A decentralized social network with end-to-end encryption and contextual information (DRAFT)

Manuel Schultheiß
manuel.schultheiss@tum.de

January 9, 2016

Abstract. Decentralized social networks with end-to-end encryption would take back control of data to the end-users. In this paper Noisecrypt is presented as such a network. First, some basic cryptographic principles are explained before focusing on the underlying protocol.

Contents

1 Prologue

I's personal. It's private. And it's no one's business but yours.

These are the first words, of Phil Zimmermann's essay "Why I wrote PGP" from 1991 [**PHIL**], arguing why emails should remain private. 24 year later, electronic communication has gained more and more attention in your everyday life. Today, letters have become instant messages and half of your real world conversations have become social network posts. But while you could be almost sure a letter remained private, today a bunch of stakeholders such as governments, technicians and ad-serving companies has access to data derived from your conversations.

Most people give a "I have nothing to hide" as an answer when addressing this point. Regarding confidentiality may be true, as long as you are not a politician, your are not responsible for a business unit, or do not have anything else to say other people are willing to listen to. But while your self forgets most of the things you said, it is probably saved in some data center waiting to get parsed by some algorithms. With more and more data you become calculable, your actions, - like what products you are willing buy or who do you vote for - can be predicted and influenced long before you perform them.

But beside ads have become the number one revenue model for internet companies and politicians and trying to expand data collection whenever it is possible [**EFF**], there is also one more problem: With the decay of classic media, such as newspaper and television, social networks have become the primary infrastructure of directing people to information now. The heterogeneous landscape of newspapers and blogs providing different information is slowly fading. The newsfeed has become the primary source of information. And it is not wise to trust a single institution here, because who controls what you read controls who you become.

2 Introduction

In contrast to classic social networks, distributed social networks with end-to-end encryption can provide self control of your data and independence from big companies. *Distributed* means, that there is no central server hosting your data, but many small servers all around the world, which everyone can setup. Just like email, every participant has a unique address like *yourname@yourserver.com*. People can interact seamlessly with people on other server. With end-to-end encryption not even the server hosting your data can see your data. This is necessary for a distributed architecture as otherwise the server admin could read your messages, for example.

Another innovation of Noisecrypt is the so called *Context*, every post can have optionally. A context contains semantic, computer-readable information about the post. An algorithm can later aggregate all people moving from point A to point B tomorrow, with having 3 seats free in their car with a tolerance of 20 kilometers, for example. Or you can look up for friends making pizza for lunch and want to share their food today. Figure 1 shows the creation of a context and the creation of a filter to search for existing context of other users. Websites already providing share-economy services can also add their data to the Noisecrypt network, using the Noisecrypt Schema JSON Syntax, as defined in the file schema.coffee. The context system introduces elements of the *Web of Data*, formerly known as the semantic web to social networks.

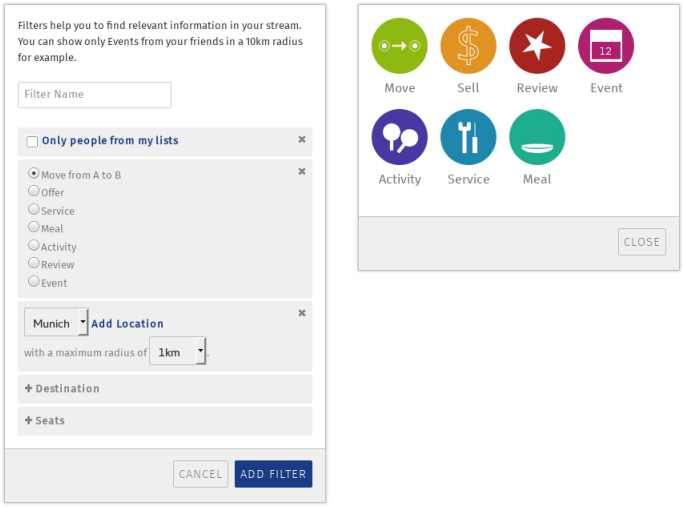


Figure 1: Context dialogues. Search dialogue on the left; Create dialogue on the right. You can perform searches for transportation, events, products, ride sharing and activities for example.

3 Existing work

With Diaspora [**diaspora**], one of the first popular distributed social networks was released. The messages are not end-to-end encrypted, however, which makes it possible for the server admin to read its users messages. A more secure concept is *Twister* [**twister**], a Twitter like peer-to-peer micro blogging platform. Using the blockchain makes it possible to encrypt messages and even meta data (including the IP Address!). Noisecrypt itself does not encrypt meta data at the moment for simplicity and performance reasons, but such an encryption scheme could be implemented later on using techniques like onion routing between sender/receiver for example.

4 Encryption and Security

4.1 Formal Notation used in this paper

4.1.1 Reach

We define the *reach* of a server: Each server S_i, on which at least one user has a public key of another user hosted on server T_j with $i, j \in 0, 1, 2, ..$ is *reached* by this server. The reach is a list of tuples (S_i, T_j). See figure 2 as an example.

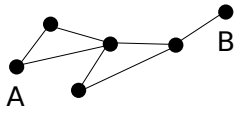


Figure 2: $|Reach(A)| = deg(A) = 2$ as the users on server A have public keys of two different servers. The bidirectional edges representing two servers having public keys of users on the respectively other server (In a real world scenario, there would be some directed edges of course).

4.2 Cryptographic Primitives

4.2.1 AES-CBC

AES is a fast, symmetric encryption algorithm that was standardized in 2001 and is still considered secure, although there exist some attacks on it. The following explanation is based on the lecture slides of Carle et al [**carle**].
A key $K_{A,B}$ is used for encryption and decryption by the two parties A and B. Cipher Block Chaining Mode (CBC) ensures that blocks that are identical in the plaintext, are not identical anymore in the chiphertext. This is essential, as otherwise images can still be recognized for example, although they are encrypted. First, a initialization vector (IV) is created:

$$c_0 = IV$$

A plaintext block p_i becomes a chipher block c_i during encryption.

$$c_i = Enc_K(c_{i-1} \oplus p_i)$$

and the same thing reversed for decryption:

$$p_i = c_{i-1} \oplus Dec_K(K, c_i)$$

It is important that the IV is shipped with integrity protection, otherwise an attacker could modify the first 256 bits in order to set the first decrypted plaintext block to a custom value. The other blocks can not be influenced however. There are several techniques to add it (`https://en.wikipedia.org/wiki/Authenticated_encryption`). Integrity protection can be achieved using the following steps using an encrypt-then-mac Scheme:

1. Encrypt Message using AES-CBC
2. Get SHA-256 hash value of encrypted message and salt with key k and crypto algorithm version a.
3. JSON Result is:

   $\{$
   a: 1,
   m: $Enc_{k||v}(text)$,
   h: $HMAC_{Sha256}(Enc_{k||v}(\mathrm{text}),\ k||v)$
   $\}$

   The algorithm version a is added, as otherwise, an attacker could extract the plaintext and use backwards compatibility to let the decryption and MAC check be performed by an old algorithm, also there is a better algorithm available.

   For example: The current implementation supports Non-HMAC AES when the information was encrypted without a HMAC. An attacker could remove the HMAC key and just send the plain AES encrypted content to the client and the client would still decrypt it correctly.

Please note that for more security, the salt for the $HMAC_{Sha256}$ should also not use k, but a derived version of k.

4.2.2 RSA

In RSA there are two different keys. The public key is visible to everyone, while the private key is only visible to the *generator* of both keys. The public key can be used by anyone to send the *generator* a message, while the private key can used by the *generator* to decrypt the message. Asymmetric encryption is a lot of slower than symmetric encryption. So usually we generate an AES key, encrypt the text with the AES key and encrypt the AES key with the RSA key, rather than encrypting the whole text with the RSA key. For RSA, we generate two large prime numbers p and q.

$$n = p * q$$

and define

$$\Phi(n) = (p-1) * (q-1)$$

Let e not equal to 1, smaller than $\Phi(n)$ and not share a common divisor with $\Phi(n)$. Now we compute d under the restriction:

$$e * d = 1 \pmod{\Phi(n)}$$

To encrypt a message M or decrypt chipher C use:

$$C = M^e \pmod{n}$$
$$M = C^d \pmod{n}$$

To provide *Ciphertext indistinguishability* RSA is used with a padding scheme like PKCS1. Here we generate n with more than two primes:

$$n = p_1 * p_2 * ... * p_n$$

with p_1 being r and p_2 being q. [**carle**]

4.2.3 Further reading

1. PKCS#1 v2.2 RSA Cryptography Standard
   `http://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf`
2. Lessons learned and misconceptions regarding encryption and cryptology
   `http://security.stackexchange.com/questions/2202/lessons-learned-and-misconceptions-regarding-encryption-and-cryptology/2206#2206`
3. HMAC versus raw SHA hash functions
   `http://dev.ionous.net/2009/03/hmac-vs-raw-sha-1.html`

4.3 Used Libraries

There are a Javascript library called Gibberish AES for Javascript and Tom Wu's RSA Library for RSA providing the main functionality in the technology preview. Google wrote with its E2E project a better implementation of cryptographic algorithms which should replace the above ones as soon as possible. The long term goal is to use the W3C web crypto API however, if supported in all major browsers, as this one is more secure regarding key deletion from memory (RAM) and random number generation.

5 Implementation

5.1 Backend Technology Stack

The current implementation is a proof-of-concept, which means it is not ready for production use yet. On server side, PHP is used as an interpreter. The reason for PHP is that it has a wide community supporting it, as it is used by many other big websites like Facebook or Wordpress for example. Furthermore it does not depend on 100 extra packages for basic functions with questionable future support like Node.js. *ZeroMQ* is used to send socket messages and communicate between different PHP threads. *Gearman* is necessary to execute background tasks like sending messages in the asynchronously. As a database, *MongoDB* is used, as it is scalable and has the ability to directly store JSON object to the database. Administrations scripts are written for GNU Bash to provide easy ssh access.

5.2 Separation of Client and Server

In contrast to classic social networks, we separate client and server in two separate projects first. In Noisecrypt, the server does not return HTML to the browser, but JSON instead which is parsed by the client software. This is necessary, as otherwise a server could return malicious javascript to the user for example and make end-to-end encryption useless. For the final release, the user has to download and install the client locally and verify the file hash to have maximum security, For testing purposes, the user can also use a web hosted version of the client. Although they are hosted on different domains, client and server can still connect to each other via cross origin resource sharing (CORS).

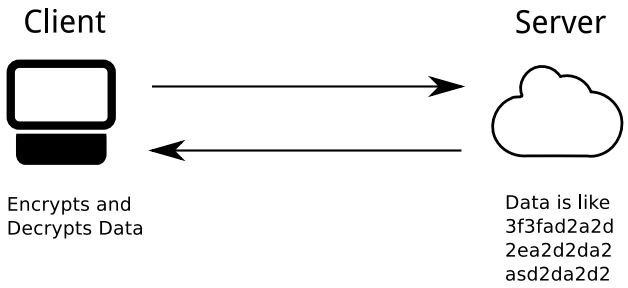


Figure 3: Client and Server code is strictly separated. The client loads the Noisecrypt application from a local webserver, a browser plugin or a trusted server. This application connects to the server that is hosting the user's data itself. Hereby, personal data such as messages or private profile information, is encrypted and decrypted on the client side, so the server sees nothing.

Hereby Noisecrypt uses an *encrypted data storage*. This means critical data is encryped with a passhprase using AES-CBC (Section 4.2.1) and hosted on the server (Figure 3).

Consequently the server can not read the encrypted user data. Such data includes

1. **Personal information** like phone number. This information can later be decrypted and encrypted with a public key to distribute it to other people.
2. **The private RSA key** used to decrypt everything that is received from other users.
3. **The key directory**, which contains the public keys of all contacts. Here it is important that the receiver B of a message sent by user A only uses the newest private key to decrypt, as a malicious server of the key directory owner could return an old public key of user B.

for example. The data must also be signed to prevent manipulation such as bit flipping etc.

5.3 Public Keys Verification

If we want to send an encrypted message to user B, we need the public key of user B first. However, as an evil server or a man-in-the-middle can provide a wrong public key, it is necessary to check the public key for correctness. If done so, we encrypt the public key of user B with our passphrase and store it in the encrypted data storage.
The user itself has to validate the fingerprint. Currently the fingerprint f is generated by applying a SHA-256 Hash function on n (modulus) and e (exponent) of the RSA key concatenated:

$$F = Sha256(n||e)$$

For a better user experience, the fingerprint can be truncated and seperated by ":", so the result looks something like:

$$12 : 3f : 3e : 2f: 12 : 3f : 3e : 2f: 12 : 3f : 3e : 2f\ (...)$$

instead of

$$123f3e3f123f3e2f123f3e2f\ (...)$$

Fingerprints are used to verify public keys of other users. Figure 4 shows the dialog to validate fingerprints.

One thing that is missing right now is to make sure that the **server always returns the newest verified public key** of a user. Hereby an evil server could provide an old, compromised key. So we got two things to fix here:

1. Other users should not see messages or parse control messages, which were encrypted with an old key, as such messages may have been faked.

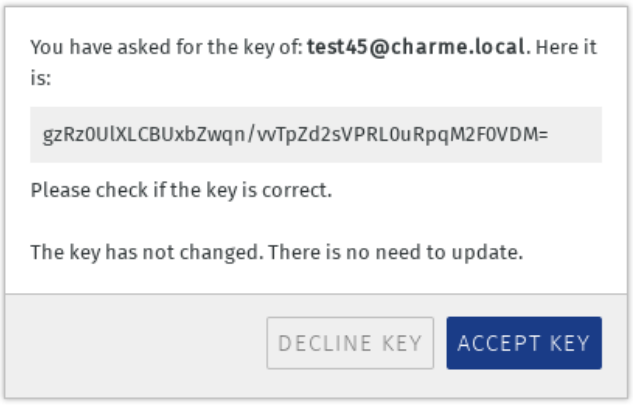


Figure 4: Fingerprint Validation Dialog

2. A man in the middle (MITM) should not be able to provide an old public key and read messages exchanged between two users. See figure 5 for an example attack. This attack requires a malicious host server however, as we assume the protocol between client and server is HTTPs. Furthermore it requires an old compromised key.

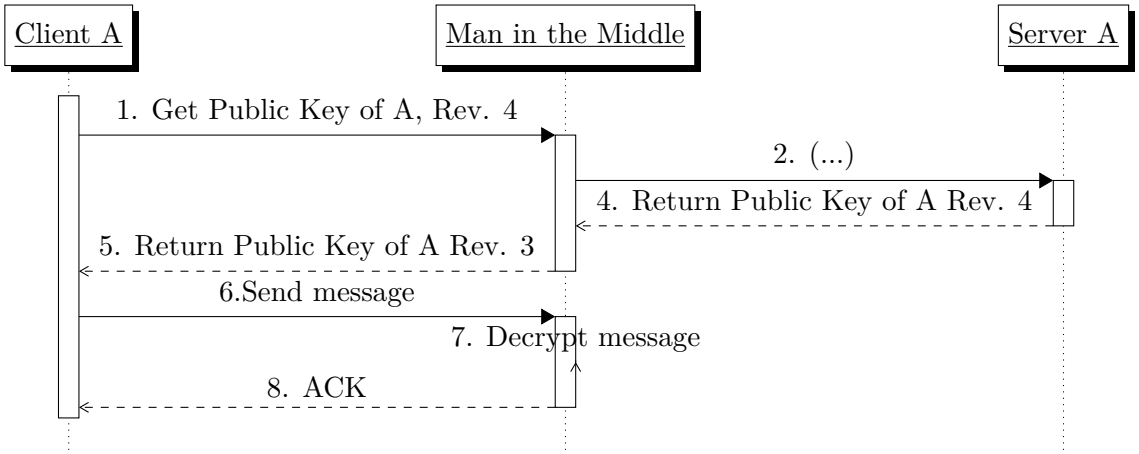


Figure 5: MITM attack on keydirectory. A nonce between client and server is not solution as we also assume that the server A has been compromised.

To fix the **first vulnerability**, we have to take care about a few things:

Old Public Keys have to be rejected by server If a message from server A is encrypted with an old public key, server B will drop it. A message from A to A can still be received by an evil server, however.

Old Public keys have to be rejected by client the client software must also drop messages that are not encrypted with the latest public key. To check this, we must add the creation time to each public key.

The **second vulnerability** (Figure 5)is harder to fix. While we can store the newest key version on the client device, on which it was added to prevent MITM on this device, it is hard to notify the other devices about the newest key version. Example:

1. Newest key revision of B is 1
2. User A adds new key of user B on device 1. Device 1 knows newest key revision of B is 2
3. User A sends messages to user B on device 2. Uses revision 1 as told by MITM (Key directory replay attack) between client and server.
4. MITM can read message with compromised revision 1 key.

The best practice would be to display a "security message" on all devices, which shows up after a key was added on another device:

1. User adds public key on device 1. Message is *"Please check your Android device (device 2) for push notification about revision 2. If you do not receive this message your server is probably evil."*. We send control message $c = Enc_{fastkey1}(\{userid, revision\})$ to our server who sends it to device 2.
2. Device 2 can decrypt and display the message "User ... has new revision of ...". We also poll the new key directory from the server. The sum of public key revisions must be greater than it was before, otherwise an attack was performed.
3. Additionally we have a security page, were the hash of the key directory is displayed, on each device.

5.4 Encryption

The data to be encrypted can be divided into three main categories:

Posts Posts of collections are encrypted for a lot of users. Each post is encrypted with a key, which needs to be distributed to the audience of the post using public key cryptography.

Messages Each message belongs to a conversation. Each conversation has a key, which needs to be distributed to the message receivers. Here we also have to take care of cases when someone is added and removed from a conversation.

Personal Information such as phone number or hometown. Here we have to make sure, that if access is revoked, we do not have to generate a new key and distribute it to all people. Therefore we introduce buckets, which make it possible to handle this case more efficient.

The following sections will give a basic overview, for a more detailed explanation see section 6.

5.4.1 Edgekeys

An edgekey is a symmetric key, that is established after public key verification and encrypted via public key by the other party. This is done, because asymmetric cryptography is a lot of slower than symmetric cryptography.

5.4.2 Post Encryption

A new post is encrypted, when its audience is restricted to a list of people. In this case we generate a postkey pk, which encrypts the content of the post. Next we need to distribute pk to other people. We can use the symmetric edgekeys $e_1, e_2, ...$ of these people to encrypt the postkeys: $k_{enc} = enc_{e_1}(pk)$. Next we distribute it to the corresponding receivers. Furthermore we generate a signature of post data with our private key, and add the public key revision of the receiver.

5.4.3 Message Encryption

Message encryption is basically working like in PGP. A simplified concept is illustrated in Figure 6). To start a new converation we obtain the public keys of the receivers from the encrypted data storage first. Then we generate a symmetric key and distribute it to the receivers. The symmetric key can change if a user is removed or a public key of a conversation participant is updated or a certain span of time has passed. In this case, we have to distribute it again. A new message is encrypted with the newest symmetric key s. A message consists of information such as *message content*, *conversation identifier*, *time* and *key identifier* of the symmetric key used for encryption. This information is signed using the senders private key.

5.4.4 Encryption of Personal Information

When disclosing private information like telephone number or email to other people, this is working slightly different to message distribution for performance reasons. If the access to a certain person is revoked, it would be inefficient to send the new key to all users. Instead we create *buckets*, each containing a maximum of k users. When a new piece of information is encrypted, we generate a random key K, and encrypt it with the bucket key B_i. As every user who has access to the information, owns the bucket key, the user can decrypt it with

$$plaintext = Dec_{B_i}(Dec_K(chiphertext))$$

When access is revoked for a user, we simply generate a new bucket key for the bucket assigned to the user and distribute the new bucket key to all users in this bucket. With this system we are able to revoke access to information and release new information to

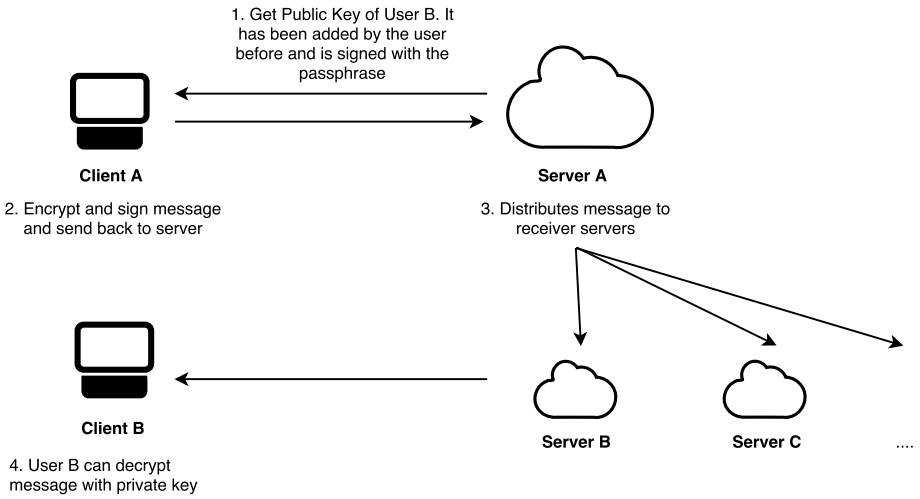


Figure 6: Basic message encryption. Server A can also send the message to more servers than only Server B. The client does not send the message directly to server B, as with increasing number of receivers this would cost to much bandwidth for the client.

a user with doing $\mathcal{O}(k) = \mathcal{O}(1)$ requests. For Noisecrypt we set $k = 10$. See figure 7 for a illustration of the process.

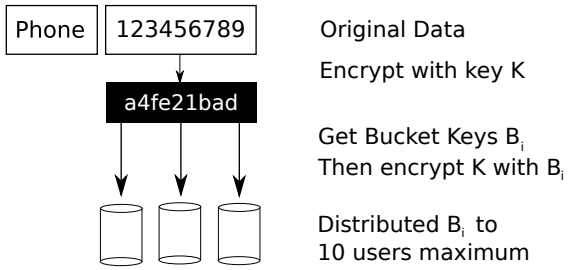


Figure 7: Encryption of personal information using buckets

5.5 Key Updates

When a private key or passphrase is compromised it is necessary to recrypt the encryped data storage to protect it from future access by the attacker. In this case, all information is transmitted to the client, decryped with the current passphrase and encrypted with the new passhprase. The revision number for the public key increases afterwards.

5.6 Primary Attack Vectors

5.6.1 Data Integrity

We must ensure, that the server can only return data, that has been created by a user. As an example, the autocomplete should only returns name/address pairs that have been added by the user. Imagine the following case: Alice adds Bob with his Noisecrypt ID bob@somewhere.com to her key directory. Now she wants to send a message to Bob and opens the auto complete. She types Bob, and the server returns a modified Id, probably on the same server (eve@somewhere.com). The Noisecrypt client would now encrypt the message with the other key and send it to Eve. So Eve could be able to get some sensitive information while Alice thinks she is communicating with Bob. Therefore we must sign most data with a pass phrase derived key.

Also conversation messages itself must be shipped with integrity protection. In contrast to the above example, here we must protect the integrity for other users than our self. When we use the conversation key for the hmac of the message, we only protect the message against attacker that are not part of the conversation. Let person A and B be part of one conversation. Now person A could still send messages with the name of person B. Using the private key would be a better solution here, but it is computationally expensive (aka slow, especially on mobile devices) to check a RSA signature in the background. It is probably the best idea to check use the first approach for a background verification and the second approach only after a click on a "Check signature" button. The second approach should also be used with a smaller RSA key length for background verification, however.

5.6.2 MITM on Client Software

The client software must be distributed with integrity protection in the final release. For example as a Apache Cordova package. To use the client software on a third party server is very dangerous and only recommended for testing purposes! The host of the client can theoretically read all your data by modifying the client software.
There is also a new technology called sub resource integrity which can be used to validate Javascript and CSS resources (See `http://www.w3.org/TR/SRI/` and `http://githubengineering.com/subresource-integrity/`). Maybe they introduce some mechanism to validate a whole web page with a hash recommended by the majority of browser users and additionally located on a (Mozilla or Google) server. So lets keep an eye on that.

5.6.3 New Conversations with malicious userId/Name pair

An attacker can initiate a new conversation with the name of a friend of you, but use another userId instead. Example:

1. Alice (alice@server.com) and Bob Alice (bob@server.com) are friends

2. Eve initiates a new conversation with Bob with Name "Alice" and userId eve@server.com.

3. Bob thinks he is talking to Alice, but he is not.

A solution for this problem would be to allow only names from the key directory.

5.6.4 XSS

One of the most critical attack vectors in Noisecrypt is Cross Site Scripting as with this technique private keys can be stolen. Therefore it is important to sanitize all displayed data from other people and servers. Never trust the server. For example imaging data is a JSON Object returned by the server:

```
\$("html").append("<a href='"+data.href+"'>TEST</a>");
```

If the server does not provide a hyperlink for data.href, but rather something like:

```
#' onload='$.post("someotherserver.com", {key: theprivatekey})
```

this is extremly dangerous. Therefore always sanitize data from the server!!!

5.6.5 Anonymity of Search Request and Profile Visits

In the current implementation, the user id is sent to other servers when requesting a profile or performing a search on this server for example. The server host could log who is visiting which profiles. Possible counter measures are

1. Providing a set of user ids, in which only one is the actual user id. This would affect the performance however and will only work if enough "fake" users are provided.

2. Use an encrypted database. (See section "Future work" for details)

5.6.6 Other Attacks

Spam All posts from non-friends should be blocked. An exception are posts containing semantic information which should be searchable all over the web. Here we can either expand the search to friends server (efficient but may contain more spam) or even check friend of friend posts only (more computational power, but less spam).

Malformed data Attackers can send malformed data to clients which results in the client to producing an exception and therefore make Noisecrypt unusable for the user. Therefore malformed requests should be filtered by the client by checking for missing and correct typed values.

DDoS Attackers can flood small servers with a lot of requests.

Session Capturing Send JSON request from another Website to host server. Make sure you can not access unencrypted user data such as lists for example.

6 Protocol Specification

6.1 Login

1st step Get password salt value via *reg_salt_get*

2nd step Login with *user_login* and parameters u (username) and $p = sha256(password||salt)$. Status PASS is returned if successful. We will use some other hash function than Sha256 later of course, for security reasons.

6.2 Search

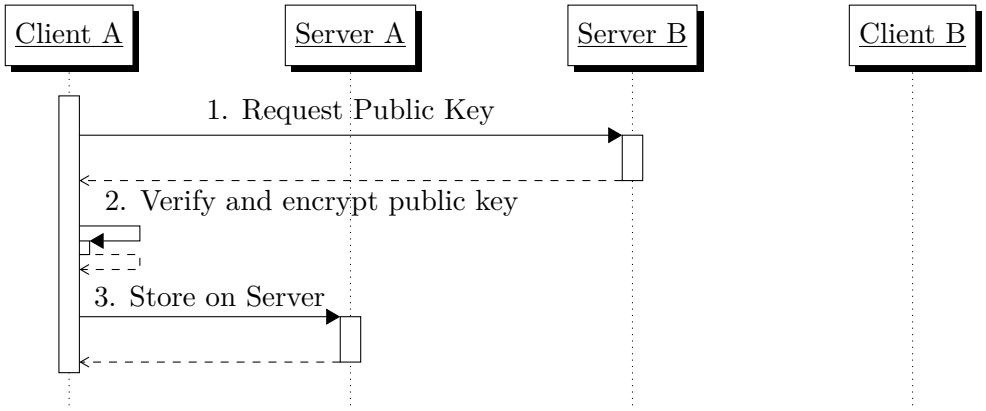
When it comes to search, we face some problems. Of course, it is not efficient to query hundreds or even more server from one client in the whole network. Filesharing applications typically use Kademlia or Chord to index data within peer-to-peer networks.Such concepts may also be useful for search, as YaCy for example has shown. Thorsten Schütt et al. have proposed an extension for distributed hash tables in their paper *Range queries on structured overlay networks* [**schuett**]. This technology could be used for contextual search such as *"Show me all friends driving to a GPS location between lat,lng=(10,12) and (12,14) tomorrow"*.
A search in social networks could also use the social graph to rank results of near people higher than from people more far away. Right now we have a simple approach, which is used for stream searches for example:

1. First we generate a list of all target servers we connect to. In case of a list is provided for stream filters, it's the hosts of the people in the list. Otherwise its the 10 most occurring severs in the key directory of the user. We also need to provide the user id u of the searching user, in case the context item is limited to a specified audience (bad privacy)

2. The target server respond by providing posts for example. If a post was limited to a audience, we check if u occurs in this list and return the post in this case. This check has to be improved for security reasons, as a bad source server could access posts he actually has no access to.

6.3 Add public key to keydirectory

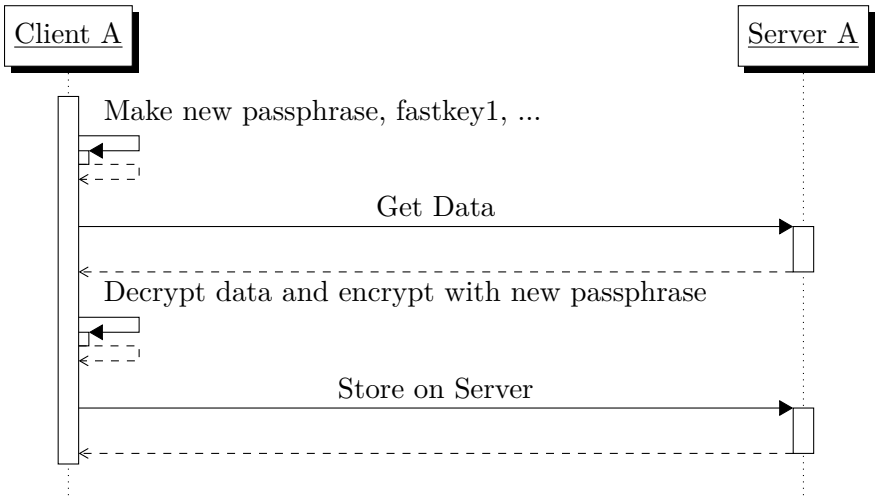
A new public key is requested, verified and added to the key directory.



After 1. the user needs to verify the key. Afterwards encrypt the public key with fastkey1 and store it on the users server afterwards.

6.4 Recryption of data

When the private/public keypair is updated, it is required to recrypt the data in the encrypted data storage. This is done in the function *updateDataOK()* in *settings_keymanager.js*.



6.5 Signup

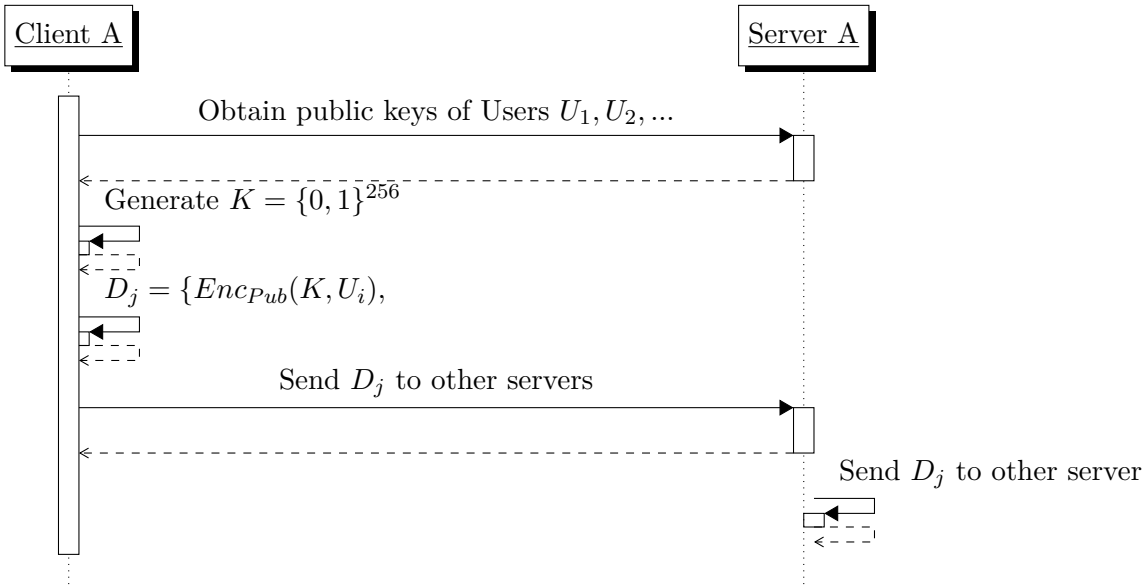
Client:

1. Get salt s from server
2. Generate password p = Sha256(s||p') from plaintext password p'

3. Generate passphrase PP and asymmetric keypair KP. Encrypt KP with PP.
4. Generate sub keys fastkey1 and fastkey2, encrypt them with passphrase.
5. Hash username with FK1 to prove integrity later
6. Send Enc(KP), Enc(FK1), Enc(FK2), Sha256(username||FK1) and other userdata to server.

6.6 Init Conversation

A conversation contains messages and people who can read this messages. Let's call D_j the payload. It contains information relevant to the conversation such as time, last preview text, members or identifier. D_j must be integrity protected of course. K is a 256 bit symmetric key used for AES encryption.



Server A distributed the payload to other servers afterwards. Technically this is done by a Gearman daemon running in the background. The receiver clients can afterwards decrypt the key K with their private key.
The JSON message schema to init a conversation from the senders client to the senders server looks like this:

- **id** = message_distribute
- **action** = initConversation
- **messageData** (signed with K)
  - **hmac**: The HMAC to check integrity

  - **revision**: Revision of K
  - **objType** = object
  - **obj**
    - **receivers**: List of userIds
    - **usernames**: List of (userId,name) tuples.

- **messageKeys**: List of public key encrypted public keys K

The server redirects the message to the receivers servers afterwards, the message JSON has the following properties:

- **id** = message_receive
- **action** = initConversation
- **messageData** (signed with K)
  - (See above)
- **key**: Public key encrypted public key K for the receiver

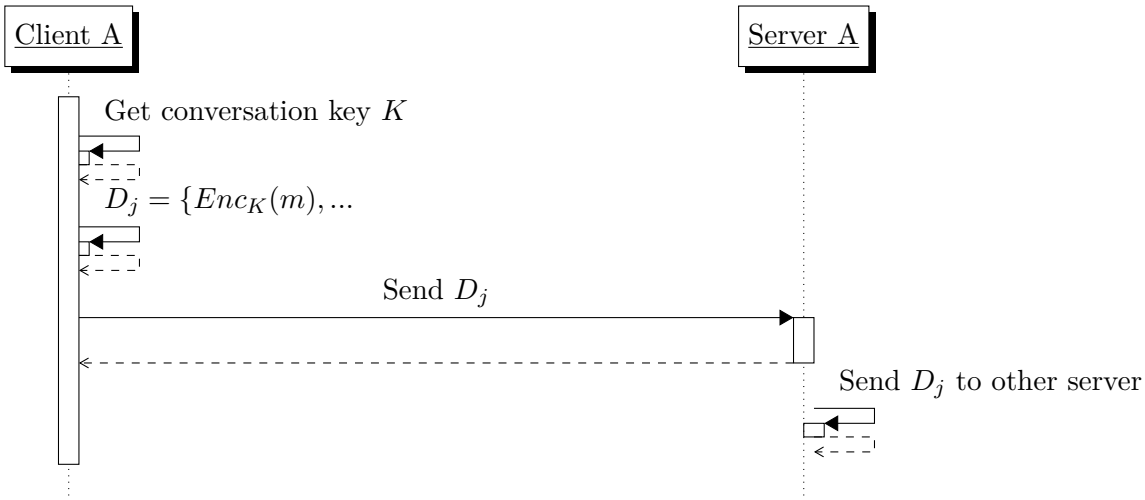
Please note that here we could be more efficient in the future by grouping the message_receive requests for same people on same servers.

6.7 Add People to Conversation

People present in the key directory can be added to an existing conversation. As they should not be able to see previous messages, a new message key $K_{(revision+1)}$ is generated in this case by the person who adds the new people. This key is distributed to the other people afterwards. The people in the conversation have now to be signed again by the person who added the new people.

6.8 Write message

Let K be the message key, which has been generated at conversation init. D_j is a JSON string, containing data such as encrypted text m, the username or the signature.


Client A
Server A
Get conversation key K
$D_j = \{Enc_K(m), ...$
Send D_j
Send D_j to other server

7 Future Work

The remaining work is primary to refactor the code, add missing security stuff and improve the search function. For searching posts, we can also try to hide the text and context parameters, such as longitude, latitude or price . For example, CryptDB [**cryptdb**] is a working SQL implementation. Its developers also responded to the paper *Inference attacks on property preserving encrypted databases* [**INF**] which questioned the security of such databases.

8 Terminology

||	Concatenation Operator. For example "a"||"b" $\Leftrightarrow$ "ab"
Conversation	A sequence of messages belonging to two or more users
Encrypted Data Storage	A JSON object in the cloud encrypted with an AES key that is only known by the user. Very vulnerable to replay attacks.
Fastkey	A symmetric key dependent on the passphrase. It is only known by the user itself, no other persons or servers.
Key Update	A key update can happen after a new keyring is generated or someone has been removed of a conversation for example. In both cases we generate an updated key, but with a different protocol.
Recrypt	Kind of neologism. Means to decrypt with an old key and encrypt with a new key. Maybe there is already another word for that...
Revision	The version of the public key or edgekey. If a new key is established, it increases by 1.
userId	An unique identifier for each person in the Noisecrypt network, similar to email. Example: ms@myserver.com

9 Sources