Anastasis: LEDGER Milestone 1

Berna Alp Vaishnavi Mohan Christian Grothoff
Belen Barros Pena Dennis Neufeld Domminik Meister

June 28, 2021

Abstract

Anastasis is a Free Software key escrow system that offers a practical way for ordinary users to bridge the conflicting requirements of keeping key material confidential and also available.

Anastasis provides a solution for secure recovery of secret keys, which works without passwords or other key material. This is achieved by splitting the key material across multiple independent Anastasis service providers, and enabling users to recover their master key by authenticating with each provider. Our protocol ensures that — without prior knowledge — the service providers learn nothing from the protocol except the minimum amount of data required to authenticate the user. Even that information is only disclosed at the time of authentication. Anastasis offers users control over the set of escrow providers, selection of authentication methods and desired policies for key recovery.

Contents

1 Introduction

1.1 Our team

Berna Alp is an economist by trade. She is currently council member at the pretty Easy privacy (pEp) foundation, a board member at ISOC Switzerland and she owns a consulting business specialized in IT transformation and ERP projects. She has worked as project coordinator on World Bank projects, as Senior FI/CO Consultant at Andersen Consulting in New York City and as SAP FI/CO & JVA team lead at a multi-national steel company implementing SAP in 28 companies and 17 countries. She is responsible for general management in the team.

Vaishnavi Mohan is a software engineer with a master's in distributed software systems. She specializes in the development and secure deployment of applications on public clouds. She will be responsible for the cloud deployment and the integration of the authentication backends with existing cloud services.

Christian Grothoff is a professor for computer network security at the Bern University of Applied Sciences, researching future Internet architectures. His research interests include compilers, programming languages, software engineering, networking, security and privacy. He will work on the cryptographic protocol design and the Gtk+ GUI.

Belen Barros Pena is an interaction designer and user researcher since 2007 with extensive experience in planning, execution and analysis of qualitative user studies and quantitative research methods as well as requirements gathering activities, participatory design workshops and expert evaluations. She does research on the design of financial technologies in collaboration with groups traditionally excluded from their production process, such as older adults and people living with mental health conditions. She will work on the user experience design.

Florian Dold is a co-maintainer of GNU Taler, an experimental network designed with the goal to provide privacy and security without the need for trusted third parties. He will work on the integration of Anastasis with the Taler wallet.

2 User stories

There are two main user stories, for backup and recovery respectively.

2.1 Secret splitting (backup)

Figure 1 illustrates the secret splitting process.

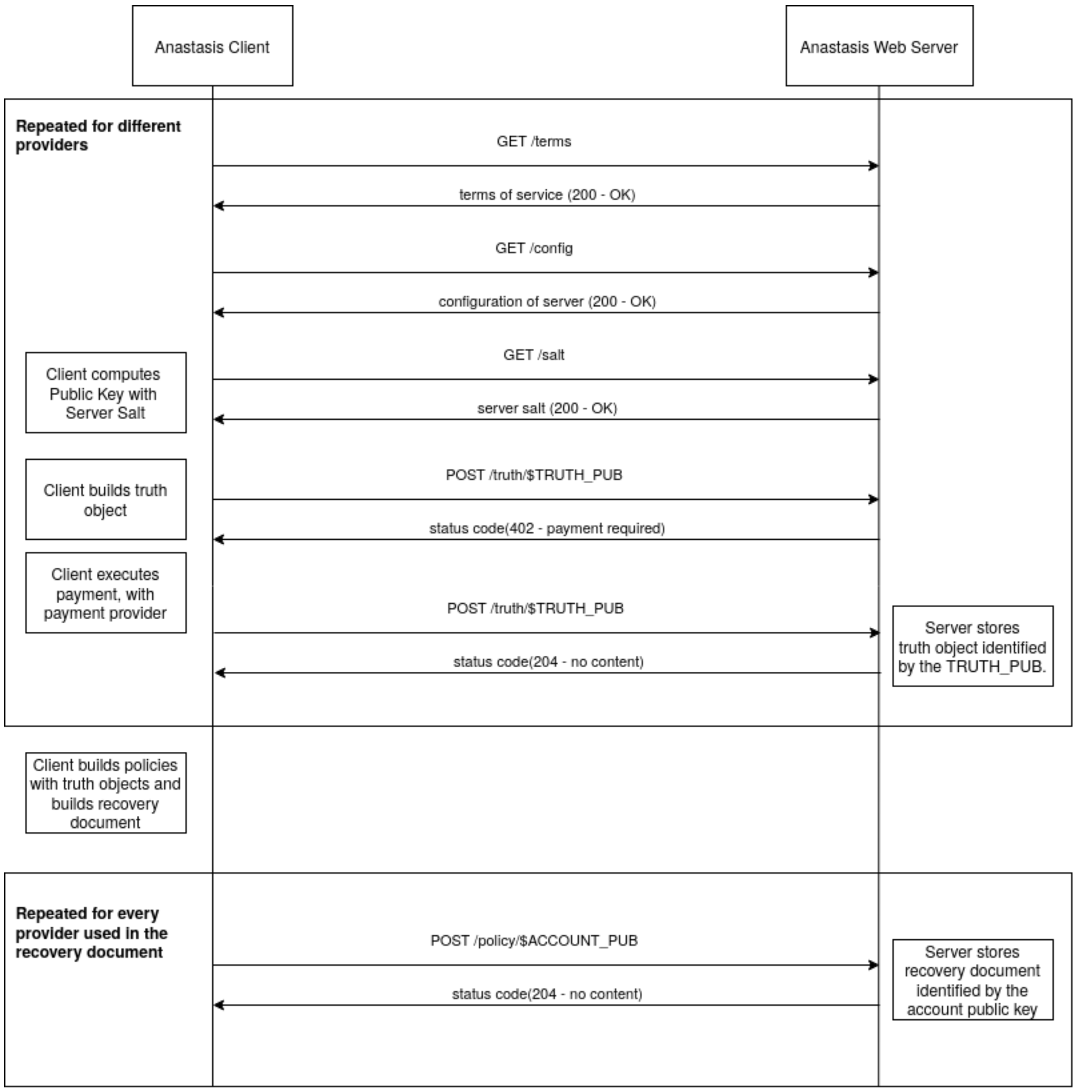

Anastasis Client
Anastasis Web Server
Repeated for different providers
GET /terms
terms of service (200 - OK)
GET /config
configuration of server (200 - OK)
Client computes Public Key with Server Salt
GET /salt
server salt (200 - OK)
Client builds truth object
POST /truth/$TRUTH_PUB
status code(402 - payment required)
Client executes payment, with payment provider
POST /truth/$TRUTH_PUB
Server stores truth object identified by the TRUTH_PUB.
status code(204 - no content)
Client builds policies with truth objects and builds recovery document
Repeated for every provider used in the recovery document
POST /policy/$ACCOUNT_PUB
Server stores recovery document identified by the account public key
status code(204 - no content)

Figure 1: Secret split process

1. The user selects a new escrow provider on which per wants to store a truth object.
2. The client software downloads the terms of service for this provider (GET /terms). This is also a check if the server is available if this command doesn't respond the client will abort the process.
3. Next the client requests the server configuration (GET /configuration). The configuration lists the available authentication methods and the protocol version of the server.
4. The client downloads the server salt (GET /salt). The salt is used to generate the server specific account public key, which identifies the user.
5. After the user has generated the public key, per will create a truth object on the client. The truth object contains all the needed information for the recovery for this key share. This truth object is sent encrypted to the server and stored under the TRUTH_PUB the client generated (POST /truth/$TRUTH_PUB).
6. In this scenario the client has not jet paid for the upload. This means the server will respond with the HTTP status code `402 Payment required`. The client first must do a payment with our payment provider — GNU Taler. After the successful payment the client will receive a payment identifier. With this payment identifier he can resend the previously failed request.
7. The user will now repeat the steps 1-6 until per thinks that they have setup a sufficient amount of authentication methods. The user can now combine these providers to create policies. For example per may have stored three truth objects at three different providers. This means per can now define combinations with these providers, for example A+B, A+C and B+C. This means the user has three ways to recover their secret.
8. After the user has generated the policies the client will generate a recovery document. The recovery document contains a list of all truth_seed's used, a list of the policies and the encrypted core secret of the user. The client will now send a encrypted recovery document to each provider used in the recovery document (POST /policy/$ACCOUNT_PUB). Through this, the recovery document is replicated and recovery can proceed without a single point of failure.

2.2 Secret recovery

Figure 2 illustrates the recovery process.

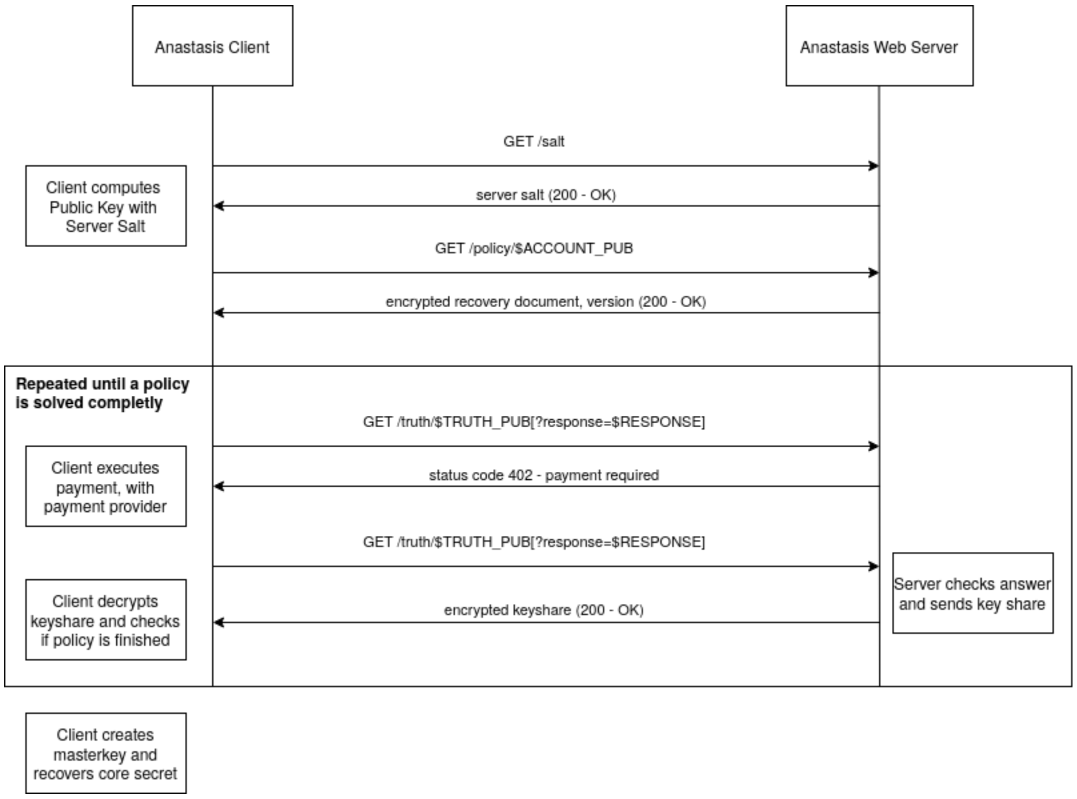

Anastasis Client
Anastasis Web Server
GET /salt
Client computes Public Key with Server Salt
server salt (200 - OK)
GET /policy/$ACCOUNT_PUB
encrypted recovery document, version (200 - OK)
Repeated until a policy is solved complety
GET /truth/$TRUTH_PUB[?response=$RESPONSE]
Client executes payment, with payment provider
status code 402 - payment required
GET /truth/$TRUTH_PUB[?response=$RESPONSE]
Server checks answer and sends key share
Client decrypts keyshare and checks if policy is finished
encrypted keyshare (200 - OK)
Client creates masterkey and recovers core secret

Figure 2: Secret recovery process

1. The user selects a server on which per previously stored a recovery document.

2. Next the client downloads the server salt to compute the server specific account public key (GET /salt).

3. After the user generated the public key, per will download the recovery document. At this point per can define a specific version or the latest version of the recovery document. In the illustration the client downloads the latest version (GET /policy/$ACCOUNT_PUB).

4. The client will now decrypt the recovery document and list all policies and authentication methods. The user now has to solve these challenges. In this example the user has to answer a secure question which was sent to them in the recovery document. (GET /truth/$TRUTH_PUB?response=$RESPONSE)

5. Note the server can define that a challenge has a certain cost, in this scenario the server rejects the first request because the user has not yet

paid for recovery. After the payment the user can resend the request. After each successfully solved challenge the client will check if one of the policies is completely satisfied. If all shares needed for one of the policies have been recovered, the client will decrypt the core secret and provide it to the user.

Figure 2 shows the flow using a secure question for the authentication challenge. If the user would have chosen a complex authentication method like SMS or E-Mail, the client would first need to start the challenge with the request (GET /truth/$TRUTH_PUB). The server would then notify the user that per will receive some token out of bounds. After that, the user would have to provide for example the PIN sent to them via SMS with the same request as before (GET /truth/$TRUTH_PUB?response=$RESPONSE).

3 Wireframes

In this section, we show some of the wire frames to illustrate how we currently imagine the user interaction to happen. We begin by the user's journey towards the backup.

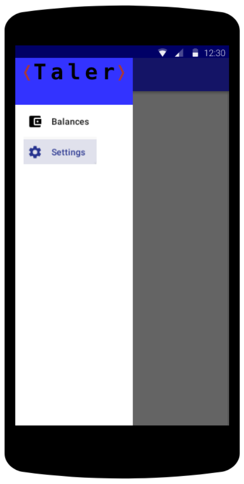
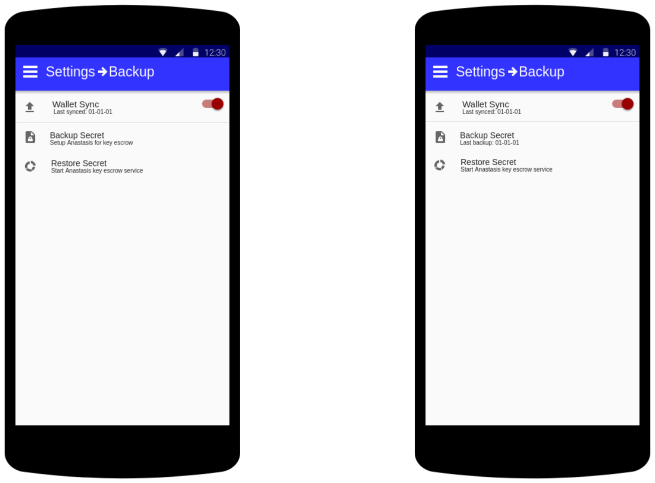


Figure 3: The user finds the “settings“ dialog of their application.



Wallet Sync
Last synced: 01-01-01

Backup Secret
Setup Anastasis for key escrow

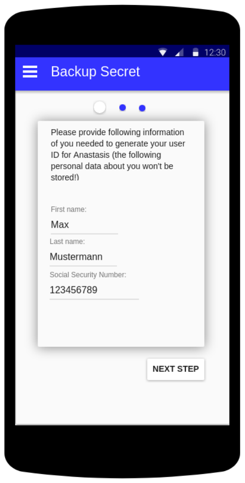
Restore Secret
Start Anastasis key escrow service

Wallet Sync
Last synced: 01-01-01

Backup Secret
Last backup: 01-01-01

Restore Secret
Start Anastasis key escrow service

Figure 4: The settings dialog has a place to begin secret backup or recovery. In the case of a Taler wallet, this is only available if "sync" is enabled as without sync, a secret that could be backed up does not make sense. Similar considerations would apply to E-mail clients where key material must first exist before it can be backed up. If a backup does exist, the dialog shows when the backup was last created.



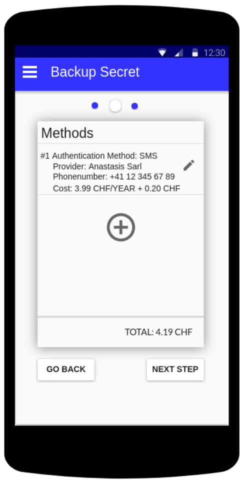
Please provide following information of you needed to generate your user ID for Anastasis (the following personal data about you won't be stored!)

First name:
Max

Last name:
Mustermann

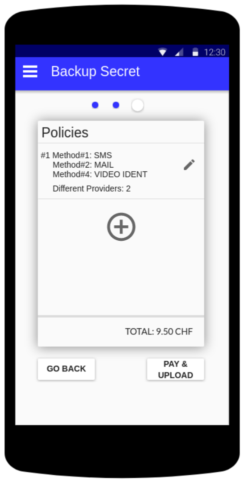
Social Security Number:
123456789

Figure 5: First, the user must provide identity information about themselves, so we can create their identity key.



#1 Authentication Method: SMS
Provider: Anastasis Sarl
Phonenumber: +41 12 345 67 89
Cost: 3.99 CHF/YEAR + 0.20 CHF

Figure 6: Next, the user adds authentication methods. For those the user has already added, a summary is shown.



#1 Method#1: SMS
Method#2: MAIL
Method#4: VIDEO IDENT
Different Providers: 2

Figure 7: Finally, the user configures the recovery policies. As before, policies that are already in place are shown.

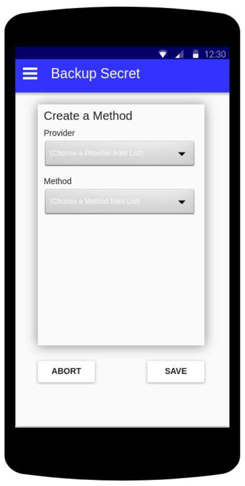


Figure 8: When adding a new policy, the user selects the authentication methods (from those added previously) to be used for the new policy.

For the recovery process, the first steps are basically identical. Once the user has added the information that allows the application to recover their identity key, the user can then begin to satisfy the challenges for recovery (SMS, e-mail, security question).

There, standard dialogs are used where the user only has to enter the PIN/-TAN code they have received, or given sufficient permissions we may ideally even simply intercept the SMS message or e-mail (if recovery is done by the e-mail application or on a mobile phone).

Once a recovery policy is satisfied, the secret is restored and control is returned to the application.

What is still *unclear* to us is how we will best design the interaction when the user suspends and resumes the recovery activity, say because they need to wait for physical mail to arrive. While suspending is relatively easy (user closes app or maybe add a "continue later" button), resuming is more tricky. Do we allow multiple recovery operations (possibly for different secrets!) to proceed in parallel? (Are there applications where this makes sense?) Do we simply resume if a recovery was in progress when the user selects resume after suspending? Do we have an open reminder / notification about the pending key recovery operation?

Another fine point is payment. If Anastasis is done for a payment app (Taler wallet, cryptocurrency), we can simply use the standard payment dialog of the respective App. However, for other clients we may need additional payment dialogs, and thus an interaction with an external payment app (or force the user to enter payment details). So payment may need to be tailored to the specific application.

A further complication could be payment for data recovery when the wallet is empty. So if providers charge for recovery, and the user needs to first recovery to pay, we have a bit of a circular dependency here. Alas, it could be resolved by simply forcing the user to first load their wallet with a small amount of money and then pay to recover their secret key.

4 Tech and architecture

The whole Anastasis application consists of multiple components. Figure 9 gives an overview over all the components.

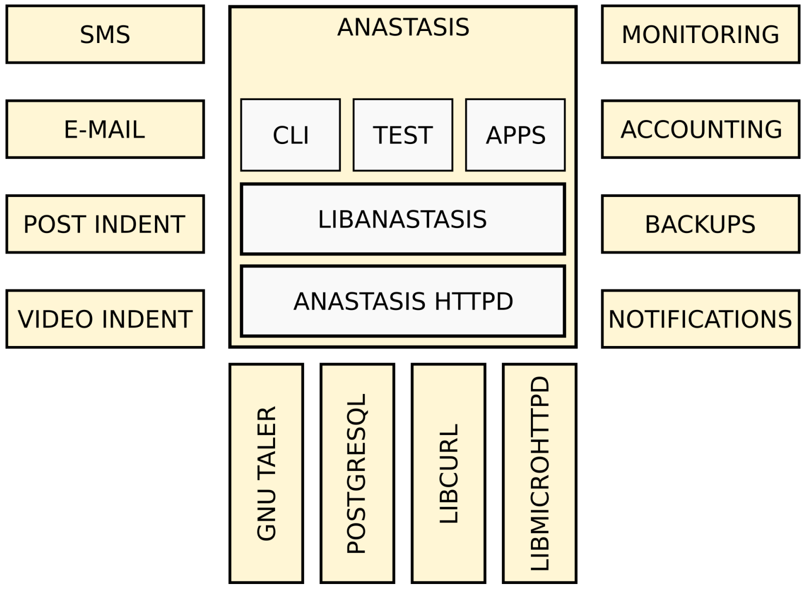


Figure 9: System architecture overview

In the center is the core implementation of Anastasis. On the left are some of the planned authentication methods from the application. On the right side of the box are the core parts which are necessary to operate Anastasis commercially. These parts are anticipated for a production deployment, but not part of the implementation for this thesis.

At the bottom section are the external libraries used for the project.

4.1 System architecture

This graphic shows the basic architecture of the Anastasis application. It shows a simplified flow of the application. The details of each component are explained later.

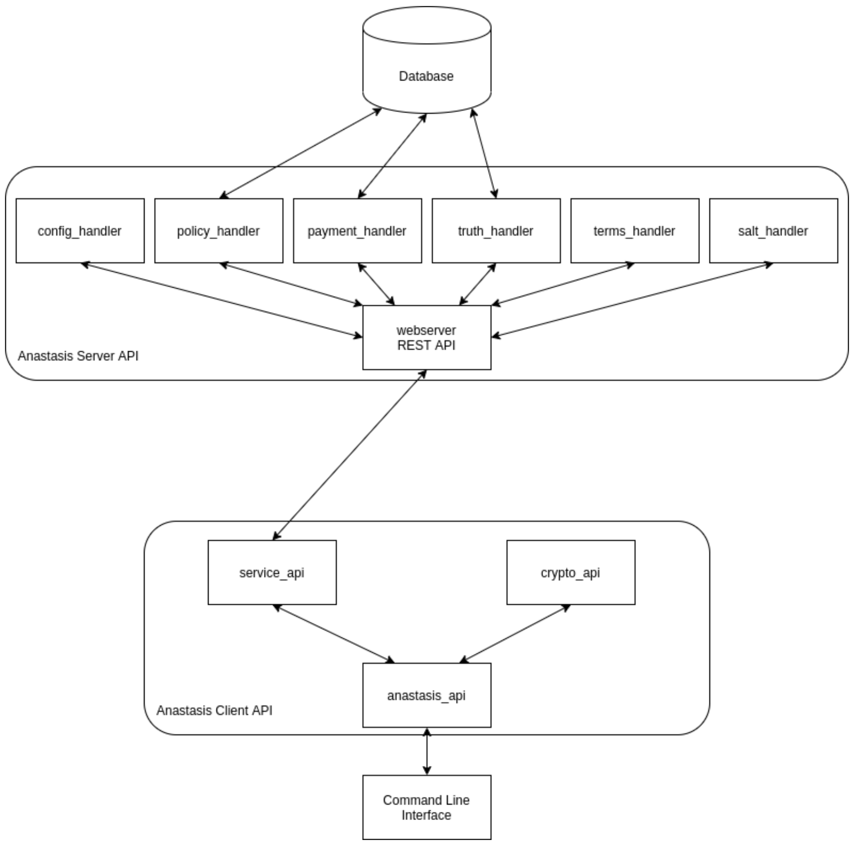


Figure 10: System design overview

1. The Anastasis CLI interacts with the Anastasis API. The Anastasis API is responsible for triggering interactions with the user, and also manages the interactions between the various client-side components.

2. After the user provided their unforgettable secret, the Crypto API derives the needed key material for the further communication. This is simplified, in reality the client would first need to download the server salt to generate the user keys. The crypto API is later also responsible for the decryption and encryption of the data, sent or received from the server.

3. The Service API is responsible for the communication with the Anastasis server. The Anastasis API sends the previously generated data and the user selected request to the service. The Service API is also responsible to handle the server's response to the request.

4. The central webserver logic handles HTTP requests sent to it by the clients. It will dispatch requests to the corresponding handler. The web-

server's core logic also returns the response and the status code of the operation to the client application.

5. Each REST endpoint of the Anastasis server is implemented by a specific handler. The handler processes the requests, typically by storing or looking up the requested data with the database. When the request is finished, the handler will send back the data or the status code to the webserver's core logic.

4.2 Server architecture

The Anastasis server architecture consists of two components. A web server with a REST API and a PostgreSQL database. The structure of these two components is shown in Figure 11.

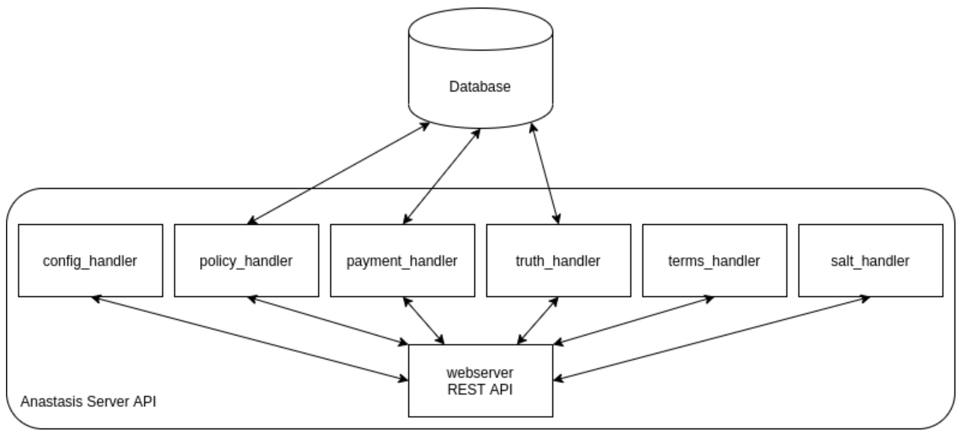


Figure 11: Anastasis server architecture

The webserver of Anastasis provides a RESTful API.

4.2.1 Database

The database schema of Anastasis is shown in Figure 12.

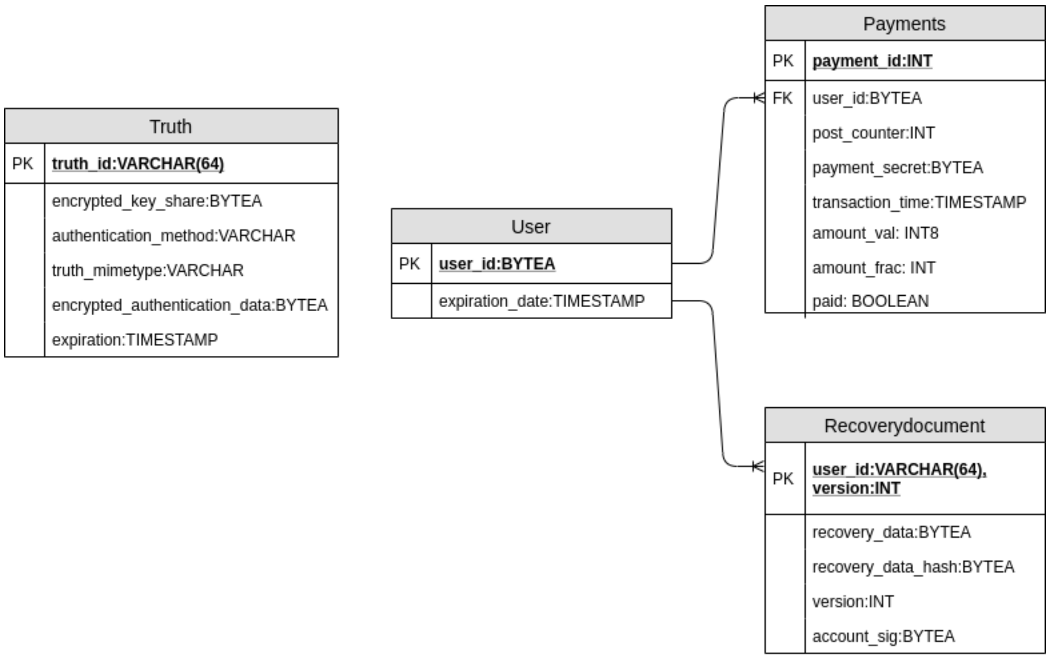


Figure 12: Anastasis database schema

The database schema consists of four main tables:

- The *Truth* table is responsible for storing the key shares and its authentication method. The key share and the authentication data are stored encrypted in the database. The authentication data is only decrypted during authentication. The key share is never decrypted for the server. This protects the privacy of the customer. Likewise, the user data is protected after a possible theft.
- The *User* table contains the identification of the user and an expiration timestamp. This timestamp is a subscription period. This timestamp is updated after each payment the user makes. Users for whom the subscription has expired are periodically deleted.
- The *Payments* table contains the details of a payment from a user. The payment is used either for the post-counter or the subscription. The post-counter is decremented after each upload of a recovery document. The user can only upload the recovery document if the provided payment contains a post-counter which is at least 1. Through this measure we can prevent people from maliciously filling our database.

- The *Recoverydocument* table contains the recovery information. The recovery document is stored encrypted in the database. This offers better protection, as explained earlier for the Truth table. Each recovery document record also contains a version, a hash of the recovery document and a signature. The version attribute allows the user to lookup a specific version of the document. The hash is used to check if the user uploads a duplicate of the document. The signature attests the integrity of the recovery data.

4.2.2 Authentication methods

This section describes an overview over the different possible authentication methods for Anastasis. In our implementation only the secure question is implemented. The other methods are just explained how they would be implemented.

In all cases, the authentication process begins by the user decrypting their (encrypted) recovery document, which contains a list of Anastasis providers, associated authentication methods, truth_seeds and associated truth encryption keys. The recovery process than varies slightly depending on the authentication method.

SMS (sms) The user tells the server with a request that they wants to authorize key recovery (via GET /truth/$TRUTH_PUB), providing a way to decrypt the truth with the phone number. The server will then generate a $PIN and send it via an SMS provider to the stored number in the truth object. The client then must send another request with the sent $PIN (via GET /truth/$TRUTH_PUB?response=$PIN). The server can now check if the two PINs match. Upon success, the server returns the encrypted key share.

Video identification (vid) This method allows the user to identify via videocall. Since the respective images must be passed on to the video identification service in the event of password recovery, it must be ensured that no further information about the user can be derived from them. Hence, the user's client software must try to delete metadata that could result in accidental information leakage about the user from the image before encrypting and uploading it to the Anastasis provider.

For recovery, the user must first send a request to server that they wants to authorize recovery (GET /truth/$TRUTH_PUB). The Anastasis provider will then decrypt the user's image and send a request with a $TOKEN to a video authentication provider that a user wants to authenticate, and respond to the user with a link to a video conference. The video authentication provider then checks via video conference that the user in the image is the same that they have on the video link. Upon success, the video provider releases the $TOKEN to the user. The client then must send another request with the $TOKEN (via GET /truth/$TRUTH_PUB?response=$TOKEN). The Anastasis provider checks that the tokens match, and upon success returns the encrypted key share.

Post identification (post) The user tells the Anastasis provider with a request that they want to authenticate using Post identification (GET /truth/$TRUTH_PUB). The Anastasis provider uses the request to decrypt the user's truth to determine the user's postal address, and sends them letter containing a $PIN. Upon receiving the letter, the client then has to send another request with the $PIN (GET /truth/$TRUTH_PUB?response=$PIN). The server can now check if the two PINs match. Upon success the server will release the encrypted key share.

Security question (qa) The user provided Anastasis with a secure question and a (normalized) answer. The secure question becomes part of the encrypted recovery document, and is never disclosed to weak adversaries, even during recovery. The encrypted truth on the server only contains a (salted) hash of the answer. The Anastasis provider cannot learn the plaintext answer. Because of the salt, and it cannot mount a confirmation attack either.

If the user wants to recover the key share from the server, they must provide the (salted) hash of the answer to the security question (via GET /truth/$TRUTH_PUB?response=$HASH). The server then checks if the stored and the provided hash match. Upon success the server responds with the encrypted key share.

4.3 Client architecture

The Anastasis client architecture consists of two main components. A client API which communicates with the server and a command line application which interacts with the user. The structure of these two components is shown in Figure 13.

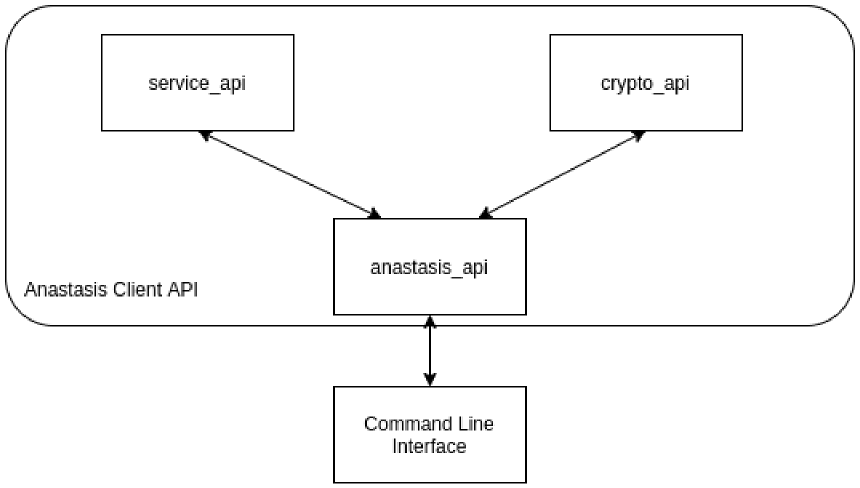


Figure 13: Anastasis client architecture

The Anastasis client implementation includes three distinctive APIs: a *Crypto API* which provides the different cryptographic functions, a *Service API* which sends the request to the server and the *Client API* which manages the main data structures and provides an abstraction for the application.

4.3.1 Crypto API

The most important data structures in the crypto API are the following:

- The kdf_id is a hash code which was generated with Argon2. The entropy source is the user's unforgettable secret. The kdf_id is used to create various key's.

  ```
  struct kdf_id
  {
    Hashcode; //512-bit
  }
  ```

- The account_private_key is used to sign the data and check the signature later. It is a 256-bit EdDSA private key. It is generated with the kdf_id as entropy source.

  ```
  struct account_private_key
  {
    eddsa_private_key;
  }
  ```

- The account_public_key is used as the user identification on the different providers. It is generated from the private_key.

  ```
  struct account_public_key
  {
    eddsa_public_key;
  }
  ```

- The truth_key is a randomly generated AES-256 GCM key. It is used to encrypt the user specify data in the truth object.

  ```
  struct truth_key
  {
    key; //256-bit
  }
  ```

- The truth_seed is a randomly generated nonce with a size of 32 Bytes. It is used to derive a truth_private_key and is stored within an encrypted recovery document.

  ```
  struct truth_seed
  {
    nonce; //256-bit
  }
  ```

- The truth_private_key is used to sign the encrypted key share and the encrypted authentication data. It is a 256-bit EdDSA private key. It is generated with the truth seed as entropy source.

  ```
  struct truth_private_key
  {
     eddsa_private_key;
  }
  ```

  The truth_public_key is used as the user identification on the different providers in case of uploaded truths. It is generated from the truth private key.

  ```
  struct truth_public_key
  {
    eddsa_public_key;
  }
  ```

- Anastasis needs different symmetric keys to encrypt data for example, the recovery document. These symmetric keys are all 256-bit large hashcodes. These symmetric keys are generated through the key routine defined in Implementation Key usage.

  ```
  struct symmetric_key
  {
    hashcode; //256-bit
  }
  ```

- Each policy has a separate policy_key. The key is used to encrypt the master_key. The policy_key is also a AES-256 GCM key. It is generated through the combination of a set of key_shares.

  ```
  struct policy_key
  {
    hashcode; //256-bit
  }
  ```

- Every truth object contains a key_share. A key_share is a 256-bit random generated bit sequence.

  ```
  struct key_share
  {
    hashcode; //256-bit
  }
  ```

- Before every encryption a random 256-bit large nonce is generated. This gives the encryption algorithm a random factor.

```
struct nonce
{
  hashcode; //256-bit
}
```

- To use AES-256 GCM an IV must be generated. It is generated with an HKDF over a salt the kdf_id and a symmetric key.

  ```
  struct iv
  {
    hashcode; //128-bit
  }
  ```

- The aes_tag is generated after each encryption, it is later used to check the integrity of the data.

  ```
  struct aes_tag
  {
    hashcode; //128-bit
  }
  ```

The functions of the crypto API basically provide the canonical set of cryptographic operations (hash, encrypt, decrypt, etc.) over these basic data structures.

4.3.2 Client API

The most important data structures in the client API are the following:

- The secret share data structure is used to upload a new recovery document.

  ```
  struct secret_share
  {
    kdf_id;
    last_etag;
    policies;
    core_secret;
  }
  ```

  - kdf_id: is used to compute the account public and private key. The hash is 512bit large.
  - last_etag: this hash is sent with the recovery document. The server will check the hash if the document on the server is the same. This prevents unnecessary uploads. The hash is 512-bit large.
  - policies: is a list of all generated policies the user wants to combine into a recovery document.

  - core_secret: is the user provided core secret. This is just a binary blob so Anastasis does not have a restriction for the user secret. This could be a for example a private key or a password the user wants to backup.

- The recovery information data structure holds a recovery document. It is downloaded within the recovery process and stored inside a recovery data structure.

```
struct recovery_information
{
  struct decryptption_policies;
  struct challenges;
  version;
  salt;
}
```

  - decryption_policies: holds all available policies within the downloaded recovery document.
  - challenges: holds all available authentication methods within the recovery document.
  - version: the version of the downloaded recovery document is stored here.
  - salt: this is the salt used for the generation of the policy keys. The salt is a 512-bit value.

- The recovery data structure is generated at the start of a secret recovery. It contains all available policies and lists which challenges are solved. Through this struct the client can check if a policy was solved completely.

```
struct recovery
{
  kdf_id;
  version;
  provider_url;
  salt;
  solved_challenges;
  struct recovery_information;
}
```

  - kdf_id: is used to compute the account public and private key. The hash is 512bit large.
  - version: hold the user desired version he wishes to download. This can be null then the client downloads the latest version.
  - provider_url: the client will download the recovery document from this provider url.

  - salt: this is the salt of the provider specified in provider_url.
  - solved_challenges: this is a list of all solved challenges. This list is updated after each successful authentication. This allows the client to check if a policy is solved.
  - recovery_information: as previously mentioned this data structure holds the downloaded recover document to process within the recovery

- A truth data structure is used to upload a new authentication method to a provider. It is identified by the TRUTH_PUB which the user creates through a HKDF over the truth_seed. The truth data structure is only used for the secret share process and not for the recovery.

```
struct truth
{
  truth_seed;
  method;
  mime_type;
  encrypted_truth;
  encrypted_key_share;
}
```

  - truth_seed: the truth_seed is the identification of the truth object. It is used as entropy source to generate the TRUTH_PUB, which later identificates the truth object. The truth objects are not linked to the user. A list of these truth_seeds are stored inside the recovery document, with this the user data is more anonymous.
  - method: this defines which method the user chose to configure, for example SMS, email, secure question.
  - mime_type: this defines in which format the truth was safed, for example jpeg, png, txt, json.
  - encrypted_truth: the encrypted truth holds the authentication specific data. It holds for example the hashed answer and the question. It is encrypted with the specific truth_key which is stored inside the recovery_document.
  - encrypted_key_share: this is the key_share protected by this truth. It is encrypted with a key which was derived with the kdf_id of the user. The server will later send this key_share to the user upon successful authentication.

- The policy data structure is used to create new policies to combine them into the recovery document. The policy data structure is only used for the secret share process.

  ```
  struct policy
  {
   truths;
   policy_key;
   salt;
  }
  ```

  - truths: every policy has a set of truths which need to be solved to recover the policy_key
  - policy_key: the policy_key is created through the combination of the different key_shares within each of the truth objects. It is later used to encrypt the master_key.
  - salt: defines the salt used to create the policy_key.

- The decryption_policy data structure is used in the recovery process. It has slightly different values as the policy structure.

  ```
  struct decryption_policy
  {
    truth_seeds;
    encrypted_master_key;
    salt;
  }
  ```

  - truth_seeds: is a list of truth_seeds which need to be solved to recreate the policy key. Each truth_seed has a corresponding challenge.
  - encrypted_master_key: holds an encrypted version of the master_key which was used to encrypt the core secret. In every policy lies the same master_key which was encrypted by the specific policy_key.
  - salt: defines the salt which was used to create this policy_key.

- The challenge data structure is used for the several key_share lookups. We named the process of authentication on the providers as challenges. It has slightly different variables as the truth data structure.

  ```
  struct challenge
  {
    truth_seed;
    url;
    truth_key;
    method;
    key_share;
    instructions;
  }
  ```

  - truth_seed: Entropy source to generate the TRUTH_PUB, which identifies the challenge on the server.
  - url: defines the provider URL on which the truth was stored.
  - truth_key: this key is sent to the server within the authentication procedure. The server can decrypt the truth with this key to start the authentication.
  - method: defines the method of this challenge, for example email, SMS, secure question.
  - key_share: After each successful authentication the key_share which was sent by the server will be saved within this variable. It is later used to recreate a policy_key.
  - instructions: this contains a string with the instructions for the user. This could for example be:" What is your favourite colour?" or" An SMS was sent to the number +41...... please provide the pin".

The functions of the client API basically provide a way to backup a core secret by providing user's identity attributes, the secret and constructing the policies, as well as a way to recover a core secred by providing the user's identity attributes and then satisfying the authentication challenges.

4.3.3 Service API

The service API is responsible for sending the requests to the REST API of the server. The client has implemented functions for every endpoint.

4.4 Technology choice

The Anastasis backend is being written in C. We decided to use C because of the various dependencies, including cryptographic libraries. Especially, GNU Taler and Sync, which are working in concert with Anastasis, are also written in C. Using the same language makes integration and testing of Anastasis much easier.

For the Anastasis client, the first UI prototype will be done in C as that will make it easy to re-use existing code fro GNU Taler to interact with the backend, and to write unit tests for the backend.

The second client will be written in TypeScript, so as to facilitate the integration of Anastasis with applications based on WebExtensions (like the GNU Taler wallet) and other software that is using NodeJS.

4.5 Cryptography overview

The Figure 14 shows the legend for the illustration of the Anastasis key usage shown in Figure 15 on page 27 and in Figure 16 on page 29. The Figure 15 gives an overview of the keys used in Anastasis. It also shows how they are created and used. Figure 16 shows how the keys to sign the (encrypted) truth data used during authentication are generated. The truth seed(s) used in Figure 16 are part of the recovery document.

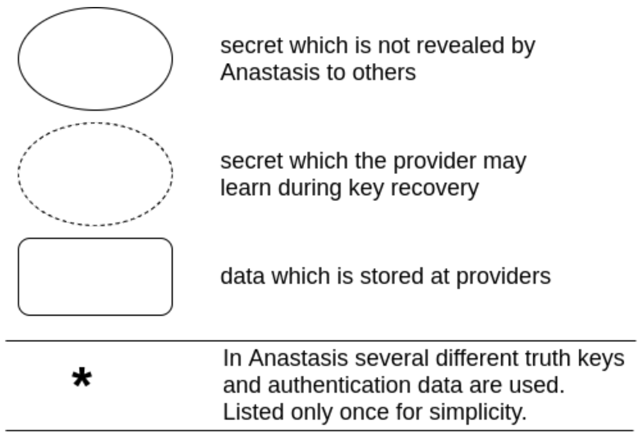

secret which is not revealed by Anastasis to others
secret which the provider may learn during key recovery
data which is stored at providers
*
In Anastasis several different truth keys and authentication data are used. Listed only once for simplicity.

Figure 14: Legend of Figure 15 on page 27

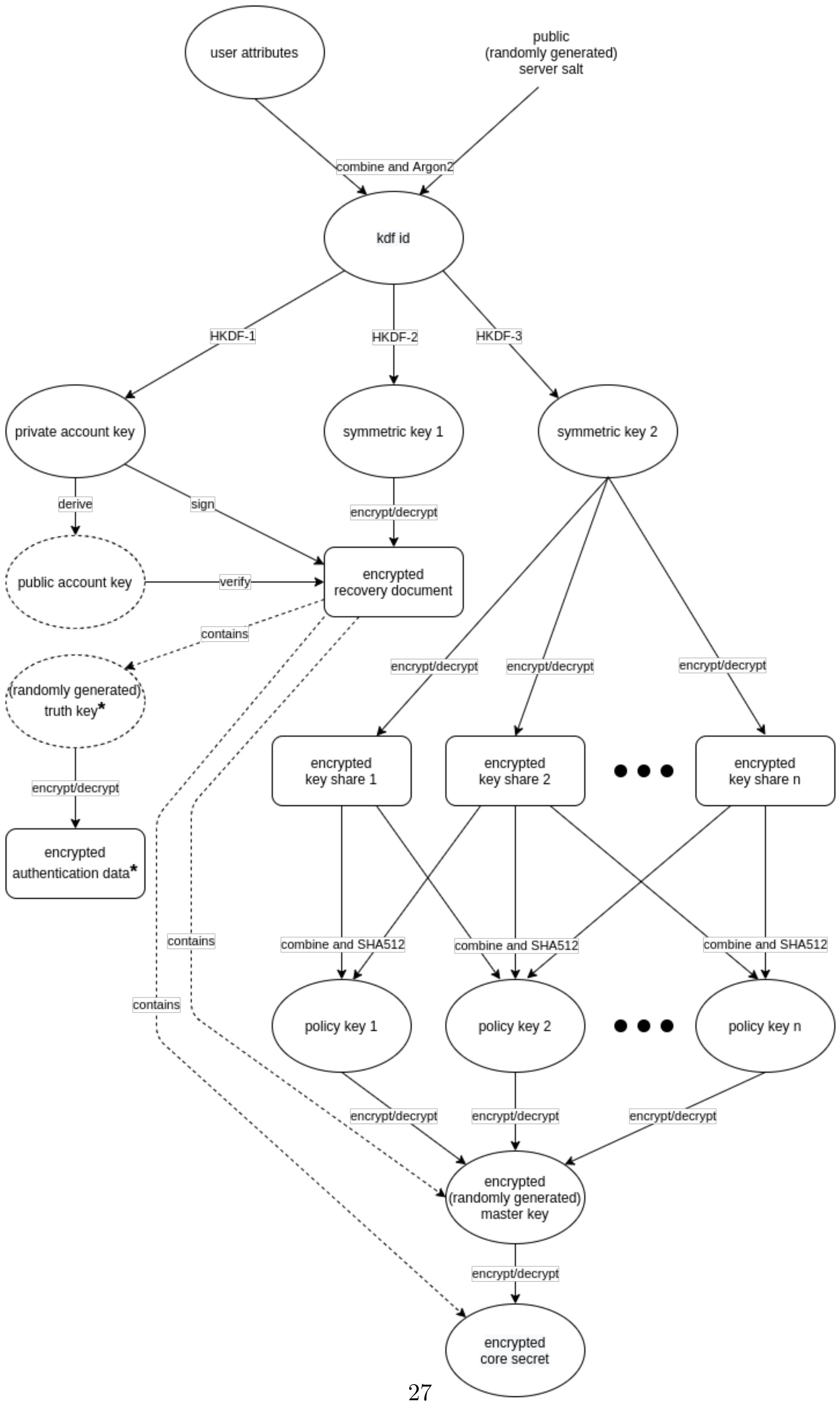


Figure 15: Secrets used in Anastasis

In the following the keys shown in the Figure 15 on page 27 are explained:

kdf id The *kdf id* is derived from the user attributes and a randomly generated public and constant salt value provided by the escrow provider using Argon2. It is used to derive the *private account key*, the *symmetric key 1* and the *symmetric key 2*.

private account key The *private account key* is used to sign the *encrypted recovery document*. It is derived from the *identity key* using *HKDF-1*.

public account key The *public account key* is derived from its corresponding *private account key*. It used to verify the signature of the *encrypted recovery document* and also is the identifier of the user which is needed by the provider.

symmetric key 1 The *symmetric key 1* is derived from the *identity key* using *HKDF-2*. It is used to encrypt and decrypt the *encrypted recovery document* which is stored by the provider.

symmetric key 2 The *symmetric key 2* is derived from the *identity key* using *HKDF-3*. It is used to encrypt and decrypt the different *encrypted key shares* which are stored by the escrow providers.

truth key The *truth key* is randomly generated for each *encrypted authentication data* and is stored within the *encrypted recovery document*. It may later be disclosed by the user to the escrow provider to let it decrypt the *encrypted authentication data* which allows the provider to then run the recovery authorization process.

master key The *master key* is randomly generated and is used to encrypt and decrypt the *encrypted core secret* which is stored within an *encrypted recovery document*. The *encrypted master key* also is stored within the *encrypted recovery document*.

policy key The *policy keys* are used for encryption and decryption of the *encrypted master key*. A *policy key* is constructed by hashing a specific combination of *key shares* specified by the user. For hashing SHA512 is used here.

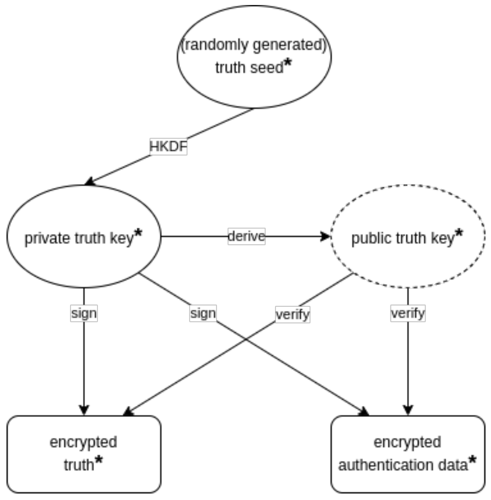


Figure 16: Key generation for signing of encrypted “Truth” data in Anastasis

In the following the keys shown in the Figure 16 on page 29 are explained:

truth seed Clients generate a random *truth seed* for each truth which is stored in the encrypted recovery document.

private truth key *Private keys* are derived per truth upload. They are used to sign the uploaded data. This way, the escrow provider can later prove that they preserved the data correctly. We use EdDSA for the signatures.

public truth key *Public keys* are used to identify the truth in the provider’s database. Providers only store the first truth upload with a valid signature. Changes to truth are thus not possible, clients must create a fresh seed for every upload.

4.6 Dependencies

In this section the dependencies used by Anastasis are presented.

4.6.1 GNU Taler

GNU Taler is one of the main reasons why we started to implement Anastasis, since the application needs a system to back up the private keys of their users. “GNU Taler is a privacy-preserving payment system. Customers can stay anonymous, but merchants can not hide their income through payments with GNU Taler. This helps to avoid tax evasion and money laundering.”

To operate GNU Taler the user needs to install an electronic wallet. Backups of the wallet are secured with a secret key. Here comes Anastasis into play, Anastasis will secure this secret key for the user.

In our implementation GNU Taler is also our payment system. We decided to use GNU Taler because both Anastasis and GNU Taler are privacy preserving applications. If we for example used credit cards for payments the user would no longer be anonymous which is helpful for the security of Anastasis as it allows us to use the user's name in the user's identity attributes. GNU Taler is also a GNU package and Free Software.

4.6.2 PostgreSQL

PostgreSQL is a Free/Libre Open Source object-relational database. PostgreSQL has over 30 years of active development which makes it a stable and reliable software.

We use PostgreSQL as our database on the Anastasis server. We decided to use PostgreSQL because it is an open source and lightweight software which has a big community. This means there are a lot of helpful documentations and forums.

4.6.3 Libcurl

Libcurl is a libre URL transfer library. Libcurl supports a wide range of protocols and a C API. Libcurl is also ready for IPv6 and SSL certificates.

For Anastasis we use Libcurl to generate the client-side HTTP requests. We decided to use Libcurl because it is also written in C and free software. The software is also well supported and has a good documentation. This makes the integration in our application easy.

4.6.4 GNU Libmicrohttpd

GNU libmicrottpd is a small C library which provides an easy way to run a HTTP server. We use GNU Libmicrohttpd in Anastasis to provide a simple webserver. The main reason why we did not use apache or nginx is that we do not need a standalone webserver. The Anastasis webserver just must handle some API requests, a standalone webserver is not needed for that and would make the infrastructure more complex to maintain and develop. GNU Libmicrohttpd is also a GNU package and Free Software.

5 Research plan

Research challenges include:

- How to minimize the complexity of policy setup for beginner users, while providing adequate flexibility for power-users. This will involve finding an algorithm that solves a complex constraint system "nicely" to provide a sane default policy.
- Integrating payment options and aligning business plan with steps where users would realistically pay.

- Determine set of user-attributes for various countries that users can be realistically expected to provide.
- Overall user interaction design to ensure users understand what they are doing, and ideally fully comprehend which data they share with whom when.

Feature list

This is an extensive list of features we eventually would like to see. We do not envision implementing all of them during LEDGER.

- Authentication methods to be supported by the back-end:
  - Security question
  - E-mail
  - SMS
  - Physical mail
  - Video-identification
  - ...
- Payment methods supported by the back-end:[1]
  - GNU Taler
  - Credit cards
  - Cryptocurrencies
  - ...
- Available user-facing front-ends:
  - Command-line interface
  - Graphical user interface
  - Integration with third-party applications:
    - GNU Taler wallet
    - p≡p e-mail application(s)
    - NYM-tech wallet
    - Re:claimID
    - ...
- Authentication methods supported by front-ends (per front-end):
  - Security question

[1] We may also separate the payment method logic from the Anastasis back-end. So this is about the business being able to process payments, not us implementing direct support.

  - E-mail
  - SMS
  - Physical mail
  - Video identification
  - ...

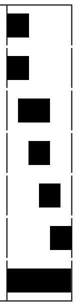
- Payment methods supported by front-ends (not all make sense for all front-ends):

  - GNU Taler
  - Credit cards
  - Cryptocurrencies
  - ...

- Supported countries (in backend, in front-end, and for payment, and translation of GUIs to dominant languages of the target country).[2]

6 GANTT

Our objective for the next 6 months is for Anastasis to implement several authentication services, have a working cloud deployment with good monitoring, and to come with a GUI for demonstration as well as being integrated with the Taler wallet as a first MVP. Furthermore, we would like to have some business leads and funding for further integration of Anastasis with other products. A GANTT chart highlighting when we expect the various tasks to be done is shown in Figure 17.



1.	Implement core logic (backend&frontend)	M0–M2
2.	Deploy 1st backend (security-question only)	M0–M2
3.	Write Gtk+ GUI (stand-alone App for demonstration)	M1–M4
4.	Deploy 2nd backend (SMS + E-mail authentication)	M2–M4
5.	Integrate with Taler wallet (backup & recovery)	M3–M5
6.	Integrate payment support	M4–M6
7.	Business development (clients, partnerships, fund-raising)	M0–M6

Figure 17: Gantt chart.

[2]Note that we need to define per-coutry user-attributes and implement per-attribute validation logic, so this is not just a business or payment method support issue.

6.1 Milestones

July 2nd The architecture and cryptography presented in this document reflects the core logic implemented in our generic backend and frontend code. Wireframes with a first idea for the GUIs have been created. LEDGER onboarding and resource planning (management tasks) are complete.

September 2nd Two Anastasis backends are operational with at least two authentication methods. A Gtk+-based GUI can be used for backup and recovery. Profiles for at least two countries exist.

November 12th Three authentication methods are supported, satisfying the business requirements from Taler Systems SA. Anastasis has been integrated into the Taler wallet, creating an integrated application usable by consumers. The backends and the Gtk+ GUI support payment for the service using at least one payment method.

6.2 Task details

This section gives detailed sub-tasks for each of the key tasks from the GANTT chart.

6.2.1 Implement core logic

- Cryptography and REST APIs (done)
- Reducer logic (done)
- Automated policy generation (kind-of works, but generated policies are terrible)
- Country-specific user-attribte specifications (DE, US, IN, CH finished)

6.2.2 Deploy 1st backend

- Server purchased, OS installed, network configured (done)
- Provide Debian/Ubuntu packages (done)
- E-mail configured, Anastasis backend running (partially done)
- Latest backend also deployed by CI (not done)
- Database backups configured (not done)

6.2.3 Write Gtk+ GUI

- Rudimentary Gtk+ GUI (done)
- Recovery authentication method selection UX working (done)
- Reviewed UX with interaction designer (not done)
- Figure out how to pay for multiple years of storage/recovery (not done)
- Backup policy editing possible (not done)
- UX adaptations based on interaction designer feedback (not done)

6.2.4 Deploy 2nd backend

- Identify cloud provider (not done)
- Extend backend to handle E-mail and SMS authentication (not done)
- Database backups configured (not done)
- Setup Taler payment integration (not done)
- Setup system monitoring (not done)
  - disk utilization (not done)
  - CPU utilization (not done)
  - bandwidth utilization (not done)
  - service availability (not done)
  - backup operation (not done)
  - test recovery processes (not done)
- Document deployment (not done)

6.2.5 Integrate with Taler wallet

- Port core crypto logic to TS (not done)
- Port REST client logic to TS (not done)
- Port cryptographic construction to TS (not done)
- Port REDUCER-style API to TS (not done)
- Design mobile-appropriate UI (not done)
- Implement UI dialogs and integrate with reducer (not done)
- Integrate UI with rest of Taler wallet (not done)
- Support authentication methods (not done)

  - security question (not done)
  - SMS (not done)
  - E-mail (not done)
  - ...

6.2.6 Integrate payment support

- Determine non-Taler payment method and technical approach (not done)
- Discuss payment UX with interaction designer (not done)
- Support payment method in backend (not done)
- Support payment method in frontend(s) (not done)

6.2.7 Business development

- Review and revise business presentation (mostly done)
- Onboard additional partners interested in our solution (got some leads)
- Finalize business plan in alignment with technical realities (difficult)
- Find investors (optional, maybe we are good?)

6.3 Human resources

Vaishnavi Mohan will focus on the integration with external authentication providers and monitoring of our cloud deployment (2&4). Belen Barros Pena will work on the user interaction design for all user-facing components. Christian Grothoff will provide the first reference implementation in C (1&3&6). Florian Dold will integrate of Anastasis with the Taler wallet to create a first consumer application with integrated payment (5).

Berna Alp will focus on business development (7).

6.4 Financial resources

Estimated costs for the next 6 months:

- Staff costs (€ 104k)
- Subcontracting cost (€ 25.5k)
- Technology assets (€ 4k)
- Travel and other indirect costs (€ 10k)
- Sales & Marketing cost (€ 7.5k)

7 KPIs

Key performance indicators are:

- Number of steps users need to perform for backup
- Number of steps users need to perform for recovery
- Number of implemented features from the feature list

A Business Canvas

A.1 Key partners

Our key partners for Anastasis are three entities. First the business partners, Taler Systems SA and p≡p Foundation, with whom we could already make contracts and wish to integrate our product. Second are the providers of Cloud services. To operate Anastasis with minimal cost we need the service of these providers. These providers can additionally provide us authentication services, this also minimizes the complexity of our solution since we do not have to implement these services by ourselves. Such a provider could be for example Amazon AWS, Azure, Google.

A.2 Key activities

The main work of our start up is the completion of our software for commercial use. This involves the integration of different authentication methods and the integration of our application into the different consumer applications. Another key activity is the maintenance and deployment of our service.

A.3 Key resources

To operate our service, we will need servers to provide our service, as previously mentioned we would provide our service on a Cloud provider. We also need developers to integrate Anastasis with various authentication providers, payment solutions and applications needing key recovery.

Additionally, we need a person responsible for the business of Anastasis. This employee would be responsible to find new business partners and present our application to investors.

A.4 Value propositions

As mentioned earlier there are many applications which need a key recovery system. Anastasis is also a privacy friendly and transparent solution. Furthermore, Anastasis will make sure that the application is user friendly and inexpensive.

A.5 Customer relationships

In the early stages of our start-up our customers are primary going to be business customers like Taler Systems SA, p≡p Foundation, Fraunhofer AISEC and NymTech, which all want to integrate our solution into their products. Thus, early on we will likely pursue B2B sales, lining up businesses that would want to integrate Anastasis with existing security products.

Once successful products exist in the market, our revenue should inherently shift to a B2C model, as then customers will pay for the recovery service. We may then also ourselves invest in integration of Anastasis with further software solutions to grow the business, even in domains where there is no significant business partner. This will be the case for applications where popular non-commercial solutions are freely available. An example for this domain would be consumer software that enables disk encryption.

A.6 Customer segments

Anastasis serves two primary markets:

- B2B2C: E-money issuers pay Anastasis to offer service to consumers with wallets to satisfy their regulatory requirements (service must exist) or offer better security. These business customers will be primarily developers of security applications which need a way to enable end-users to securely backup end-user key material.
- B2C: Consumers pay Anastasis for safekeeping and/or recovery (subscription). End-users paying for the recovery service will be all users using privacy-enhancing technologies, where the putting the user in charge of their data also burdens the user with taking care of their private keys. Specific applications include payment services including crypto-currencies and end-to-end encrypted communication services.

Furthermore, some wallet operators may pay Anastasis to assist with the technical integration.

We envision the segmentation to result in the following offerings:

- B2B2C: yearly subscription model based on number of users:
  - Tier 1 — up to 1K users
  - Tier 2 — up to 10K users
  - Tier 3 — up 100K users
- B2C: yearly subscription model with three levels of service:
  - Standard – no service, no warranty
  - Premier – with warranty
  - Platinum – with warranty and support

A.7 Cost structure

The main cost for our start-up is the salary of our employees. However, the core shareholders of the team will do some of the work without drawing a salary.

To provide Anastasis as a service, we expect to make use of existing public Cloud services, which also cost a little bit.

A.8 Revenue streams

In the beginning, businesses like Taler Systems SA will pay us to operate an Anastasis server and to help them integrate our protocol with their software. Once we have many end-users utilizing Anastasis, they will have to pay directly for the service. The users have to pay a subscription fee and possibly additional fees for expensive recovery operations.

Model		Price
Subscription: End-User Standard		€ 10
Subscription: End-User Premier		€ 30
Subscription: End-User Platinum		€ 100
Subscription: Partner Company Tier 1	0–1,000 users	€ 5,000
Subscription: Partner Company Tier 2	1001–5,000 users	€ 25,000
Subscription: Partner Company Tier 3	5001–50,000 users	€ 50,000

Figure 18: Envisioned price structure.