A decentralized approach to publishing confidential data

Konstantin Welke

Master Thesis

July 2nd 2009

supervised by

Prof. Dr. Gerhard Schneider

Technical Faculty

Albert-Ludwigs University of Freiburg

DECLARATION

I hereby declare that this thesis has been composed by me without any assistance and I have not used any sources or tools other than those cited. Furthermore I declare that this thesis has not been accepted in any other previous application for a degree.

Location, Date, Signature

dedicated to my loving wife

Acknowledgments

I want to thank Klaus Rechert for his valuable feedback and patience, which greatly improved this thesis. I am grateful to my supervisor Prof. Dr. Gerhard Schneider for furthering my interest in computer security and for supervising this thesis.

Abstract

This thesis proposes a protocol to allow a user to publish data to a selected group of peers, with confidentiality, authenticity, integrity and in a decentralized way. New users can be added and removed from the group efficiently. A social network was implemented as an example application that uses the protocol presented in this thesis. Using the protocol allows building a social network that, unlike existing social networks, respects the users privacy. This implementation is realized as a web application. The security and privacy implications of the protocol are discussed.

Contents

1 Motivation

Bob took photographs at his party and wants to share them with his friends but with no one else, specifically not with his parents, his employer or a third party like Facebook or Google. Alice maintains an electronic calendar and wants to share certain parts of it with different groups of people: her work appointments should be visible only to her colleagues, some appointments should be visible only to her friends and her personal appointments should only be visible to her alone. How can Alice and Bob share their data in a controlled way?

1.1 Introduction

Traditionally, with some exceptions, users stored their personal data on their own, personal computers. In the recent years, this changed with the emergence of *social networks*. Personal data is moving more and more to central locations. Not only the amount of data that is shared in social networks is increasing, the kind of data is also getting more and more personal. A typical user profile in a social network usually consists of the user's real name, both real-world and Internet contact information, age and interests. Users typically maintain a list of contacts that they know and can access parts of their personal information. Users can upload pictures and *tag* them, i.e. adding semantic information which users are shown on the picture. Using a looser definition of the term social network, which not only includes services such as Facebook[1] or Myspace[2], but also Google[3] and MSN[4], the information that a user stores in the social network extends to text documents, spreadsheets and even his GPS position[5]. In all of these cases, the content is stored in plaintext on the server. The user usually can set permissions, so that only a defined set of their contacts can access a piece of content. Nevertheless,

[1]Facebook: http://www.facebook.com (last retrieval: May 29th 2009)
[2]Myspace: http://www.myspace.com (last retrieval: May 29th 2009)
[3]Google: http://www.google.com (last retrieval: May 29th 2009)
[4]MSN: http://www.msn.com (last retrieval: May 29th 2009)
[5]Google Latitude: http://www.google.com/latitude/ (last retrieval: May 29th 2009)

the social network provider will always be able to access the content. This has drastic security implications, as it means that the user essentially cedes control over his personal information to the provider: the provider could e.g. sell the data to a third party[6], or if the provider's database is broken into, the attacker can easily access the data.

The privacy problems in such centralized systems, where content is stored in plaintext are manifold: The most obvious problem, fairly covered in newspapers, is that users disclose information that would usually be considered private to the public, with various consequences. One of the most well-known examples are party pictures uploaded to a social network, that a future employer might learn about. Another are small misdemenours that users admit or boast about in social networks, that have real-world consequences as authorities learn about them[7]. As a recent, more extreme example of March 2009, a man charged with carrying a loaded gun had that charge dropped, partly because of the content of the arresting cops MySpace page[8]. What all these examples have in common is that information that was designated for a close social group wound up publicly available. Acquisti and Gross [2006] showed that even privacy-aware users have problems judging the extend to which their personal information becomes visible in social networks, and that users generally disclose more information than they are aware of.

1.2 The Confidential Publishing Problem

Privacy has many aspects and facets that greatly depend on context and research focus. On its own, privacy cannot be covered by one single definition. In this thesis, the term privacy is used as defined by Westin [1970]:

[6] As a real-life example, there are reports that German public health insurances sold highly confidential patient information to private competitors. cf. Welt Online http://www.welt.de/politik/article3729919/ (German newspaper article from May 13th 2009) (last retrieval: May 29th 2009)

[7] The Times Argus: http://www.timesargus.com/apps/pbcs.dll/article?AID=/20060226/NEWS/602260375/1002 (Newspaper article from February 26, 2006) (last retrieved June 7th 2009)

[8] New York Times: http://www.nytimes.com/2009/03/11/nyregion/11about.html (last retrieval: May 29th 2009)

> *„Privacy is the claim of individuals, groups and institutions to determine for themselves, when, how and to what extent information about them is communicated to others.“*

Current web services, as well older technologies such as e-mail, do not respect the user's privacy in the sense that they disclose information to third parties without giving the user a choice: In current social networks, any content stored by the user is always accessible by the social network provider. Each user should have control over what information about the user is disclosed to the public. Jiang et al. [2002] introduced the concept of *asymetric information*: In the relationship between a service provider ("data collector") and user ("data owner"), the service provider usually knows much more about the user than the other way around. They introduce the *Principle of Minimal Asymmetry*:

> *"A privacy-aware system should minimize the asymmetry of information between data owners and data collectors and data users, by:*
>
> - *Decreasing the flow of information from data owners to data collectors and users*
> - *Increasing the flow of information from data collectors and users back to data owners"*

In order to protect the user's privacy, the user needs to be able publish content in a way that minimizes the information flow to data collector. More specifically, a user should be able to publish content so that only authorized peers can access the content. This is called the *confidential publishing problem*. The confidential publishing problem consists of publishing content with

- confidentiality
- integrity,
- authenticity,
- a decentralized structure

- and usability.

The content needs to be published with confidentiality, so that the specific peers selected by the user can access it, without any third party being able to access the encrypted content. The content needs to be published with integrity, so that each authorized user can detect whether the information was manipulated. The content needs to be published with authenticity, so that each authorized user can verify that the content is from the user it claims to be from. The content needs to be published in a decentralized structure, prohibiting any single entity or set of entities from easily exerting control over a user (e.g. excluding the user from the network). In a decentralized structure, the user can simply switch to a service provider that does not, for example, censor the user. Ultimately, users can become their own service provider to avoid censorship. It needs to be published in a usable way, because usability is an important practical aspect of security[9]. This includes both a user being able to easily try out the protocol and all security decisions being presented in a way that allows a user to can make an informed decision.

1.3 Outline

This next chapter discusses existing approaches that can be used to solve the confidential publishing problem (section 2). Following that, the cryptographic techniques needed to solve the confidential publishing problem are discussed (section 3). After establishing the cryptographic techniques, an abstract description of the protocol, that excludes a number of details, is presented (section 4). This is followed by a description of how the protocol can be realized on top of HTTP(S) (section 5). The design of an actual implementation of a social network that uses the protocol, is described thereafter (section 6). Afterwards, the security and privacy of this realization discussed (section 7). Finally, the results of this thesis are summarized and an outlook for possible future improvements and applications is presented (section 8).

[9]cf. Ross [2008] Section 26.2 “System Evaluation and Assurance”, p. 859

2 Previous and Existing Approaches

This section discusses how existing approaches can be used to solve the confidential publishing problem.

Single-User Content Encryption Various ready-to-use solutions exist to encrypt data for a single user including programs like Truecrypt[10] or PGP Desktop[11]. These approaches encrypt data, such as hard drive partitions, folders or single files so that only the user him- or herself can read it. In order to securely sent a message to another user, S/MIME (Kaliski [1998]) or PGP (Callas et al. [2007]) can be used. If Bob wants to share a folder with a group of peers, he would encrypt it using one of the programs mentioned above and publish it. He would then share the key needed to decrypt the folder with that group of peers. While this approach works, it has a number of drawbacks: First of all, the entire folder is stored as one big file so just retrieving one file from the folder at a time would be impossible. Second, if Bob decides to exclude one peer from this group, he would need to send all other users a new key, which is inefficient. Third, how Bob can share a folder with multiple groups of users is unclear. In conclusion, a user can use the above techniques to store confidential data for him- or herself, or send confidential data to another user. However, publication of data that changes over time to group of peers is inefficient and cumbersome.

Multicast Security Wallner et al. [1999] describe how IP multicast networks can be secured, so that the transmitted data can only be decrypted by group members. New group members can be added to and existing group members can be removed from the multicast group in an efficient way. Arkko et al. [2003] proposes how the approach for IP Multicast networks can be applied to securing real-time multimedia protocols, such as SIP (Handley et al. [1999]), RTSP (Schulzrinne et al. [1998]) or SRTP (Baugher et al. [2004]). Welke and Rechert [2009] showed how these techniques can be used to allow

[10]Truecrypt: http://www.truecrypt.org/ (last retrieved: May 29th 2009)
[11]PGP: http://www.pgp.com/ (last retrieved: May 29th 2009)

users to share their location (e.g. their GPS position) with exactly the peers they authorized over existing internet infrastructure. As these protocols are used to secure real-time data transmission, they are not directly suitable to publish files or send messages to users which are not currently connected to the network. Thus, they cannot directly be applied to the confidential publishing problem, in which data is first published and then retrieved. Instead, this thesis uses the general techniques proposed by Wallner et al. [1999], such as group cryptography, and applies them to the confidential publishing problem.

Social Networks Social networks such as Facebook and Myspace allow users to publish certain types of data (e.g. text, images, audio and video) to the public (here: other members of the social network). Depending on the social network, users can set access rights so that the published data is only accessible by a defined group of peers, e.g. their group of friends. In any case, the information is accessible in plaintext to the social network operator. Thus, the data is not stored in a confidential way.

While established social networks such as Facebook and Myspace use a central structure to exert complete control, the majority of current research focuses on decentralized structures[12]. FOAF+SSL[13] (Story et al. [2009]) uses public key cryptography to verify user credentials, but the actual user data is stored in plaintext and can thus be read by the server. HelloWorld (Ackermann et al. [2009]) uses public-key cryptography to encrypt user data, but only concerns itself with one-to-one communication. Thus, an important use case in social networks, a user publishing content for a group of peers, cannot be done efficiently.

Freenet Darknets, such as Freenet[14] (Clarke et al. [2001]), aim at anonymous data publishing and retrieval. The data in Freenet is encrypted, meaning the server that stores it does not know what it is storing. Given the

[12]cf. W3C Workshop on the Future of Social Networking: http://www.w3.org/2008/09/msnws/papers/ (last retrieved May 29th 2009)
[13]FOAF+SSL: http://esw.w3.org/topic/foaf+ssl (last retrieved May 29th 2009)
[14]Freenet: http://freenetproject.org/ (last retrieved May 29th 2009)

unique ID of data, anyone can retrieve and decrypt it. The data is routed through intermediate nodes to anonymize the content access. Note that while the data is being encrypted, the encryption is set by the Freenet network and the plaintext is available to any user. However, the 'plaintext' of the data stored in the Freenet network could be encrypted independently. HelloWorld (Ackermann et al. [2009]) mentions that it plans to use Freenet as a publication platform for its one-to-one encrypted data. In itself however, Freenet offers no mechanism to efficiently solve the confidential publishing problem, as it does not concern itself with efficient group cryptography.

Digital Rights Management Digital Rights Managements (DRM) concerns itself with distributing content to a specified group of users and aims to exert control over what the user can do with that content. Specifically, DRM aims at inhibiting the user from further sharing that content with third parties or otherwise creating unauthorized copies. DRM schemes such as Fairplay[15] or Windows Media DRM[16] generally use a trusted server to send licence keys to authorized users. Alice cannot simply use a DRM service to publish her calendar, as the existing services only work for certain file types (typically audio and video). Sun's DReaM initiative[17], an open approach to DRM, tries to solve this problem by decoupling the protocol from the DRM service provider and by not limiting itself to certain types of files. It does however still rely on a trusted server. As Alice wants to share her calendar with her contacts and not reveal it to the infrastructure, current DRM approaches do not solve the confidential publishing problem.

TeamDrive TeamDrive[18] is a service for sharing folders over the Internet among multiple users. While each authorized user can access the shared folders plaintext, the folders are stored on the server in encrypted form. The

[15]Fairplay is the DRM that was used in Apple's iTunes: http://www.apple.com/itunes/ (last retrieved May 29th 2009)
[16]Windows Media DRM: http://www.microsoft.com/windows/windowsmedia/forpros/drm/default.mspx (last retrieved May 29th 2009)
[17]DReaM: https://dream.dev.java.net/ (last retrieved May 29th 2009)
[18]Team Drive: http://www.teamdrive.net/ (last retrieved May 29th 2009)

folders are published to the target audience with confidentiality, authenticity and integrity. Moreover, the software is easy to use. As any WebDAV-Server can be used, it is considered decentralized. Thus, it can be seen as a solution to the confidential publishing problem.

The internals on how the system works have not been disclosed to public. Instead, TeamDrive carries the "Data Protection Seal of Approval" which was awarded by the Independent Center for Privacy Protection in Schleswig-Holstein, Germany[19].

Summary This section dicussed how existing programs and techniques relate to the confidential publishing problem. Many the examined approaches offer partial solutions to the confidential publishing problem. The following chapter presents the cryptographic techniques needed to efficiently solve the confidential publishing problem. Afterwards, a protocol to solve the confidential publishing problem is proposed, implemented and discussed.

[19]Data Protection Seal of Approval: https://www.datenschutzzentrum.de/guetesiegel/ (last retrieved: May 29th 2009)

3 Cryptographic Techniques

To solve the confidential publishing problem, the user needs to be able to publish files with confidentiality, integrity, and authenticity. This section introduces the cryptographic techniques needed to achieve these desired properties.

3.1 Confidentiality

If data is transmitted or stored in a way that can be read by an unauthorized party, such as an eavesdropper, confidentiality is breached. Data must be encrypted in a way that can only be decrypted by authorized parties. Usually files are encrypted using *symmetric encryption*. In symmetric encryption, the same key is used for both encryption and decryption. This poses the problem that if Alice wants to encrypt data so that only Bob can decrypt it, Alice must first send Bob the key. If an eavesdropper, here called Eve, can listen to Alice sending Bob the key, she can also decrypt the data that was only intended for Bob[20]. Using *public-key cryptography*, two parties can agree on a symmetric key, which only these two parties know about, in the presence of an eavesdropper.

3.1.1 Public-key Cryptography

In public key cryptography, every party has a key pair consisting of one *private key* P and one *public key* K. The private key is to be kept secret, while the public key can be published. In this thesis, only the RSA cryptosystem (Rivest et al. [1978], Jonsson and Kaliski [2003]) is taken into consideration. Other public-key cryptosystems include ElGamal [1985] or Curve25519 (Bernstein [2006]).

If Alice wants to send Bob a confidential message using RSA, she first retrieves Bob's public key K_{Bob} and then encrypts the message using this key. Now only Bob, using the corresponding private key P_{Bob}, can decrypt

[20] If no eavesdropper can listen to the key transmission, then the data transmission would not need to be encrypted.

the message. Note that in practice, public-key cryptography is orders of magnitude slower than symmetric cryptography. Public-key cryptography is only used to encrypt a symmetric key with which the actual data is encrypted. Now only Bob can decrypt the symmetric key used to encrypt the message.

Man-in-the-Middle Attack While public-key cryptography allows a symmetric key to be transmitted confidentially in the presence of an eavesdropper, an attacker who can manipulate the data transfer can conduct a *man-in-the-middle attack:* When Alice tries to retrieve Bob's public key K_{Bob}, Mallory exchanges it with the public key $K_{Bob'}$, to which Mallory knows the private key $P_{Bob'}$. If Alice does not detect this manipulation, she encrypts the confidential message using $K_{Bob'}$, which is only readable by Mallory. Mallory intercepts this message, decrypts it to plaintext, encrypts it using Bob's true public key K_{Bob}, and sends it to Bob. Bob has no way of detecting that a man-in-the-middle-attack occured. As Mallory is able to access the message's plaintext, confidentiality is breached on a successfull man-in-the-middle attack.

Human-Readable Fingerprints As Mallory had to exchange Bob's true public key K_{Bob} against a different public key $K_{Bob'}$, Alice could detect the man-in-the-middle attack if she were able to verify Bob's key over a different, authenticated channel. Such a channel could be, for example, a phone conversation between Alice and Bob, which is authenticated if Alice recognizes Bob's voice. Another authenticated channel could be Bob's business card, which could contain a means to Bob's public key. What these authenticated channels have in common is that they are low-bandwidth and error-prone. When a human is involved in comparing two keys, the data that is compared should be presented in a form that is as short and human-readable as possible.

The SSH protocol (Ylonen [2005]) generates a fingerprint of the public key, using a one-way hash function. It relies on the user to compare the fingerprint of the public key retrieved from the server with a given source. The fingerprint is presented as as 32 hexadecimal digits (Friedl [2002]). Bååth

and Kühn [2002] introduced the concept of *human-readable fingerprints* in a student report. Human-readable fingerprints represent the public key as a number of human-readable sentences. The rationale is that comparing sentences is easier for human beings than comparing numbers. As an example, an SSH fingerprint is usually represented as

c1:b1:30:29:d7:b8:de:6c:97:77:10:d7:46:41:63:87

In contrast, the human-readable fingerprint of the same public key would be

Happy:
Lisa:chased:the cookie:
Savah:jumped over:the flower:
Chip:smiled at:the world:
Nacho:joked about:the pizza

It should be evident that the latter is much easier for humans to compare and to remember than the former. (Both examples taken from Bååth and Kühn [2002]).

Kaminsky [2006] independently presented the concept, calling it *crypto-mnemonics*. Although no scientific evaluation of human-readable fingerprints is currently available, intuition dictates that the they are much easier to remember and to compare than hexadecimal fingerprints.

3.1.2 Symmetric Cryptography

Symmetric cryptography is used to encrypt and decrypt data using a key that all authorized parties know. In this thesis, only the Salsa20 *stream cipher* (Bernstein [2008a]) is considered. Salsa20 was chosen because it has been scientifically validated (Fischer et al. [2006], Tsunoo et al. [2007], Crowley [2006], Bernstein [2008b]), fast[21] and easy to implement in a number of languages, such as C, JavaScript or PHP. Stream ciphers take as input a cryptographic key, which should only be disclosed to the authorized parties, and an initialization vector (IV), which does not need to be kept secret. As

[21] cf. http://cr.yp.to/snuffle/speed.pdf (last retrieved May 29th 2009)

output, Salsa20 generates a *key stream* of up to 2^{70} octets. This key stream is XORed with the plaintext to generate the ciphertext. Two encryptions should use different initialization vectors: If two encryptions use the same key and IV, an eavesdropper can observe the difference between both plaintexts, as illustrated in table 1. This is especially critical if encrypted data has been modified, as an observer can detect how the data changed. Depending on the format of the encrypted file, such information can be valuable. As an exception, an encrypted piece of data can be appended to using the same IV. This should only be done if it is not considered breach of confidentiality for an observer to be able to detect both the fact that data is appended and how much data was appended. Table 2 illustrates this.

key stream	0100 1110...
plaintext a	1000 1111
ciphertext a (plaintext a $\oplus$ key stream)	1100 0001
plaintext b	1100 1110
ciphertext b (plaintext b $\oplus$ key stream)	1000 0000
plaintext a $\oplus$ plaintext b	0100 0001
ciphertext a $\oplus$ ciphertext b	0100 0001

Table 1: XOR of two stream ciphertexts using the same IV yields the XOR of both plaintexts

key stream	0100 1110 0101 1100...
plaintext	1000 1111
ciphertext	1100 0001
plaintext to append	1000 1011
resulting ciphertext	1100 0001 1101 0111

Table 2: Appending to encrypted data

3.1.3 Group Cryptography

While symmetric cryptography is suitable for one-to-one communication, it is inefficient when used to communicate in a dynamic group. Consider Alice

has a group of n peers that she wants to confidentially share a piece of data with. She could encrypt it using a symmetric key, called *group key,* and send each group member that key confidentially. She could then publish the encrypted confidential data, so that each group member can retrieve it. From then on, she could publish more encrypted data using this key, and only the members of the group could access the plaintext. If she wants to add a new member to the group, she simply sends him the group key. If she wants to remove a member from the group, for example because she caught him disclosing confidential data to a third party, she needs to generate a new symmetric key to be used as the group key. Alice needs to send this new key to every remaining group member, encrypted by the members public key. Needing to send $n-1$ messages to remove a member from a group of n peers is considered inefficient. Group cryptography concerns itself with efficient group management.

Both Wallner et al. [1999] and Wong et al. [2000] introduced the notion of key trees: The group key is at the root of the tree. Each leaf in the tree represents a key that exactly one user knows. Intermediate keys are known by a subset of users. This is illustrated in figure 1: The group key is labelled 6, while leafs 0, 1, 2, 3 represent keys only known by exactly one user. Keys 4 and 5 are known by exactly two users. The group key 6 is known by all four users. The key tree can be published in encrypted form: Every node is encrypted by each of it's children. This way, every group member, knowing one leaf node, can iteratively decrypt the parent node until they reach the group key at the root of the tree. In figure 1, key 4 is encrypted by keys 0 and 1, and key 6 is encrypted by keys 4 and 5. The member that knows key 0 can decrypt key 4. Using key 4, the member can decrypt the group key 6.

When a new user joins the group and the tree is not full, a new leaf node is added for the user. If the tree is full, a new group key is generated, using the existing group key as a child. The parent of the leaf node is encrypted with the leaf node key, so that the new member can access it. In figure 2, member 3 joins the group. As the tree is full, a new group key (5) is generated, while the existing group key 2 is appended as a child. Note that the tree is kept at a consistent depth.

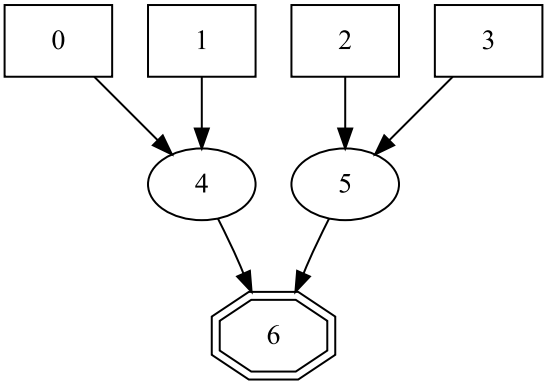
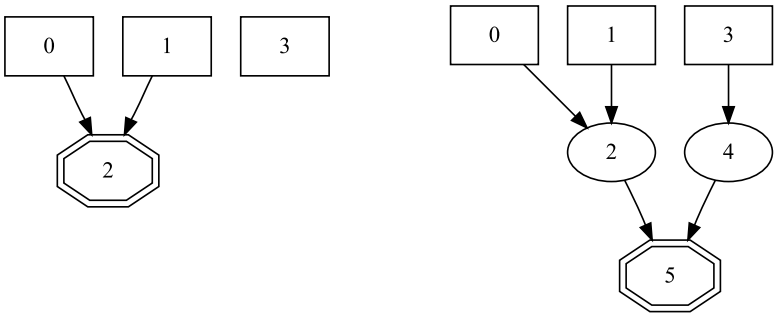


Figure 1: A Key Tree

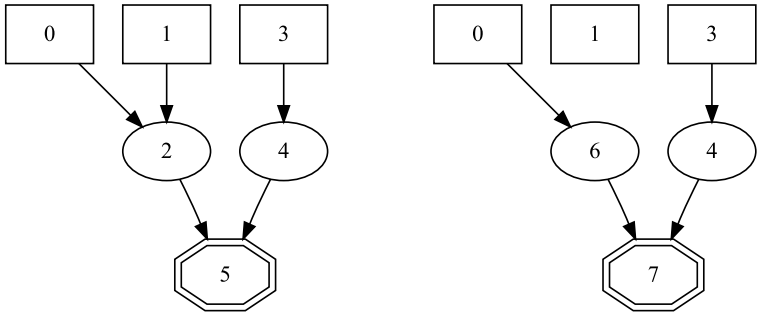


(a) (b)

Figure 2: Adding a new member

When an existing user is removed from the group, all keys that the user knows about must be removed, so that the user cannot decrypt data designated for this group in the future. The keys that the user knows are exactly the keys in the path from the user's leaf node to the root of the tree. The leaf node is simply removed, while all other keys in the path are replaced by new keys. This is illustrated in figure 3: User 1 leaves the group. All keys that user 1 knows need to be replaced: key 2 is replaced by key 6 and the group key 5 is replaced by a new group key 7.

While the owner of a group needs to know about structural information, such as a keys parent and children or if it is a leaf node, the members of the group only need to know key IDs and values. In the example from figure 3,



(a) (b)

Figure 3: Excluding a member

user 3 knows keys 3, 4 and 5. After user 1 is excluded, user 3 would learn about a new key 7. No structural information needs to be transmitted. It is sufficient if group members only retrieve published data. If group members also publish data designated for the group, they should know which key is the current group key so they can encrypt data with it.

Depending on the use case, the new keys are broadcasted to the members or published at a specific location. Independent of the use case, each key is encrypted by both its children. This way, the exact users who should know the key can decrypt it.

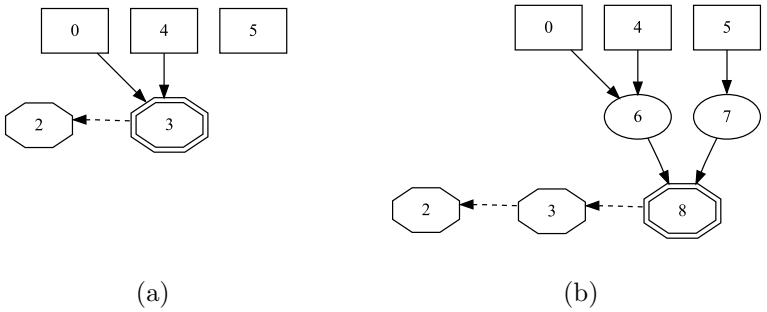
Forward and Backward Secrecy The terms *forward secrecy* and *backward secrecy* are often used to describe cryptosystems. However, these terms are sometimes used with different or unclear meanings (Shirey [2007]). Here, the term forward secrecy is used to describe member m not being able to understand any future data designated for the group after being excluded from it. The term backward secrecy is used to describe member m not being able to understand past data, published before m joined the group. The algorithm, as currently described, is forward secret in the sense that members that have left the group can no longer follow group communication. The system is not backward secret as a newly joined agent might receive an existing group key. If that were the case, the agent could follow at least parts of past

communications. If backward secrecy is desired, then all keys in the path from the new user to the group key must be changed before adding the new user (Xu [2005]).

Backward Key Exposure Group cryptography initially aimed at securing IP multicast networks (Wallner et al. [1999]) in which new group members should not be able to understand data published in the past. When publishing confidential data, users might want the opposite property: A new member should be able to access all data published in the past. This is called *backward key exposure* in this thesis. It can be achieved by publishing each old group key, encrypted with it's succeeding group key. If a new group key is introduced because the tree was full, the old key must be replaced by a new key. Group keys that replace existing group keys have point to the "previous" group key. The previous group key is encrypted by the current group key, so that all group members can access it. This way, all group member can access all content encrypted in the past.

This is illustrated by figure 4. Straight lines point to a node's parent, while dotted lines point to the previous group key. In (a), a group tree with two users, 0 and 4 is shown. Note that before user 4 joined, user 1 (not shown) was excluded from the group. This lead to the generation of a new group key 3. The existing group key 3 points to the previous group key 2 so that user 4 can access content published under group key 2. In (b), user 5 joins. As the current tree is full, this leads to a new group tree (8). The new user is only given the user key 5, which points to key 7. Key 7 in turn points to group key 8. Key 8 points to the previous group key 3, in turn points to the previous group key 2. Using these old group keys, user 5 can access all content published in the past for this group. Users 0 and 4 can decrypt key 6 and use this key to decrypt key 8. Even if they did not know keys 2 and 3, they can still decrypt them using key 8. Thus, all group members know the current group key as well as all previous group keys.

Efficiency The naive approach uses just one constant-sized message to add n users to a group. However, it needs $n - 1$ messages to remove a



(a) (b)

Figure 4: Backward Key Exposure

user from a group. When using group cryptography with forward secrecy and backward key exposure, $O(\log n)$ constant-sized messages are needed to create a group. To add one user to a group, amortized $O(1)$ constant-sized messages are needed. To exclude a user, $O(\log n)$ constant-sized messages are needed (cf. Wong et al. [2000]). Thus, group cryptography adds a slight overhead when adding users to a group compared to the naive approach. However, the ability to efficiently remove members from a group is deemed important, as this allows to exclude misbehaving members (who e.g. breach confidentiality) at low cost.

3.2 Integrity and Authenticity

Integrity and authenticity share a similar goal: Integrity ensures that the content has not been accidentally modified, while authenticity ensures that the content comes from the party it claims to be from (and has not been modified by a third party). Integrity can be established by hashing the plaintext using a *one-way hash function*. One-way hash functions generate a constant-sized digest from input data. While generating the digest from the input data is fast, it is computationally hard to find input data that generates a given digest. Authenticity can be established by *signing* the resulting digest. For RSA, signing a digest is the same operation as decrypting data with a private key. The resulting signature can be verified using the public key. As verification is the same operation as encryption, the plaintext of the digest is restored. This digest is then compared to the digest of the received plaintext.

If both match, the data was received with authenticity and integrity. If not, either the data or the signature were manipulated.

Message Authentication Codes *Message Authentication Codes* (MACs) use a symmetric key to establish authenticity and integrity. They are suitable for proving that a piece of data originated from someone who knew the symmetric key. In this protocol, MACs are used to authenticating data designated only for one user, as they are generated faster than public-key signatures. In this thesis, the HMAC (Mineta et al. [2001]) scheme is used, in which the symmetric key and the data to be authenticated are hashed to produce the MAC.

3.3 Password-based Security

One-way hash functions can be used to turn a password into a symmetric key. This allows, for instance, the private key to be stored in encrypted form in a public location.

If confidential data stored in a public location is protected by a password, attackers can try to guess the password to access the confidential information. Similar to reversing an encryption, the success of a password guessing attack is simply a matter of time. The main factors in a password cracking attack are a) how many passwords need to be tried out and b) how many passwords can be tried out per time interval. The number of passwords that need to be tried mainly depends on the password length and the number of possibilities for one character. As an example, the number of different alphanumerical four-letter passwords is 62^4 or 14776336. If an attacker can try out one password per second, it takes about 85 days to find the correct password. If the attacker can try out 12,500 passwords per second, the correct password is found after about 10 minutes. Thus, the distinction beween *online* and *offline attacks* is important. An online attack consists of an attacker that must ask another party, such as a server, whether a password is correct. This implies that the server might be able to detect the password guessing and slow it down. An offline attack consists of attackers trying passwords on

their own, paying no regard to third parties. Such an attack is much faster, as the rate at which passwords are guessed is only limited by the attackers computing power.

To emphasize the need for a strong password, that is hard to guess, the term *passphrase* is often used in cryptography. Throughout this thesis, the term passphrase will be used for passwords that generate symmetric keys.

4 Abstract Protocol Description

This thesis proposes a protocol to solve the confidential publishing problem. In this section, a high level view of the protocol without much technical detail is presented. The protocol follows three steps: First, the user encrypts the files and stores them on the server. The user then distributes the cryptographic keys to the target audience. Finally, members of the target audience can retrieve and decrypt the files. This way, the files are distributed with confidentiality, integrity, and authenticity.

Here, the term *server* is used for any publicly available service that allows for the storage and retrieval of files. The server could be realized using a classic HTTP server, as well as a peer-to-peer network, a cloud computing provider or even a darknet. It is assumed that there is an authenticated way for a user to store files on the server. Anyone can then retrieve these files. The term *access* is used to describe that a user can access the plaintext of data, i.e. the data is either stored in plaintext or the user has the appropriate key to decrypt the ciphertext. If the user can retrieve the ciphertext but not decrypt it, the data is not accessible by the user.

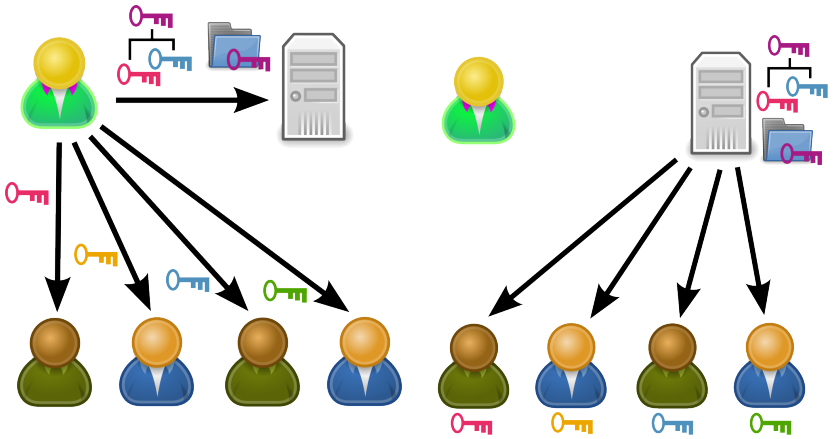
To achieve user-friendliness and make the protocol easy to try out, it will first be implemented as a web application. This way, a modern web browser is all the user needs to be able to use the protocol.

Publishing Confidential Files A user who confidentially publishes files is called *publisher.* To ensure confidentiality, the publisher generates a symmetric group key to encrypt the files and publishes them on the server. To ensure integrity and authenticity, the files are signed. Similar to filesystems, users organize files into folders. Each folder consists of a file containing *metadata* that describes certain folder properties and contents. This metadata is encrypted with the group key. As filenames potentially carry confidential information (e.g. “Bob kissing Alice”), every file has a *public name* and a *private name.* The private name, that is actually shown to users, is confidential. The public name is not confidential. It can be revealed to the public and be used as the actual filename for the encrypted file. Just like in filesystems,

folders can contain subfolders.

The published files have a defined target audience. Each member of the target audience must receive the symmetric key used to encrypt the files in order to decrypt them. The publisher thus creates a key tree using the group key as the root of the tree, adds leaf key nodes for all intended recipients and publishes the encrypted key tree. The publisher sends each member of the target audience a user key, which is a leaf in the key tree, together with the location of the root folder and the folder name of the key tree. This with confidentiality, authenticity and integrity. Figure 5 (a) illustrates this. Each target audience member m can retrieve the path in the key tree from the personal key (a leaf in the key tree) to the group key (the root of the tree). They can then retrieve any desired parts of the published folder. Using the group key, they can decrypt any file designated for their group. This is shown in figure 5 (b).

Using the method described above, the publisher can publish files with confidentiality, authenticity, and integrity. The following section discusses how one can modify data, add and remove users, or verify up-to-dateness. How exactly the publisher establishes authenticity is also discussed.



(a) (b)

Figure 5: Publishing Confidential Files

4.1 Additional Features

In this section, we refine the protocol using a number of enhancements that were omitted from the above description for simplicitie's sake. They all are optional in the sense that the above protocol functions without them.

Modifying Data The publisher may want to modify published data. When using stream ciphers, encrypted data can be appended. If modified in any other way, it is replaced with a completely new ciphertext, using a different initialization vector. To facilitate appending, files are split into two parts: One containing the actual data, which can be appended, and another that contains data that will be overwritten with every modification (such as data for authentication, integrity and up-to-dateness).

Mutliple Groups and Read Permissions In the above description, the publisher creates one cryptographic group of all peers that have read access to the published content. Instead of having one single group, users should be able to organize their peers into different groups, such as their friends, their family or their co-workers. Each such group is a cryptographic group with one key tree. When publishing content, the user selects a target audience which can access the content. This target audience can consist of groups and individual users. The content file is encrypted with a symmetric key, which is accessible exactly by the target audience.

To add a user to a group, the publisher sends the user a message containing a leaf key and the location where the key tree is stored. Using these, the user is able to access content encrypted for that group. If the group key changes, a new key, decryptable using a key the user posseses, will be stored in the designated location.

To remove a user from a group, the publisher needs to change the group key. The new group key is then stored in the group's key tree folder. The new group key is encrypted in a way that every member of the group, except the excluded user, can decrypt it. Future new content and content modifications will use this new group key so that the excluded user can no longer understand it. However, existing content can still be accessed by the

excluded member. The publisher would need to re-encrypt it with the new group key to avoid it being read by the excluded user. As re-encrypting all published data is a potentially costly operation, the default behavior would be to leave these files accessible to the former target audience member.

It should be noted that read permissions for an individual file are orthogonal to the concept of groups. As an example, Alice maintains three cryptographic groups: her friends, family, and co-workers. When she publishes her birthday party as a calendar appointment, she wants her family, her friends, and one co-worker to be able to read it. Thus, she makes the symmetric key that encrypts the actual content accessible to her family, her friends and the co-worker. If she later adds another user to the group of friends, that friend will be able to access the appointment, as well as all other content accessible by the group of friends. Instead, Alice could also give the user read permissions for the appointment only instead of adding the user to the group of friends. This would allow the user to access the appointment, independent of group memberships.

Up-to-dateness Up-to-dateness is the property that a user always retrieves the latest revision of a piece of content. It can be achieved by appending a counter that increases upon each modification. If clients have a way to remember this counter for each file, they can detect if a file has been replaced by an older file. Note that this technique can detect manipulation only if the newer file has been accessed by the client before it was exchanged with an older file.

Generating Public Names from Private Names In order to retrieve a file, it's public name must first be known. In order to generate the public name from the private name without needing to retrieve a list of files in a folder, the following mechanism is used. The folder's file list is split from other folder properties and is only retrieved when needed. Every content folder defines a random salt. Every file's public name is the hash of its private name concatenated with that salt.

Key Management The publisher has one or more public keys to authenticate the published content. Furthermore, the public keys are used by other users to encrypt data - e.g. when sending leaf keys for their key tree. All of a user's public keys are stored in a publicly accessible folder. Public keys have a defined period of validity and are self-authenticated. This implies that after a certain period of time, a new key must be generated to replace the old key. *Key messages*, authenticated by the key that is replaced, designate which key it is replaced by. Key messages can also declare keys invalid if, for instance, they were compromised.

Private Data A user generates a symmetric key from his or her passphrase. Using this symmetric key, users can access content only designated for them, such as their private key. Users can have multiple passphrases: One *main passphrase* than can access all data, and zero or more *short-term passphrases* that can only access a subset of data. When encrypting folders and files, the user can choose which passphrase can access them, similar to the way the user selects a target audience. The user should typically use the main passphrase in a trusted environment such as their own computer. A short-term passphrase can be used in untrusted environments like Internet cafes.

Every passphrase defines a folder called *passphrase secrets*. This folder stores one private key which corresponds to a publicly accessible public key. The main passphrase stores the *long-term public key*. The long-term key is valid for a long period of time, such as 5 years. Short-term passphrases have *short-term public keys* associated with them, which are valid for a shorter, user-defined time of, for example, a few weeks. Each passphrase secrets folder also stores other data only accessible to that user's passphrase, such as received symmetric keys.

Messaging Users can send messages to other users. Every user has a globally unique ID. To achieve confidentiality, the message is encrypted using the receivers public key. To achieve integrity and authenticity, the message is signed using the senders private key. To achieve up-to-dateness, the message contains a timestamp. In order to hide the sender's identity from the

receiving server, the signature can be part of the ciphertext. This message will then be stored in the user's message inbox.

Write Permissions In this protocol, every file is public in the sense that anyone can read its ciphertext. Read permissions are realized by giving the user access to the key to decrypt the file. In contrast, write permissions need to be enforced by the server. By default, users can only create or modify files using their newest long-term key. In a more refined approach, the user defines write permissions for each filesystem folder[22]. This makes *wall* functionalities possible, where other users can append messages to a person's wall, which is shown to all users in the same group. Such wall functionalities are popular in social networks. The user must inform the server to allow messages to be appended for a certain group key. This functionality is also used to realize the user's message inbox.

4.2 Example Use Case

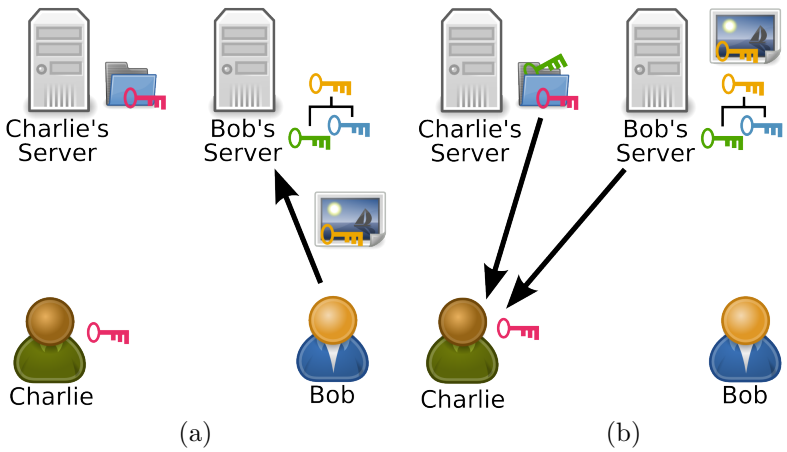


Figure 6: Example Use Case

Bob made pictures of the party he hosted last night and wants to share them with his group of friends, which consists of Alice, Charlie and Dorothy.

[22]The server is not aware whether some file is folder metadata or actual content, so file permissions can not be realized on a protocol folder level.

As shown in figure 6 (a), he uploads the pictures into a folder, with both the folder metadata, the content metadata and the pictures encryted in a way only accessible to this group of friends.

Charlie wants to retrieve the folder where Bob usually stores his pictures. First, he generates a symmetric key from his passphrase. Using this key, he can access keys that Bob sent him in the past, as they are stored in his passphrase secrets folder. After retrieving the folder metadata, his client notices that metadata exists that is encrypted with an unknown key. The client checks the location where the key tree for Bob's group of friends is stored and notices that the unknown key is accessible there, using a key that Charlie knows. This means that either a new user joined that group when the key tree was full, or an existing member was excluded. Charlie retrieves this key and can now decrypt metadata, which points him to a subfolder called "last night's party pics". Charlie can now look at the images that Bob published. Alice and Dorothy can access the pictures in a similar way.

5 Realization

The protocol described so far allows to solve the confidential publishing problem. A user can publish content with confidentiality, integrity, authenticity and in a decentralized way. However, numereous technical details were ommited. These technical details are presented in this section, together with eloborations on the design decisions made.

First the concept of user IDs is explained, then the general data format that is used in the implementation is described, followed by format descriptions for keys, folders, and content. Afterwards, the protocol to store files on the server is specified. Finally, it is described how users can send messages to other users.

5.1 IDs

User IDs Every user has a unique ID. Similar to an email address, it consists of a server-local user name, the ASCII character '@' and the server's domain name. The user name can consist of any unicode character contained in UTF-8. The idea is that the protocol allows users from any culture to use their native script, with the least restrictions possible. This however poses a problem: If the user name contains characters not found on the average keyboard, e.g. 猫, users probably do not know how to enter it. It cannot be considered user-friendly to let the user search through unicode tables for the right character. Thus, if non-ASCII characters are contained in the user ID, the user should supply a URI-encoded[23] form of the ID. E.g. the ID '猫`@crypto.uni-freiburg.de`' should also be given as '`%E7%8C%AB@crypto.uni-freiburg.de`' or '`%u732B@crypto.uni-freiburg.de`'. Similarly, the user ID '`Heinrich Müller@crypto.uni-freiburg.de`' should also be given as '`Heinrich M%C3%BCller@crypto.uni-freiburg.de`'.

[23] cf. Berners-Lee et al. [2005]

Key IDs As users have different keys with which they can encrypt data, they need to be able to specify which key they used to encrypt their content. Every key has a unique ID. The *key ID* consists of the *key name*, the ':' character and the user id that the key belongs to, e.g. '`longterm-0:猫@crypto.uni-freiburg.de`'. Note that key names starting with '`longterm`' and '`shortterm`' have special meanings.

Restrictions Based on the above description, some restrictions for key and user names follow. The characters '%', ':' and '@' are not allowed. Additionally, the character '/' is not allowed, as it is reserved for use in the protocol. To avoid confusion, no quotation marks, such as " or ' are allowed. Non-printable characters are not allowed. The only whitespace character allowed is a single space, and cannot be followed by another whitespace. If a key or user ID contains non-ASCII characters, it should also be given in URI-encoded form. IDs are not case sensitive, upper-case characters are converted to lower-case characters internally.

Base URL A user ID implies a URL at which the user has a directory accessible via HTTP(S). The user's *base URL* is constructed by appending the user name to the server from the user ID. An example of this would be '`猫@crypto.uni-freiburg.de`' with the base URLs '`https://crypto.uni-freiburg.de/%E7%8C%AB/`' and '`http://crypto.uni-freiburg.de/%E7%8C%AB/`', respectively.

Note that this protocol tries to make as few assumptions as possible about the capabilities of the server that it runs on. If the server does not support the user URL given here, workarounds such as the user IDs '`~alice@crypto.uni-freiburg.de`' or '`users/alice@crypto.uni-freiburg.de`' are possible. Similarly, HTTPS connections are preferred because they additionally protect the user's privacy (eavesdroppers cannot see what file the user accesses). However, as not all web servers support HTTPS, so HTTP is tried if HTTPS is unavailable.

5.2 Netstring Object Notation

The data format used for the metadata needs to efficiently contain binary data so that encrypted metadata can be embedded. This is supported by many formats, such as ASN.1 (X.680 [2008]), Protocol Buffers[24], Thrift[25], Etch[26], ICE[27] or SDXF (Wildgrube [2001]). However, none of these formats are human-readable: Even a computer-savvy user with a hex editor cannot easily interpret the data format. On the other hand, human-readable formats like XML (Bray et al. [2008]) or JavaScript Object Notation[28] do not allow binary data to be embedded. Typically, binary data is encoded as base64, which increases its size by about 40%.

This thesis defines the *Netstring Object Notation* (NSON). NSON is similar to the JavaScript Object Notation in syntax, but it is binary safe. Binary data never needs to be escaped. Similar to *netstrings* described by Bernstein[29], the size in octets (8 bit bytes) is prepended to any type. This size is represented as a human-readable ASCII string, so that it can be easily parsed by humans and in any programming language with no confusion about endianness. This makes parsing objects particulary easy. NSON distinguishes between numbers, strings, arrays and objects. Any type starts with the payload length in octets, followed by one character that specifies the type: `'%'` for numbers, `'"'` for strings, `'['` for arrays and `'{'` for objects. After the specified number of octets, a terminating character is expected: `'%'` for numbers, `'"'` for strings, `']'` for arrays and `'}'` for objects. Between types, exactly one comma (`','`) must appear and any number of whitespaces, as defined by the Unicode Character Database[30], is allowed. An array is simply a list of types. An object is a list of keys or key-value pairs. A key is a string. A key-value pair is a key, followed by `':'`, followed by any type. An EBNF

[24]Protocol Buffers: http://code.google.com/p/protobuf/ (last retrieved May 29th)
[25]Thrift: http://developers.facebook.com/thrift/ (last retrieved May 29th 2009)
[26]Etch: http://cwiki.apache.org/ETCH/ (last retrieved May 29th 2009)
[27]ICE: http://zeroc.com/ice.html (last retrieved May 29th 2009)
[28]JavaScript Object Notation: http://json.org/ (last retrieved May 29th 2009)
[29]Netstrings: http://cr.yp.to/proto/netstrings.txt (last retrieved May 29th 2009)
[30]Unicode Character Database: http://unicode.org/Public/UNIDATA/PropList.txt (last retrieved May 29th 2009)

grammar describing NSON can be found on page 71.

In order to allow appending of NSON types without needing to change anything else in a file, NSON types can just be concatenated to form an implicit array.

Examples Example of an NSON string:

```
12"Hello World!"
```

Example of an NSON number:

```
6%-27.67%
```

Example of an NSON array:

```
43[21"A string and a number", 7%5.632e4%]
```

Example of an NSON object:

```
96{
 4"type": 7"key:rsa",
 7"private",
 5"value": 45"Here would be the value of an RSA private key"
}
```

Example of an NSON implicit array of a string, a number and another string:

```
5"Seven", 1%8%, 4"Nine"
```

Example of appending the number 10 to the above implicit array:

```
5"Seven", 1%8%, 4"Nine", 2%10%
```

Note that in explicit array notation, a header specifying the array length precedes the array and the character `']'` succeeds the array. This makes it impossible to append an NSON type to the array without changing the array header. However, appending is the only file modifying change that can be made to data that is encrypted using a stream cipher without changing the initialization vector. This is the reason for the implicit array notation.

NSON Object Types The plaintext of each metadata file, such as a folder, can be seen as the concatenation of NSON objects. This way, new objects can easily be appended to the end of the file. Stream ciphers are used so that the ciphertext can be appended as well. NSON Objects contain one *type* and can contain a *subtype*. The type is defined in the "type" property. A subtype is defined by appending ":" and the subtype name to the type property. For instance, an object with the type property "key:rsa" has the type "key" and the subtype "rsa". Objects usually also have either one "value" or "url" property. In the former case, the value that the object describes is directly read from the value property, while in the latter case, the value the of URL given in the property contains the value. In some cases, an object contains neither a "value" nor a "url" property. Then, the URL is constructed from another property, usually from the "name". Note that loading data from another URL means that that file either needs to be included in the signature of the metadata or needs its own signature to ensure integrity and authenticity.

5.3 Content, Folders, and Files

Just like in traditional filesystems, the protocol allows users to organize their files into folders. Note that when the user creates one *content file* (e.g. a text file), multiple files might be created in the actual filesystem (e.g. one for the encrypted text file and one for the signature). Similarly, one *content folder*, created by the user, does not necessarily correspond to an actual *filesystem folder*. The terms *content folder* and *content file* are to be distinguished from *filesystem folder* and *filesystem file*.

Folders Folders are split into two filesystem files: One containing the folder properties, and another containing the list of content files. The folder properties set important options for the folder, such as the folder salt. The file list contains a list of all content files contained in the folder. The reason for splitting the folder into two files is that this way, the user can retrieve the folder salt to randomly access files expected in the folder without having to

retrieve a potentially long list of files.

The file list consists of a concatenation of NSON objects of the type "content" or "folder". Folder entries stand for subfolders, while content stands for actual user content, e.g. images, or text files. An additional "content-type" property can be provided to describe the MIME type of the data. Both folder and content have a mandatory property "name" describing the private name of the object. This name is used to generate the public file name(s). If a content file is encrypted, its metadata is encrypted with the same key. The encrypted metadata is then stored in the "value" field of an NSON object of the type "meta". The "trans" property specifies how to decrypt the data.

Read Permissions When creating a content file, the user decides which users or groups should be able to access the file's content. The content is encrypted with a symmetric key. This key is stored in a files key list, encrypted in a way that exactly the intended users and groups have access to. The key list is a concatenation of "meta" NSON objects, which decrypt to NSON objects of type "key". If the content should be publicly available, the content is not encrypted and no key is added to the key list. If a user or group should have access to the file in the future, a key accessible by the user or group is appended to the key list.

Signatures and Revisions Signatures are used to establish authenticity and integrity. They are calculated using the newest long-term private key or any valid short-term private key of the user, and can be verified using the corresponding public key. An exception is the signature of passphrases, which is technically a Message Authentication Code (see "Passphrases" on page 46). The signature is always based on the plaintext of the file it signs. In the case of folders which can contain encrypted meta objects, the signature is based on the encrypted meta objects, as clients who cannot decrypt all meta objects should still be able to verify the signature.

Revisions ensure up-to-dateness. A revision is simply a NSON number. If the client has a way to remember the revision number and the signature, it can compare the remembered number and signature with the retrieved

number and signature. If the signature is different and the retrieved revision number is not greater than the remembered revision number, the content was replaced with an old version. Such content is rejected.

The NSON objects that defines a salt can specify which keys are acceptable for signing the content. This is done in the "signedby" property, which contains either a single NSON string or an array of NSON strings. Note that any valid, non-compromised long-term key is always allowed to sign any content. The "signedby" property is mainly useful to allow additional short-term keys to sign content.

Private and Public File Names User content and folders have private names, which should only be visible to authorized users. The public name, which is the name in the filesystem, is constructed by the hex representation of the hash of the concatenation of the folder's salt, the user name, the ":" character, the namespace, the ":" character and the private names of all objects in the path. The namespace is used to distinguish different types of files, e.g. actual user content, signatures, folder properties and folder file lists, without limiting the possibilities for content names. Table 3 lists the namespaces used in this thesis.

Note that a folder can define multiple salts. If multiple salts exist, one public filename for each salt is tried. As this can easily become ineffective, implementations are encouraged to limit the number of salts present in a folder. Note that two files with the same private name but different public names (using different salts) can exist in one folder. While this is unavoidable under some circumstances, implementations should try to avoid it, as the resulting behavior is undefined.

As the folder properties define the salt for this folder, the same salt cannot be used to generate the folder properties public name, from which the salt must first be retrieved. This is why the folder's parents salt is used to generate the folder properties public name. As the root folder has no parent, no salt can be used for it. The root folder is defined as being unencrypted and signed by the newest long-term key. Folder names end in a "/" character. By default, SHA-256 is used as the hash function.

content type	namespace
user content	content
folder properties	folder-properties
folder file list	folder-list

content type	namespace suffix
revision and signature*	-revision
file content*	-separate
key list**	-keys

Table 3: Namespaces for various file types
*: If not contained in the file containing the NSON content object
**: If the content is encrypted

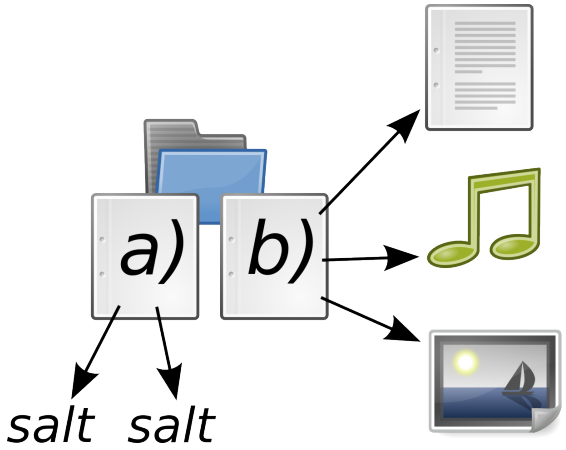


Figure 7: A folder consists of folder properties (a), defining salts, and a file list (b).

Retrieving Content, Revision, Signature and Key List After retrieving the file under the public name, exactly one NSON object is parsed from the file. This object is either of the type "content" or of the type "meta", which decrypts into an object of the type "content". If it is of type "meta" and no key name is given in the "trans" property, the same key that encrypted the salt is used. The "content" object defines the key to decrypt the actual content as well as the revision number and signature. If the "signature" or "revision" properties are missing from the object, they are loaded from a separate file whose public filename is generated by appending "-revision" to the

namespace. If the revision number and signature are stored in a seperate file, the actual content file can be appended. The “content” property is either set to “follows” or “separate”. If set to “follows”, the actual content, encrypted with the specified key (or in platext if no key was specified), follows the NSON object. If set to “separate”, the actual content is retrieved from a seperate file whose public name is generated by appending “-separate” to the namespace. Usually, the “content” property is set to “follows”. It is set to “separate” when publishing plaintext data, such as image files. This way, the plaintext data does not need to be modified for use with this protocol. It can also be used to refer to a file on a different server. If the file is publicly accessible, i.e. stored in plaintext, the content object contains a “plaintext” field. Otherwise, the key list that stores the keys necessary to access the content is retrieved from a seperate file using the “-keys” namespace suffix. Table 4 shows examples of content files and their public names.

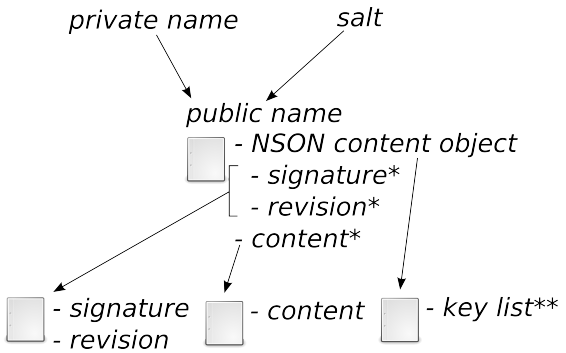


Figure 8: Filesystem files used to store one content file.
* indicates that the file is only retrieved if the property is not present in the first file.
** indicates that the file is only retrieved if the content is encrypted

5.4 Keys and Key Messages

So far, it was discussed how folders specify a salt and a file list and how content files are retrieved. Content files can be encrypted and authenticated by cryptographic keys. How cryptographic keys are organized exactly is discussed in this part.

Each user has one main passphrase and multiple short-term passphrases.

private name	“/” (root folder)
file type	folder properties
user ID	alice@crypto.uni-freiburg.de
public name	sha256("alice@crypto.uni-freiburg.de:folder-properties:/") = 022409585a2b776920ae68b672e09cc24ac6504dd6af6a52812a25cd1b6c4d03

private name	“/university/networks/lecture notes”
file type	content
user ID	alice@crypto.uni-freiburg.de
salt	ssss
public name	sha256("ssssalice@crypto.uni-freiburg.de:content:/uni/networks/lecture notes") = 6de6a1432776e1d31e4f3e5407b37b10b127b4c5c1de37fa1c91c38d89d53652

private name	“/uni/networkings/lecture notes”
file type	content (revision and signature)
user ID	alice@crypto.uni-freiburg.de
salt	ssss
public name	sha256("ssssalice@crypto.uni-freiburg.de:content-revision:/uni/networks/lecture notes") = b2a4c704c219ee334da8f6b7acbad4a0d6a15ce7b02d91cc91a8396943b65c38

Table 4: Content files and their public names

Each passphrase is associated with one public/private key pair. Keys from public key algorithms have a special naming pattern. Long-term key names start with the string “longterm-”, followed by a non-negative integer encoded as ASCII. Short-term keys start with the string “shortterm-” and can continue in any user-defined way. Both are stored in the folder with the private name “/public-keys/” and the public name is generated with no salt. This folder is publicly readable. It also contains the file with the private name “active” (and public name generated with no salt) that lists the newest long-term key and all short-term keys that are valid or will be valid in the future. This allows other users to quickly select appropriate public keys to use when sending a message to the user, for example.

Passphrases The protocol is designed so that users only needs their passphrase, and nothing else, to access their account. Using the main passphrase, the user can access all data. Short-term passphrases can only access the

subset of data that the user authorized. A symmetric key is generated from the passphrase. This key is used to access a special folder, only accessible by the user. This folder is called the passphrase secrets folder. It's private name is "passphrase-secrets-" followed by the passphrase name. It's public key is generated using the symmetric key generated from the passphrase as salt. The passphrase-secret folder's integrity and authenticity is established by a Message Authentication Code (MAC) using the symmetric key generated by the passphrase definition.

This folder contains a subfolder "private-keys/", which contains the private keys accessible by the passphrase, i.e. the long-term keys for the main passphrase or one short-term key for a short-term passphrase. The subfolder "passkeys/" contains a number of *passkeys*. Passkeys are only accessible by the passphrase with which they are associated (plus the main passphrase). They are used to facilitate passphrase changes: If the passphrase changes, this changes the user's symmetric key. Thus, changing a passphrase would result in needing to re-encrypt every file encrypted with the passphrase. To avoid this, files are not directly encrypted with the passphrase, but with a passphrase's passkey (except the passphrase-secrets folder and the passkeys folder). Whenever the passphrase changes, a new passkey is added. Future encryptions are done with this passkey. In the case of the main passphrase, this folder also stored the symmetric keys generated from all short-term passphrases, so that the main passphrase can access all data, even from the passphrase secrets folders of other passphrases.

Keys sent to or received from other users are stored in the subfolders "to/" and "from/", respectively. Every time a user receives a key from another user, it is saved in the "from/" subfolder. Every time the user sends another user a key that only this user knows about, it is saved in the "to/" subfolder. As an example, if Alice uses her main passphrase to log in and accesses Bobs public key "longterm-0:bob@crypto.uni-freiburg.de" for the first time, she stores it under the private name "passphrase-secrets-main/from/longterm-0:bob@crypto.uni-freiburg.de". In the future, she retrieves this key from her secrets folder, where it can be accessed in an authenticated way.

If a user is a member of another users group, the user needs to know

about the location where future group keys are stored. The object with this information is stored in the subfolder "group-memberships/". One folder with the private name of each group owner exists, which contains the group membership information.

Preventing Man-in-the-Middle-Attacks The protocol uses two mechanisms to dwarf man-in-the-middle attacks. First of all, users are urged to verify another user's public key over an authenticated channel, such as a phone call or in person, before using it. Human-readable fingerprints are employed to make this fingerprint comparision as user-friendly as possible. Second, as described in the previous paragraph, once a public key is retrieved from a user, it is saved into the passphrase secrets folder. This way, if the first retrieval of the public key was authentic, future retrievals will be authentic, too. Public keys from other users, stored in the secrets folder, contain a "verified" property to indicate that the user verified this key over an authenticated channel and that the key definitely comes from the user it claims to be from. If the user compares the key and the human-readable fingerprint is different, the "verified" property is set to "failed" and all data signed with the key automatically fails verification.

Generating a Symmetric Key from a Passphrase The passphrase is turned into a symmetric key by concatenating the user name, the ":" character, the passphrase name, the ":" character and the actual passphrase, and repeatedly hashing this string. RSA Laboratories recommends hashing a passphrase at least 1000 times (Kaliski [2000]). One file with the private name "passphrases" (and the public name generated with no salt) contains a concatenation of NSON objects. Each such object has the type "passphrase" and defines the passphrase parameters in its properties. These parameters include the encryption algorithm used to encrypt the passphrase-secrets folder, the hash type used to generate the symmetric key from the passphrase, the number of times the hashing is repeated, and the hash function used to generate the public name of the passphrase folder.

To ensure the authenticity and integrity of the "passphrases" file, it is

signed by the newest long-term key. As the authenticity of this long-term key cannot be established before the user generates the appropriate passphrase, the human-readable fingerprint of this long-term key is shown to the user. This lets the user verify that the long-term key is authentic and reminds the user of the long-term keys human-readable fingerprint on every login. Implementations should reject symmetric keys with fewer than 128 bits, unsecure algorithms, or less than 200 iterations[31].

The generated symmetric key equals

$$\text{hash}^{\text{iterations}}(\text{user id} + ":" + \text{name} + ":" + \text{passphrase})$$

where + is the concatenation operator, "name", "hash" and "iterations" are the properties defined in the passphrase NSON object.

Replacing Keys Instead of changing a key, it is replaced with a new key that has a different name. In the case of symmetric keys, the new key is simply stored in an appropriate folder, encrypted in a way that all authorized users can access. New short-term public keys are placed in the "/public-keys/" folder and signed by the newest long-term key that is valid when the new short-term key becomes valid.

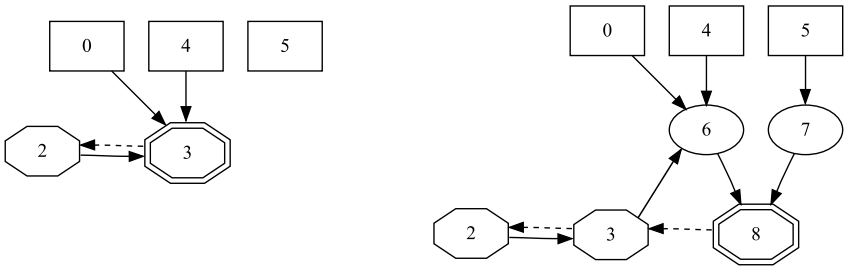
New long-term keys require a key transition message. This message is simply appended to the file that contains the new key and is signed with the preceeding long-term key. The first long-term key, named "longterm-0", does not have a corresponding key transition message. Keys can be changed for three reasons: First of all, the existing keys validity period could soon come to an end, so the user introduces a key that is valid for a longer time. Another reason would be the existing key being compromised so that a new, uncompromised key must be introduced. The third reason is that the user lost access to the existing key by forgotting the main passphrase, for example. In the first case, the new key is marked as verified by users who marked the previous key as verified. In the latter two cases, the new key is not marked as verified. In the first two cases, the key transition message is signed by

[31] On slow computers or portable devices, 1000 iterations take "too long". E.g. on the Apple iPhone, the web browser terminates the JavaScript after about 200 iterations.

the preceeding key. In the last case, the message cannot be signed as the user no longer has access to the preceeding key. This makes this messages extremely dangerous, as any third party without prior knowledge could use it to convince a user to accept a new key. In the two last cases, the user is urged to verify the key over an authenticated channel before trusting these keys. The reason for the key change is embedded in the key transition message. Note that short-term keys can be invalidated by key transition messages, but in this case, no new key is introduced. Note that in the case of long-term key messages, the validity periods are ignored: for each long-term key, there must simply be a key-message signed by the preceeding long-term key. Otherwise users could not introduce new keys after their last long-term key expires.

Group Keys For every group, a special folder exists that contains all keys for that group. It's private name is "group-keys-" prepended to the group name. The public and private names are known to all members of the group. Whenever keys in the group change, they appear in this folder, in a way that is only readable by intended users, i.e. encrypted by their two children. In this folder, only the key names and key values are stored, as this is the only information the user needs. Note that every such key contains a "next" property that points to the "next" key in the tree. This allows users to efficiently retrieve all keys that they should know. The current group key also contains a "previous" property, which points to an old group key. From there, more "previous" properties point to more old group key. If a user encounters an unknown key, the user should re-read the newest non-branched key without a "next" property in each of the key owner's groups of which the user is a member. If a new key was introduced in that group, it would be pointed to by the last key without a "next" property. This is illustrated in 9. In (a), the current group key 3 has a "previous" property pointing to the previous group key 2. After user 5 joins in (b), the new group key 8's "previous" property points to the previous group key 3. When existing users 0 or 4 re-read key 3, a new "next" property is added, pointing to key 6, whose "next" property in turn points to key 8.

The group owner also maintains a folder that contains the key structure.



(a) (b)

Figure 9: Next and previous properties in a key tree

This folder has the private name "group-structure-", followed by the group name. The public name of the folder and all information therein are only accessible to the user. In this folder, structural information about group keys is stored, specifically its index in the key tree, the keys parent key, its child keys, which key it replaces and by which key it is replaced, whether it is a leaf in the tree (i.e. only known by a single user) and whether the subtree that this key represents is full (i.e. all children have the maximum amount of children or are leaf nodes). In the group-structure folder, one file with the private name "group key" contains the name of the current group key.

Symmetric Key Names While public key names have cleartext names, chosen so that users can clearly identify their purpose, symmetric key names hide their meaning to protect the users privacy: Authorized users may know which group a key belongs to. Unauthorized users should not be able to infer which group of users is authorized by reading the key name. Whenever a group is defined, a key salt is defined together with it. Group key names are generated by concatenating the groups key salt, "group-", the group name, the "-" character and the group key index, defined in the group key structure file and hashing it. This ensures that while keys belonging to groups are just consecutively numbered, non-group members can neither infer which groups are authorized, nor how many members belong to a group. Note that group members can roughly estimate an upper limit to the number of users in a group by counting the number of keys they know in that group. Each

passphrase defines a key salt called “passkeys”. To create a passkey name, this salt and the index of the passkeyare concatenated and then hashed. The resulting string, encoded as a hexadecimal number, is the public name for this passkey.

5.5 Storing Data on the Server

Read access control was implemented in a flexible way by letting any user retrieve any file, but only giving the authorized users the keys needed to decrypt the file. This implies that the server is completely unaware of any access control mechanisms and simply distributes files. For write access control, the server must act as a gatekeeper. While established techniques, such as (S)FTP or SSH can be used to upload files, this protocol defines a message format that is used to upload files. This is needed to establish a coherent, user friendly experience, in which the users truly only needs their passphrase to use the protocol, and no additional credentials for an FTP account, for instance. Furthermore, certain features such as multiple passphrases or allowing group members to add files, cannot easily be mapped to, e.g. FTP accounts. Essentially, storing data on the server works in two steps: First, the account owner specifies which keys are authorized to write in which filesystem folder. Then, using one of these authorized keys, a user can write to any file in an folder authorized for that key.

Server Message Format Message are sent to the server with confidentiality, authenticity, integrity and up-to-dateness. A message is a concatenation of NSON objects. The first object has the type “server-message:header”. It contains the messages timestamp, the ID of the key that signs the message and the signature of the message. Optionally, it can specify the encryption, signature, and hash algorithms used. If the message contains file operations, the header object can specify a file that must be locked in order to perform the operations specified in the message. This assures that if two users try to modify one file simultaneously, one user succeeds and the other one fails and should try again later. The header also contains the hash of the file

to be locked. This assures that the filesystem was in the state expected by the client, similar to the compare-and-swap operation in multithreaded applications (cf. Sundell and Tsigas [2004]).

Authenticity is achieved by embedding a signature in the message header that is based on the concatenation of the timestamp and the message following the header. To achieve up-to-dateness, messages with a timestamp in the future or with a timestamp older than 30 seconds are discarded. Note that according to the HTTP/1.1 standard (Fielding et al. [1999]), servers send their current time with every response. This way, clients know about the server clock and adjust the timestamp accordingly. Also note that generating the timestamp and the signature are the last two actions of the client before sending the server message.

Apart from the header, which must always be present, the following "server-message" object subtypes can be concatenated: "key" defines keys that the server recognizes. These can be used later to sign server-messages. "write-permission" and "append-message-permission" specify one or more keys that can write in the directory. Giving a key "write-permission" means that the user can use that key to modify files in any way in that filesystem folder. The user can add new files and modify, rename, and delete existing files. These operations are realized using the subtypes "new-file", "modify-file", "rename-file" and "delete-file". Note that the server always knows the newest longterm key and always allows it to modify all files in the user's directory.

Giving a key "append-message-permission" means that using this key, a user can add messages to that folder. This can be used to realize the users inbox, for example. Using a key with "append-message-permission", a user can upload an "append-message". The server then gives this message an index and stores is in a single filesystem file. An NSON object pointing to the file is then added to the folder's file list. Additionally, a file with the private name "latest-index" points to the latest index number that was given by the server. The user can also specify "append-message-salt" and "append-message-key", which lets the server both use this salt to generate the public names and this key to encrypt and sign the actual files and metadata. Specifying the key

"*" in an "append-message-permission" lets anyone append messages in that filesystem folder. This allows a "world-appendable" folder that can only be read by the user to be created.

5.6 User Messaging

Each user has a folder with the private name "inbox" and the public name generated without salt. The user sets a key and salt for this folder and sets permissions so that anyone can append messages. This way, users can send messages to peers. To check for new messages, the user retrieves the file "latest-index". If the index in this file is newer than the one known by the client, the client retrieves all files between the known index and the latest index.

User Messaging and Passphrases Usually, a user chooses which content can be accessed by which passphrase. When it comes to messaging, the sending party must decide for which of the receiver's passphrases it makes the message accessible (i.e. for which short-term key in addition to the newest long-term key). By default, a message is encrypted so that the message is readable by all passphrases valid at the time of sending.

6 Implementation: A Social Network

This section sketches a social network as an example application that uses the protocol described in the previous chapter. First of all, common features of social networks are defined. The section thereafter discusses how these properties are implemented. It should be noted that web applications can be seen as the easiest way for a user to try out an application. Finally, the current state of the implementation is described.

6.1 Features of Social Networks

Depending on how "social networks" are defined, the actual features vary. Here, we focus on the "minimal feature set" of social networks, such as Facebook, MySpace, and studiVZ. The most important features are probably the user's profile and the user's friends. The profile describes personal information about a user. It consists of properties, such as their name or their email address (see table 5 for an incomplete list of typical properties). A user's friends are members of the social network that the user knows, for example, in real life and considers friends. To become another user's friend, a user sends the other user a "friend request". The other user can either accept or decline the friend request. If accepted, both users are each other's friend (it is a symmetric relationship). Users have a list of friends visible on their profile page.

Other important features are the user's current status, messaging, groups and the user's wall. Social networks have the advantage that users can send each other messages virtually spam-free. This is possible because the service provider have a global view of the social network and can easily exclude any misbehaving, i.e. spamming, user. On the flip side, this also allows the social network to exert censorship as it sees fit, subject to the service provider's interpretation of "misbehavior"[32]. Groups in the context of social

[32] As a recent example, Facebook censored pictures of women breastfeeding their child. New York Times: http://query.nytimes.com/gst/fullpage.html?res=9401E5D61531F936A35752C0A96F9C8B63 (newspaper article from January 5th, 2009) (last retreive May 31st, 2009)

networks are mostly discussion forums that are usually only readable by members of the forum. Non-members can apply to join and the user who "owns" the group can accept new members and exclude existing members. Each membership is usually displayed on a user's profile page, which often leads to group names that are statements[33]. Finally, the wall allows any friend of a user to leave a short comment on it, which is usually displayed on a user's profile page.

Finally, users can upload pictures to the server and *tag* them, i.e. add information about which users are visible in the picture (including showing the area *where* the user is in picture). This functionality allows to search for pictures in which a certain user appears. This reveals potentially sensitive information about a user: As an example, a friend of a user could upload a picture in which the user is clearly drunk. On its own, this picture may be overlooked by many people. However, if it is tagged, it appears in the list of pictures in which the user appears, i.e. can be easily associated with the user. This is why social networks usually inform a user that a picture of was tagged with the user and allow a user to remove the tag from the picture.

name	email address	education and workplaces
sex	phone number	"looking for"
birthday	current town	political/religious views
relationship status	instant messaging addresses	activities
interests	favorite music	favorite tv shows
favorite movies	favorite quotes	...

Table 5: Typical profile properties in social networks, extracted from Facebook and studiVZ

[33]Such group names lead to a political party's youth organisation's vice president stepping down, as he was member in a studiVZ group called "Nach Frankreich fahr ich nur auf Ketten", (rough translation: "I only drive to France on chains"), which implied that he (jokingly) approved of invading France. cf. Tagesspiegel http://www.tagesspiegel.de/politik/deutschland/CDU-Hessen-Junge-Union-StudiVZ;art122,2538091 (German newspaper article from May 26th 2008) (last retrieved: May 30th 2009)

6.2 Implementation of a Social Network

User Profile Some functionalities, such as messaging, are already present in the actual protocol. Other functionalities needs to be implemented. The user's profile is a list of properties. A single property can be implemented as an NSON object with a name and value. As all properties together are relatively small in size, all properties can simply be concatenated into one single file. If one property is modified, this means that the whole modified file needs to be re-uploaded. The user can choose which properties are public and which are limited to certain groups. Non-public profile properties are, as usual, wrapped into encrypted "meta" objects.

Contacts Instead of adding users as "friends", they are added as "contacts". Additionally, users can add their contacts to user groups, such as "friends", "family" and other groups of the user's choice. Adding a contact consists of sending a contact-request object in a message. Note that in the protocol, being a contact is not necessarily symmetric, as there is no central instance that could enforce the symmetry. For each group of users, the user can publish a list of group members, that can be accessed by that group.

Groups "Groups" in a social network can be mapped to "groups" in the protocol. Each group has a special filesystem folder for messaging. The group owner instructs the server to allow messages to be appended to that filesystem folder using a key calculated by hashing the group key. Messages in the group are encrypted by the group key to ensure confidentiality.

Wall Similar to the group functionality, the user defines a special filesystem folder for the wall, which can be written to using a key that all contacts can access. The hash of this key is used to sign messages sent to the server.

Tags Tags are defined by an NSON object of the type "tag". In its value property, it contains either a user ID or freeform text. An optional "rect" property, consisting of an array of 4 numbers, describes the top-left and bottom-right points in which the tag is valid in, for instance, a picture. Other

properties can be introduced to annotate the point in time a user appears in a video, as an example. As tags are data that potentially changes often (i.e. more often than an uploaded picture), tags span a new namespace: "tags". This allows any content to be tagged but requires two more HTTP requests to download the tag file. The retrieved file consists of an implicit array of tags, with the revision and signature in a separate file. This allows for efficient appending.

Note that many services allow searching for tags such as all the photos in the social network in which a particular user appears. This requires a central instance that catalogues all tags for all photos, which clearly is not compatible with the confidentiality requirement. While such a service may be useful for public content, a different approach is taken for confidential content. Each user has a folder with the private name "tags", accessible only by the users contacts. In this folder, the user creates a file with the private name of the tag, which lists all file tagged with this tag. Each tag is accessible by the same set of users as the tagged content. Note that this allows a contact to see if tags exist that are not accessible by that contact.

If a user wants to retrieve all content with a certain tag, the user needs to retrieve all such tags from all contacts. This scales linearly in the number of contacts and linearly in the amount of tagged content a user has. When using a standalone program instead of a web client, retrieving these tag lists could be done automatically in the background. The way that tags are stored is a good example for a limitation imposed by not trusting the service provider: In a "classic" web application, the server can take care of indexing all tags. Here, this must be done by the user to preserve confidentiality. However, data can be organized in a way that is still not as efficient as using a server yet, eases the computational burden on the user.

6.3 State of the Implementation

As a proof of concept, parts of the protocol were implemented as a web application in JavaScript (client-side) and PHP[34] (server-side). The implemen-

[34]PHP Hypertext Preprocessor: http://www.php.net/ (last retrieved May 31st 2009)

tation served to try out ideas and to test whether various concepts worked. At the time of writing, the implementation was severely incomplete and out-of-date. For example, it does not use the Netstring Object Notation but the JavaScript Object Notation in which binary characters need to be encoded. Currently, a user can log on using different passphrases, store and retrieve, encrypt and decrypt, verify and sign content for him- or herself. However, no interaction between users is implemented. The implementation uses open-source JavaScript libraries from third parties such for RSA encryption and decryption, for example, or to calculate cryptographic hashes. The code for Salsa20 encryption and decryption and for RSA signature creation and verificationwas written by the author. The author also implemented Salsa20 encryption and decryption in PHP.

7 Discussion

The following sections discuss whether the protocol solves the confidential publishing problem, by examining each requirement of the confidential publishing problem. Afterwards, the implications of implementing the protocol as a web application are discussed.

7.1 The Confidential Publishing Problem

According to Ross [2008],

> *"A working definition of* evaluation *is 'the process of assembling evidence that a system meets, or fails to meet, a prescribed assurance target'."*

Here, the assurance target is solving the confidential publishing problem, i.e. allowing a user to publish content with confidentiality, integrity, authenticity, a decentralized structure and usability.

In the following, two types of attackers and their impact on protocol security are considered. The *third-party attacker* is an attacker who cannot intercept traffic, but can access all information that is publicly available. As the only requirement for such an attacker is establishing a connection to the server, anyone can be regarded as such an attacker. The *network attacker* can eavesdrop and manipulate any data sent over the network, i.e. between user and server. This allows this attacker to perform, for example, man-in-the-middle attacks. This type of attacker is generally assumed in the literature, e.g. in BAN logic (Burrows et al. [1990]). If the server itself is seen as an attacker, it is a network attacker: the server knows which files are requested and can manipulate them. Also, all data on the server that is relevant to the protocol was sent to the server or will be retrieved from the server. As such, it is the same data that the network attacker knows about when eavesdropping on an unsecured connection. Note here, only the case where data is transmitted over unsecured HTTP connection is considered, as this reveals more information to the network attacker than data that is transferred over an HTTPS connection. It is also assumed that

the cryptographic methods (i.e. Salsa20, RSA and SHA-256) are secure and an attack on them would be computationally infeasible.

7.1.1 Confidentiality

Confidentiality is achieved by encrypting content so that the plaintext is accessible by the target audience. A third-party attacker can see whether encrypted metadata in the user's root folder exists, but cannot access it. If no filesystem file list is accessible[35], the attacker cannot even access ciphertext outside of the root folder's metadata, as the attacker does not know the public names of any other files[36]. The network attacker can intercept any request from the user or response from the server. Most data is stored as ciphertext, with a hexadecimal filename, so little information is disclosed. What the attacker can learn is the public file name, the order of access (e.g. the first file read after reading the passphrase specifications probably is a passphrase secrets folder) and the file size (e.g. a kilobyte is not sufficient to store a long movie). Note that from the file name one can usually infer the user that the file belongs to, so the attacker can see which peer's files a user is interested in.

Password Guessing Confidentiality can be breached if an attacker can successfully guess a user's passphrase, accessing all content that is accessible by that passphrase. In this protocol, a symmetric key is generated from the passphrase, which is used as salt for the passphrase secrets folder. A third-party attacker can generate a list of possible passphrases and the associated passphrase secrets folder's public names and then request that file name. If a file from the list exists, it is highly likely to be the passphrase folder. The attacker can verify this by decrypting the file using the symmetric key generated from the guessed passphrase. The protocol employs various countermeasures to slow down such an attack. First of all, the passphrase folder's filesystem name is dependent on the user ID, the passphrase name

[35]This is the default behavior on most HTTP(S) servers.

[36]Guessing one public name, generated using a 256 bit salt, yields a success chance of about $\frac{1}{2^{255}}$ per attempt and is thus deemed infeasible.

and the passphrase. Thus, an attacker cannot use data that was generated to attack one user to attack a second user. Second, it calculates the symmetric key by hashing a large number of times, at least 1000, to slow down the attacker. If a third-party attacker cannot access a list of filenames, the attacker can only perform online attacks, which are slower. As an improvement, web server modules such as mod_evasive[37] can limit the number of concurrent connections that one IP address can establish to a server, thus slowing down online attacks. A network attacker can see the transferred public filenames. If these filenames include the user's passphrase secrets folder, the attacker can perform an offline password guessing attack.

Password Recovery If the user forgets a passphrase, how can the user access data encrypted by that passphrase? In the protocol, if a user forgets the main passphrase, all the user can do is use a "key lost" message to publish a new long-term key. This implies that the user cannot access past data and poses the question of how the user, without their current private key, can authenticate to the server to be able to publish the key message.

E-mail providers (e.g. AOL, Microsoft, Google and Yahoo!) often use so-called "secret questions" to authenticate the user to change the password. Schechter et al. [2009] showed that secret questions are much easier to guess than passwords. As an example, US vice presidential candidate Sarah Palin's Yahoo! account was hacked because of easily guessable "secret questions"[38]. As such, "secret questions" to authenticate the user should be avoided. It is unclear how else users could be authenticated without their private key. Bruce Schneier notes[39]:

> *"I like to think that if I forget my password, it should be really hard to gain access to my account. I want it to be so hard that*

[37] mod_evasive for apache: http://www.zdziarski.com/projects/mod_evasive/ (last retrieved May 29th 2009)
mod_evasive for lighttpd: http://redmine.lighttpd.net/projects/lighttpd/wiki/Docs (last retrieved May 29th 2009)

[38] Wired.com: http://www.wired.com/threatlevel/2008/09/palin-e-mail-ha/ (last retrieved on May 29th 2009)

[39] From "Schneier of Security": http://www.schneier.com/blog/archives/2005/02/the_curse_of_th.html (February 11, 2005) (last retrieved May 29th 2009)

an attacker can't possibly do it. I know this is a customer service issue, but it's a security issue too. And if the password is controlling access to something important – like my bank account – then the bypass mechanism should be harder, not easier."

In typical web applications, authenticated users can access all the content stored in the past, even after forgetting their passpword. In contrast, while the protocol allows an authenticated user to publish a new long-term key, the user can no longer access past content if he is lacking the passphrase and the associated symmetric keys. While this is frustrating for the user, the alternative is that the service provider possesses something that the user can use to access their data. As the user should be able to use this "something" even after forgetting their password, i.e. without further knowledge required, the service provider would be able to use it, too. As this clearly violates the confidentiality requirement, it seems that the only possible solution is to educate users. They must undestand that forgetting their passphrase means that they cannot access their account anymore. This would be frustrating for many users.

7.1.2 Integrity and Authenticity

To achieve integrity and authenticity, all data relevant to the protocol is cryptographically signed. If the data or the signature is manipulated - by a transmission error or by a network attacker - this is detect when the signature is verified. A network attacker can perform a man-in-the-middle attack, which exchanges the public key to be retrieved with a public key to which the attacker knows the private key. The protocol tries to prevent this by requiring the user to verify the key's human-readable fingerprint when retrieving a key for the first time. Each retrieved key is then stored in an authenticated way, which prevents future man-in-the-middle attacks. When Alice retrieves content from Bob, the network attacker could fake a key message, claiming that Bob lost his long-term key and supplying a new long-term key. This requires no knowledge of the old key. While Alice is shown a warning message, there is a probability that the user will accept this key.

Note that over an unsecure connection, the server is not authenticated. Thus, the attacker could not only suppress, for example, files uploaded by the user, but also simulate to the user that these files were uploaded successfully. The protocol includes no separate mechanism to authenticate the server. The server should be authenticated by HTTPS.

7.1.3 Decentralized Structure

The protocol uses a decentralized structure. Anyone who can set up an HTTP server can be a server in this protocol.

7.1.4 Usability

With only a prototype implementation, it is hard to discuss the usability of the protocol. It should be noted, however, that the use of human-readable fingerprints should improve the usability of public key fingerprint verification.

7.2 Web Application Limitations

Web applications have the advantage of being accessible from any web browser connected to the Internet. As such, they are accessible almost anywhere without a user needing to install any additional software. This is an important property, as it allows the user to easily try out the protocol. Web applications have three limitations relevant to the protocol. First of all, the code that implements the protocol comes from the server. This usually is the same server that confidential data needs to be protected from. Second, web applications are written in scripting languages, the most portable being JavaScript[40]. JavaScript performance varies, depending on the browser used. Third, JavaScript is only allowed to access content coming from the same domain as the visited webpage.

Code Coming from the Web Server When using locally installed applications, such as an email client, the program code comes from one source,

[40]Standardized as ECMAScript [2009]

and the emails come from another source. When using web applications, code and data come from the same source: the web server. In the protocol, this is important as confidential data should not be accessible by the server. As the server controls the code that runs, the server could modify it in a way that exposes the user's secrets - such as the passphrase - to it, breaching the confidentiality requirement. It should be noted that in constrast to existing social networks, which can always access the plaintext, this could be considered a criminal act. However, this is not a legal thesis and such a classification depends on the terms of service and the legislation of the service provider's country.

In conclusion, the web server can manipulate the JavaScript code in any way. Thus, the web application of the protocol should only be used with short-term passphrases that can only access limited data. The main passphrase should only be used on a trusted computer running a stand-alone application.

JavaScript Performance When using the protocol, the main area where JavaScript is noticeably slow is cryptography. Two techniques are so slow that they cannot be considered user-friendly: Signing or decrypting using an RSA private key, and iteratively hashing the passphrase. Table 6 shows how much time it takes to sign a piece of data and to generate a symmetric key from a passphrase for two current web browsers. Note that to "log in", the user needs to generate the symmetric key once. When reading one content file, one signature must be verified, which is at a usable speed on all tested platforms (e.g. about 150 ms to verify using a 2048 bit key on Firefox 3.5). When uploading a file, because it is new or modified, the user needs to sign the file once. Additionally, the user either needs to sign the upload message to the server with a longterm key or with a symmetric key (which is faster).

In conclusion, waiting more than 5 seconds to sign a file with a 2048 bit key (in addition to the actual upload time for a file) is a frustrating user experience. While by today's recommendations, 2048 bit keys should be used, doing so in a browser other than Google Chrome is unacceptable for

Operation	Firefox 3.5	Google Chrome 2.0
Signing with a 2048 bit private key	5896 (1867)	552.55 (9.8)
Signing with a 1024 bit private key	956 (304)	81.7 (5.0)
Signing with a 776 bit private key	493 (156)	36 (0)
Signing with a 512 bit private key	219 (69)	13.1 (0.31)
Hashing SHA-256 1000 times	848 (269)	250 (3.5)
Hashing SHA-256 200 times	173 (55)	51 (0.82)

Table 6: Time to perform various cryptographic operations in JavaScript
All timings in milliseconds, standard deviations in parantheses. measurment was taken 10 times. All timings taken on an Intel(R) Core(TM)2 CPU T5600 running at 1.83GHz.

users. Instead, this thesis recommends the use of a 1024 bit key for short-term passphrases that will be used in web applications, even though 1024 bit are too short by most recommendations (Lenstra and Verheul [2001], Barker et al. [2005], Orman and Hoffman [2004], Rechberger and Rijmen [2008]). Note that it is expected that a 778 bit key will be factorized soon (Montgomery [2008]).

Accessing other Domains The third limitation of web applications is that JavaScript is not allowed to send data to or retrieve data from domains other than the visited webpage. This collides with the protocol's decentrality, i.e. a user needs to be able to download another user's content from that user's server, which can be on another domain. Both Internet Explorer and Firefox allow signed JavaScript to cross domain boundaries[41].

7.2.1 Typical Vulnerabilities in Web Applications

Web applications can contain a multitude of security vulnerabilities. Most of them can be avoided by properly escaping user input. Failing to properly escape user input in web pages can lead to *Cross-Site Scripting* (XSS) attacks. In the implementation, the JavaScript `document.createTextNode()`

[41] Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/ms970704.aspx (last retrieved May 31st 2009),
Mozilla.org JavaScript Security: http://www.mozilla.org/projects/security/components/signed-scripts.html (last retrieved May 31st 2009)

function is used to add user-dependent data to the web page, which treats all input data as text. If not properly escaped data is embedded into an HTTP header, this can lead to *HTTP Header Injection* and *HTTP Response Splitting* attacks. In these attacks, additional headers or the beginning of the actual data transmitted via HTTP are inserted. In the implementation, the response to server messages sets an HTTP header, which depends on user input. The inserted header is selected from a number of static strings, so no data can be injected. A user performing a *Directory Traversal Attack* tries to access files for which the user does not have permission by changing the local path of the file access. This is avoided by always using the full canonized path before verifying whether a user may access a file. Finally, in a *Cross-Site Request Forgery* (XSRF) attack, an attacker embeds an easily guessable link to a site requiring authentication in a web page. If a user authenticated at the linked site and visits the malicious site, the linked site interprets the embedded link in the malicious site as a legitimate request from the user. XSRF attacks can be dwarfed by requiring a token that an attacker cannot easily guess. In the implementation, this applies to server messages which are signed by a cryptographic key, which an attacker should not be able to guess.

8 Conclusion & Outlook

In this thesis, the confidential publishing problem was introduced, in which a user wants to publish content to a selected group of peers, in a way that only these peers can access the content. This needs to be done with confidentiality, integrity, authenticity, a decentralized structure and in a usable way. A protocol to solve the confidential publishing problem was proposed. In the protocol, a user maintains a number of cryptographic groups, which correspond to e.g. friends or family. When publishing content, the user can choose a target audience, i.e. which users and groups can access the content. Only the target audience, and no one else, can access the content. Content files have a private name, that is only revealed to the file's target audience, and a public name that is used as the actual filename. From the public name, a third-party cannot infer the private name. The protocol includes key management, i.e. replacing a key whose validity expires soon with new keys or revoking compromised keys. The user can also define short-term passphrases which can only access a subset of published data. When the user uses a short-term passphrase in an untrusted location, such as an Internet cafe, only the data that the passphrase has access to can be compromised. The protocol also describes server messages, using which the user can upload content to the server. Server messages are also used to create confidential discussion groups or sent confidential messages to the user.

This protocol was implemented to create an example application that mimics current social networks. Using the protocol allows to build a decentral social network that respects the user's privacy. Especially in comparision to existing approaches, very few information about a user is disclosed to third parties, such as the social network's service providers. Just like in real life, users must be careful with whom to share content, as a peer with access to the content can breach confidentiality by sharing it with a third party.

After discussing the protocol requirements, the protocol was found to be a solution to the confidential publishing problem. The security implications of implementing the protocol as a web application were also considered.

The protocol described in this thesis touches many aspects that need

further elaboration. The most prominent would be a in-depth discussion on spam, as this is a big problem in today's messaging systems. Other aspects that need consideration are an efficient "push" model, in which new content is efficiently sent to the user, so that the user does not need to regulary retrieve, for instance, the inbox folder to check for new messages. Note that such a model cannot easily be realized over HTTP, so that another protocol must be used, e.g. psyc[42]. Currently, the protocol is inefficient in modifiying big files, as randomly changing a file requires the user to upload an amout of data linear in the size of the file. Research is needed on how to make modifying files more efficient. Finally, a complete implementation of the protocol is needed, so that users can actually use the protocol. This should include a web application, so that the protocol can be used from anywhere with web access, as well as a stand-alone application that can be used on a computer that the user trusts. It is currently unclear how a web application can be implemented in a way that the user can trust it, if retrieved from a potentially malicious server.

Using the solution to the confidential publishing problem proposed in this thesis Alice and Bob both can publish their personal data to a group of peers of their choice. Alice shares parts of her calendar with different groups and Bob shares pictures with his friends. The protocol can be used in all cases where users want to rely on third party infrastructure to publish content to a group of users without revealing the content to neither the infrastructure nor any other party.

[42]PSYC (Protocol for SYnchronous Conferencing): http://psyc.eu/ (last retrieved June 6th 2009)

A NSON Grammar

This appendix section defines the NSON grammar in EBNF (ISO14977 [1996]).

```
<integer> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
<nson-number> ::= <integer> '%' ('+'|'-')? <integer>
    ('.'<integer>)? ('e' ('+'|'-')? <integer>)? '%' ;
<nson-string> ::= <integer> '''' <octet>* '''' ;
<nson-array> ::= <integer> '[' <nson-list> ']' ;
<nson-object> ::= <integer> '{' <nson-keyvalues> '}' ;
<nson-type> ::= <nson-number> | <nson-string> | <nson-array> | <nson-object>;
<nson-list> ::= (<nson-type> ',' <ws>+)* (<nson-type>)? ','? <ws>* ;
<nson-keyvalue> ::= <nson-string> ':' <ws>* <nson-type>
<nson-keyvalues> ::= ((<nson-keyvalue> | <nson-string>) ',' <ws>+)
    (<nson-keyvalue> | <nson-string>) ','? <ws>*
```

The `<octet>` terminal maps to one octet (8 bit). The `<ws>` terminal is any whitespace character, as defined by the Unicode Character Database[43].

[43] cf. http://unicode.org/Public/UNIDATA/PropList.txt (last retrieved July 1st 2009)

B List of Figures and Tables

List of Figures

List of Tables

References

M. Ackermann, K. Hymon, B. Ludwig, and K. Wilhelm. HelloWorld: An Open Source, Distributed and Secure Social Network. In *W3C Workshop on the Future of Social Networking*. World Wide Web Consortium (W3C), 2009. URL http://www.w3.org/2008/09/msnws/papers/HelloWorld_paper.pdf.

A. Acquisti and R. Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *Privacy Enhancing Technologies*, volume 4258 of *Lecture Notes in Computer Science*, pages 36–58. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-68790-0. doi: 10.1007/11957454_3. URL http://www.springerlink.com/content/gx00n8nh88252822/.

J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman. Mikey: Multimedia internet keying. IETF Draft, June 2003. URL http://securemulticast.org/draft-ietf-msec-mikey-07.txt.

E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for Key Management. *NIST Special Publication*, pages 800–57, 2005. URL http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf.

M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. URL http://www.ietf.org/rfc/rfc3711.txt.

T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005. URL http://www.ietf.org/rfc/rfc3986.txt.

D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. *Lecture Notes in Computer Science*, 3958:207, 2006. URL http://cr.yp.to/ecdh.html.

D. J. Bernstein. The salsa20 family of stream ciphers. pages 84–97, 2008a. doi: http://dx.doi.org/10.1007/978-3-540-68351-3_8. URL http://cr.yp.to/snuffle.html.

D. J. Bernstein. Response to 'On the Salsa20 core function', eSTREAM report 2008/011. 3, February 2008b. URL http://www.ecrypt.eu.org/stream/papers.html.

T. Bray, Paoli J., Sperberg-McQueen C. M., E. Maler, and Yergeau F. Extensible markup language (xml) 1.0 (fifth edition), November 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/.

M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990.

C. Bååth and J. Kühn. Ssh and the man in the middle, June 2002. URL http://www.d.kth.se/~d99-lba/doc/report-jkuhn-cbaath.pdf.

J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. URL http://www.ietf.org/rfc/rfc4880.txt.

I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, pages 46–66, 2001. URL http://www.ecse.rpi.edu/Homepages/shivkuma/teaching/sp2001/readings/freenet.pdf.

P. Crowley. Truncated differential cryptanalysis of five rounds of Salsa20. In *Workshop Record of SASC*, 2006.

ECMAScript. ECMAScript Language Specification, April 2009. URL http://www.ecma-international.org/publications/files/drafts/tc39-2009-025.pdf.

T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. URL http://www.ietf.org/rfc/rfc2616.txt. Updated by RFC 2817.

S. Fischer, W. Meier, C. Berbain, J.F. Biasse, and M.J.B. Robshaw. Non-randomness in eSTREAM Candidates Salsa20 and TSC-4. *Lecture Notes in Computer Science*, 4329:2, 2006.

M. Friedl. Ssh fingerprint format. IETF Draft, March 2002. URL http://tools.ietf.org/html/draft-ietf-secsh-fingerprint-00.

M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543 (Proposed Standard), March 1999. URL http://www.ietf.org/rfc/rfc2543.txt. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265.

ISO14977. Extended BNF. International Organization for Standardization, 1996. URL http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=26153.

X. Jiang, J. I. Hong, and J. A. Landay. Approximate information flows: Socially-based modeling of privacy in ubiquitous computing. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 176–193, London, UK, 2002. Springer-Verlag. ISBN 3-540-44267-7.

J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003. URL http://www.ietf.org/rfc/rfc3447.txt.

B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315 (Informational), March 1998. URL http://www.ietf.org/rfc/rfc2315.txt.

B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. URL http://www.ietf.org/rfc/rfc2898.txt.

D. Kaminsky. Black ops 2006: pattern recognition, toorcon r3mix, 2006. URL http://www.doxpara.com/slides/dmk_blackops2006.ppt.

A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of cryptology*, 14(4):255–293, 2001.

N. Y. Mineta, C. L. Shavers, and R. G. Kammer. The keyed-hash message authentication code (hmac). Federal Information Processing Standard Publication, 2001. URL http://csrc.nist.gov/publications/fips/dfips-HMAC.pdf.

P. L. Montgomery. Preliminary design of post-sieving processing for rsa-768. In *CADO workshop on integer factorization*, oct 2008. URL http://cado.gforge.inria.fr/workshop/slides/montgomery.pdf.

H. Orman and P. Hoffman. Determining Strengths For Public Keys Used For Exchanging Symmetric Keys. RFC 3766 (Best Current Practice), April 2004. URL http://www.ietf.org/rfc/rfc3766.txt.

C. Rechberger and V. Rijmen. ECRYPT yearly report on algorithms and keysizes (2007-2008). Technical report, D. SPA. 28, IST-2002-507932, 2008.

R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359340.359342.

J. A. Ross. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2008. ISBN 0470068523.

S Schechter, A. J. B. Brush, and S. Egelman. It's no secret: Measuring the security and reliability of authentication via 'secret' questions. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998. URL http://www.ietf.org/rfc/rfc2326.txt.

R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational), August 2007. URL http://www.ietf.org/rfc/rfc4949.txt.

H. Story, B. Harbulot, I. Jacobi, and M. Jones. FOAF+TLS: RESTful Authentication for the Social Web. 2009. URL http://CEUR-WS.org/Vol-447/paper5.pdf.

H. Sundell and P. Tsigas. Lock-free and practical deques using single-word compare-and-swap. Technical Report 2004-02, Computing Science, Chalmers University of Technology, March 2004. URL citeseer.ist.psu.edu/sundell04lockfree.html.

Y. Tsunoo, T. Saito, H. Kubo, T. Suzaki, and H. Nakashima. Differential cryptanalysis of Salsa20/8. In *Workshop Record of SASC*, 2007.

D. Wallner, E. Harder, and R. Agee. Key Management for Multicast: Issues and Architectures. RFC 2627 (Informational), June 1999. URL http://www.ietf.org/rfc/rfc2627.txt.

K. Welke and K. Rechert. Privacy-aware location sharing (under peer review). In *Proceedings of the 4th International Conference on Pervasive Computing and Applications*. ACM New York, NY, USA, 2009.

A. F. Westin. *Privacy and Freedom*. The Bodley Head Ltd, 1970. ISBN 0370013255.

M. Wildgrube. Structured Data Exchange Format (SDXF). RFC 3072 (Informational), March 2001. URL http://www.ietf.org/rfc/rfc3072.txt.

C. K. Wong, M. Gouda, and S. S. Lam. Secure group communications using key graphs. *IEEE/ACM transactions on networking*, 8(1):16–30, 2000.

X.680. Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. International Telecommunication

Union, November 2008. URL http://www.itu.int/rec/T-REC-X.680-200811-P/en.

S. Xu. On the security of group communication schemes based on symmetric key cryptosystems. In *SASN '05: Proceedings of the 3rd ACM workshop on Security of ad hoc and sensor networks*, pages 22–31, New York, NY, USA, 2005. ACM. ISBN 1-59593-227-5. doi: http://doi.acm.org/10.1145/1102219.1102224.

T. Ylonen. Ssh transport layer protocol. IETF Draft, March 2005. URL http://tools.ietf.org/html/draft-ietf-secsh-transport-24.