ÐSignal: A decentralized privacy-enhanced Signal protocol

Dewmini Marakkalage, Andrea Piccione, Sergio Roldán

February 15, 2020

1 Topic Changes

The idea of building a decentralized, secure and private chat application is mostly unmodified but we changed the scope of the project according to the feedback received and the fact that the team has found another member. The new idea is building a decentralized messaging protocol based on the well-known Signal Protocol (https://en.wikipedia.org/wiki/Signal_Protocol), which is widely adopted by the major messaging applications. We would like to produce a simpler yet functional version of the Signal protocol to make it work in the current gossip network which was built throughout the semester. In order to achieve this goal, we plan to build a decentralized storage that can be used to replace the server needed by the protocol (a central authority that stores pre-keys used by the Double Ratchet Algorithm). Finally, in order to complement the project and further improve the protocol from a privacy and anonymity point of view, we want to implement a simple version of Whisper, a flexible and secure decentralized gossip protocol that aims to protect user metadata and preserve sender and receiver anonymity.

2 Introduction

The advent of chat applications has created a huge change in our daily life and nowadays these apps are widely spread and used. However, we live in a world where data is the new source of revenue of big companies, digital surveillance is widespread and recent scandals (e.g. Cambridge Analytica) clearly show that invasions of privacy are real. A chat application should not require personal data in exchange for functionalities. The data requested should solely be used locally on the personal device and the owner decides how to preserve, share and manage their own information. So, why use applications that can potentially "steal" your data and allow companies to watch your private conversations? At the current day, the number of "zero-data" chat apps is very small and their usage is very limited. Our goal is to build a fully-functional decentralized chat protocol with a simple design and idea: making messaging secure, private and easy-to-use, with no central authority or server required.

3 Related Work

In this section, we briefly discuss some prior work related to each component of our project.

3.1 Signal Protocol

As confidentiality section is directly derived from the Signal protocol our purpose is to modify the algorithms proposed to be able to work under a decentralized infrastructure. If we would want to find a vulnerable point to Signal we would probably aim to the server in charge of store and deliver both the public keys (global and ephemeral) and messages in offline mode. Our aim is to use a decentralized structure to avoid the need of one central authority that could be untrustworthy or more easily attacked.

Moving Signal to a serverless structure present mainly two problems related with the server used to store information to be used offline. The first point is message delivery:

- Given liveness of both parties involved in the message exchange, we could use a straightforward approach to assure delivery just using the Peerster private message structure with confirmations. If one node dies or gets unreachable we need a mechanism that store messages for the moment that the receiving party becomes available again. For the confidentiality block we assume an underlying decentralized storage and we will focus on defining the algorithm such as each time a new message is sent it will be a direct message. If the sender doesn't receive a confirmation in Tout the sender then changes the mode to store the messages in the decentralized storage, until it receives any message coming from the receiver, this would assure liveness again and the messages could continue in private mode. Using private direct messages is more efficient than store them for space and processing reasons as well as to assure delivery. Also because each confirmation is a message itself we assure the refresh of DH Ratchet enhancing forward security.

The second point is the key delivery:

- In this case we also assume an underlying decentralized storage that keeps the general public keys and a set of X ephemeral public keys. As we lack a central authority we need some way to assure validity of the public keys so an attacker can not impersonate a certain node, to do that we could use a truncation of the hash general public key as unique identifier in the network (similarly to some Blockchains) to mask the public key and get a smaller and usable version. With that we only believe keys signed by the owner of the unique identifier so the owner of the real general public key. Then if we want to send a message to an unknown node (a node which we don't know yet the public keys) we just request in the storage its keys and check if its UID matches, then we could derive a shared key using 3XDH and use it to initialize the Double Ratchet.

3.2 Whisper Protocol

The Whisper protocol is built on top of the Ethereum's ÐΞVp2p wire protocol suite (https://github.com/ethereum/devp2p) to gossip messages around a distributed network and combines aspects of both DHTs and datagram messaging systems: it is basically a cryptographic peer-to-peer network and protocol suite which provides a general-purpose transport and interface for applications to communicate via a P2P network. Since the Ethereum architecture is incompatible with the current codebase, we need to adapt our Peerster to reproduce the same abstractions needed to build the underlying Ethereum's p2p protocol. We plan to follow the original protocol specifications as much as possible and do some changes when required. We will try to build even the lower-level protocol features directly on top of Peerster in order to guarantee the best compatibility and performance. For example, it is likely we're gonna need a DHT or IPFS-like implementation in order to store per-entry configurable time-to-live and conventions for the signing and encryption of values.

Another interesting and quite popular project that uses the Whisper protocol is Status, a decentralized messaging app (https://status.im/). This app, among the other functionalities, provides a P2P messenger which is built on top of the Whisper protocol. Although the protocol is a modified version of the original one, the whole messaging stack of the application is similar to what we have in mind: building the Signal protocol on top of the Whisper protocol in order to guarantee both security (confidentiality, Perfect Forward Secrecy) and privacy guarantees (darkness and anonymity). The app seems to have successfully implemented the two protocols and make them work together in a reasonable way, so we're confident we can do the same by following some of their guidelines outlined in their documentation and do the right adjustments according to our case scenario which will be using a decentralized storage and the Peerster gossip protocol underneath. Status uses different approaches, some more secure and others less, to overcome the need of using a centralized authority to store bundle of pre-keys, which should be retrieved by whoever wants to start communicating with another peer in a secure way. According to our judgment, the most appropriate way is to use the decentralized permanent storage (which will be built in parallel during the development) but we don't exclude the possibility to even use the Whisper protocol directly to share this information because both should be suitable for the task according to the Status documentation.

Of course, our solution will take inspiration from the official source and the status protocol but at the same time it will be completely different from these references because it will be built on a different codebase. However, we believe that implementing the Whisper gossip protocol and the underlying abstractions on the current Peerster should be a challenging yet doable task.

3.3 Decentralized Storage

There are many works on different kinds of decentralized storage systems, most of which are based on DHTs. BitTorrent, KAD file network, PAST, and the recently popularized InterPlanetary File System (IPFS)[3] are few examples for decentralized storage systems that benefit from DHTs.

PAST [1] is introduced by Druschel and Rowstron in 2001. It replicates each file in multiple nodes which are, with high probability, distributed uniformly throughout the network. This is achieved by computing a cryptographic hash of each file and storing them in k nodes whose node identifiers are close to the computed hash. To retrieve files when requested by a client, PAST relies on the routing scheme Pastry [2], which ensures that the total number of PAST nodes through which a request is routed is at most logarithmic (with high probability) in the total number of PAST nodes in the network. Additionally, PAST also uses a smart card system provides various security guarantees.

IPFS [3] is launched in 2015 by ProtocolLabs Inc. and has features like the ability to have versioned file system. This is facilitated by modeling all stored data as Merkel DAGs. It additionally uses a credit system called BitSwap credits to incentivize nodes for offering storage. Another interesting feature of IPFS is that it allows several different cryptographic hash functions to be used for hashing file chunks. The identifier for a chunk there for contains the type of the hash function, the length of the hash, and the actual hash. IPFS stores chunks based on Kademila [4] and uses Distributed Sloppy Hash Tables (DSHTs) for routing; A DSHT is a variation of DHTs and stores only a subset of values associated with a key. This is efficient, for example, when replicas of a file are stored in multiple nodes, but a receiving node only needs to know the address of one of those nodes.

4 System Goals & Functionalities

In this section, we discuss goals and functionalities of each component of our project.

4.1 Adding confidentiality through Signal

As part of our secure and private message service we need that message exchanges or chats are kept confidential. This concept of confidentiality or privacy of messages is even stronger when working on a decentralized system where any node could be able to eavesdrop a conversation because we need of untrusted and unknown peers to forward the information. In order to solve this problem, we will implement the algorithm proposed in Signal called Double Ratchet to assure confidentiality between two peers, and because Double Ratchet needs of a shared common key we will also implement the X3DH proposed in Signal which achieve our requirement.

X3DH or Extended Triple Diffie-Hellman is a key agreement protocol which establishes a shared secret key between two parties who mutually authenticate each other based on public keys. As an extension of the Diffie-Hellman protocol it provides forward secrecy, and also it provides cryptographic deniability. Double Ratchet extends this properties with future secrecy.

Double Ratchet works by the parties deriving new keys for every Double Ratchet message. Double Ratchet is the combination of symmetric-ratchet and Diffie-Hellman ratchet using KDF chains to generate the key for each round of each of the Ratchets. While symmetric ratchet is used to encrypt and decrypt sent and received messages respectively, Diffie-Hellman ratchet is used to update the symmetric-ratchet when new public keys are received. With this we know that the parties also send Diffie-Hellman public values attached to their messages. The results of Diffie-Hellman calculations are mixed into the derived keys so that later keys cannot be calculated from earlier ones.

The differential property of this protocol is that is designed for asynchronous settings where one user is offline but has published some information to a server (in our case in a node or set of nodes of the network because nodes should help the decentralized system for being a part of it). Another user wants to use that information to send encrypted data to the first one, and also establish a shared secret key for future communication.

The main goal of this section is implementing key exchange and message encryption/decryption using both the algorithms mentioned above.

As a last remark we will used as cryptographic algorithms and cryptographic primitives like hash functions recommended in the Signal documentation with some changes to keep consistency between the different blocks (Status recommendations):

- The elliptic curve Curve25519 with X point compression
- SHA-256 from the SHA-2 family for the KDF chains, X3DH
- Encryption based on AEAD using AES-256 in **GCM** mode plus HMAC
- The info parameter for X3DH will be set for now to “DSE DAS”

4.2 A Secure Distributed Storage

A key component of our overall system is a decentralized storage service. There are many such existing services built for different purposes, most of which are based on Distributed Hash Tables (DHTs) or their variants. The basic idea of a decentralized storage service is to split files into multiple chunks and store each chunk in several nodes across the network. Usually, a cryptographic hash of a chunk is taken to

be the identifier of the chunk, which allows the identifier itself to be used for checking the integrity of the file chunk. Having replicas of the same chunk stored in multiple nodes improves reliability as a file chunk would still be available even if all but one of the nodes who possess it leave the network. For this project, we implement a simple decentralized storage system which allows both public and private file sharing. Our implementation is loosely based on the InterPlanetary File System.

For this project, we implement a secure persistent distributed storage system that is loosely based on IPFS. For our application, we do not need all the features offered by IPFS.

We expect our storage system to meet the following requirements:

1. Ability reliably share public files.
2. Ability to securely and reliably store/share private files so that only the owner/intended recipient can view the content of the files.
3. Ability to verify the owner of the files.

How to achieve security:
To share a public file, we first compute cryptographic hash of it and then encrypt the hash with owner's private key to make a fingerprint. Then we create an augmented file with the original file content and the fingerprint then share the augmented file.

To share a private file, we additionally use GPG encryption. That is, we first generate random encryption key, and then encrypt the file with that key. Then we encrypt the key itself with the recipient's public key so that it can only be decrypted by the recipient (who holds the corresponding private key). Then we share the encrypted file and encrypted key together as a single file, treating it as just another shared public file. Although this makes it available to everybody in the network, only the recipient can decrypt it.

How to store:
Our file storage service builds on the simple file chunk storage model of Homework 2. Basically, for each file, we first split them into chunks and store each chunk separately, where we create a meta file consisting of hashes of individual files. However, to allow larger files, we allow the meta file to be chunked as well. We also allow the owners to choose the kind of SHA-2 hash (SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256). We prepend each hash with two additional bytes. The first one is to identify whether a hash corresponds to a meta file or a chunk of basic file, and the second byte is to identify type of SHA-2 hash used. This scheme is a simplified version of the one used in IPFS for its hashes.

Where to store and retrieve from:

We store each chunk in nodes whose identifiers are close to the hash of the chunk, according to some metric as suggested in Kademila [4] (which is also used in IPFS and PAST). For this, we assume that the node identifiers are (almost) uniformly random, and to ensure that, we take a cryptographic hash of a node's public key to be its identifier. Then, for example, we store a file chunk in nodes whose identifiers first bits are equal to those of the chunk hash. We use the same method proposed in [4] to find the content given their hash values.

4.3 Adding privacy and anonymity through the Whisper protocol

Most of the known protocols used by messaging apps usually provide some sort of encryption: for instance, the Signal protocol or its variations are implemented by the most known IM apps like Whatsapp, Facebook Messenger or Skype. However, security doesn't guarantee any form of privacy from whoever might have access to the metadata used to exchange messages. By imaging a possible threat model, we can clearly see that attackers (either a malicious MITM or the company that has access to the metadata) are able to compromise the privacy of messages and to get to know whom one communicated, when, and how often. Simple network packet-sniffing, logs analysis or more sophisticated statistical attacks on timing and bandwidth can all reveal a significant amount of information that can affect the anonymity of communication.

In order to guarantee robustness from information leakage of metadata, we plan to implement a simplified yet fully-functional version of Whisper, a peer-to-peer, multicast, end-to-end encrypted, "dark" protocol. We believe that the existent gossip protocol built throughout the semester can be adapted to implement Whisper because there are some common traits with the underlying DEV-P2P protocol used by the original Whisper implementation.

The design goals of the Whisper protocol we have in mind are:

- Simple and relatively efficient multicasting and broadcasting through an intuitive API.
- Good level of plausible deniability and darkness that can be configured by several parameters.
- Low traffic latency and bandwidth, according to the level of privacy chosen to guarantee.

The basic protocol works by broadcasting each message (which is wrapped in an "envelope" that includes other fields) to the whole network, even if the packet is intended to himself. Messages are encrypted (the protocol uses its own public-private or symmetric key pairs to encrypt messages) with the intended recipient's public/symmetric key, so that the real recipient can correctly identify packets that were sent to himself. By following the Status documentation guidelines (https://status.im/research/pfs.html), we

plan to use AES-GCM for symmetric encryption and ECIES with SECP-256k1 for asymmetric encryption.

An envelope is the packet that is handled by the protocol and contains fields that are required by nodes to handle the packet correctly. Since broadcasting all the messages is very expensive, not only because it affects latency and performance but also because it can be easily attacked by DDoS on the network, an envelope contains a Proof-of-Work parameter that indicates the amount of work that should be done to send a message and prevent spamming. A TTL field is also used to prevent packets to stay in the network forever (as an example, Peerster already implements a similar mechanism by using the Hop-limit field). Moreover, packets cannot be tracked and do not leak any information because everyone broadcasts even messages intended for them until the TTL expires, therefore it is very difficult to identify unique senders and receivers (plausible-deniability). It's already clear that there's an evident trade-off between privacy and efficiency: if all nodes broadcast all messages they receive, full darkness is easily achieved but, on the other side, there's a huge cost of bandwidth and computational cost. Therefore, full darkness and plausible deniability comes at the expense of high-latency and relatively high bandwidth. This tradeoff will be taken into account during the development in order to guarantee a normal usage in stable conditions with a reasonable wait-time before effectively delivering a message and make the whole protocol scalable to larger networks. Moreover, the idea is to make this tradeoff configurable as in the original protocol, so that users of the protocol can decide how much information they want to reveal at the expense of efficiency.

A mechanism to "relax" the need of broadcasting all messages is the Topic field in the envelope structure, which is used as an indication of whether a message is "of interest" to a node. Thanks to a bloom filter structure (computed by every node) which takes into account all topics that the client declared to be interested to, it is possible to give just a probabilistic indication of whether I should be interested in attempting to decrypt a message. In fact, for the darkness principle, nodes should not know whether they are one of the recipients of a message until they have tried decrypting it using the keys at their disposal. The last version of the protocol (version 6) uses packets to advertise peers of their requirements in terms of bloom filter and PoW requirement: this way, nodes should forward envelopes to a peer that only meets the requirements received, otherwise the peer is disconnected.

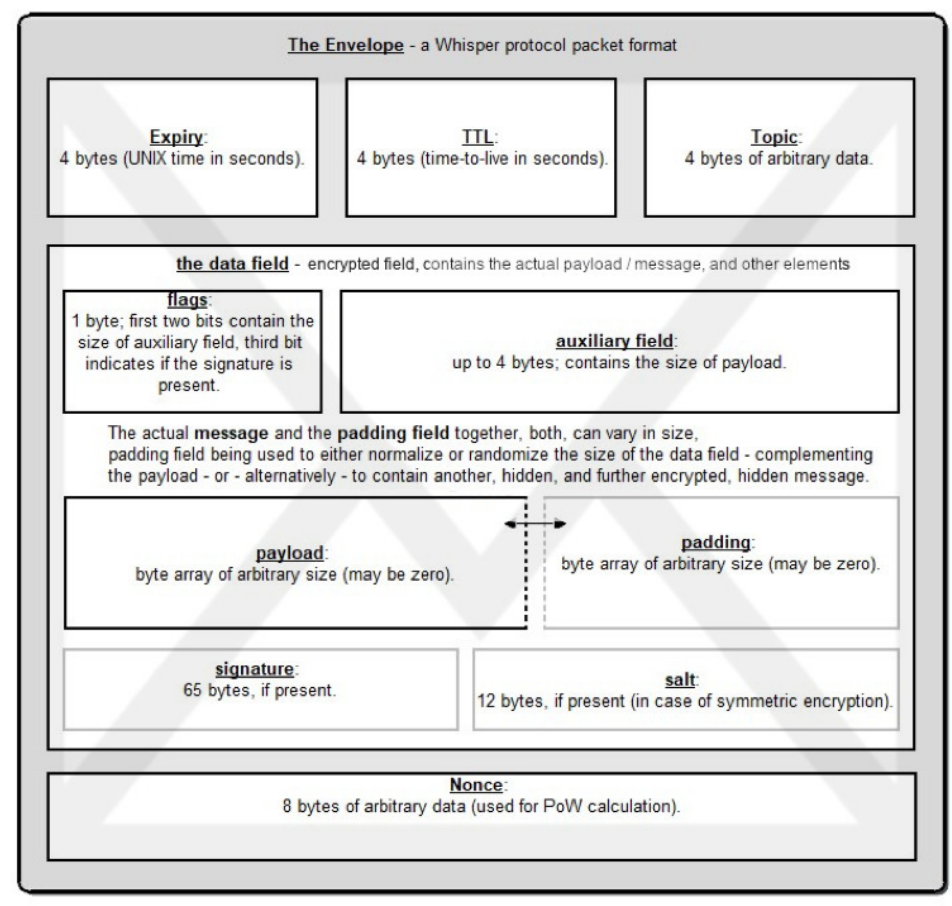

Figure 1: Components of a Whisper protocol envelope. https://status.im/research/extended_features.html

Even with these adjustments, the protocol is still quite expensive and there are other several drawbacks (e.g. network partitioning due to topic distribution in case of forwarding envelopes that might be interesting to my neighbours only). Our idea is to implement the core parts of the protocol and then improve it along the way in order to make it suitable to our use case. This way, we should be able to ensure a good level of anonymity and make the protocol usable for the final chat application we have in mind.

5 Architecture Design

The Figure 2 shows the simple sketch of the proposed architecture of the project.

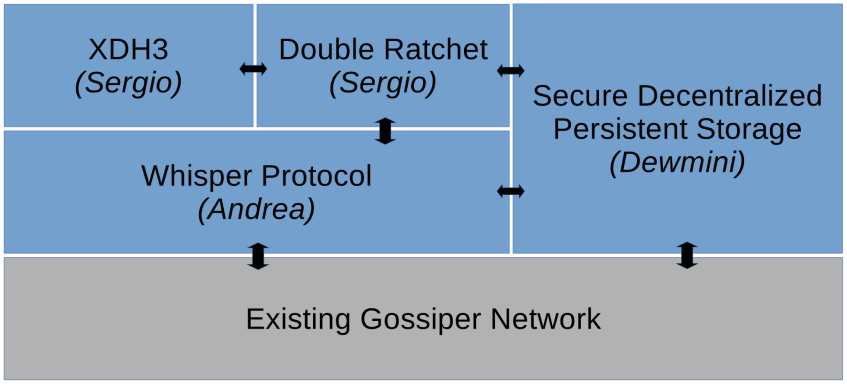

Figure 2: Sketch of the possible architecture design of the project

As outlined in the previous sections, we plan to build both Whisper and the Decentralized Secure Storage on top of the current Gossiper architecture provided by Peerster. The Signal protocol (X3DH and Double Ratchet) will use Whisper to guarantee a secure and anonymous communication, without leaking meta-information. The Decentralized storage will play a key role for making the Signal protocol fully decentralized by giving the possibility to store messages and keys in the network so that they can be easily retrieved and we can efficiently make up for the absence of a central server authority. By using these state-of-the-art protocols, we guarantee a fully private and secure application that works in a decentralized environment.

6 Implementation

6.1 Secure Distributed Persistent Storage Implementation

We first describe the implementation of Distributed Persistent Storage and then show how to add security on top of that.

6.1.1 Distributed Persistent Storage

We implement Distributed Persistent Storage, which we call *DStore*, as a separate module that can easily integrated to any Peerster implementation. Our implementation is based on Kademlia [4], and uses separate DStore node identifiers which are pseudorandomly generated by taking the SHA256 hash of a node's (IP address, port) pair, and trimming it to be 32 bits.

Spliting files into chunks. When storing a file, DStore first splits it into four kilobytes chunks, compute their hashes using one of the hash functions in SHA-2 family according to the user preference, and stores the chunk in k different DStore nodes in the network. Then, the hashes of the chunks are concatenated to create a metafile. As long as the metafile is larger than four kilobytes, we continue splitting it into chunks, computing their hashes, storing them in k different nodes, and concatenating the hashes to

make a new meta file. Finally, when we have metafile of size at most four kilobytes, we compute its hash and store the metafile in k different nodes. The hash of the metafile is used as the key for the original file.

Given such a key, to reconstruct the original file, DStore first needs to retrieve the metafile. Then, it uses its content to retrieve other associated chunks. It continues doing so until it retrieves the original file chunks and concatenate those chunks to recreate the original file. To facilitated this, DStore needs two additional information regarding each hash value. First it needs to know which hash function is used to generate the hash, and secondly, it needs to know whether hash corresponds to a metafile chunk or a chunk of an original file. Thus, we prepend each hash value with two bytes indicating the type of the chunk and the type of the hash function.

The k nodes a chunk is stored are the k nodes whose DStore node identifiers are closest to the first 32 bits of the chunk hash. The closeness is measured using the XOR distance. The XOR distance of two 32-bit strings is the base-2 integer value of the bitwise XOR of the two bit strings.

Routing table. To efficiently find the closest node identifiers to a given 32-bit string, each DStore node maintains a routing table, which is independent of that of the Peerster node. This routing table has a bucket for each prefix of its DStore identifier, and are numbered from 0 to 31. Bucket i stores the addresses of at most k other nodes whose node identifiers are equal to the local DStore node identifier in first i bits and are different in the $(i+1)$-th bit. A bucket is implemented as a doubly linked list where the node in the head position is the least recently seen node.

Updating the routing table. The routing table is updated every time the DStore receives a message from another DStore node. If the sender is already in the routing table, we find the corresponding bucket and move the sender node to the end of the linked list. Otherwise, if the bucket has less than k nodes, we add the sender node to the end of the linked list. However, if the bucket already has k nodes and the sender is not there, then we first check whether the least recently seen node in the bucket (i.e., the head node) is still online by pinging it. If it is online, we ignore the sender. Otherwise, we remove the current head node, update the head pointer, and add the sender to the end of the list.

Remote procedure calls. A DStore node supports four remote remote procedure calls (RPCs).

- **Ping** checks whether a remote node is online.
- **FindNode(id)** asks for the k closest nodes to *id* in a remote node's routing table.
- **FindValue(key)** asks for the value associated with *key* in a remote node's local storage. If it is not present, the remote node should reply as in FindNode, giving the k closest nodes to *key* according to its routing table.

- **StoreKeyValue(key, val)** ask a remote node to store (key, val) pair in its local storage.

Finding the k closest nodes. A node has to find the k closest nodes to a given bit string in several situations. For example, to store a chunk or to retrieve a chunk, a node has to find the k closest node to the chunk hash. This is done by first creating short list of k closest nodes to a given bit string using the node's local routing table, and then iteratively refining the short-list by asking the nodes that are currently in the short list about their knowledge of the k closest nodes to that bit string.

Storing a file chunk. To store a file chunk in the distributed storage, a node finds the k closest nodes to the hash of the file chunk, and sends them *StoreKeyValue(key, val)* RPC requests. Afterwards, each node that stores a given (key, val) pair periodically *republish* them. That is, they periodically finds the k closet nodes to key and ask them to store the (key, val) pair. If the network is stable with no nodes leaving or entering the system, republishing has no effect. However, if a node leaves a joins the system, republishing ensures that each (key, val) pair will eventually stored in the k closest nodes to key. This guarantees persistence as long as no more than $k - 1$ nodes simultaneously fail.

6.1.2 Providing security

DStore provides security on top of the distributed storage using standard encryption techniques. For this, we assume that each node independently generate private and public key pairs, share the public key with the others, and keeps the private key secure.

Verifying the authenticity. To support the authenticity of the files, they are signed before sharing using the owner's private key, so that the other's can verify it with the owner's public key. To sign a file, the owner first computes a secure hash of the file content, and encrypts the hash with RSA encryption with Optimal Asymmetric Encryption Padding (OAEP).

Providing secrecy. To provide secrecy, a signed file is encrypted using AES encryption in Cipher Block Chaining (CBC) mode using a secure random initialization vector, and secure random symmetric key. The key is then encrypted using the recipient's public key using RSA encryption together with OAEP. Finally, both the encrypted file and the encrypted key is stored in DStore and its DStore key can be shared with the recipient over a non secure channel. The recipient node, after retrieving the file from DStore, uses its private key to decrypt the encrypted symmetric key, and then use that symmetric key to decrypt the encrypted file data. Once decrypted, the recipient can verify the authenticity by verifying the owner's signature as before.

6.2 Signal protocol implementation

The Signal protocol has been build as a library initially on top of the entire Gossip protocol of Homework 1 and lately integrated over the Whisper protocol following the specifications in the official Sig-

nal documentation for X3DH and Double Ratchet (https://signal.org/docs/specifications/x3dh/ and https://signal.org/docs/specifications/doubleratchet/ respectively) with very few changes. Finally it was integrated with the Decentralized Storage to allow identity sharing in a non-interactive way

We decided to change the mode of operation of AES from the initially proposed GCM to CBC with HMAC for AEAD. The change was made due to an initially unseen recommendation in Signal documentation to use this second mode. We implemented CBC with HMAC according to the IETF proposal "Authenticated Encryption with AES-CBC and HMAC-SHA" [11]. Is worth to note that changing the mode to GCM would be quite straightforward, in case we want reduce the computational overhead.

The implementation works in three main steps: The first one is produced when the Peerster is initialized, at this moment the implementation tries to retrieve a previously saved identity from the "_SignalFolder". This identity contains the data necessary to start the X3DH protocol including the public-private key pair: identity key IK, signed prekey SPK with its signature, and an array of one-time prekeys OKs. In case that we are unable to retrieve the identity a new one will be automatically generated and stored in the previously mentioned folder. If we were able to retrieve the identity we will try to retrieve any previously stored Ratchet State from the same folder. Those States contain the information necessary to continue with the previous Double Ratchet without the necessity to make a new X3DH agreement. Finally the identity is uploaded to the Decentralized Storage.

The second step is produced when any Peer wants to communicate with another (here we suppose that the Peerster has already build the routing tables to communicate). In case that the sending Peer (lets call it Bob) does not have a Ratchet State with the receiving Peer (lets call it Alice), Bob will try to retrieve Alice's identity from the Decentralized Storage. This is the case for the non-interactive X3DH where Bob can use the identity to initialize his Double Ratchet, first making a Diffie-Hellman Ratchet step and then doing a Symmetric Ratchet step to obtain the key to encrypt its message, and will send Alice a X3DHMessage including the RatchetMessage with its plaintext encrypted. In case Bob is unable to retrieve the identity he will try to start an interactive X3DH session with Alice sending his own identity though a X3DHIdentity message. Upon receiving a X3DHIdentity Alice will start her Double Ratchet following similar steps to the ones done by Bob in the non-interactive version, and she will compose a X3DHMessage for Bob including her identity and a Ratchet message that will allow Bob to initialize his Double Ratchet and send the intended message. The mentioned RatchetMessage includes as additional data both parties identity keys for authentication purposes and some irrelevant message (in our case the id of the receiver ":" the id of the sender).

The third step is produced once the Double Ratchet has been initialized and focuses on the sending and reception of messages doing Diffie-Hellman Ratchet steps when new public keys are received using a new

freshly generated DH key pair to encrypt new message and the last key pair to decipher old messages, defining a new pseudo-session for the new message. Upon each sending we make a Symmetric Ratchet step to encrypt the message, and upon each receiving we make the same to decipher.

X3DHMessages used to agree a mutual root key for the Double Ratchet already includes a RatchetMessage that is used in the receiving end to check that all the computation where correctly made, that the information received is correct and that both Double Ratchet states are synchronized to send and receive new messages. Every RatchetMessage includes the ciphertext, the additional data and a header including a public key that can be new or already used, depending if we are sending more than one message per pseudo-session, the length of the current chain (aka pseudo-session) and the message number apart from the origin and destination. The public key is used to make Diffie-Hellman Ratchet steps as already mentioned and we use the updated Symmetric Ratchet to decipher any number received messages which chain is the same, which means that public key is shared between those messages.

Identities and Ratchet States are represented using Elliptic Points for the Public Key and byte arrays for the private key that should be transformed to bigInts to compute the Diffie-Hellman calculations. In order to be able to store them in Decentralized Storage and in "_SignalFolder" we need to compress them, we use x-coordinate compression (as described by NIST) of the Elliptic Points and convert them to byte arrays, this also reduce the overhead while sending. The same applies for EDCSA signatures that are represented as bigInts and sent/stored as byte arrays.

In terms of throughput we won't take in consideration X3DHIdentity or even X3DHMessages because we only will have one per conversation required for initialization. In case of RatchetMessages we have quite an overhead because we need to send, apart from the original origin, destination and message, a 33 bytes array representing the public key in addition to two int64 for the message id and chain length in the header. In the message we have the cipher version of the message padded to 16 bytes, which should not increase much the total size, plus a tag as a 16 bytes truncated version of the 256 bits HMAC, calculated with the header plus other additional data if required. Final throughput is determined by the overhead and propagation through Whisper and the Gossiper network.

The implementation uses only standard Go libraries mainly of crypto package including SHA-256 for hashing, AES for AEAD encryption/decryption, Elliptic for all operation related to public-key cryptography, ECDSA for the signatures, HMAC and HKDF for KDF.

6.3 Whisper protocol implementation

The Whisper protocol has been built as a library on top of the entire Gossip protocol of Homework 1 and follows the specifications given in the official Ethereum documentation (github.com/ethereum/wiki/wiki/Whisper-

pages) and in the status documentation (status.im/research/) with very few changes.

The client has the ability to manage its own identity with the creation, addition and deletion of cryptographic keys (both symmetric and private/public keys) through a clear set of public functions. Keys are safely stored in memory and can be used at any time for the main functions offered by the protocol.

The client can use the main API method to send a new message in the network by specifying a Topic, a required time of Proof-of-Work (PoW) for sending the message, the Time-To-Live of the envelope, the key to be used for the encryption (either a symmetric or public key) and the payload. The message is then prepared (with the addition of random padding to the payload in order to hide the length of the message), encrypted with the specified key if requested and finally wrapped in an envelope with the specified parameters (the nonce field for the PoW is computed from the required time of work parameter specified).

A client message is treated in the same way as any other incoming message. Each envelope is handled by checking for its validity: if the topic is interesting for the node, if its PoW meets the node's requirement, if the envelope is not already expired according to its TTL, if the message's size is not above the limit imposed by the client. An invalid envelope causes the blacklisting of the peer who forwarded the message (i.e. all its future messages will not be considered anymore by the node). If the envelope passes all the checks, it is stored until its TTL expires. Whisper periodically prunes all the expired envelopes and periodically broadcasts all messages still in cache. The protocol also periodically creates a random packet (in the same way as specified above but all the parameters are randomly generated) and sends it to the network in order to not reveal information about its identity and its specific interests in the case it sends similar messages often.

In order to forward an envelope, a node uses a routing table which is updated every time it receives a status packet. A status packet is a simple clear-text packet which contains the PoW and Bloom filter of a node. Nodes use the underlying rumor-mongering and anti-entropy features to share status packets with everyone in the network. They notify nodes about Bloom filter and Proof-of-Work requirements at their start, upon any change and periodically. The use of the Gossip protocol is suitable for the task because updates must be ordered and nodes can keep track of the latest updates with the ID field of a Rumor message, otherwise they could mistakenly forward a packet when they should not and be blacklisted. Since the status packet propagation can take some time in larger networks, there's a configurable tolerated time so that if a node sets a new value of PoW and/or change its Bloom Filter, the old value is still accepted for a limited amount of time and the peer not blacklisted.

The routing table works in a similar fashion as the routing protocol implemented in Homework 2: when a node receives any status packet from peer 1, the node will never forward to peer 1 envelopes which don't

meet the aggregated requirements received. So, every time a node receives a status packet, it computes an aggregated Bloom filter between the one stored in the table (which takes into account all previous status information received by any nodes from peer 1) and the new status just received; a similar approach is used for the PoW but we just keep track of the minimal value received. Status packets can either contain both PoW, Bloom Filter or only one of them: the distinction is provided with specific packet codes. This way, if the network has operated for some time and there are a sufficient number of topics specified, each peer will eventually receive semi-interesting information most of the time: the forwarding of packets will be relatively efficient but still private because it only reveals some probabilistic indication of node's interests. As a good metaphor, it can be seen like forming a probabilistic topic-reception vortex around nodes, a sort of "topic-space" gravity for interesting messages.

The client can install a new message filter in order to be notified upon incoming messages that match the specified filter. For adding a filter, it is necessary to add a minimum PoW parameter, a set of topics to be watched and optionally the key to be used for the decryption (either a symmetric or private key). Every time a new envelope is received, all filters that (1) match the topic of the envelope and (2) are able to decrypt the message with the specified key will be notified. The filter will keep every decrypted message in memory until it will be asked to be retrieved by the client. The client can then ask at any time for all the messages received by a specific filter (through an id received when creating the filter) from the last time it asked.

The client can also remove filters, manage keys, configure the main parameters dynamically (PoW and Bloom Filter). All other parameters (timers and constants) can instead be easily modified in the config.go file in the whisper directory.

Apart from the standard Go libraries, the implemented protocol only takes advantage of one external library (github.com/ecies/go) for asymmetric encryption that uses the standard Elliptic Curve Integrated Encryption Scheme (ECIES) with a SECP-256k1 public key (the official go-ethereum package was also available but it includes too many (not required) features).

7 Results

The following is an example of a simple demo application that uses the protocol.

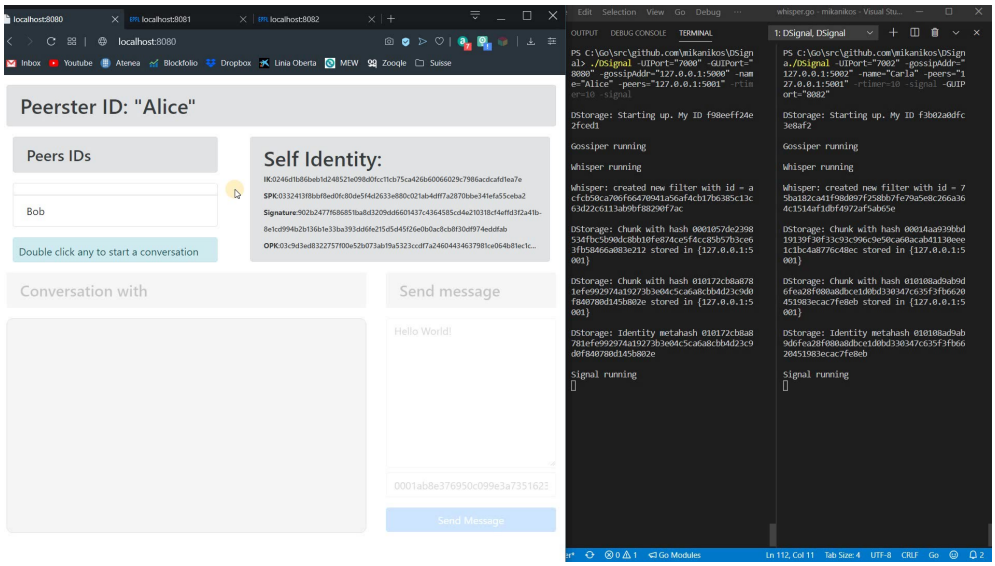

Figure 3: Initialization of two Gossipers

The first step is to initialize the Gossiper, which means to first initialize the Peerster as usually, then initialize the Decentralized Storage using the Peerster address to generate its ID, later initialize Whisper with a new filter and finally initialize Signal trying to retrieve stored identity and ratchets (if any) and then uploading the identity to the decentralized storage.

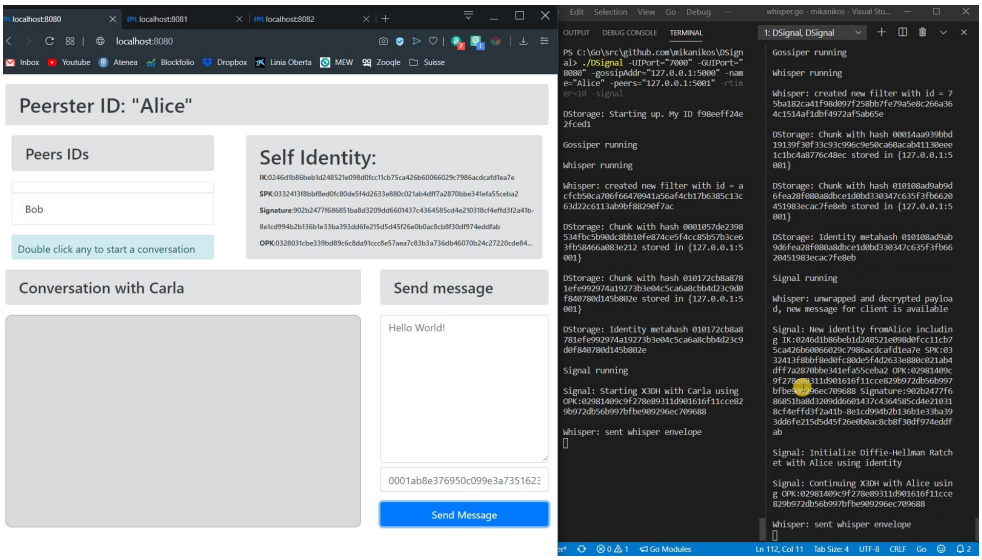

Figure 4: X3DH initialization with Alice's identity

The second step, upon a message exchange, is to start the X3DH protocol by sending our own identity (Alice). The identity will be wrapped in a Whisper envelope and sent. Upon receiving the envelope the destination node (Carla) will check its validity according to the topic and PoW, and unwrap it.

Then, Carla will continue with the above Signal protocol to initialize its Ratchet and send back a X3DH message including a Ratchet message and its identity. This will allow to continue the protocol.

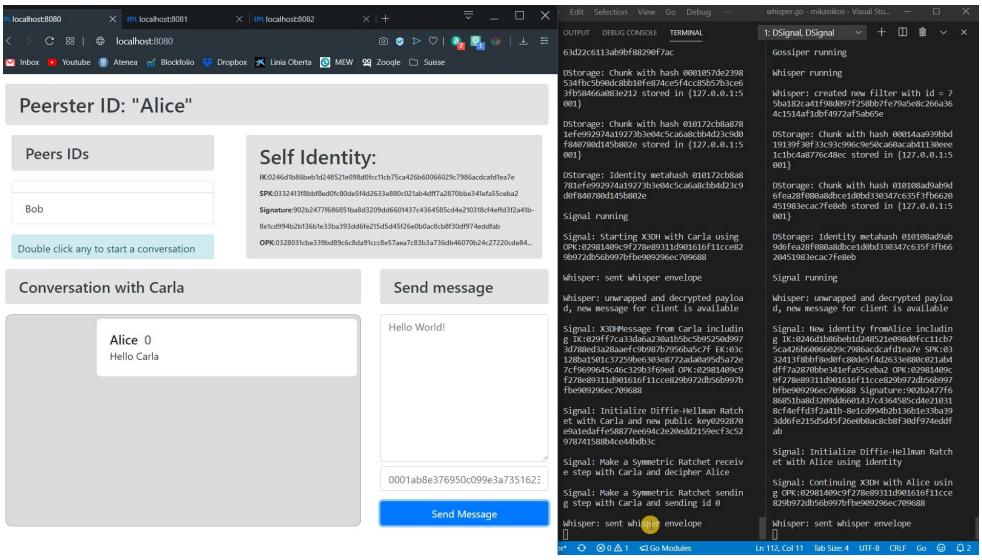

Figure 5: Alice Double Ratchet initialization

Upon receiving the message Alice will initialize its Ratchet and will send the first intended message encrypted, as an envelope. The X3DH protocol completes by storing the new Ratchet for later usage.

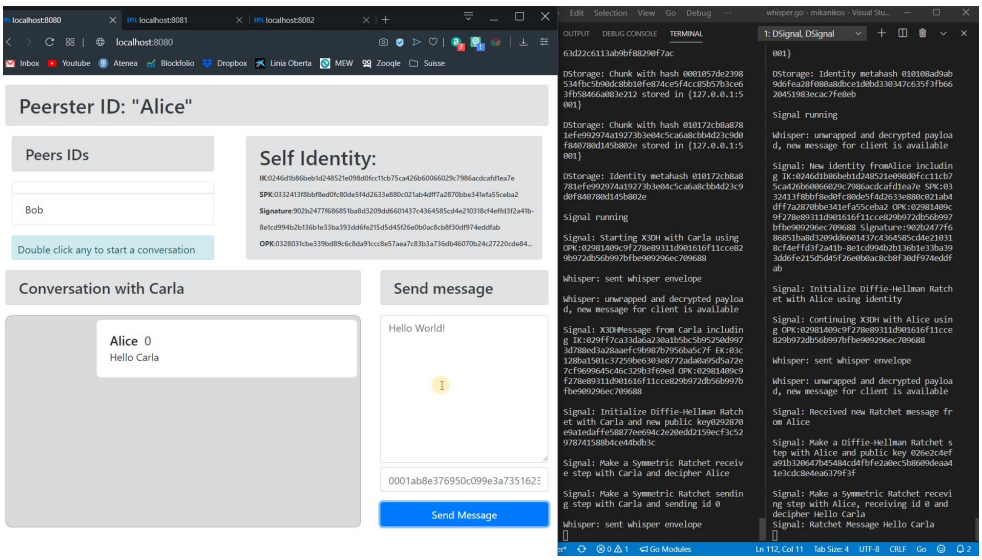

Figure 6: Alice first Double Ratchet message

Carla will unwrap the received envelope, decipher and display the message, making a DH and Symmetric Ratchet step.

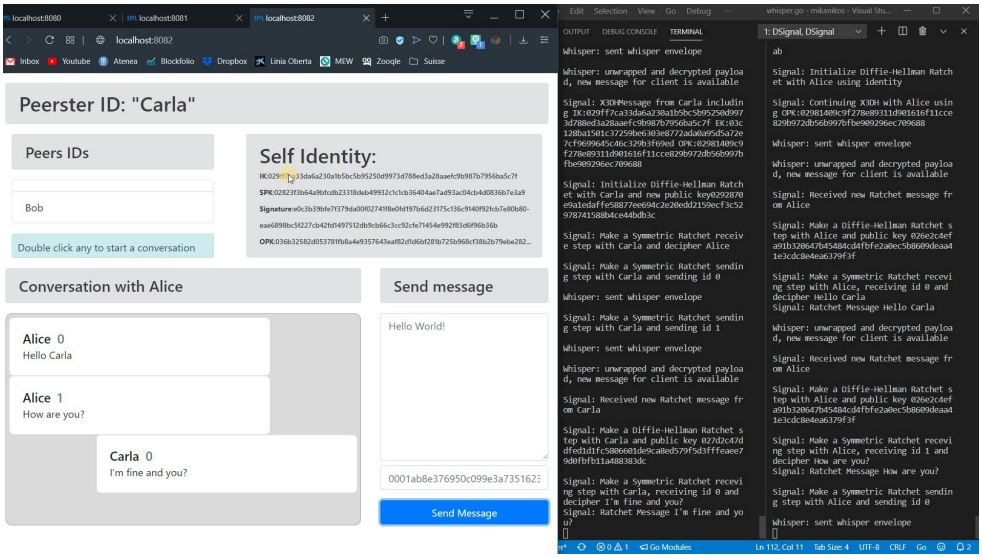

Figure 7: Execution after two more steps

Upon a message exchange we are able to see the different envelopes being sent, received and Ratchet messages being encrypted and decrypted. And how the different Ratchet chains advance and update.

References

[1] Druschel, Peter, and Antony Rowstron. *PAST: A large-scale, persistent peer-to-peer storage utility.* Proceedings Eighth Workshop on Hot Topics in Operating Systems. IEEE, 2001.

[2] Rowstron, Antony, and Peter Druschel. *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems.* IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2001.

[3] Benet, Juan. *IPFS-content addressed, versioned, p2p file system.* arXiv preprint arXiv:1407.3561 (2014).

[4] Maymounkov, Petar, and David Mazieres. *Kademlia: A peer-to-peer information system based on the xor metric.* International Workshop on Peer-to-Peer Systems. Springer, Berlin, Heidelberg, 2002.

[5] https://github.com/ethereum/wiki/wiki/Whisper

[6] https://www.mycryptopedia.com/ethereui/wiki/Whisper-PoC-2-Protocol-Spec

[7] https://signal.org/docs/specifications/doubleratchet/

[8] https://status.im/researc

[9] https://signal.org/docs/specifications/x3dh

[10] https://moodle.epfl.ch/pluginfile.php/1632974/mod_resource/content/6/cryptosec19handout.pdf [Case Studies: Signal - COM401 Cryptography and Security - Serge Vaudenay]

[11] https://tools.ietf.org/id/draft-mcgrew-aead-aes-cbc-hmac-sha2-02.html