Private Notes: Encrypted XML Notes Synchronization and Sharing with Untrusted Web Services

Paul Klingelhuber
University of Applied Sciences Upper Austria
paul.klingelhuber
@students.fh-hagenberg.at

Rene Mayrhofer
University of Applied Sciences Upper Austria
rene.mayrhofer@fh-hagenberg.at

ABSTRACT

Personal notes, even when shared with others, often contain highly sensitive information. From a security and privacy point of view, currently available (web) services that upload such personal notes to potentially untrusted third party servers are therefore problematic and we suggest to encrypt all notes before transferring them from the user's personal device. However, synchronization and sharing of encrypted data is a non-trivial issue, because conflict resolution and merging algorithms need to be applied to plain-text content. With *Private Notes*, we propose an architecture for client-side encryption, merge, and conflict handling of personal notes stored in XML format. We adopt the OpenPGP standard for symmetric and asymmetric encryption and WebDAV for synchronizing and sharing notes on arbitrary web servers. Specific implementations in the form of a plug-in for the Tomboy desktop note taking application and the Android and iOS mobile platforms demonstrate the ease of use of encrypted notes sharing.

Categories and Subject Descriptors

H.4.3 [**Information Systems Applications**]: Communications Applications

Keywords

note taking applications, XML synchronization, client-based encryption

1. INTRODUCTION

Management of personal or public notes is one of the main components of so-called Personal Information Management (*PIM*), and was one of the early applications of mobile devices. Use cases for note keeping are diverse and include the typical shopping and to-do lists, meeting protocols, or "notes to self". Mimicking the way people take structured or unstructured notes during their daily life, mobile devices should support quick creation, modification, and retrieval of arbitrarily formatted notes with little overhead.

Mobile phones are attractive devices for note taking because they are usually kept with the user and are therefore (mostly) always available. On the other hand, entering longer texts is awkward and slow on small keyboards or touch screens. Therefore, one important feature of note management applications is *synchronization* of specific notes or sets of notes between multiple devices. With notes synchronization, users may create or modify notes on devices where and when it is most appropriate (e.g. using a laptop during a meeting or a tablet at home) and retrieve them independently (e.g. with their mobile phone). Consequently, various commercial applications already address this issue of note management across multiple devices, e.g. Evernote[1], Remember the Milk[2], and Springpad[3] as web services with mobile phone clients.

One critical problem with these services is that all notes are stored in plain text on the central web service, and users therefore have to trust the service operators to keep their notes secure from other users (against the plethora of web service attack techniques currently available) and not to exploit this data themselves or intentionally share them with others. Considering the highly personal nature of many of the notes users keep, there is a significant risk involved in sending them to third party services. With *Private Notes*, we aim to solve this issue by encrypting and decrypting notes on end-user devices (such as mobile phones and laptops) and only storing fully encrypted versions of personal notes on third-party services. Therefore, these services, typically realized as web services, only need to be trusted to provide storage with reasonable availability, but do not need to be trusted to guarantee security and privacy.

However, there are two main issues that need to be addressed for client-side encrypted notes synchronization:

a) Synchronization of arbitrary binary files – including the encrypted versions of textual notes – is a non-trivial issue. With disconnected mode of operation and client-side caching, which is the standard use case for most mobile applications, binary files may be changed independently on multiple devices between two full synchronization rounds, and may therefore diverge from each other. Automatic merging of arbitrary binary formats cannot be guaranteed and would therefore be left to the user

iiWAS2011, 5-7 December, 2011, Ho Chi Minh City, Vietnam

[1] `http://www.evernote.com`
[2] `http://www.rememberthemilk.com`
[3] `http://springpadit.com`

to solve (e.g. the Dropbox[4] file synchronization service simply creates copies of files when such conflicts occur).

b) Sharing encrypted content requires all users (respectively their devices) to have access to the cryptographic key material required to decrypt the data. This means to either use a shared password for deriving the same secret key (which is problematic in case access should be revoked for some users) or to use session keys which are in turn encrypted for each of the authorized users.

In the Private Notes project, we solve the issue of synchronization by not applying any conflict handling to the encrypted binary data, but including text merge capabilities for the XML formatted notes within the client applications. Updating the encrypted versions of notes on the (untrusted) web service is the sole discretion of clients, after handling conflicts in the plain-text version. Sharing encrypted context is solved by encrypting the required session key to decrypt the actual note with each authorized user's public key based on the OpenPGP standard, which allows interoperability with well-known cryptographic tools and potentially (in future versions) with email clients.

The present paper makes three contributions:

- We present an architecture for client-based encryption and synchronization of notes with untrusted, storage-only web services. All necessary methods for en- and decryption and merge/conflict handling are included in client applications for privacy reasons and to support a wide range of web services for cloud based storage of notes. To support a wide range of client devices and applications, we define three different encryption schemes and discuss their respective advantages and disadvantages.

- We introduce a user-friendly sharing mechanism that relies on the OpenPGP standard for handling multiple (private) decryption keys and an URL scheme for flexible distribution of "links" to shared notes.

- We describe specific implementations for desktop operating systems (Windows, Linux, MacOS/X) based on a Tomboy[5] plug-in, for Android by extending the previously incomplete Tomdroid application, and a new implementation for iOS (e.g. for iPhone). An initial implementation for the storage-only notes sharing web service is based on the WebDAV standard and minor server-side extensions for creating note specific URLs derived from GUIDs (globally unique identifiers).

2. RELATED WORK

In the field of data synchronization, there is a breadth of solutions available, because of the greatly different areas of application. When narrowed down to PIM data synchronization, there are still several established technologies such as ActiveSync [6] or SyncML [18]. While ActiveSync is a widely used protocol especially for business use with Microsoft Exchange servers or newer alternative implementations such as Zarafa Z-Push, SyncML is an open alternative that is also widely adopted and has been in use on mobile devices (feature phones) before smartphones became popular. Both of these protocols support secure transport via HTTPS, which only secures the transport from the synchronizing client to the server. The server therefore has full access to the data, which is also necessary because in both protocols the server plays a central role in the synchronization process, e.g. in handling conflicts. With both protocols, capabilities are mostly defined by client devices. Furthermore, both protocols define data types for elements that can be synchronized (like messages, contacts and other PIM related information) and serialization for these types which are mostly XML based [6, 8].

ActiveSync and SyncML use XML as a serialization format, and therefore the servers can perform synchronization on the raw data. In contrast, there are also systems that allow distributed handling of XML structures such as the middleware XMIDDLE [23] which aims at making it easy for multiple clients to access and change XML data while abstracting possibly unreliable network connections. Another example is the work on cooperative access to XML documents via lock based protocols by Helmer et al. [17], which provides locks on sub-trees of XML documents which can be used to coordinate write access. In contrast to ActiveSync and SyncML, these systems aim more at simultaneous cooperation on documents. When thinking about notes as the digital counterpart of sticky notes a simultaneous cooperation on editing them seems less likely, however when thought of as sketch-pads, simultaneous editing would be a reasonable use case.

However, all the mentioned protocols and approaches lack the possibility to be used with an untrusted server, and therefore violate one of our design goals. With untrusted (web) services, all en- and decryption needs to be done on the client side. One possible solution is the OpenPGP message format [10] which provides both encryption with a shared password and with public/private key cryptography for multi-party notes sharing. As mentioned above, OpenPGP does not support synchronization of its (binary) encrypted output. When the server is not fully trusted, synchronization needs to be done on the client, identifying conflicts and resolving them on the decrypted (plain text) format, e.g. with state-based merging [19].

One closely related approach with more emphasis on (distributed) versioning is to use Git with OpenSSL for en-/decrypting the backend repository files [25]. This setup may work in a mixed platform environment with implementations like JGit[6] (the Java Git implementation), but has not yet been demonstrated in practice. Although it imposes more work on the clients, it provides full versioning capabilities, which our Private Notes project currently lacks. An additional open issue is the integration of public key cryptography, because the setup described in Shang's paper [25] requires a deterministic encryption to not disturb the operation of the underlying version control system.

3. NOTE ENCRYPTION

In this section, we describe a simple format for symmetric encryption of files as an alternative to the more complex OpenPGP standard. It was designed with a focus on small files such as notes, to be usable on devices with significantly limited resources, and to support client-side synchronization. Synchronization is based on a timestamp encoded in

[4] `http://www.dropbox.com`
[5] `http://projects.gnome.org/tomboy/`
[6] `http://eclipse.org/jgit/`

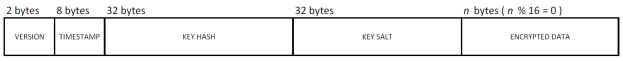

Figure 1: Top Level of the encryption scheme.

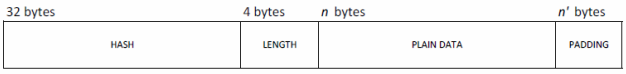

Figure 2: Contents of encrypted part.

the header of the encrypted data, which is used to speed up synchronization when only files in a certain time interval need to be considered. Validating the timestamp based on the embedded hash is only possible after decryption, but the order for processing different notes can be determined beforehand, therefore speeding up the whole process. Furthermore, the filename is part of the verification hash because in the Tomboy notes file format, the filename identifies the individual notes, which consequently needs to be validated as well.

The used cryptographic primitives are: the SHA-256 hash function (for all hashes) and the AES encryption algorithm in CBC mode. An encrypted file consists of a version information, followed by the timestamp, the key hash, the key salt, and finally the encrypted data part. This is depicted in figure 1. The encrypted data contains a message authentication code (MAC), the length of the plain text, the plain text and padding to multiples of the AES block size (see figure 2). The MAC is a SHA hash of the version, the timestamp, the filename, and the plain text including padding. It is used to protect the data against unnoticed altering of the data (either malicious or accidental). This does not mean that we can recover the original data if it has been changed, but detect it. The input data for the hash can be seen in figure 3. This scheme is deliberately kept very simple on the design principle by Ferguson and Schneider "Complexity is the worst enemy of security." [15].

Our scheme takes a password as input with UTF8 encoding and uses it as a symmetric key for the encryption. As keys for the actual encryption we use the result of hashing the password bytes with the used salt once. The data that is stored in the file is the result of repeating this operation again using the same salt. We use a salt length of 256 bits as suggested by [15]. We do not use stretching, which is the repeated application of the hashing algorithm before using the result after a fixed number of computations as the key, because the scheme should be easily adoptable on devices with limited resources. It is questionable if there are parameter combinations for stretching that impact the performance of brute-force attacks on state-of-the-art machines while not diminishing performance on limited mobile devices to the point of decreasing usability.

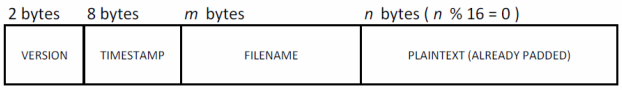

Figure 3: Data that is used as an input for the modification detection hash.

Comparison to OpenPGP

Compared to the OpenPGP format, our simplified scheme is similar in some respects such as support for salted keys, but differs in the usage (OpenPGP uses only 64 bits of salt). The OpenPGP structure is composed of so-called packets which are put together to form an OpenPGP message. Because of this structure, there are many different types of information that can be contained in such a message, such as a signature packet or a symmetrically encrypted data packet [10]. Undoubtedly the OpenPGP format offers greater versatility, but therefore requires significantly more effort to support even if only the required features are implemented for compliance. On platforms without OpenPGP libraries (such as the BouncyCastle [3] implementation in Java), this implementation effort may be prohibitive. We chose to use a straightforward and simpler, albeit less flexible design as an alternative for limited mobile platforms. Its implementation only needs to support SHA-256 and AES with a blocksize of 128 bits and a 256 bit key.

In terms of execution speed, we tested a C# implementation of the OpenPGP format with the BouncyCastle library and a C# version of our scheme on a file of 455 bytes. Our implementation took 141 ms to run whereas the BouncyCastle implementation took 525 ms. This difference in execution speed is partly probably due to the fact that BouncyCastle uses its own AES implementation and ships with many other capabilities built in, however it shows that our scheme can be easily implemented and perform well even without optimizing the code for execution speed.

A major advantage of OpenPGP over our scheme is that it supports public key cryptography, which is especially useful when implementing our proposed notes sharing functionality described in more detail below.

Interchangeability

Since the structure of our scheme is not defined in terms of smaller units (like the OpenPGP packets) and the header format is different as well, these two schemes are obviously not directly interchangeable. However, it would be possible to easily handle data in both formats. Because our scheme is only specified with one version number, the first bit always has the value 0, OpenPGP messages on the other hand always start with a 1 bit (except ASCII-armored files, which could however be identified by their textual header). With this distinction, the appropriate scheme could be automatically selected for decrypting notes.

3.1 Encryption with Shared Passwords

Shared passwords are the most familiar encryption mechanism for end-users, because they are comparable to the passwords used every day to log into their computers, e-mail accounts, or PIN codes for cash machines to withdraw money. In the context of this paper, shared does not necessarily mean that these passwords are shared with other people, but in the sense that it is shared between users and the party they interact with when using the key — such as multiple personal devices that synchronize the user's notes.

Shared keys can have diverse characteristics; as stated above, they can be simple PINs or strong pass-phrases. Typically, these types of passwords are not directly suitable for mass data encryption, as encryption keys need to have a defined length (such as 128 bits) and need to be distributed uniformly. The transformation from an arbitrary password

to such a key is described by so-called string-to-key parameters. This can be for example a specific hashing algorithm and values that describe the number of applications and a salt [15].

Shared passwords are practical when the encrypted data only needs to be accessible by one user. They become a problem when a user wants to share access with another user to a certain resource. Even if the user chooses a new password for this purpose, it becomes more tedious and users may therefore tend to choose weaker passwords or make other mistakes. As surveys and statistical analysis frequently show, people are typically not very good at choosing good passwords [5, 2, 21].

We decided to use shared passwords for the encryption of the notes that are synchronized between the devices of one user. Depending on the device it is important to give the user control about the handling of the password. On a smartphone for example, a user might want to only temporarily cache the password, if they are afraid that the device might get lost or stolen.

Both the simplified encryption scheme described in section 3 and the standard OpenPGP symmetric encryption scheme are suitable for this use case. Our application designs described below allow for easy exchange with other encryption formats. In fact, there is already a module for encryption via GPG and it would be trivial to make it configurable by the user which one to use. When using GPG, we paid attention to set the appropriate options that promise robust security, such as using AES and forcing the use of an *MAC* to prevent undetected modification of encrypted files.

3.2 Encryption with Asymmetric Keys

Asymmetric keys, also called public- and private-key pairs, are a widely accepted alternative when dealing with multiple users who want to share information with each other. By encrypting the message with the public key of the receiver, only they can decrypt it with their private key. The OpenPGP format for example allows the encryption of a message with multiple receivers. This is achieved by creating a new intermediate (random) key, also called a session key, which is used to encrypt the actual message. This key is then encrypted with the public key of every recipient. This has several advantages, one being that the amount of data that needs to be transferred only increases minimally with each user. A second reason is that the session key will be different each time and therefore the amount of cypher text from the same key an attacker has to do analysis on stays small and makes attacks more difficult [10].

The collection of public keys a user has is referred to as their public keyring. The private keyring of a user contains their personal keys. When a user receives a public key from another person they know, they should contact that person and verify this key by comparing the fingerprint over an out-of-band channel (such as a phone call when the voice of the other party is known). Key trust is the way to handle bigger sets of keys. This allows users to mark somebody they know (respectively their keys) as trusted, which can mean that they will trust every key that the other person trusts, depending on the trust-level [9].

One might argue that when dealing with multiple devices the handling of the key pairs is again a problem. This is only partly true because the public keys don't have to be kept secret (as their name suggests). Concerning the private keys it is indeed not trivial to distribute them securely to all devices that need it. Because the private keys are a very sensitive part, they are typically protected by a passphrase that is only known to the key owner [9] and is entered on every start of the respective application. These applications typically cache them for a short time, so that the user can execute several operations consecutively without having to enter it over and over again. However, after this time the passphrase is discarded from memory again and will be re-requested if necessary.

When looking at scenarios with multiple recipients, where new messages are encrypted to everybody from the group there might still be the problem that one of the participants does not know all the others and might not trust the one that created the initial message enough to accept all relevant public keys. This is a difficult situation in which one would have to check directly with the others if their public keys are correct. This however does not make much sense if one really does not know the other person beforehand. In this case one can only refuse to send messages to them or accept their keys for the sake of being able to share messages with the whole group.

To summarize, we suggest to use our simple encryption scheme only on mobile devices with significant resource limitations and to use the OpenPGP standard, either with symmetric encryption for personal notes or with asymmetric key handling for shared notes, whenever the devices are sufficiently capable.

4. NOTE SYNCHRONIZATION

Synchronization of notes between devices can be done in a variety of options. On the one hand a distinction can be made between synchronization that always transfers the whole note as an update versus only transferring the deltas. On the other hand the transport and connection topology between the synchronizing devices is a key differentiation factor. There can be ether a client-server setup or a peer-to-peer (*P2P*) topology (or also hybrid forms). In the case of a client-server infrastructure there is again ether the possibility to have a server that actively assists with the synchronization process or to have a passive server that is basically only a global storage node. Examples for active servers are amongst others ActiveSync and SyncML as described in section 2. Servers that only offer storage can be implemented via public standards such as NFS [26] or WebDAV [14], or it can also be services such as Dropbox[7] or Ubuntu One[8].

As discussed before, the synchronization methods with active servers typically provide security only for the communication channel — the server, its operators, and the country it is hosted in need to be implicitly trusted. Regarding the P2P methods, there is for example a system that extends and adapts the CVS to allow peer-to-peer synchronization [22]. This one also does not tackle encryption as well. A similar approach is the already mentioned Git with OpenSSL [25] which brings encryption of the files contained in the VCS. This is roughly the direction that we also took, but without the full versioning functionality. Additionally, we only use the client server based approach because there are fewer options that need to be covered compared to peer-to-peer systems, like multiple interwoven version trees. Another reason

[7] `http://www.dropbox.com`
[8] `https://one.ubuntu.com/`

is that in a mixed environment of PCs and mobile devices it is very likely that not all of them are always reachable at the same time, therefore it is the preferable solution to have a server that is always available.

4.1 Private Notes synchronization

Our basic strategy is to use a globally accessible storage server, in our case WebDAV because it is a standardized file-access protocol that was built for the web and is currently widely used (for example by Microsoft SkyDrive[9] or Apples iCloud). All notes are encrypted before transfer to the server with a shared key of the user, because these notes are only for their private use and therefore a shared key is sufficient and also applicable for users who don not have their own key pairs.

Because a file based storage service is used, we adopt the approach implemented by the Tomboy file based synchronization, which means that the server directory used for notes synchronization includes a manifest file that holds the latest version number and a list of all notes with their respective version numbers. Every note has an associated version that is incremented with each change when synchronized to the server. In other words, this count can locally only increase by 1 between two synchronization transactions. Versions are kept on a per-note basis and the global version is simply the maximum across all values.

When multiple clients (say A and B) access the same server, it can obviously happen that version numbers change by more than 1 on the server between two synchronization transaction of client B because client A has made multiple changes and synchronized them individually. This is not necessarily a problem, only when changes are made to the same file a conflict arises. This is similar to version control systems like CVS, except that these systems typically attempt automatic merging and only report these states as conflicts if merging fails [11].

The conflict handling in Tomboy normally does not do any automatic merging but simply allows the user to prefer one of the versions or save both versions so he can resolve them later. On the mobile client Tomdroid, we chose to implement a automatic merging for conflicts, because on smartphones it is difficult for a user to compare two notes and make a manual merge. The automatic merge still contains some annotations which show the user where changes happened so they can quickly fix the content without having to look at two different notes.

In our case, all files that are stored on the WebDAV server are encrypted, but their names and their content (apart from being encrypted) are the same. When a client synchronizes with the storage location, i.e. the WebDAV URL for this user, it first retrieves the manifest file, verifies the versions of all notes, and downloads those that have changed. The client then locally decrypts the files to be able to access their content. When there are local changes, the manifest file is updated and the updated notes and the new manifest file are uploaded to the WebDAV server after being encrypted again.

This system with file-based synchronization endpoints only supports one actively synchronizing client at a time. This can be achieved for example by lock-files, which is widely used in practice (apache web server, eclipse IDE etc.). With WebDAV, locking functionality is provided by the protocol itself. However, because the specification does not require compatible implementations to support this feature, it depends on the specific WebDAV server implementation if it can be used [14].

[9] `http://skydrive.live.com/`

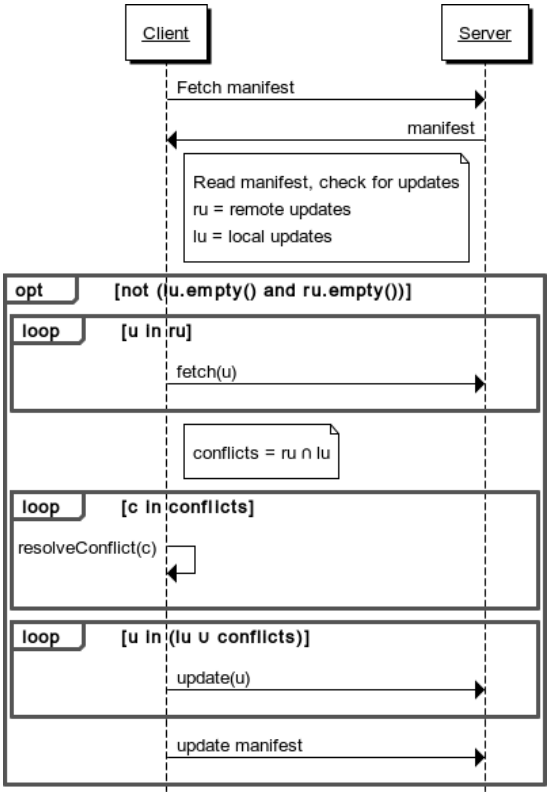

Figure 4: Notes synchronization scheme.

Figure 4 summarizes the basic process of synchronizing with a WebDAV server as described above. Setting and resetting of locks is not shown, as it is implementation specific and not important for the synchronization procedure itself.

4.2 Note Sharing

Sharing encrypted notes with other users requires changes to the applied synchronization scheme. Encryption is only one part for which a different approach is required. As discussed before, asymmetric key cryptography with a session key encrypted with each participant's public key is the most straight-forward means for notes sharing. An alternative to automatic encryption key handling are various group-keying protocols (e.g. [20]), which are unsuitable for our purpose as they require online connectivity between all parties. The last possibility – to let the user choose new pass-phrases for every share – is not realistic because this would favor poorly chosen passwords and necessitate a secure channel to transmit the password to every participant before they can take part in using the shared note.

For Private Notes, we define the following aims for note sharing:

- It should be easy to mark something as shared.
- For every share, it should be possible to choose different partners.
- Sending references/links to shares and adding them at

the other (receiving) party should be as user-friendly and flexible (in terms of the way of transmitting the link) as possible.

Encryption of shared notes

Because we rely on the OpenPGP message format, dealing with multiple recipients is unproblematic, because it is supported to encrypt a message with multiple public keys [10]. A precondition is that we already have the public keys of all the parties we want to share with and that they have ours (cf. section 3.2 about multiple recipients and mutual acquaintanceship).

In our implementation, when the user wants to share a note, we present a searchable list of all the keys that are available to his OpenPGP tool (e.g. the platform `gpg` binary). Although the OpenPGP standard already supports managing lists of public key ids in its messages, the actual recipients may be hidden when creating the OpenPGP message. For future extensibility concerning the applied encryption format, we therefore explicitly store this list of all key ids with access to a note. The manifest of a shared note, which contains the note id and revision number, is extended by the key ids of all participating parties. An example of such a shared manifest is shown in listing 1. This manifest is then stored alongside the shared note at the respective location as represented e.g. by the WebDAV URL for this note.

Listing 1: Shared manifest

```
1 <sync revision="0" server-id="8ca05352-f733-4
      ff0-a7c9-bdae0198978e">
2   <note id="9d60b164-e3aa-4382-b783-
      ae0d2f9c115d" rev="0" />
3   <shared>
4     <with partner="8D7E 68DD" />
5     <with partner="EF11 D60B" />
6     <with partner="B164 5EE8" />
7   </shared>
8 </sync>
```

WebDAV storage of shared notes

Another factor is that unlike synchronization, a user might not want to share all their notes with other people. People participating in any share must always be prevented from gaining access to any of the other notes via the same storage location. Therefore, we define shares on a per-note basis, where the user can decide for every note if it should be shared and with whom they want to share it.

To secure all other notes from being accessible via the shared note, we always create a separate sharing location on the web service. For our implementation we decided to automate the process of creating new storage locations for shares on the server. The "passive" server is involved in this action insofar as it needs to allocate a new WebDAV account with an associated URL and username/password combination for each share. Our demo server uses an apache server with WebDAV support, a PHP script for invoking this process and a shell script for coordinating the account creation. In our system, a simple script creates a new WebDAV account with a random name and random credentials. This means that shared items are no longer synchronized to the main synchronization location.

Shares are also not restricted to the server which is responsible for the normal synchronization. This means that shares could in theory be stored on many servers, or a user could also use a private server if they think it is better suited to hold their shared notes. The only point that prevents a user from doing this currently is that since the creation of the WebDAV accounts for sharing is automated, the server is assumed to be our demo server, because no other server currently runs this script. Of course everybody can deploy the same (or similar) solution on an arbitrary web service. To support this in our Tomboy add-in, one would need to add an option to configure the server or let the user put in the service URL that should be used for the share. This is again only needed for the party that creates the share, other participants would not have to do any extra work.

Distributing share access is handled in Private Notes in the following way: the client offers an option to create a *share link*. This link contains an application specific URI prefix `note://tomboyshare/` followed by a link which contains all information needed to access the share location. The application specific prefix is included for several reasons. First of all, it helps users to see that they are not dealing with a normal link to a website. Secondly, the `note://` prefix is already used to trigger certain operations in Tomboy via the command line and can be registered (depending on the platform and operating system) for URI handling. If the platform supports it and the application is correctly registered, the user is then able to open the link to a shared note as easily as opening a webpage link. This is supported by Microsoft Windows via registry entries, Mac OS X via Launch Services, on Linux depending on the desktop environment, and by some mobile device operating systems such as Android via Intent Filters. Without platform support for URI prefix handlers, users can manually import the share link into the Private Notes client by pasting it into a standard input field.

There is an additional issue to solve for integrating shared notes: when synchronizing all items (including the shared ones) as if they came from the same source, it can easily happen that a shared note has not changed but some of the private notes have, and therefore the global version number has increased. If we now copied this global version number into the share manifest, it would be misleading for other clients by forcing a re-synchronization without changes to the note content. Therefore, the shared items should have their own isolated maximum version number and Private Notes clients need to be adapted to deal with these multiple locations and synchronization revisions.

If we look again at figure 4, the changes we need to make to integrate sharing into this synchronization remain reasonable. When fetching the manifest, we also have to retrieve the manifests from all the shares. The shared manifests also need to be checked for updates. In the context of the diagram, we would consider these updates part of the remote updates. When fetching or updating (uploading) the notes, the client then needs to use either the normal sync location or a share location. In the end, we have to update the normal synchronization manifest (if there were updates to the notes that are covered by it) and also all share manifests of notes that have been updated. Additionally, the selected encryption format has to be applied to every upload and download.

Comparison to similar services

When comparing our sharing approach to other storage services that allow sharing such as Dropbox, there are several differences. Dropbox is a closed-source service which has its own client programs and APIs. We only use WebDAV for our storage which is a public standard. As explained above, when a user wants to share something with others a new WebDAV account gets created. The credentials for this share are encoded into the share link that they can freely distribute to all participants. This means that anybody who can acquire the link gets access to the files stored at this location. However, only encrypted files reside there, which is not useful to potential attackers without the (session) key. If we in contrast look at how Dropbox handles shares, a registered user can mark a folder as a shared folder and add other registered users as participants. They will receive an invitation (an email and it will appear on their account website) and can then join the share. The Dropbox approach obviously has some advantages, such as that a user can remove users later (if you are the initiator of the share) which are then no longer able to change the contents of the share, however they still keep a copy of the version of the files just before being removed from the share [13].

The advantage of our approach is that because the Dropbox service is a user account centered service where one has to create an account and use the proprietary clients or use their API, sharing with anybody who does not use this service is not possible. In our architecture, everybody could run their own share server and there is no need for registration to access any other share server as long as a correct share link is distributed.

Furthermore, with a service like Dropbox, the user is at the mercy of the respective company. Although Dropbox for example claimed that their content was encrypted somehow on their servers, this is subject to software errors in their server infrastructure [12], abuse by administrators with sufficient access level, and law enforcement overrides. Data stored remotely can only be assumed to remain private if the server is only entrusted with encrypted data which is impossible to decrypt on the server side like in our approach.

5. DISCUSSION AND CURRENT LIMITATIONS

The described functionality was implemented as a prototype which includes an add-in for the notes application Tomboy, an adapted version of the mobile client Tomdroid, a new mobile client for iOS, and a WebDAV server set-up including the required scripts for automatic share URL handling. The add-in enables users to declare notes for sharing and to select partners from a list of public keys. It also automatically interacts with the extended WebDAV server for creating new shares. Mobile clients are able to import those shares and update the notes, while handling of share participant is not supported due to the restricted user interface.

There are still a few areas for future improvement, mainly in terms of shared notes security and conflict handling.

5.1 Security related

Concerning the manifest file and the handling of share participants there are some issues that we need to handle. As described in section 4.2, the manifest file contains a list of participant keys. Assuming that a possible attacker somehow gains access to the location where the shared file is stored, the following scenario would be possible: The attacker is unable to decrypt the files. However, the layout of manifest files is publicly documented, the attacker could construct a new one. He would have to know somebody else who is participating, for example Bob. The attacker would now insert his own key-id plus the key-id of Bob into the manifest. The name of the item that is shared is simply the name of the file that is stored along with the manifest. The next time Bob synchronizes, he reads the keys he needs to synchronize to, and encrypts the files (including the shared item) for these keys (which now includes the attackers key). Afterwards, the attacker could simply decrypt the shared item with his own private key.

This of course only works under the following preconditions: The attacker is somehow known to Bob (he knows his public key); Bob doesn't check that all files were encrypted by the same person; and Bob doesn't get suspicious that the recipient-list has changed. To make things worse, it could be that Bob has never synchronized with the share before, so it could be possible that he doesn't even know the old recipients.

Obviously this scenario would mark a huge risk so it has to be prevented. There are a few points that need to be done to achieve this: Bob needs some way to verify that the files he is looking at have not been tampered with. As stated above, he might not have had access to the files yet, but what he already has is the share link. So to give Bob the possibility to verify the share, there needs to be sufficient information encoded in the link to verify the authenticity of the manifest file. One possibility is to encode (parts of) the public key fingerprint of the "owner" of the share location, i.e. the person who originally created the node, and to sign the manifest file with the associated private key. The disadvantage is that only the original owner could make modifications to the manifest, which is problematic for bidirectional synchronization.

There is a possible variation to this, namely including all the key-ids (or again alternatively some hash value of their concatenated values) in the manifest as well as the encrypted OpenPGP message, allowing the modified content to be signed by any of these keys. This has however some negative side effect: one cannot introduce new people to the share after it has been sent to all the participants. Depending on the application scenario, this could be a desired effect, but very likely it is not.

A second option is to use the server-id, which is not related to any other value (like the server path, shared file or anything else) so it cannot be calculated from anything else. When an attacker creates his own new manifest file, he cannot know which server-id was in there originally, so it will be different. An important improvement would then again be to not directly include the server-id but a hash value of it. This would make the share link less sensitive against compromise, because even if an attacker accomplishes to get the share link, he would have to find a matching server-id before being able to trick a user.

A totally different issue is key distribution, which obviously plays a big role in this system, because every party needs the public keys of the others. One way is to send them along with the share links, or to rely solely on PGP key servers. However, there should be at least some method to inform the user if they need some additional keys. The

concrete solution is again application and/or scenario specific and is therefore out of scope of this paper.

5.2 Conflict resolution

When multiple people work on the same note, it is obvious that version conflicts occur more often, compared to a single person that only synchronizes different devices. As conflicts are more prominent, they should be easy to deal with for the user. Conflict resolution can be automated to a certain degree. We recommend putting those parts that are conflicted together in the same "conflict resolution required" block in the note (one version under the other, with some symbol indicating that there was a conflict and which part was found in which version) and inserting items that only changed in one version. Some of the currently prevalent versioning systems such as Subversion or CVS use the diff3 algorithm for this task [19].

If one needs to implement similar functionality for their own application the Google diff-match-patch library is another candidate. The library is available in a wide range of languages including JavaScript, $C++$, Objective-C, and Lua for example and it is also used by Google itself for their Documents web application [16]. As described in section 4, we use it for automatic merging in the Android mobile client and it was very easy to integrate into our implementation.

There are also systems for peer-to-peer shared synchronization which embrace the fact of uncertain availability in these distributed environments. These systems use slightly different approaches towards handling conflicts between different versions. There, not only the latest state of files is relevant for the merging, but also their full history of changes. A system for handling such version conflicts in a flexible way was for example created at the Helsinki Institute for Information Technology [24].

6. CONCLUSION AND FUTURE OUTLOOK

While our current implementation works and shows how secure sharing can be done, there are undoubtedly numerous possible improvements. From the usability point of view, one of the first things would probably be a better integration into the tools that are responsible for the PGP encryption. Especially in our android prototype where APG[10] is used for the PGP encryption, there are some disrupting screens that appear while syncing with which a user has to interact with. This could certainly be improved via tighter integration into our application.

A possible direction in which the project might develop in the future is to usability and on-line cooperation. It would be a big improvement in terms of interaction, if a system would bring together the nearly instant cooperation capabilities similar to EtherPad or Google Documents with the security of our sharing system. Obviously this would not be realized with file uploads to WebDAV shares, but via some server, achieving this in a way that is both responsive while collaborating and still offers the possibility for others to synchronize with the last changes would definitely be a challenge.

One step that could be a first one into that direction could be experimenting with encrypting only diffs and sending them to the collaborators. This would be in some ways comparable to adding encryption to a versioning system. As already mentioned in section 2, something similar has been proposed [25], however in a slightly different way where the encryption lies before the creation of the diffs.

All parts of our implementation of Private Notes, including the Tomboy plugin, the Android and the iOS clients, and the server scripts are available under open source licenses at `https://gitorious.org/privatenotes`.

[10]Android Privacy Guard http://thialfihar.org/projects/apg/

7. REFERENCES

[1] Android developers web page. `http://developer.android.com/`.

[2] Rsa security survey reveals multiple passwords creating security risks and end user frustration. `http://www.rsa.com/press_release.aspx?id=6095`, 2005.

[3] The Legion of the Bouncy Castle web page. `http://www.bouncycastle.org`, 2008. accessed 2008-10-13.

[4] Mac os x developer library. `http://developer.apple.com/library/mac/documentation/`, 2009.

[5] Password survey results. `http://www.symantec.com/connect/blogs/password-survey-results`, 2010.

[6] Activesync http protocol specification, March 2011.

[7] Microsoft developer network web page. `http://msdn.microsoft.com/`, 2011.

[8] O. M. Alliance. Syncml representation protocol, July 2009.

[9] J. M. Ashley. The gnu privacy handbook. 1999.

[10] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC2440: OpenPGP message format, November 1998.

[11] P. Cederqvist. *Version Management with CVS*. Free Software Foundation, Inc., 2008.

[12] cnet News. cnet news - dropbox confirms security glitch–no password required. `http://news.cnet.com/8301-31921_3-20072755-281/`, June 2011.

[13] Dropbox. Dropbox help. `http://www.dropbox.com/help/156`, November 2010.

[14] L. Dusseault. RFC4918: HTTP extensions for web distributed authoring and versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007. Updated by RFC 5689.

[15] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing, 2003.

[16] Google. Diff, match and patch libraries for plain text. `http://code.google.com/p/google-diff-match-patch/`, 2010.

[17] S. Helmer, C.-C. Kanne, and G. Moerkotte. Lock-based protocols for cooperation on xml documents, 2003.

[18] A. Jönsson and L. Novak. Syncml - getting the mobile internet in sync. *Ericsson Review*, 03/2001, 2001.

[19] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3.

[20] Y. Kim, A. Perrig, and G. Tsudik. Tree-based group key agreement. *ACM Transactions on Information Systems Security*, 7(1):60–96, May 2004.

[21] D. V. Klein. "foiling the cracker": A survey of, and improvements to, password security, 1990.

[22] H. Larkin. Applying concurrent versioning to serverless mobile device synchronisation. In *ACIS-ICIS*, pages 157–162. IEEE Computer Society, 2007.

[23] C. Mascolo, L. Capra, and W. Emmerich. An xml based middleware for peer-to-peer computing. In *1ST IEEE INTERNATIONAL CONFERENCE OF PEER-TO-PEER COMPUTING, LINKOPING (S.*

[24] K. Rimey. Version headers for flexible synchronization and conflict resolution. HIIT Technical Reports 2004-3 2004-3, November 2004.

[25] N. Shang. Git transparent encryption, October 2010.

[26] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. RFC3010: NFS version 4 protocol, December 2000.