Passwords in Peer-to-Peer

Gunnar Kreitz, Oleksandr Bodriagov, Benjamin Greschbach, Guillermo Rodríguez-Cano, and Sonja Buchegger
KTH Royal Institute of Technology
School of Computer Science and Communication
Stockholm, Sweden
{gkreitz, obo, bgre, gurc, buc}@csc.kth.se

***Abstract*—One of the differences between typical peer-to-peer (P2P) and client-server systems is the existence of user accounts. While many P2P applications, like public file sharing, are anonymous, more complex services such as decentralized online social networks require user authentication. In these, the common approach to P2P authentication builds on the possession of cryptographic keys. A drawback with that approach is usability when users access the system from multiple devices, an increasingly common scenario.**

In this work, we present a scheme to support logins based on users knowing a username-password pair. We use passwords, as they are the most common authentication mechanism in services on the Internet today, ensuring strong user familiarity. In addition to password logins, we also present supporting protocols to provide functionality related to password logins, such as resetting a forgotten password via e-mail or security questions. Together, these allow P2P systems to emulate centralized password logins. The results of our performance evaluation indicate that incurred delays are well within acceptable bounds.

I. Introduction

Most of the peer-to-peer (P2P) systems deployed today do not authenticate users. While this is is often acceptable, or even preferable, there are some problems for which user authentication is a requirement. These include P2P storage, backup, and online social networks. In such applications, the data accessible to a client depends on who is using it.

We discuss how to implement a username-password scheme for authentication in P2P systems. Our goal is to construct an authentication component that can be reused across different P2P applications, which we assume authenticate via possession of cryptographic keys. Thus, from an API perspective, the login system shall allow a user entering a username and a password to recover a set of cryptographic keys which can then be used by the actual application. These keys can also be updated as needed by the application.

The goal from an end-user point of view is to emulate current behavior of centralized password-based login mechanisms. More specifically, we include schemes to remember logins, change passwords, and provide recovery if a password is forgotten. We aim to follow best practice in password authentication, acknowledging that users often re-use passwords between systems. By remembered logins, we mean that a user can opt to have a device store information such that it can log in again without storing the user's password in plain text on the device. Similarly, password change requires knowing the old password, and for password recovery, the user is able to set a new password but does not learn her previous one.

A. Why password authentication?

There is a rich literature on various approaches to authentication, ranging from the traditional username-password pair to hardware tokens and biometry. Of these, the traditional view is that passwords should be replaced by some better mechanism. However, as argued by Herley and van Oorschot [1], despite significant research efforts into dislodging passwords, they are still by far the most common authentication mechanism today. Reasons for their prevalence include simplicity, price, and very strong user familiarity.

When authentication is required in the P2P setting, it is typically done via the security-wise stronger mechanism of generating and storing cryptographic keys on a user's machine. This approach is taken in systems such as OneSwarm [2], Safebook [3], and Tribler [4]. This works well until the user wants to access the service from a second device. To do so, she would need to transfer the keys, or assume a new identity. This is an added complexity and user-perceived drawback for P2P services competing against client-server systems.

One concern is that using passwords may lead to added security risks for skilled and security-conscious users who can easily copy keys between devices. However, nothing prevents such users from choosing passwords of similar strength as cryptographic keys. Another issue pertains to remembered logins, where one must consider theft. We cannot prevent a thief from accessing the user's account, but with our protocols, the thief cannot change the user's password, and the legitimate user can always revoke the remembered credentials that are on the stolen device.

B. Our Contribution

We develop and describe a suite of protocols for password authentication in P2P networks: account registration, login, password change, remembered logins, logout also of a remote device, and password recovery, following best practices and adapting standardized criteria from centralized systems to P2P environments, and start a discussion on usable authentication in P2P systems.

Our password authentication is based on standard cryptographic techniques and can be used with standard P2P components. As a first step toward a security analysis, we discuss the security implications of our protocols. Then, we evaluate the performance of our protocols under various scenarios.

C. Paper Outline

We discuss related work in Section II, give a system overview in Section III and outline our basic scheme for password-based login in Section IV. We then describe password recovery mechanisms as extensions to the basic login mechanisms in Section V. Next, we discuss security in Section VI and report our evaluation results in Section VII before concluding in Section VIII.

II. Related Work

The subject of securely establishing stable identities in P2P systems has been previously studied, for instance by Aberer, Datta and Hauswirth [5]. The need for identities mainly arose from technical concerns, such as handling dynamic IP address assignment, or avoiding Sybil attacks [6]. Authentication of a node is done via a signature key, automatically generated and stored on the node.

As P2P systems began providing more complex functionality [2], [3], [4], [7], the need to authenticate *users*, rather than nodes, arose. It seems that often, authentication via a signature key has been carried over to this problem. While a solution of automatic identification of a node is preferable as long as users use a single device, equating a node with a user fails as users increasingly access services from multiple devices.

Illustrative is the case of backup systems, where an important use case is to restore data on a different system from where it was backed up. Here, two different approaches to authentication have been taken. All approaches build on encrypting backed up content, and the approaches vary in whether the keys are randomly derived [7], or derived from a password [8]. In the former case, a user must manually back the keys up, as these keys are required to restore the backup. The systems deriving a key from a password are related to our proposed protocol, and use some related techniques. However, to the best of our knowledge, they do not consider the additional protocols required surrounding password authentication, such as remembered logins, and recovering lost passwords.

Some P2P storage systems also use techniques which are related to ours. For example, the DHT-based systems GNUnet and Freenet use keyword strings to derive a public-private key pair whose private key is used to sign data and the hash of the public key to identify the data in the storage. Both of these systems use a keyword string as a seed to a pseudo-random number generator that produces the key pair [9], [10]. Knowing only the memorable keyword string the user can store and retrieve information.

Related to forgotten passwords, recovery of information in a P2P scenario has been studied by Vu et al. [11] who proposed a combination of threshold-based secret sharing with delegate selection and encrypting shares with passwords.

Frykholm and Juels [12] proposed a password-recovery mechanism based on security questions very similar to our protocol for the same task. They offer better, information-theoretic security properties, something not applicable to our scenario. We treat the subject of password change, which is not applicable to their scenario, although their proposal could be extended to support password change using our techniques.

III. System Overview and Assumptions

We have designed our system around standard primitives, as depicted in Figure 1. In particular, our protocols build on: a DHT [13], [14], for user lookup; a peer sampling protocol [15], [16] for randomly choosing peers; and a distributed storage [17], [18] for storing data required for our solution. Both the DHT and distributed storage are P2P protocols, run by the peers participating in the system. The storage could be implemented as a DHT, or even be the same as the user lookup DHT. However, we put different requirements on the user lookup DHT and the distributed storage, as detailed below.

To make the system flexible across different implementations, we require as few non-standard features as possible. The exception to this is the DHT that handles account registration, mapping each registered username to a reference in the storage. For resilience against account hijacking, we propose modifying the DHT to be write-once on keys: once an account has been registered, nobody else can register that username.

From the DHT we require two operations, `put`(key, $value$), and `get`(key). The put operation associates the $value$ with the key, and subsequent get operations on that key will return the $value$. As the DHT is write-once, a second put operation with the same key will not affect the system state.

The distributed storage functions for data manipulation are similar to the DHT, with three differences: we allow the distributed storage to select the "filename" for us; we require that data can be updated; and we assume (minimalistic) access control when writing. We refer to what is stored in the distributed storage as files, to simplify our description. While the storage component can be implemented as a distributed file system, we emphasize that our requirements are significantly weaker than full file system semantics.

We formalize the API to the storage as having three operations. First, `create`($data$) which generates a new file and returns a filename. Second, `write`($filename$, $data$) that overwrites the file $filename$ with content $data$. Third, `read`($filename$) which reads the content from a file. Our security does not require overwritten data to be inaccessible, so a solution similar to GNUnet [10] or Freenet [9] where a new version is stored and pointed to suffice in our protocols.

We require the storage system to support some minimalistic access control. Each stored file has an owner, which is the user who created the file. Only the owner can perform the `write` operation. To authenticate ownership of files, we assume that a public-key cryptographic system is used.

Finally, for the peer sampling component, we require a `getPeer()` method, returning a randomly selected peer, with a distribution close to uniform.

IV. Password-based P2P Login

For password-based authentication in P2P systems, the basic functionality involved is registering an account, and logging in. We also consider password change and remembered logins,

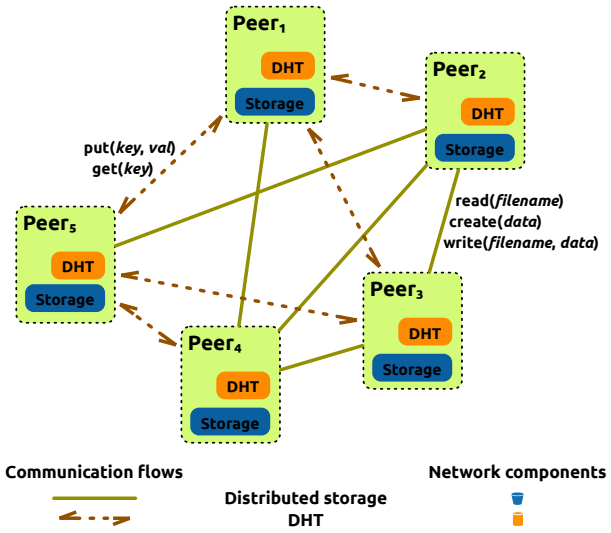

Fig. 1. Overview of the system.

allowing a device to store sufficient information to log in later without asking for credentials anew. Following recommendations from the ISO 27002 standard [19], we define the following requirements for our login procedure and add our own (preceded by a star) to account for several devices.

- passwords should neither be stored nor transmitted in clear text
- a user should be able to choose her own passwords and change them
- files with passwords should be stored separately from application data
- ⋆ a user should use the same password to log in from any device
- ⋆ it should not be possible to recover a password by stealing a device with remembered credentials
- ⋆ it should be possible to block access to the account from a stolen device

The standard also defines limitations for password login procedures that our system cannot provide fully due to the lack of rate-limiting possibilities in P2P networks: to limit the number of unsuccessful login attempts and the maximum and minimum time allowed for the login procedure. Adapting a multi-party password hardening scheme [20] could, in future work, be a way to achieve similar properties in a P2P network. Besides this limitation, our protocols fulfill the requirements as outlined in the standard, and our own added requirements.

We now describe our protocols based on the system model from Section III. Figure 2 shows the information objects and their storage locations, with arrows for the abstract flow of the login procedure, Table I lists the terms used in the algorithms.

A. Account Registration

To register a new account (see Algorithm 1), the user first has to choose a username $uname$ and a password $passwd$. Next, the user creates a key store file F_{KS}, containing all the keys used by the P2P application the user wants to log in to

TABLE I
PROTOCOL TERMINOLOGY

$uname$	Username
$passwd$	Password
$salt$	Random byte string
K_W	Cryptographic key for write authentication
F_{KS}	Key store file
f_{KS}	File name of F_{KS}
K_{KS}	Cryptographic key (used to encrypt F_{KS})
F_{LI}, f_{LI}, K_{LI}	Login information file, its file name and key
F_{DL}, f_{DL}, K_{DL}	Device login information file, its file name and key
D, D_{ID}	User device and the identifier of D
$K_{x1}, K_{x2}, \ldots$	Cryptographic keys for usage after logging in
$devmap$	Mapping from device identifiers to device login information files and corresponding keys

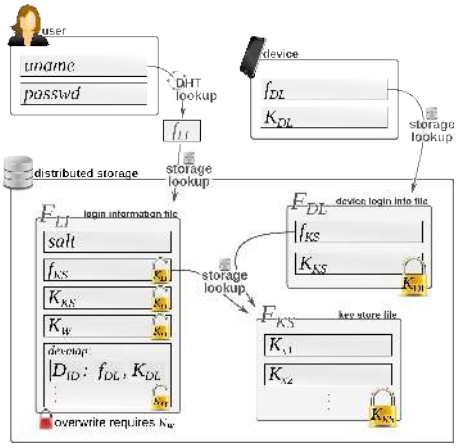

Fig. 2. Storage Locations (boxes) and Login Procedure (arrows)

(and an additional storage key, authenticating write operations on this file). The user then creates a symmetric key K_{KS}, encrypts the file content with this key and puts the ciphertext into the storage, obtaining a file name f_{KS}. Now, the user creates a login information file F_{LI} by creating a random byte string $salt$, deriving a symmetric key K_{LI} from the password $passwd$ and the $salt$, encrypting f_{KS}, K_{KS} and K_W (a generated storage key, required for overwriting F_{LI} later) with K_{LI}. The salt and the three encrypted values are put into the storage, obtaining a file name f_{LI}. The salt is stored in plaintext, so that the user later can derive the decryption key K_{LI} by only providing the password. Finally, the user performs the write-once operation `put` on the DHT with $uname$ as key and f_{LI} as value. If the username was taken, the user is prompted for a new username. Once all operations have succeeded, the user is registered in the system.

B. Login

Once registered, a user is able to log in – that is, to retrieve the cryptographic keys stored in the key store file F_{KS} – from any device by only entering her username and password (see Algorithm 2). A `get` request with the parameter $uname$ to the DHT results in the filename f_{LI} for the login information file F_{LI}. This file is retrieved from the distributed storage and contains the $salt$ in plaintext. The latter is fed into a key-

Algorithm 1 Account Registration

$uname \leftarrow$ User.input("Choose username:")
$passwd \leftarrow$ User.input("Choose strong password:")
$K_{KS} \leftarrow$ generateKey()
$F_{KS} \leftarrow$ encrypt$_{K_{KS}}(K_{x1}||K_{x2}||\ldots)$
$f_{KS} \leftarrow$ Storage.create(F_{KS})
$salt \leftarrow$ generateSalt()
$devmap \leftarrow$ createMap()
$K_{LI} \leftarrow$ KDF($salt$,$passwd$)
$K_W \leftarrow$ generateKey() // suitable for the storage system
$F_{LI} \leftarrow salt||$encrypt$_{K_{LI}}(f_{KS}||K_{KS}||K_W||devmap)$
$f_{LI} \leftarrow$ Storage.create(F_{LI}) // using K_W
while DHT.put($uname$,f_{LI}) fails
    $uname \leftarrow$ User.input("Choose new username:")
end while

Algorithm 2 Login

$f_{DL}, K_{DL} \leftarrow$ Device.readLocalStore()
if $f_{DL} \neq$ `NULL` **then** // non-interactive login
    $F_{DL} \leftarrow$ Storage.read(f_{DL})
    $f_{KS}, K_{KS} \leftarrow$ decrypt$_{K_{DL}}(F_{DL})$
    $saveLoginLocally \leftarrow$ `False`
else // interactive login
    $uname \leftarrow$ User.input("Enter username:")
    $passwd \leftarrow$ User.input("Enter password:")
    $saveLoginLocally \leftarrow$ User.input("Remember?")
    $f_{LI} \leftarrow$ DHT.get($uname$)
    $F_{LI} \leftarrow$ Storage.read(f_{LI})
    $salt \leftarrow F_{LI}.salt$ // stored in plaintext
    $K_{LI} \leftarrow$ KDF($salt$,$passwd$)
    $f_{KS}, K_{KS}, K_W, devmap \leftarrow$ decrypt$_{K_{LI}}(F_{LI})$
end if
$F_{KS} \leftarrow$ Storage.read(f_{KS})
$K_{x1}, K_{x2}, \ldots \leftarrow$ decrypt$_{K_{KS}}(F_{KS})$
if $saveLoginLocally$ **then**
    $K_{DL} \leftarrow$ generateKey()
    $F_{DL} \leftarrow$ encrypt$_{K_{DL}}(f_{KS}||K_{KS})$
    $f_{DL} \leftarrow$ Storage.create(F_{DL})
    Device.writeLocalStore($f_{DL}||K_{DL}$)
    $devmap$.append(Device.ID, $f_{DL}||K_{DL}$)
    $F_{LI} \leftarrow salt||$encrypt$_{K_{LI}}(f_{KS}||K_{KS}||K_W||devmap)$
    Storage.write(f_{LI},F_{LI}) // using K_W
end if

derivation function together with the user password to derive the key K_{LI}. This key allows the user to decrypt all other content of the login information file, including the filename f_{KS} of the key store file and the corresponding key K_{KS}. Finally, the user fetches the key store file F_{KS} from the storage system and decrypts it, using K_{KS}. This concludes the login procedure as the user is now in possession of the keys $K_{x1}, K_{x2}, \ldots$, required by the P2P application.

If the user chose to remember the login information on the local device, a new device login information file F_{DL} is created and saved to the storage system (which returns a filename f_{DL}). This file contains the filename f_{KS} of the key store file as well as the according key K_{KS} and is encrypted with a new key K_{DL}. On the device, only the filename f_{DL} and the key K_{DL} are stored locally. Additionally, a reference to the device login information file is stored in the $devmap$ value of the login information file F_{LI}. It contains a mapping from a device identifier to the filename and key of the device login information file, allowing password changes and device revocation as described later.

When the user wants to log in from the same device again, the locally stored values (f_{DL}, K_{DL}) are used to retrieve the device login information file, decrypt it, and thereby gain access to the key store file. Thus, the remembered login feature allows the user to log in without entering the password, while nothing password-related is stored on the device. Furthermore, remembered logins remain valid even if the P2P application changes keys in the key store file.

C. Password Change

Before the user can change the password, she must log in using her password to obtain K_{LI}. With this information, the password change can be accomplished (see Algorithm 3): the user is asked for a new password and a new salt is generated. The key-derivation function is used to generate a new key K_{LI}^{new} for the login information file. Then, the content of the key-store file is fetched and decrypted (with the old key). A new key K_{KS}^{new} is generated and used for encrypting the key-store content again before it is saved to the storage system, obtaining a new filename f_{KS}^{new}. Finally, the login information file is updated: $f_{KS}^{new}, K_{KS}^{new}$, the write credential K_W as well as a new empty device mapping $devmap^{new}$ are encrypted with the new key K_{LI}^{new}. Together with the new salt, this ciphertext is written to the distributed storage, using the reference f_{LI} and the credential K_W, to authenticate the `write` operation. Lastly, the keys stored in the key store should be updated by the application using our P2P protocol. See Section VI-E for a discussion. At this point, old device login information files can also be deleted from the storage to reclaim space.

D. Logout

To log out from the system, the user does not have to interact with the DHT or the storage system. Simply wiping her local cache from application data and all key material restores the pre-login state. If the user chose to remember the login on a device, the corresponding device login information file F_{DL} can also be deleted from the storage.

A problem related to logging out is revoking remembered credentials on another device, e. g., a user's stolen phone. To accomplish this, we first run the password change operation, which locks out all devices with remembered logins, because the key store key K_{KS} changed (as well as the filename f_{KS}). Next, we use the device mapping $devmap$ to inform all devices about the new key (and filename), except the device that is to be revoked. To inform a device about the change, we update

Algorithm 3 Password Change

Input: $uname, K_{LI}^{old}$

$f_{LI} \leftarrow$ DHT.get($uname$)
$F_{LI}^{old} \leftarrow$ Storage.read(f_{LI})
$f_{KS}^{old}, K_{KS}^{old}, K_W, devmap^{old} \leftarrow$ decrypt$_{K_{LI}^{old}}(F_{LI}^{old})$
$passwd^{new} \leftarrow$ User.input("Enter new password:")
$salt^{new} \leftarrow$ generateSalt()
$K_{LI}^{new} \leftarrow$ KDF($salt^{new}$,$passwd^{new}$)
$devmap^{new} \leftarrow$ createMap()
$F_{KS}^{enc-old} \leftarrow$ Storage.read(f_{KS}^{old})
$F_{KS} \leftarrow$ decrypt$_{K_{KS}^{old}}(F_{KS}^{enc-old})$
$K_{KS}^{new} \leftarrow$ generateKey()
$F_{KS}^{enc-new} \leftarrow$ encrypt$_{K_{KS}^{new}}(F_{KS})$
$f_{KS}^{new} \leftarrow$ Storage.create($F_{KS}^{enc-new}$)
$F_{LI}^{new} \leftarrow$
$salt^{new}$|| encrypt$_{K_{LI}^{new}}(f_{KS}^{new}||K_{KS}^{new}||K_W||devmap^{new})$
Storage.write(f_{LI},F_{LI}^{new}) // using K_W
Refresh keys stored in key store
Old device login information files may be deleted

the corresponding values in the device's login information file F_{DL} which can be accessed from the device by using the locally stored credentials.

Algorithm 4 describes this necessary extension. After running the password change operation, all devices that should *not* be revoked and that have remembered logins (and therefore are referenced in the device mapping $devmap$) are processed. The device login information filename f_{DL} and its key K_{DL} are read, and the new key store key K_{KS}^{new} and filename f_{KS}^{new} are written to the device login information file F_{DL}, encrypted under the device key K_{DL}. Finally, the modified $devmap$ is saved back to the login information file F_{LI}.

Algorithm 4 Logout Other Device

... // run Algorithm 3 (Password Change)
$deviceToLogout \leftarrow$ User.input("Select device:")
$devmap.remove(deviceToLogout)$
foreach D_{ID} **in** $devmap$ // all devices to keep
    $f_{DL}, K_{DL} \leftarrow devmap$.get($D_{ID}$)
    $F_{DL} \leftarrow$ encrypt$_{K_{DL}}(f_{KS}^{new}||K_{KS}^{new})$
    Storage.write(f_{DL},F_{DL})
end
... // save modified $devmap$ back to F_{LI}

V. Password Recovery

An important part of password-based logins is the possibility for users to recover their accounts if they forget their passwords. We refer to this as a *password recovery mechanism*. The goal of a password recovery mechanism is to provide a secondary way of authenticating the user. There are a number of password recovery mechanisms used in practice. In our experience, three of the most common ones are password hints, security questions, and e-mail based recovery. Other approaches (beyond the scope of this paper) include vouching for identity by social contacts [21], or using trusted devices.

Password hints means that the user may enter a hint at the same time as she sets this password. The hint will be displayed to her if she forgets her password, and should be selected such that it helps her recall her password, but does not make it significantly easier for someone else to guess it. The hint is not truly a secondary authentication mechanism, but rather a means to recovering the original password-based authentication mechanism. A basic version of password hints would be straightforward to implement in our system: the hint can be stored in plaintext in the login information file. Security questions and e-mail based password recovery are more complex to adapt. We described their implementation in detail after listing requirements.

As in Section IV for the login procedure, we define a set of functional requirements for password recovery, based on the ISO 27002 standard [19] as follows. We also augment the list with requirements of our own (preceded by a star).

- establish methods to verify the identity of a user prior to allowing the user to choose a new password
- communicate with those affected by or involved with recovery security incidents
- have procedures to allow recovery and restoration of business operations and availability of information in a time-scaled manner
- a legitimate user should be able to recover lost (forgotten) or broken (device's) keys
- ⋆ the recovery procedure should allow a user to set a new password, not reveal the old password
- ⋆ the process of recovery should be easy to use
- ⋆ sensitive information for recovery should be kept secret

Our protocols support these requirements. The sole exception is that if a password is reset via security questions alone, the system would not "communicate with those affected" (e.g., send an e-mail notification that the password had been reset, as is common in centralized services). We remark that the last item is a stronger property than many centralized systems provide. In our system, no one learns the answers to a user's security questions. We consider this to be important, since many systems use similar security questions.

The operations described in this section imply minor additions to the protocols of Section IV, i. e., invoking the update procedures after each password change (to sustain transaction safety, the updates have to be included in the final write operation of the password change operation).

A. Security Questions

Security questions is a password recovery technique that relies on answers to questions the user is asked during registration. The answers should be such that they cannot be easily guessed or researched by an attacker, but still stable over time, memorable, and definite [22]. Rabkin [23] underlines the importance to choose good questions especially in the era of social networks. Frykholm and Juels [12] discuss a related technique that is similar to our adaption of this scheme.

TABLE II
RECOVERY PROTOCOL TERMINOLOGY

Security Question Recovery	
qS_i	(n,k)-secret sharing share of K_{LI}
Q_i	Security challenge question
A_i	Answer to question Q_i
$qsalt_i$	Random byte string
qK_i	Key to encrypt the share qS_i
E-mail Based Recovery	
K_R	Long-term recovery key
eS_i	(n,k)-secret sharing share of K_R
$email$	Recovery e-mail address of the user
$peer_i$	Randomly selected peer
$esalt_i$	Random byte string (to seed the e-mail commitment)
$ksalt_i$	Random byte string (to seed the key eK_i)
C_i	Cryptographic commitment to the e-mail address
eK_i	Key to encrypt the share eS_i

We assume that the user provides n answers A_i to suitable security questions Q_i. In order to recover the password, we require the user to answer any k out of these n questions correctly. The choice of k constitutes an obvious trade-off between security and usability. A successful recovery yields the key K_{LI} to the login information file, allowing the user to change the password, using Algorithm 3. Our implementation does not require the user to provide new answers after a regular password change. Additionally, we avoid storing the plaintext answers to the security questions.

For the setup of the question based recovery mechanism (Algorithm 5), we first create n shares $qS_1, \ldots, qS_n$ of the key K_{LI} under an (n,k)-secret sharing scheme. For each of these shares, we create a salt $qsalt_i$, derive a key qK_i from this salt and the answer A_i, and use it to encrypt the share, yielding qS_i^{enc}. Furthermore we encrypt the key qK_i with the login information file key K_{LI}, for the update procedure described later. Finally, the login information file is extended with all questions Q_i, the salts $qsalt_i$, the encrypted shares qS_i^{enc} and the encrypted keys qK_i^{enc}. When recovering, the user has to reproduce at least k answers, which together with the stored salts can be used to derive k keys qK_i, which in turn can decrypt k shares qS_i.

When K_{LI} changes (e. g., due to a regular password change), we update the recovery information as in Algorithm 6: for the new key K_{LI}^{new}, a new set of shares is created. Next, the keys qK_i are decrypted and used to encrypt the new shares. Neither the keys qK_i nor the salts $salt_i$ change, so the user can still use the same answers for recovery. Finally, the updated shares (and re-encrypted keys, to allow further updates) are saved back to the login information file.

Algorithm 5 Security Questions Setup

```
qS_1, ..., qS_n ← createShares(n,k,K_LI)
for i ← 1, n do
    Q_i ← User.input("Enter question i:")
    A_i ← User.input("Enter answer i:")
    qsalt_i ← generateSalt()
    qK_i ← KDF(qsalt_i,A_i)
    qS_i^enc ← encrypt_{qK_i}(qS_i)
    qK_i^enc ← encrypt_{K_LI}(qK_i)
end for
add to F_LI: qS_i^enc, qK_i^enc and the plaintext values of
    Q_i, qsalt_i   ∀i ∈ {1, ..., n}
```

Algorithm 6 Security Questions Update (on K_{LI} change)

```
qS_1^new, ..., qS_n^new ← createShares(n,k,K_LI^new)
for i ← 1, n do
    qK_i ← decrypt_{K_LI}(qK_i^enc)
    qS_i^{new-enc} ← encrypt_{qK_i}(qS_i^new)
    qK_i^{new-enc} ← encrypt_{K_LI^new}(qK_i)
end for
update in F_LI: qS_i^{new-enc}, qK_i^{new-enc}   ∀i ∈ {1, ..., n}
```

B. E-mail Based

In e-mail based password recovery, the user is sent an e-mail containing some information, typically a link with a token, by which she can reset her password. This link is sent to an e-mail address she has registered with her account.

We adapt this scheme by randomly choosing a number of peers, that collaboratively provide this functionality to the user. We use (n,k)-secret sharing to enable password recovery even if not all of the involved peers are online when the user wants to recover the password. We discuss parameter choices of k and n in Section VII-B.

To provide persistence of the recovery mechanism independent of a changing key K_{LI} (e. g., due to a password change), the result of the recovery process is a recovery key K_R, that always encrypts the current version of K_{LI}. Algorithm 7 describes the setup procedure: From the recovery key K_R, n shares $eS_1, \ldots, eS_n$ are generated using (n,k)-secret sharing. For each share, a random peer $peer_i$ is picked, two salts $esalt_i$ and $ksalt_i$ are created and a cryptographic commitment C_i is derived from the salt $esalt_i$ together with the $email$. This commitment will be used to authorize the user to the peer, and bind it to this specific e-mail address. Next, a key eK_i, to encrypt the share eS_i, is derived in the same way as the commitment, but with salt $ksalt_i$. A different salt is needed so that the peer cannot decrypt the share (before learning the address). The commitment and the encrypted share are stored at the peer. The login information F_{LI} file is extended with a list of the chosen peers $peer_i$ and the according salts $esalt_i$, $ksalt_i$, as well as K_{LI}^{enc}, encrypted with the recovery key, and the recovery key, encrypted with K_{LI} (to allow for password changes).

To recover the password (Algorithm 8), the user looks up the available information in the login information file, including the list of peers to be requested for assistance. Each request is authorized by the commitment C_i, that the peer can derive from the salt $esalt_i$ and the e-mail address, that the user provided (Algorithm 9). If the request was legitimate, the peer sends the encrypted share to the e-mail address. As soon as the user collected k answers, she can recover K_{LI}.

Algorithm 7 E-mail Recovery Setup

$K_R \leftarrow$ generateKey() // long-term recovery key
$K_{LI}^{enc} \leftarrow$ encrypt$_{K_R}(K_{LI})$
$eS_1, \ldots, eS_n \leftarrow$ createShares(n,k,K_R)
$email \leftarrow$ User.input("Enter recovery e-mail address:")
for $i \leftarrow 1, n$ **do**
    $peer_i \leftarrow$ getPeer()
    $esalt_i \leftarrow$ generateSalt()
    $ksalt_i \leftarrow$ generateSalt()
    $C_i \leftarrow$ KDF($esalt_i$,$email$) // commitment
    $eK_i \leftarrow$ KDF($ksalt_i$,$email$)
    $eS_i^{enc} \leftarrow$ encrypt$_{eK_i}(eS_i)$
    store at $peer_i$: C_i, eS_i^{enc}
end for
$K_R^{enc} \leftarrow$ encrypt$_{K_{LI}}(K_R)$
add to F_{LI}: K_{LI}^{enc}, K_R^{enc} and the plaintext values of $peer_i, esalt_i, ksalt_i \quad \forall i \in \{1, \ldots, n\}$

Algorithm 8 E-mail Recovery: User

$uname \leftarrow$ User.input("Enter username:")
$email \leftarrow$ User.input("Enter e-mail:")
$f_{LI} \leftarrow$ DHT.get($uname$)
$F_{LI} \leftarrow$ Storage.read(f_{LI})
$K_{LI}^{enc}; \forall i : peer_i, esalt_i, ksalt_i \leftarrow F_{LI}$ // plaintext part
$\forall i$: send $(email, esalt_i)$ to $peer_i$ // send n requests
$eS_1^{enc}, \ldots, eS_k^{enc} \leftarrow$ read e-mail // wait for k e-mails
for $i \leftarrow 1, k$ **do**
    $eK_i \leftarrow$ KDF($ksalt_i$,$email$)
    $eS_i \leftarrow$ decrypt$_{eK_i}(eS_i^{enc})$
end for
$K_R \leftarrow$ useShares($eS_1, \ldots, eS_k$)
$K_{LI} \leftarrow$ decrypt$_{K_R}(K_{LI}^{enc})$
... // run Algorithm 3 (Password Change)

To provide long-term persistence of this recovery mechanism, K_{LI}^{enc} has to be updated whenever K_{LI} changes. Algorithm 10 describes the necessary steps, including updating K_R^{enc} to allow subsequent updates.

Algorithm 9 E-mail Recovery: Peer

Stored: C_i, eS_i^{enc} // stored at peer
Input: $email, esalt_i$ // provided by the user request
if $C_i =$ KDF($esalt_i$,$email$) **then** // legitimate request
    sendMail($email$,eS_i^{enc})
end if

Algorithm 10 E-mail Recovery Update (on K_{LI} change)

$K_R \leftarrow$ decrypt$_{K_{LI}^{old}}(K_R^{enc})$
$K_{LI}^{enc-new} \leftarrow$ encrypt$_{K_R}(K_{LI}^{new})$
$K_R^{enc-new} \leftarrow$ encrypt$_{K_{LI}^{new}}(K_R)$
update in F_{LI}: $K_{LI}^{enc-new}, K_R^{enc-new}$

C. Combining Approaches

The approaches presented above can be composed, either sequentially or in parallel. By sequential composition, we mean that the user must *both* correctly answer security questions and receive e-mail. By parallel composition, we mean that either mechanism can be used alone to recover the password. The latter is achieved by using both systems in parallel.

For sequential composition, the user picks a uniformly random string r of the same length as K_{LI}. The user stores r in one of the mechanisms, and $K_{LI} \oplus r$ in the second mechanism, where $\oplus$ denotes the exclusive-OR operation. If one recovers both of these, K_{LI} can be computed. If one learns only one of the pieces, nothing is gained, as both r and $K_{LI} \oplus r$ are uniformly random. More generally, to combine n mechanisms in arbitrary ways, (n, k)-secret sharing can be used. What we describe here are two trivial such schemes for $n = 2$.

VI. Security

The goal we set is to emulate the security provided by a centralized solution. Some security risks are inherent to the password functionality, and apply regardless of implementation technique. For instance, in e-mail based recovery, an attacker compromising the victim's e-mail account can reset her password.

In this section, we elaborate on security concerns of our protocols. We do not have full cryptographic security proofs of our protocols, something which is important future work.

Concerning safety, we have designed our protocols such that persistently stored data remains in a consistent state if the protocol is aborted at any point. Some protocols may, if an operation fails, leave orphan files in the storage. If our protocol for revoking remembered credentials is interrupted, it may revoke more devices than intended. Apart from this, our operations have transactional semantics, assuming small writes (both creation, and updates) to the storage are atomic operations and that operations block until successful.

A. Adversary Model

To capture the concept of collusions, we consider an adversary that corrupts a number of nodes. Upon corruption, the adversary gains all information known to that node, and in the case of an active adversary, can also control its future actions. The adversary can also make requests to the underlying system, e. g., read files from the distributed storage. As almost all our protocols mainly operate on publicly readable (encrypted) data, this ability is important. The only computation made by nodes different from the one logging in are in verifying write operations, and in e-mail based password recovery.

B. Risks in Used Components

As our protocols make use of several standard components, vulnerabilities in those components can also affect our system. For instance, an adversary may prevent a user from logging in by attacking the victim's ability to read her login information file from the distributed storage. We note that there are many security techniques in DHTs that can mitigate such threats, and refer to Urdaneta et al. [24] for a recent survey.

C. Offline Guessing

An issue in password authentication based on a (distributed) file system is that the information required to verify a password attempt is inherently exposed. This means that in our scheme, we cannot prevent an attacker from mounting an offline attack against our encrypted passwords. This is a considerable drawback from centralized schemes, where the encrypted password database is kept protected.

As a partial mitigation, we utilize a KDF with a per-user salt. This forces an attacker to evaluate the KDF individually for each user on a password guess, defeating parallel attacks against multiple users. We recommend the system be instantiated with a slow KDF, such as bcrypt [25] to throttle offline guessing. The protocol could be modified to reduce storage by using the username as a salt, but we recommend against that as it would be vulnerable to pre-computation (before the system is started, or between instances using the same KDF) attacks against common username-password pairs.

The problem of a server performing offline attacks against its password database was treated by Ford and Kaliski [20]. Their techniques are client-server based, and require all servers to be online for a login. We leave it as future work to investigate modifying their protocol to be applicable also in a P2P setting. This would prevent offline guessing attacks.

D. Colluding Nodes in E-mail Based Recovery

In the mechanism for e-mail based recovery, we employ (n, k)-secret sharing, and secrets are stored on n random nodes in the system. If an attacker controls k or more of these nodes, she can recover the secret and access the victim's account. In Section VII-B we discuss the choice of these parameters.

A peer sampling protocol is used to select the n nodes where the shares are stored. An active attacker may influence this protocol, in order to ensure that she controls k out of the selected nodes. This can be mitigated by a peer sampling protocol designed to tolerate active attacks, such as Brahms [16].

The protocol is designed to reveal only minimal information to the peers. Thus, even when colluding, peers have to get hold of both, the salt $ksalt_i$ and the recovery e-mail address $email$ to mount an attack. The e-mail address will be revealed to a peer only when the user initiates the recovery process. Malicious peers might try to guess it earlier, but to verify guesses either $ksalt_i$ or $esalt_i$ are required. These are stored in the login information file of the user, which can only be found knowing the username. Therefore the setup process should be anonymous, where the peer does not learn the username of the user for whom it stores the share.

E. Updating Application Keys

When a password is changed, or device is revoked, access is effectively revoked from future updates to the key store. However, a malicious device may have stored the last keys it was able to access. Thus, in the password update procedure (which is also used for device revocation), the keys for the P2P application itself need to be updated, and then these new keys need to be written to the key store. How to update them, if possible, is beyond the scope of this paper, as it is a functionality of the application protocol.

F. Denial-of-Service

An adversary may mount a DoS attack by writing many user names into the DHT, thus blocking those from registration by legitimate users. This attack is also possible in centralized systems, but there detection and counter-measures (e.g., removing the fake accounts) is significantly easier. One can solve this issue by assuming a lightweight CA dealing only with checking user identification before account creation, similar to Safebook [3]. We remark that this attack only target the availability of registration of a new user account, it does not affect existing users.

A related attack can occur when an adversary can prevent a user from accessing some of the data required to log in (e.g., by controlling all replicas holding the victim's entry in the DHT). In such attacks, existing users can be prevented from logging in, possibly permanently by overwriting or deleting the key. This illustrates the importance of applying security techniques for underlying components [24] and indicates a large replication factor should be chosen.

G. Security Summary

Aside from the concerns outlined, we believe that our schemes produce a similar level of security as client-server based password authentication schemes. The cryptographic design of our protocols relies on relatively standard techniques. This leads us to be confident that the security of our protocols can be formally proven using cryptographic techniques.

We also provide some features which are not commonly present in centralized systems. One of these is the ability to revoke stored credentials from only some specific devices. A second one is the ability to set up e-mail based password recovery without revealing your e-mail address before recovery actually occurs, offering an additional privacy protection.

VII. Evaluation

We developed two lightweight custom simulators, one to evaluate the efficiency and security of our protocols, and one to assist in setting the n and k parameters for e-mail based password recovery. For the performance analysis, we take as input the time to perform required cryptographic operations, as well as the time of our network operations. For the analysis of n and k parameters, we need data on node uptime to evaluate the availability of the password recovery system for a given choice. We used latency data from Jiménez et al. [14], and node availability data from Rzadca et al. [26].

To measure the computational cost of the necessary cryptographic operations in a prototype implementation, we used OpenSSL's built-in benchmark function on a 2.26 GHz Core 2 Duo running Mac OS X, as well as an ARM 1 GHz Cortex-A8, similar to modern smart phones. The times for all required cryptographic operations (using DSA as a public-key scheme) was negligible, below 5 ms (2 ms on the faster CPU).

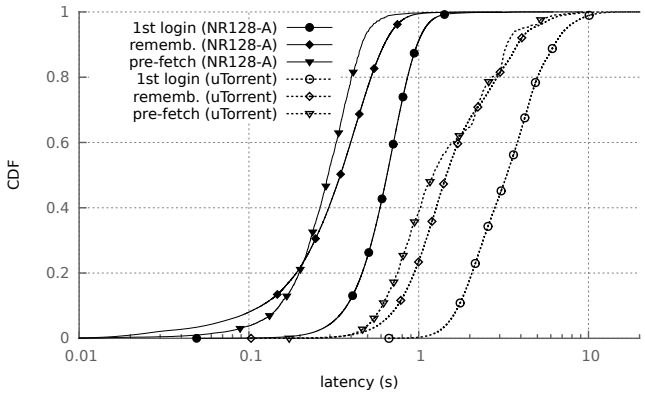

Fig. 3. CDF for login latency in three modes: First time login, remembered logins, and first time login after password entry (pre-fetch). Network operations are assumed to have costs of BitTorrent mainline DHT lookups, using NR128-A (solid lines) or μTorrent strategy (dashed lines) [14].

As our protocols can be applied with any DHT and file storage combination, we used the BitTorrent Mainline DHT as an example for our numeric performance evaluation. Jiménez et al. [14] recently ran experiments to evaluate the performance of their proposed algorithmic improvements. From their measurements, we received a CDF for the latency of real-world DHT lookups. We assume that all our *network operations*, writes and reads, both from DHT and distributed storage, take the same amount of time as a DHT lookup in their study. This can be motivated, as our distributed storage could be implemented via a DHT. We remark that their measurements are performed with a "warmed up" client with filled routing tables for the DHT. Thus, these numbers may be overly optimistic for a newly started client.

Finally, we believe, node availability will vary considerably between applications. As a representative case, we considered a distributed storage system by Rzadca et al. [26], which featured such data in their evaluation. In the distribution, 10% of nodes have availability 95%, 25% have availability 87%, 30% have availability 75%, and 35% have availability 33%. To this rough distribution, Gaussian noise with $\sigma = .1$ is added, and the resulting availability is capped between 3% and 97%.

A. Performance

We believe that the main performance-critical operation is logging in [27]. We believe that for all other operations, latency on the order of a few seconds can be acceptable, and even a minute if they are run in the background. Thus, we only present results for logging in, but note that as the operations for other protocols are similar, results are expected to be similar. The performance cost of our protocols is dominated by network operations. However, to slow down password guessing attempts, one may wish to force the key derivation procedure to be slow, to the point of making that cost dominant.

When evaluating the protocols, we parallelized network operations where possible. Logging in for the first time and remembering the credentials for future logins is then a sequence of two network operations, followed by key derivation, followed by three parallel network operations. In a password login, it is also possible to pre-fetch some of the information after the user has entered her username, but before she enters her password. In particular, as soon as the username is known, the filename F_{LI} can be retrieved from the DHT, and the file can be read. Decryption of the file and further processing is then only possible after the user enters her password. To evaluate this speed-up, we computed the time it takes to finish the login after the user has entered her password. The time to fetch the two files is identical to the time to do a remembered login, and it is sufficiently small that the data can realistically be retrieved while the user is typing her password.

TABLE III
LATENCIES OF PROTOCOLS, IN MILLISECONDS.

	Network Op. [14]		First Login		Remem. Login	
DHT	median	99^{th}	median	99^{th}	median	99^{th}
NR128-A	164	567	650	1362	346	915
μTorrent	647	5140	3299	10154	1456	7148

To determine the sensitivity of our performance to implementation characteristics, we evaluated our protocol for two different client strategies in the BitTorrent Mainline DHT: The NR128-A algorithm [14], and the μTorrent client's implementation. We present these performance numbers in Figure 3 and Table III. Firstly, we observe that with a fast storage, our login protocol is very fast, with a median login time of 650 ms the first time, and 346 ms for remembered logins. Comparing the results, we observe that our protocols are indeed sensitive to storage latency. While performance results building on μTorrent data are slightly above recommended levels [28], we still consider them within range of acceptability for P2P.

When evaluating these numbers, we assumed that the runtime of the KDF function is negligible. As a system designer, one may wish to pick a slow KDF (e. g., bcrypt [25]), as this slows down password guessing attempts. Any latency intentionally added via the KDF would affect the first-time login (after the password entry) times.

B. Parameters for E-mail Password Recovery

There are trade-offs between availability, security, and storage space in our e-mail based password recovery protocol.

For the selection of k, the minimum number of peers required to recover the password, there is a direct trade-off between security and availability. Lower choices of k increase the risk of an adversary, controlling a significant number of nodes, to break into the user's account. A higher k reduces the availability of the recovery functionality, which reflects the chances of a user to immediately succeed with the password recovery. However, if the user does not instantly receive k answers, she can simply wait until enough peers are online.

We believe a reasonable choice of parameters is $n = 16$ and $k = n/2$. With these numbers, using the availability data from Rzadca et al. [26], there is a 96% probability of immediate recovery of a lost password. A very strong attacker, corrupting 25% of the nodes in the system, would still only be able to

access the user's account with probability 3%. The analysis of parameter choice here is similar to any P2P system using secret sharing, and we refer to e.g., Vu et al. [11] for a more in-depth discussion.

C. Scalability

The latency of our protocols will scale similarly to DHTs or other distributed storage systems. The data we used for evaluation is based on measurements on the largest deployed DHT, demonstrating that performance is good with extremely large user numbers. Performance may in fact be worse for a small system, as there are then fewer nodes, meaning that it is less likely to find data at a nearby node. To bootstrap the system with good performance when it is small, a very simple distributed storage using one or a few super-nodes would be one approach. Storage requirements per user are also small, with a few files per user and small file sizes. From this, we conclude that our system is likely to scale well with the number of users.

VIII. Conclusions and Future Work

The pros and cons of password-based authentication have been extensively debated. We believe that for some applications, a username-password pair provides an appropriate level of security. We argue that incorporating a well-known authentication scheme may assist in user adoption of P2P systems for more complex tasks than file sharing. To the best of our knowledge, ours is the first work to focus on password-based logins in a P2P setting, including mechanisms to recover a forgotten password. Our protocols are new (but our security questions are similar to [12]), relatively straightforward, and we believe, they are an important first step towards usable authentication in P2P.

The performance of our mechanisms in terms of delay varies according to the underlying DHT or P2P system in general and in relation to how much intentional delay is added by parameterizing cryptographic functions. Overall, however, our evaluation results show that for user satisfaction [27], the delays can be kept at a very acceptable level [28].

While we have provided an initial discussion of the security properties of our protocol here, future work will include a thorough security analysis. Our scheme allows offline password guessing attacks, which will also be addressed in future work.

Acknowledgments

We thank Raúl Jiménez et al. [14] for sharing their measurement results and Jay Lorch for excellent work as a shepherd of this paper. This research has been funded by the Swedish Foundation for Strategic Research grant SSF FFL09-0086 and the Swedish Research Council grant VR 2009-3793.

References

[1] C. Herley and P. C. van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Security & Privacy*, vol. 10, no. 1, pp. 28–36, 2012.

[2] T. Isdal, M. Piatek, A. Krishnamurthy, and T. E. Anderson, "Privacy-preserving P2P data sharing with OneSwarm," in *SIG COMM*. ACM, 2010, pp. 111–122.

[3] L. A. Cutillo, R. Molva, and T. Strufe, "Safebook: A privacy-preserving online social network leveraging on real-life trust," *IEEE Communications Magazine*, vol. 47, no. 12, pp. 94–101, 2009.

[4] S. M. A. Abbas, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips, "A gossip-based distributed social networking system," in *WETICE*, S. Reddy, Ed. IEEE Computer Society, 2009, pp. 93–98.

[5] K. Aberer, A. Datta, and M. Hauswirth, "Efficient, self-contained handling of identity in peer-to-peer systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 858–869, 2004.

[6] J. R. Douceur, "The Sybil attack," in *IPTPS*, vol. 2429. Springer, 2002, pp. 251–260.

[7] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard, "A cooperative internet backup scheme," in *USENIX*. USENIX, 2003, pp. 29–41.

[8] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: Making backup cheap and easy," in *OSDI*. USENIX Association, 2002.

[9] I. Clarke, O. Sandberg, M. Toseland, and V. Verendel, "Private communication through a network of trusted connections: The dark freenet," 2010. [Online]. Available: https://freenetproject.org/papers/freenet-0.7.5-paper.pdf

[10] K. Bennett, C. Grothoff, T. Horozov, and J. T. Lindgren, "An encoding for censorship-resistant sharing," 2003. [Online]. Available: https://gnunet.org/svn/GNUnet-docs/WWW/download/ecrs.pdf

[11] L.-H. Vu, K. Aberer, S. Buchegger, and A. Datta, "Enabling secure secret sharing in distributed online social networks," in *ACSAC*. IEEE Computer Society, 2009, pp. 419–428.

[12] N. Frykholm and A. Juels, "Error-tolerant password recovery," in *CCS*. ACM, 2001, pp. 1–9.

[13] K. Wehrle, S. Götz, and S. Rieche, "Distributed hash tables," in *Peer-to-Peer Systems and Applications*, vol. 3485. Springer, 2005, pp. 79–93.

[14] R. Jiménez, F. Osmani, and B. Knutsson, "Sub-second lookups on a large-scale Kademlia-based overlay," in *P2P*. IEEE Computer Society, 2011, pp. 82–91.

[15] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Computing Surveys*, vol. 25, no. 3, 2007.

[16] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer, "Brahms: Byzantine resilient random membership sampling," *Computer Networks*, vol. 53, no. 13, pp. 2340–2359, 2009.

[17] K. Bennett, C. Grothoff, T. Horozov, and I. Patrascu, "Efficient sharing of encrypted data," in *ACISP*, ser. Lecture Notes in Computer Science, L. M. Batten and J. Seberry, Eds., vol. 2384. Springer, 2002, pp. 107–120.

[18] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," in *FAST*. USENIX, 2003.

[19] International Electrotechnical Commission, "ISO/IEC 27002:2005. Information technology – Security techniques – Code of practice for information security management," International Organization for Standardization, 2005.

[20] W. Ford and B. Kaliski Jr., "Server-assisted generation of a strong secret from a password," in *WET ICE*. IEEE Computer Society, 2000, pp. 176–180.

[21] J. G. Brainard, A. Juels, R. L. Rivest, M. Szydlo, and M. Yung, "Fourth-factor authentication: somebody you know," in *CCS*. ACM, 2006, pp. 168–178.

[22] G. Scoville, "Good security questions," http://goodsecurityquestions.com/ (19th April, 2012).

[23] A. Rabkin, "Personal knowledge questions for fallback authentication: security questions in the era of Facebook," in *SOUPS*. ACM, 2008, pp. 13–23.

[24] G. Urdaneta, G. Pierre, and M. van Steen, "A survey of DHT security techniques," *ACM Computing Surveys*, vol. 43, no. 2, p. 8, 2011.

[25] N. Provos and D. Mazières, "A future-adaptable password scheme," in *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 1999, pp. 81–91.

[26] K. Rzadca, A. Datta, and S. Buchegger, "Replica placement in P2P storage: Complexity and game theoretic analyses," in *ICDCS*. IEEE Computer Society, 2010, pp. 599–609.

[27] A. Rushinek and S. F. Rushinek, "What makes users happy?" *Commun. ACM*, vol. 29, no. 7, pp. 594–598, 1986.

[28] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying interactive user experience on thin clients," *IEEE Computer Society*, vol. 39, no. 3, pp. 46–52, 2006.