Passtör[1]

Cedric Maire, Guillaume Michel, Xavier Pantet
Version 2.0

1 Introduction

According to TeleSign[2], 6 passwords are used to secure slightly more than 25 accounts on average. Moreover, no less than 68% of Internet users say that they want companies to provide an extra layer of security. We are offering to the billion people browsing the web everyday a new tool to better manage the security of their online accounts by letting them create unique and secure passwords they can easily retrieve anywhere on earth and at any given time. Just relax and enjoy safe browsing again with Passtör.

2 Related Work

2.1 KeePass

KeePass[3] is an open source password manager that was first developed only for Windows. It now has official support for macOS and Linux through the use of Mono[4] and, since it is open source, there exists a lot of unofficial native ports to many different operating systems such as iOS, Android and many more.

KeepPass is first a local-only password manager, which means that the passwords are only stored on the device itself (unless configured differently). Which also means that the user has the responsibility to back-up all his passwords such that he is guaranteed to not lose them. This task can become quite cumbersome for the average user since the backup has to be made and kept on a secondary disc or system such that the backup persists even if the first device has a failure that erases the passwords or makes them unavailable. Even if the user does these backups on a regular basis, recent password changes or passwords that were created after the last backup might be lost if a failure occurs.

With the creation of Passtör we want to remedy this problem of having to constantly backing-up the passwords to a secondary disc or device. Passtör will instantly share updated or newly created passwords in the network such that a client will be able to access his own passwords from anywhere in the world without having to worry about backups.

[1] Passtör = Password Manager + Peerster + Deutsche Qualität.

[2] `https://www.telesign.com/site/wp-content/uploads/2015/06/TeleSign-2FA-Infographic.pdf`

[3] `https://keepass.info/`

[4] `https://www.mono-project.com/`

2.2 Bitwarden

Bitwarden[5] is another popular open source password manager officially supported on Linux, macOS and Windows. It is different than KeePass in the sense that Bitwarden is cloud-synchronised, which means that all created passwords are uploaded[6] to Bitwardens servers. This means that the user no longer has the responsibility of backing-up all the passwords. This now happens instantly by synchronising each change with Bitwardens servers. Since Bitwarden is fully open source, one can also self-host the cloud-synchronisation and not upload them to Bitwardens servers.

Overall, this is an improvement of user experience compared to KeePass but introduces a central trusted entity (in the case one decides to use Bitwardens cloud). One does not need to backup his passwords but has to trust Bitwarden to not misuse the shared private data, to correctly implement and host the open sourced code and to not remove ones access to their personal account. If access is revoked, once still got a local copy but loses the ability to synchronise the database and has to setup the self-hosted solution and then again has to worry about backups.

This vicious circle of either trusting a central entity with our data and passwords or worrying about backups and availability is something we want to get rid of with the creation of Passtör. Passtör is distributed and hosted on many different nodes in a trust-less manner. The users do not have to worry about backups since the whole database will be distributed in the network and access cannot be revoked since even if a client is ignored by a node, the client will have the choice to choose a different one a regain access to all his passwords.

2.3 Apple Keychain

Apple Keychain is another example of password manager developed by Apple, as its name suggests, designed for smooth integration within the Apple ecosystem (Macs and iOS). It includes automatic replication of stored passwords across all devices linked to the same Apple account and the system is secured using traditional encryption mechanisms. As we might expect, the way Apple Keychain works is poorly documented and this is obviously one of its main drawbacks when it comes to trust. On the other hand, it makes it extremely easy for users to store passwords and use them in Safari or even in apps whichever device they are using and wherever they are located.

Such products demonstrated the clear feasibility of developing easy-to-use solutions to help people create, use and maintain safe passwords. However, the way they are implemented requires the users to fully trust the service providers. In particular, they have no guarantee that their passwords are stored in a safe manner, nor that the system cannot prevent them from accessing their data unless they agree to pay a ransom, for example.

In that sense, we claim that Passtör offers a better solution since users won't have to trust a single entity. Passtör aims at being an open-source project, hence every user is free to inspect the code and trust the system as a whole.

[5] `https://bitwarden.com/`

[6] `https://help.bitwarden.com/article/can-bitwarden-see-my-passwords/`

3 System Goals & Functionalities

3.1 System Goals

Our systems goal is to provide its users with a reliable and trusted password manager. The password manager is failure resistant in the sense that even if a few nodes fail or are not accessible, all users can still access their passwords. In the case a user looses a device, he/she should be able to easily recover all passwords.

3.2 Functionalities

The functionalities associated with the system goals are the following. To achieve reliability and trust for the password manager, we are going to make it... distributed! It will resist node failures or inaccessible nodes, because the data will be replicated and stored at different nodes. It should be possible for a user to query passwords from any node running the password manager. In the case where a user looses a device, he/she will be able to recover all passwords using his/her Master Password, and can get them from any node.

3.3 How we split the work

Cedric and Xavier took care of the cryptography side of the project. We did obviously not develop custom solutions but rather relied on existing implementations of trusted and commonly-used standards. Hashing and Key Derivations are performed using SHA3 and ChaCha20, respectively, from Golang's hashlib package while the crypto library provides us with building blocks for symmetric encryption using ChaCha20, digital signatures over elliptic curves (using ECDSA) and public-private key pairs generation.

Guillaume implemented a DHT with replication factor functionality to allow passwords to be replicated on different machines, without overloading all machines with all passwords from all users. The challenges were to make sure that the passwords are actually stored accordingly to the replication factor. They are easy to find, update thank to the routing provided by the DHT.

3.4 GUI

As a GUI, we decided to implement a simple TUI (Text-based User Interface). According to the KISS principle[7], we aim at keeping our interface minimal and clean with just elementary features. In a nutshell, users will be able to request, add, delete or replace a password for a given service. Decryption is performed only on the client side using the Master Password so that no data ever need to be decrypted on the Passtörs.

[7] Keep It Simple, Stupid

4 Cryptography

To improve the end users UX, we want them to only remember two things: their username (*ID*) and master password (*MasterPass*). As in all password managers, we assume that the master password will never be lost and is strong enough to resist traditional brute-force or dictionary attacks.

From *ID* and *MasterPass*, the client can (locally) derive what we call the client secret (*S*), computed as follows:

$$S = KDF(ID, MasterPass, Salt)$$

The salt is a random value that can be stored in clear. Its only purpose is to mitigate the risk of time-memory trade-off attacks such as rainbow tables. *KDF* is a Key Derivation Function. Many KDFs could be used (in principle any hash function, for example) and we decided to use Argon2[8] which was especially tailored for secure password hashing. The time and memory consumption of Argon2 can be tweaked to adapt to its environment and provide good security guarantees against brute-force attacks.

Again, we stress at this point that *MasterPass* shall never be recovered, hence the user may want to write it down on a piece of paper and store it in a safe. Indeed, *S* will be used as symmetric key to encrypt other keys. Therefore, without their secret, no client will ever be able to recover their signature and data encryption keys to read their passwords or perform updates.

In addition, the client also generates a triplet of keys $K = (pk, sk, k)$ which are respectively a public and private key for digital signatures, and a symmetric key used to encrypt and decrypt their data. Signatures will be useful to prove that data updates are legitimate in the nodes before actually writing them.

User data will be stored in a network of Passtörs using a DHT (see later). To make sure a user can recover their data anywhere, we need to store their credentials in that network in an encrypted manner, of course. What other Passtörs will see is a triplet $K' = (pk, sk', k')$ of keys where *pk* is the public (plaintext) signature key and the other values are computed like this:

$$sk' = Enc_S(sk)$$
$$k' = Enc_S(k)$$

These are the symmetric encryptions of the keys under *S*. Since no Passtör knows *S*, none is able to decrypt the private and symmetric keys, only the user will be able to perform this operation locally. Also note that *pk* is stored in clear, since we want all the nodes of the network to be able to verify the validity of the data they receive.

As encryption scheme, we decided to use ChaCha20[9], which is quite new and believed to be an efficient replacement for AES but it is clear that any other secure encryption scheme would have been fine to perform this task.

[8] https://en.wikipedia.org/wiki/Argon2

[9] https://www.cryptopp.com/wiki/ChaCha20

The data about users is stored in the network as a giant map, kept synchronized among peers using the DHT. The keys of this map represent the users themselves and are computed using the following formula:

$$mapKey = H(ID)$$

To avoid conflicts, we would need to require that usernames are unique in the system, which could be ensured if we considered usernames being of the form *user@instance* where *instance* is the instance where the user creates their account and has a unique identifier and each instance makes sure that no two users can have the same identifier when registering to them. Due to time constraints, this is not implemented in the current version of Passtör. Here H can be any secure hash function, in this project we decide to use SHA3.

The value associated to each entry contains the encrypted keys and data as well as salts and nonces in plaintext and a signature that validates the whole block of data.

The values themselves are also maps, where keys identify the service with its associated username. We compute the key as $H(service|username|k)$, where H is still SHA3. In this case, k is used as salt. The value is the password encrypted under k. Each account also contains a signature validating the data. As signature algorithm we use a standard variant of ECDSA called ed25519[10].

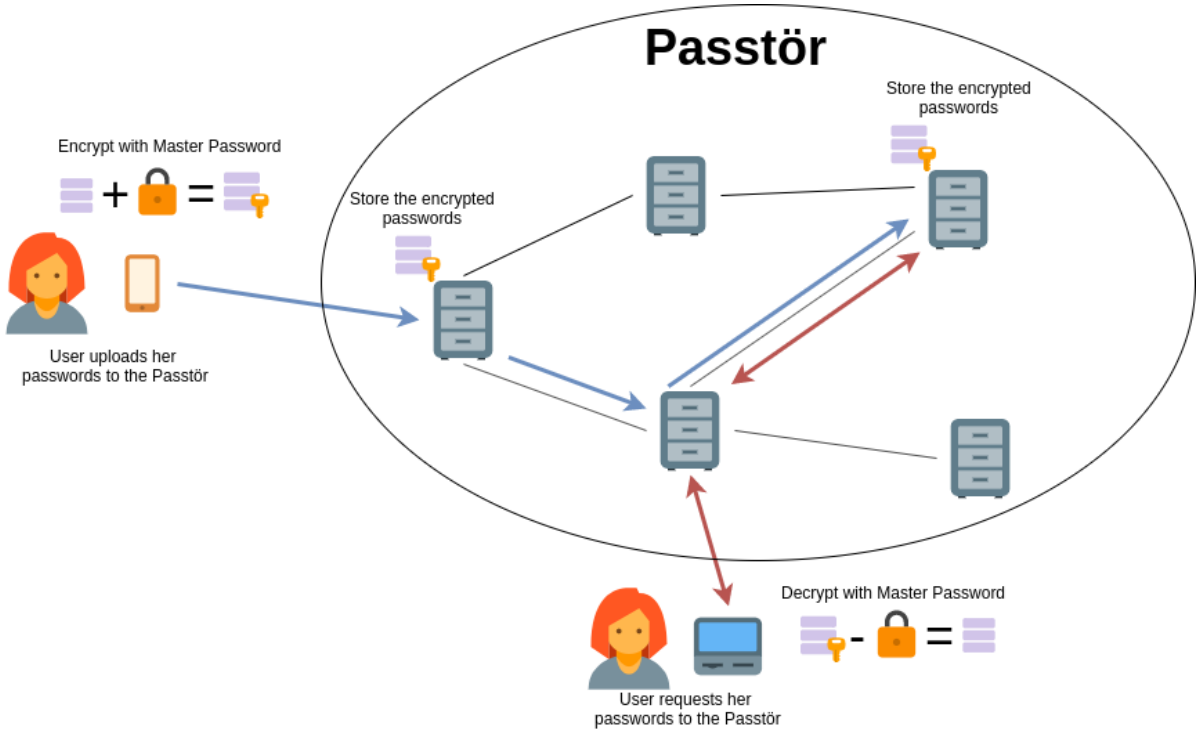

Figure 1: Passtör architecture design

[10]https://en.wikipedia.org/wiki/EdDSA

4.0.1 Data and Cryptography

Since we store all the data *publicly* we need strong guarantees against potential adversaries. The data is highly sensitive since it is composed of services with the corresponding usernames and passwords. We store all the data for all accounts in Go structures in encrypted form unless it has to be readable by anyone, like the public key to verify changes. We have the following data structures:

```
type Hash [HASH_SIZE]byte
type Salt [SALT_LENGTH]byte
type Nonce [NONCE_SIZE]byte

type PublicKey = ed25519.PublicKey
type PrivateKey = ed25519.PrivateKey
type SymmetricKey [SYMM_KEY_SIZE]byte

type Signature [SIGNATURE_SIZE]byte

type EncryptedData []byte

type Credentials struct {
    Username EncryptedData // Encrypted under k
    Password EncryptedData // Encrypted under k
}

type LoginMetaData struct {
    ServiceNonce  Nonce
    UsernameNonce Nonce
    PasswordNonce Nonce
}

type Login struct {
    ID          Hash
    Service     EncryptedData  // Encrypted under k
    Credentials Credentials
    MetaData    LoginMetaData
}

type Keys struct {
    PublicKey       PublicKey       // Plaintext
    PrivateKeySeed  EncryptedData   // Encrypted under S
    SymmetricKey    EncryptedData   // Encrypted under S
}

type AccountMetaData struct {
    SecretSalt          Salt // Plaintext
    PrivateKeySeedNonce Nonce
    SymmetricKeyNonce   Nonce
}

type Account struct {
    ID        Hash
    Keys      Keys
    Version   uint32 // Plaintext
    Data      map[Hash]Login
    MetaData  AccountMetaData
    Signature Signature
}
```

When a user register a new account it's *ID* is simply a hash of it's chosen username. This

does not guarantee much privacy as a user is expected to choose simple usernames like *Bob* or *Alice*, our system isn't built to guarantee much privacy. If one is able to find a pre-image he can discover how many passwords this user is storing and how many times they updated some information on their account. An adversary will also be able to recover the used salt per account, the salt will make a pre-computed dictionary attack more difficult but the security still depends on the master password. We recommend generating a random passphrase of 15 words from the *Oxford English Dictionary* which contains around 170'000 words[11]. This provides about the same security as a 256-bit key.

As one can see from the data structures we do not store much useful data in plaintext. Each *Account* has a *Version* attribute in clear text that is used to make replay attacks impossible, each node ever only accepts *correct* updates. A correct update is one that has a signature that verifies using the accounts public key, has a greater version number and has the same public key as the old version already stored on the node. As of now we do not support public and private key changes but could accept them if the update was signed with the old keypair. An account signature authenticates every single byte of the data such that no tampering is possible without breaking the signature. Nonces can be stored in plaintext.

5 Accounts storage

We use a Distributed Hash Table (DHT) as seen in class to store the encrypted accounts information. Our DHT takes inspiration mostly from Kademlia DHT [1], and allows replication of each encrypted account with a factor specific to each account. The accounts are distributed in the DHT, and thus each node needs to store only $\frac{\#accounts}{\#nodes} \times repl$ accounts on average, $repl$ being the constant average replication factor. Thus the required storage capacity is linear in the number of registered accounts per node which scales well with a large number of nodes.

The nodes forming the DHT interact over UDP. Each node in the system has a unique name, and is identified by the hash of this name, its *NodeID*. Thus the hash of all nodes name is evenly distributed.

The specificity of Kademlia is that it uses the XOR distance as distance metric between two hashes (e.g. $\delta(h_0, h_1) = h_0 \oplus h_1$). We use 512-bits identifiers for nodes and encrypted accounts. Each node keeps a list of k ($k = 5$ by default) node addresses and IDs for nodes at a distance between 2^i and 2^{i+1} from itself for $0 \leq i < 512$, thus forming 512 buckets of k-nodes. Buckets do not need to be full, small buckets will in practice even be empty. Whenever a node receives a message from a peer, it adds it to the corresponding bucket, if it is not already full. Those buckets implement a least-recently seen eviction policy, except that nodes answering to periodic ping requests are never removed from the list.

Our system provides strong consistency. Each update overrides the previous version of the data, thus there is no stale data in the system.

[11] `https://en.wikipedia.org/wiki/List_of_dictionaries_by_number_of_words`

5.1 Ping

We use a `Ping(node)` RPC as described in Kademlia to verify if a remote node is still alive. It is useful to replace dead nodes by alive ones in the *k-buckets*, and keep them up-to-date.

5.2 Lookup

The `Lookup(value)` RPC returns the k closest nodes to a given hash, relatively to the XOR distance. It is similar to the `FIND_NODE` as described in Kademlia. To sum up, when a node n performs a lookup for a value v it will first take the k closest nodes to v it knows of and insert them in a list l. Then, it will query all the peers in l, asking them for the k closest nodes to v. Once n got all the answers, it will add the peers in the answers to l. Then it will recursively query all the peers in l it didn't query before, until n has queried all peers in l. Then, n select the k closest peers to v in l to return them. The recursive step is done concurrently, multiple remote peers are queried in parallel to fasten the lookup process. The concurrency parameter α is a constant of the system.

5.3 Allocate

The `ALLOCATE(account, repl)` function allocates an encrypted `account`, identified by the `hash` of the user to `repl` peers in the DHT. It consists of calling `LOOKUP(hash)` to get the k closest peers to `hash`, and selecting the `repl` closer peers to `hash` to send them the `account` to store. This method is called to store a new account in the DHT or to override an existing one. The `account` is stored by the remote peers only if it is correctly signed. An override is accepted by the remote peer if the public key of the new version is the same as the public key of the old version, if it is signed correctly, and if the version identifier is higher in the new version than in the old version. This method only returns to the client once the encrypted account was stored on `repl` peers in the DHT.

5.4 Fetch

`FETCH(value)` can be called by a client to get the encrypted account information of the account identified by `value`. It first call `LOOKUP(value)`, to know where the file is stored, and then queries the nodes that are the closest to `value` for the `account`. It wait for $repl \times threshold$ identical replies before sending the `account` corresponding to `value` back to the client. *repl* is the replication factor of the given account, and $0 < threshold < 1$ is a system parameter defining the trust level, by default $threshold = 51\%$.

5.5 Republish

Each node periodically republishes every encrypted account it stores, as described in Kademlia. This allows churn in the system. Each file is republished on average every x minutes, x being a system parameter, $x = 5$ by default. Thus, each node storing a replica of an encrypted account will erase it locally and allocate it on average every $x \times repl$ minutes. This allow to handle churn

in our system. If a node storing a replica of a file disconnects from the system, the other nodes storing the same file will allocate it to another peer. If a new node n_5 joins the system and is closer to the identifier of an `account` than a node storing it n_0, that `account` will be allocated to n_5, and removed from n_0.

5.6 Joining the DHT

When new node n_6 joins the DHT, it needs to know the address of at least another peer in the DHT. It will then call `LOOKUP(own_identifier)` to fill its *k-buckets*. It will shortly get encrypted accounts allocations from remote peers, and is directly available for clients to store or fetch encrypted accounts.

6 Evaluation

Our system works perfectly as expected, and it doesn't really make sense to measure anything as we only intend to provide a secure distributed password manager. The security is the same as the security provided by the algorithms we use. And the performance of the DHT is the same as Kademlia. We will describe the time performance for simple operations and show beautiful screenshots in this section.

6.1 System performance

As passwords are mostly formed of short strings of characters (even if one uses very long passwords they rarely go beyond 64 characters), we can consider them as small messages, like tweets, even shorter.

The replication factor is used in order to ensure that passwords won't be lost in case of a failure and at the same time to spare resources and avoid that any node is overloaded with all the passwords of all the users. As for the users vision of performance he will be able to connect to any node and access all his personal passwords in matters of seconds.

Assuming that our system has a maximum delay of max_RTT between any two pairs of nodes. The DHT `LOOKUP` operation takes $\mathcal{O}(log\, N \times max\_RTT)$. Joining the DHT, consists in performing a `LOOKUP` for its own identifier, thus it is the same cost. When a client wants to `STORE` or `GET` it's encrypted account, it needs to reach a node ($\frac{1}{2} \times max\_RTT$), then the node has to `LOOKUP` for the peers to allocate or fetch the data $\mathcal{O}(log\, N \times max\_RTT)$. Send or get the encrypted account to or from the remote peer ($1 \times max\_RTT$), and send it back to the client ($\frac{1}{2} \times max\_RTT$). Thus at most $2 \times max\_RTT$ more than a `LOOKUP`. Reallocating an encrypted account is again a cost similar to `LOOKUP`.

The DHT handles churn in the system, the republish parameter (x) can be adapted for different networks. Thus the system can adapt to any churn level. If there is a lot of churn, and the system replication factor is small, the system may generate a lot of messages, because encrypted accounts are being frequently republished.

For each `LOOKUP` request, at most $k \times log\ N$ are sent in the network. The packets containing encrypted accounts are relatively small, their size is linear in the number of password stored. For example, an account containing 1,000 credentials size would be less than 200KiB. Moreover those packets are only used to allocate or fetch encrypted accounts, not for `LOOKUP` requests that are more numerous. No individual node is going to be overwhelmed by fetch or allocation requests as encrypted accounts are personal, thus there is no *popular* file that everyone wants to access to.

6.2 Screenshots

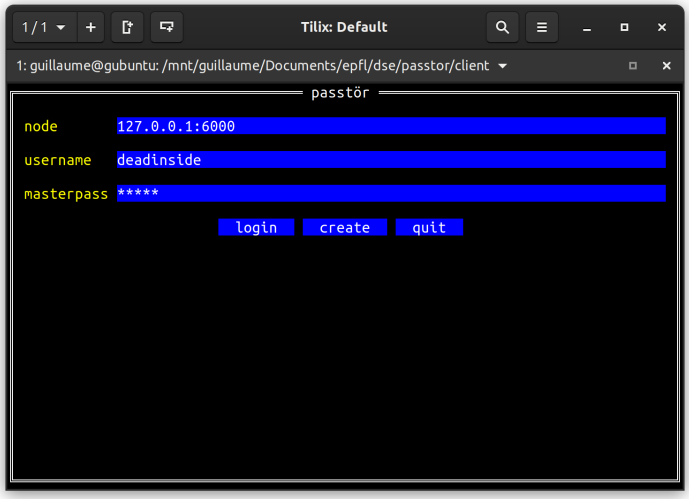

Figure 2: Login screen. It is possible to connect to any node participating in the DHT. To create an account, pick a username and a master password and press the create button. If the username is already taken, the node will return an error. When logging in an existing account, one just need to remember its global username and password.

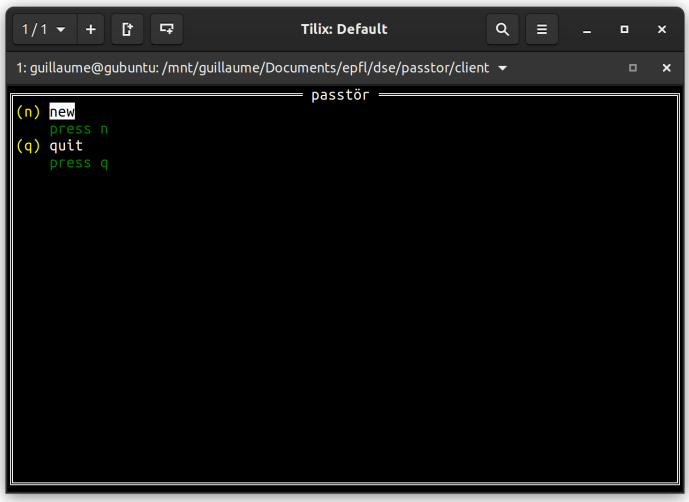

Figure 3: List of stored passwords after an account creation. It is possible to create a new credentials entry by pressing `n`, or quitting by pressing `q`.

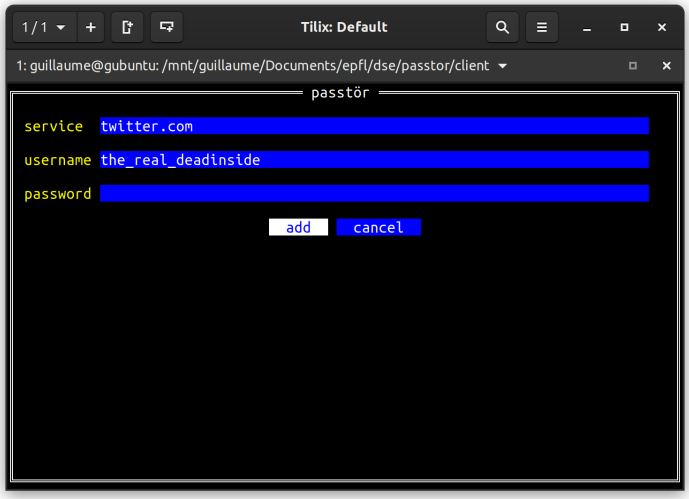

Figure 4: Creation of a new credentials entry. One need to input the service name, service username, and optionally the password. If no password is typed, the client will generate a random pass phrase composed of 8 random words, separated by dots. When modifying a credential entry, it is possible to modify the password manually, or to let it empty to generate a random password.

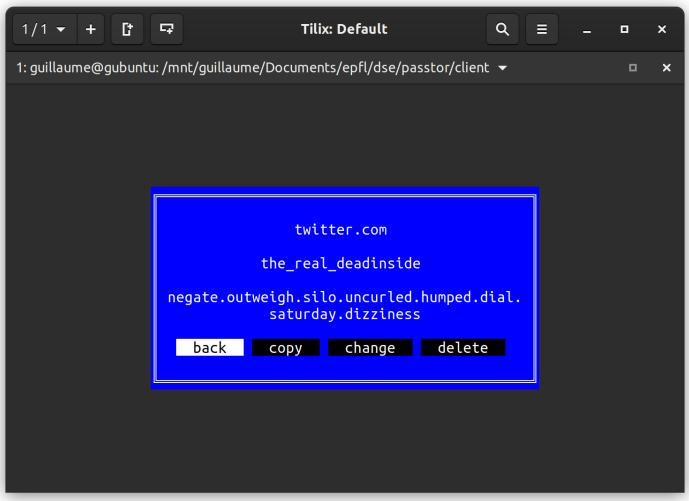

Figure 5: Show password for the entry `twitter.com`, that was randomly generated.

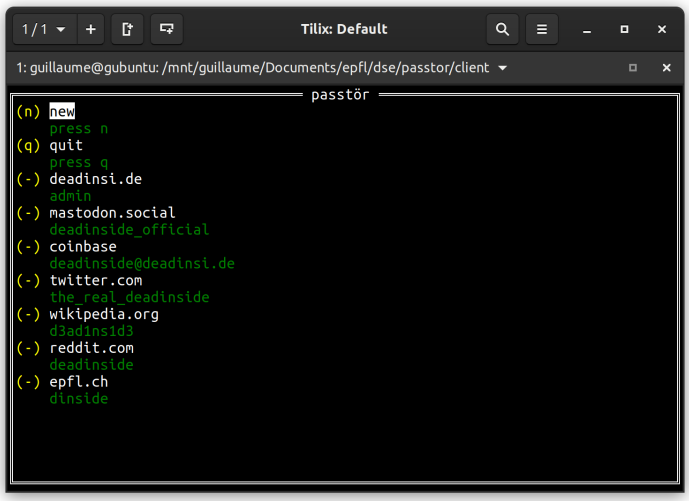

Figure 6: Shows the password list with multiple logins.

7 Future work

With our current implementation, it is easy to know whether the account associated with a username exists in the system or not. For example, an attacker can simply compute the hash of `bob`, and fetch this value from the Passtör revealing the account existence. Although the data is encrypted, the attacker can infer the number of credentials stored on `bob` account, and its current version, giving clue on the number of updates performed. This is not a desired effect, thus it would be good to improve the system to give a higher anonymity and privacy to users, although it is not a critical feature.

For now, the DHT is vulnerable to sibyls attacks, where an attacker could generate nodes close to a target account identifier, in order to have a majority of nodes storing this account. Then the attacker can modify the account information, by generating a new PKI key pair, and the original account is lost. To tackle this problem, it possible to use the cryptographic hash of a public-key, created with S/Kademlia's static crypto puzzle [2] as node identifier. This makes identity generation expensive and mitigate sibyls attacks. S/Kademlia also allows nodes to sign the messages they exchange for a better security in the DHT.

Another performance improvement for large networks with high latency between hosts could be to improve geolocality in the DHT *k-bucket*. Those buckets currently implement a least-recently seen eviction policy, except that nodes answering to periodic ping requests are never removed from the list. We could change it to a highest latency eviction: if a bucket is full and a new node should be added, the node with the highest latency from the reference node is evicted and the new one is inserted. This improvement would reduce the lookup time by querying the closest known latency-wise nodes at a given XOR distance, but may induce some instability.

If we want people to actually use our password manager, it would be good to improve the user interface, and make it more user friendly and beautiful. Web browser add-on, and mobile applications would make it convenient for users to use Passtör.

References

[1] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," *Peer-to-Peer Systems*, pp. 53–65, 2002.

[2] I. Baumgart and S. Mies, "S/kademlia: A practicable approach towards secure key-based routing," in *2007 International Conference on Parallel and Distributed Systems*, Dec. 2007, pp. 1–8. DOI: 10.1109/ICPADS.2007.4447808.