique ID—just a large enough random number) in its metadata (see Figure 7). Importantly, no other state is synchronized, and users do not have to use this UUID unless they wish for some messages to be considered "in the same group" as some messages from other people in the UI. Each member of the group controls their own `group-secret` and stores a copy of the member list and other group info. When a group member sees additional members in another membership list (and often a corresponding message), they add the new members to their own list. Because there is no verification, it is possible for membership lists to be incomplete or contain extra accounts. The specific audience each user broadcasts to is at their discretion, but by default we expect people will want to keep it roughly in sync with the other group members. A benign example of not synchronizing is if different people wish to have different nicknames or group names for the same logical chat. Another reason to allow your group state to drift from the others is if you wish to block or mute other users in the chat, where it wouldn't affect everyone else.

6.6.1 Joining a Group Chat

The easiest way to join a group chat is to be invited. To do this, a current member adds them to their own copy of the group and membership list (and often will send an accompanying message). The new member can now see at least the copy of the group information of the person who added them and can create a group with the same UUID (and by default the same member list) on their profile.

Once the new member is on the inviter's membership list, when other users go online they should see the new entry and (by default) add the new member to their copies of the group. The new member will eventually be able to see posts by all other users in the group as each one of them adds the new user. If users wish to avoid this slower process of people being added, they may use the same `group-secret` as other members of the group. This way, a new member could find out the secret from one person and immediately have access to the past messages of anyone else using that symmetric key for their copy of the group.

Note that a member of a group will only see group posts by other people in the group that they follow, because group posts are read directly from profiles. However, this should not be a problem because they know who to follow from the membership list of the person who added them.

The other way to join a group is to know the `group secret` key and add yourself. Since a user would discover that through a child group, we discuss this method for joining in the next section.

6.6.2 Child Groups

Some social media and chat applications, such as Facebook and Slack, allow groups within broader organizations. In Slack, all members of a workspace may preview and join any public channel under that workspace. On Facebook, a broad MIT group contains many subgroups. In each of these, members of the top-level group are able to read messages in lower-level groups without joining them. We allow this multi-level hierarchy in our system with *child groups*.

The `child groups` section of each `group` folder (see Figure 7) is where users can indicate that another group is a child of that one. If they do not include the group key, then this is a "private channel" (to use the Slack parlance), meaningful only to those already in it. If the secret key of the child group is included, then members of the parent group may decrypt the child's info and posts to preview it. If they like what they see, they can add themselves to the group—knowing the member list and UUID from having decrypted it, they can create a group of their own with the same UUID and members, and broadcast a message to the other members that they have joined. The other members must be subscribing to this user to see that message at all, but presumably this is the case since they were already in the parent group together.

6.6.3 Leaving a group

To leave a group, a member may stop checking the group for updates and post a request to other members to remove them from their copies of the group. There is no guarantee to be removed from membership lists, but the user is under no obligation to check other profiles for updates to that group any more and the UI can ignore them.

6.6.4 Removing a group member

A member removed from a group must be unable to read future data published to that group. If Alice wishes to remove Bob, she must do two things. She must change the `group-secret` as in Section 6.3.1, but now also must alert the other members of the group. She sends a message to the group saying she's removing Bob so they too can remove Bob from their copies of the group.

If someone does not perform this process, their messages will still be visible by the partially-removed member. However, the partially-removed member will be unable to see messages from anyone else in the group so we argue this is fine.

We decided against implementing any form of traditional access control for groups, such as having specific admins that may add and remove users. This is because achieving consensus and guaranteed access control in a distributed system of untrusted parties is a hard problem. For groups of well-behaving people, an admin could simply be a user whom everyone has agreed to only and automatically accept membership changes from.

6.7 Multi-Device Support

Users have multiple devices. We want a social network identity accessible through any number of devices a user owns. However, key management is hard. Usernames + passwords are more familiar and understandable to users, making our system more usable as well as allowing a seamless transition from other social media.

When a user first creates a profile, that user selects a username, password, and birth date (MM/DD/YYYY). We use these three to generate a private/public RSA key pair for an IPNS identity. Next, we make that IPNS identity point to this new user's profile. Finally, we store one copy of the master key pair encrypted with the new public IPNS key in the "device keys" section of the profile (see figure 6).

In order to access their identity from a new device, the user may re-enter their username, password, and birth date. This new device can generate the public/private key pair for the IPNS identity described above, see the existing entry, view the existing profile, and read the encrypted master key pair. After this, the user is logged in.

An attacker could find a username, password, and birth date that generate the key pair used for this and decrypt the key stored in the user's profile. So, it is important that this function is one-way. The birth date serves as a "salt" that increases entropy.

More generally, IPNS entries and profile entries similar to the above can be created for multiple "device keys." This method can be extended for e.g. Yubikeys, password managers, and device-specific secret stores or keychains.

7 Lower latency via peer-to-peer communication

When a user connects to the network after the while, they have to scan through all their friends' profiles to see all the messages they have sent. Messages are separated by groups. So, for each friend, you would have to scan through all the shared groups you are in and then download all the messages they have sent since the last recorded message. The runtime of this naive approach would be O(#Friends + #Groups + #Posts). We can improve this runtime by lazy loading. Instead of downloading all the message, you could download the last C messages or only the messages in the last unit of time. If the user wanted to see more, they could choose to load more. This is a very common feature in existing centralized or federated platforms.

They way we described the system involves reading/writing to the profile. While this is needed for robust communication, it is inefficient, because it requires users to constantly poll for changes in their friends' profiles. For real-time messaging, this would cause high latency. Instead, if possible, users may create a peer-to-peer connection with their friends. When a change is made to a user's profile, the user may notify all of their online friends that a change was made. This enables real-time updates with minimal latency.

8 Evaluation

8.1 Security

When designing our system, we wanted to guarantee users would be able to own their data. In a centralized model, users trust a third-party to safely secure their information, granting ownership to the third-party. Here, the user must only ensure the channel between themselves and a social networking service like Facebook is secure(e.g avoiding attacks like MITM and Phishing). From there, the burden of responsibility for the information is on Facebook(which recent events have shown is not the most wise). In IPFS, unlike a centralized regime, the threat model is a bit more intimidating. Users maintain their own profile but now route said profile through a p2p network of nodes, some we can trust and some we cannot. Thus, unlike the centralized regime, adversaries can now read user's information without having to eavesdrop on a network or phish for passwords. In the following sections, we'll go over our threat model and address how our system guards against these threats allowing users to independently maintain their profiles without worry.

8.1.1 Threat Model

When designing our system, we considered a computationally bounded adversary who is interested in the following:

- Reading sensitive profile updates
- Associating someone's profile with their real-life identity
- Tampering with profile content
- Writing fake profile updates

We provide the security guarantee that an adversary is immune to these threats(except the identity association when the adversary intercepts network activity) as long as they use a sufficiently strong password. Our system is vulnerable to an adversary intercepting network activity and correlating activity patterns between nodes and peers. We investigate how to mitigate this in section 9. When discussing identity, we assume adversaries do not correlate network activity. In the following sections, we review how a user finds their rsa-key-pair and how said key pair enables the aforementioned security guarantee.

8.1.2 Multi-device Login

Only the profile owner knows their `rsa-private`. The user retrieves the (`rsa-public` , `rsa-private`) pair by logging into their profile with their username/password/birthday(see 6.7) . Provided users use a strong password, according to **Assumption 1**, no computationally bounded adversary can crack their login.

However, a major limitation of owning your data in a decentralized regime is the lack of a trusted third-party to help recover login information. In the centralized regime, users trust Facebook to maintain their login information and help them recover said information if they lose their password. In a decentralized regime, a user must guard their own login information with no third-party help. Revealing said login is a clear no-no as that would compromise their profile by revealing the `rsa-private`. If a user was to lose their login, they would lose access to their profile. Therefore, we strongly suggest users carefully devise a method to guard their password.

8.1.3 Privacy

Due to our system's implementation of access control, our system provides a good guarantee of confidentiality. All sensitive information is encrypted using one of AES and RSA. No secret keys are made visible and due to our adversary being computationally bounded, we can be assured that we will be secure against attacks. We also hide the groups, subscribers a user is a part of to guard against adversaries pinpointing the identity of a user from the identity of their friends and groups as explained on how we perform metadata-hiding(see **??**) and is secure according to **Assumption 1**.

We do leak some information. First, we don't hide when your profile changes. So if it was someone constantly polling your profile, they could see you changed some field or sent a message but could not see the new value is or learn any more about the message. Of course, if desired, this data leak could be spoofed by just making frequent, arbitrary changes to ones profile so an adversary could learn nothing. We also leak the size of the profile, so adversaries could determine whether a profile is that of a significant figure based on it's size. We believe users, if this is a concern, can simply add noise to their profile to mitigate this.

8.1.4 Spoofing and Tampering on IPFS

Content addressing ensures that adversaries cannot tamper with content and IPNS(see 5.1.4) ensures adversaries cannot spoof profiles. These guarantees ensure adversaries cannot publish fake updates. For example, if an adversary was to alter a post, a subscriber would detect it when comparing the post with the content address from the user's profile received from IPNS. And if an adversary tried publishing a fake profile, IPNS ensures subscribers would detect it, assuming the adversary is computationally bounded and the node key pair is not revealed. Thus, all content subscribers see is authentic.

8.2 Features

One of the goals of the system is to achieve a level of privacy while maintaining the same conveniences of a modern social media/group messaging application. The system as described naively supports simple group chats, but more complicated features can be built on top it including admins, read-receipts, custom reacts etc. It is easy to set state and implement arbitrary functionality because the method of sending messages is so general.

8.2.1 Basic Features

The system can clearly handle the following features, as described below.

- Privacy(e.g Access Control, Metadata-hiding)
- Instant messaging/ Group Chats
- Read Receipts, Reacts, Mentions and other metadata posts
- Replies comments/threads/etc
- Multi device support
- Sign-in with username/password
- Nested groups

However, given how general the core components are, there are some additional features which take only a little more effort to obtain.

8.2.2 Deletion

To delete a message, send a post with the deletion type and the parent being the message you want deleted. Other clients will see your deletion request and can then choose to not display that message. This is functionality similar to many other applications; if you delete a message before anyone sees it, then nobody will ever see it. Unfortunately, it people have seen the message, then you can only politely ask them to honor your deletion request. Since they already have the message downloaded, they could have immediately taken a screenshot and there would be nothing our program to do to counteract it. While certain platforms may brag on their 'self destructing messages', messages that disappear after they are seen once, they don't really do anything more than politely asking. They may, however, enforce that a message may only be downloaded once per user. Such a restriction is impossible in our platform as a user has no reliable way to know when someone else views their profile.

8.2.3 Friend Discovery And Requests

A trade off between user privacy is user discovery. Since we don't have a index of users and users can have arbitrary amounts of information hidden, we cannot do a global look based on common factors like name or location in general.

One method of discovering friends, is offline/through some third party site. As mentioned earlier, you can exchange public keys and then exchange messages easily. Instead of physically meeting up, you could just paste your public key in a different social media site. If neither option is desired, we recommend looking at your friends' friend list. Your friends could have list of people they talk to publicly available in their profile. In such cases, you could browse through the union of these list which you could search by name. Current social media platforms often assume you are more likely to looking to friend someone if you already have mutual friends to this assumption is not unrealistic. Depending on your friends, you could keeping searching in this manner to an arbitrary depth.

However, knowing who you want to friend is only half the battle. No one will receive a message from you unless they know to look at your profile. So you won't be able to message them to see if they want to receive a message from you. One solution is to immediately start the friending handshake with some subset of your friend's friends. If they also want to be communicate with you, they will have done the other half and now a connection has been established.

8.3 Flexibility

We aimed to design a universal social networking platform that supports all the standard social media features without sacrificing privacy. We provide a robust privacy model that gives users full control of who can see their data. We implement an impressive set of features that cover the main cast of affordances expected of social networking services. In additional, our system supports a wider range of features that 3rd parties could add on. For example, third parties can augment posts through `meta.json` to support reacts and implement Slack-like channels with multi-level groups. Lastly, our design is open thereby opening the flood gates for many different front ends which will let users customize their experience. So did we accomplish our goal of creating a universal social networking platform? We hope

so. With more users and third-parties adding custom experiences, we believe it will be interesting to see if our system is as flexible as we think.

8.4 Scaling and Reliability

With our optimizations, our system scales with the number of friends and number of groups. This doesn't scale well to arbitrarily large friends/groups. However, we don't believe this will be a huge hindrance in practice. The average number of Facebook friends is under 400. We also anecdotally observed that number of multi-person groups seems to be smaller than the number of friends. We believe this is due to not being equally likely to talk any subset of friends at once. With these assumptions and lazy loading we arrive at a runtime solely proportional to the number of friends.

With IPFS, we also do not run into problems like in Secure Scuttlebutt. Once a user publishes an updated profile to IPNS and pins it, subscribers can eventually view the content according to **Assumptions 2 and 3**.

9 Future Work

We are extremely pleased with our system currently, but we believe there is room for improvement. We would like to delve deeper in the following areas:

9.1 Group Moderators

We would like to investigate group roles. We believe a good example is that of Reddit. Think of Reddit moderators and Reddit Owners. These are two different classes of users, where moderators are delegated their roles by the owner of the subreddit. Above the Reddit Owner is the Reddit Admin, who has the ability to govern all activity on the site. We believe exploring an abstraction to flexibly allow for different privileges among group members can lead to an even easier experience for developers to create their desired applications. A promising approach would be to have the owner of the group serve a role similar to that of the Reddit Admin and delegate roles to others, with the option of giving them the power to delegate roles as well. This creates a flexible hierarchy of roles and privileges that we think can be adapted to any group.

9.2 Trusted Timestamps

Currently, when users send each other messages, there is no central authority that can be used to time stamp messages. We would like to consider a design that makes it easy for developers to allow for a trusted timestamps. This is enabled by some trusted third-party, a Trusted Timestamp Authority that signs a hash of the message appended with the timestamp.

9.3 Network Anonymity

ISPs have the ability to use network activity to discover a user's friends, whether through IPFS or direct peer-to-peer. For some users who desire stronger anonymity, this would be undesirable. We would like to investigate the effects of incorporating optional onion encryption(e.g Tor) in our system to mask a user's network activity.

```
Put(data) -> key:  Clients execute the PUT protocol to store data
under a unique identifier key.
Get(key) -> data:  Clients execute the Get protocol to retrieve
data that is currently stored using key.
```

Figure 9: Example API for FileCoin[6]

9.4 Generalizing to Social Media

We designed our system with social networking services[insert reference] in mind. However, we believe our platform could also support the gamut of social media as well. In addition to social networks, we can support personal blogs trivially but would also like to be able to support social media like large forums(e.g Reddit), social gaming(e.g Twitch), video sharing(e.g Youtube), collaborative editing(e.g Wikipedia). We believe these applications provide the challenge of aggregating group state where the size of the network is unbounded(i.e scaling becomes an issue since personal social networks are relatively small). In the medium case(thousands of people), we believe group aggregation can be done by simply designating users who offer to retrieve and aggregate data, perhaps as a group role. In the larger case(tens of thousands and up), we believe we need a better scaling solution.

9.5 Scaling to Billions of Users

The weak incentives provided by IPFS for storage and retrieval of data does not provide a convincing guarantee of performance. Especially in regards to large groups. Luckily, there are solutions[5,6] that use the blockchain and cryptocurrency to provide a stronger incentive of storage and retrieval of data with a similar API to that of IPFS's DSHT. In this regime, we believe we can reliably provide a decentralized solution, putting to rest **Assumption 2**, and provide performant mutable state updates, dealing with **Assumption 3**. Users will be able to instantly find groups of interest(of any size) their friends are in, see viewable members, and browse content of any form(aggregated or not), allowing third parties to create a wide variety of social media experiences.

10 Conclusion

Frustrated by the monopoly Facebook has, its practices regarding user privacy, and the impunity it has enjoyed from transgressions like the Cambridge Analytica scandal, we felt driven to seek an alternative that could truly replace it and provide an even better experience. After surveying the landscape and seeing the existing alternatives were underwhelming, we were driven and motivated to design our alternative using the security principles we've learned as well as innovative technology in the p2p space.

We are proud to have designed a system that does exactly that. Our system delivers on design goals that provide a system that has superior reliability and data ownership compared to social networking services like Facebook as well as the alluring promise of a universal profile that can be used for any social networking application. We provide strong security guarantees and a platform that is flexible and scaleable to a user's social networking needs.

In the future, we believe our system could motivate work to support greater experiences. Currently, our system makes a strong assumption on the performance of IPNS and the stor-

age capabilities of IPFS. We believe on the scale of friend networks, we believe this will not be a problem as nodes will deliver real-time updates through peers and occasionally provide offline-updates through the less performant IPNS and IPFS. However, to cater to greater social media experiences, we believe further innovation in the p2p space such as Filecoin will incentivize faster retrieval and better storage that make decentralized applications more attractive and hopefully make centralized monopolies like Facebook a relic of the past.

11 Acknowledgments

We would like to thank our instructors, Prof. Ronald Rivest and Prof. Yael Kalai for teaching such an enriching course and providing the foundation and guidance for our project. We also would like to thank the 6.857 TA's for their useful advice and feedback.

References

[1] *Mastodon* `https://docs.joinmastodon.org/usage/decentralization/`

[2] *Secure Scuttlebutt* `https://ssbc.github.io/scuttlebutt-protocol-guide/`

[3] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System* `https://ipfs.io/`

[4] *Mike Thelwall. Social Network Sites: Users and Uses* `https://doi.org/10.1016/S0065-2458(09)01002-X.`

[5] David Vorick, Luke Champine *Sia: Simple Decentralized Storage* `https://https://sia.tech/sia.pdf`

[6] Protocol Labs. *FileCoin: A Decentralized Storage Network* `https://filecoin.io/filecoin.pdf`

12 Appendix

12.1 Attempted Prototype

We attempted to prototype a working model of our system for the class but found there were many nuances(and lack of documentation) working with IPFS to get it in a good enough condition. For those interested, our code base can be found at `https://github.com/npfoss/gravity`.