Privacy-Preserving Access Control in Publicly Readable Storage Systems

Daniel Bosk(✉) and Sonja Buchegger

School of Computer Science and Communication,
KTH Royal Institute of Technology, Stockholm, Sweden
{dbosk,buc}@kth.se

Abstract. In this paper, we focus on achieving privacy-preserving access control mechanisms for decentralized storage, primarily intended for an asynchronous message passing setting. We propose two modular constructions, one using a pull strategy and the other a push strategy for sharing data. These models yield different privacy properties and requirements on the underlying system. We achieve hidden policies, hidden credentials and hidden decisions. We additionally achieve what could be called 'hidden policy-updates', meaning that previously-authorized subjects cannot determine if they have been excluded from future updates or not.

Keywords: Privacy · Access control · Cloud storage · Decentralized storage · Hidden policies · Hidden policy-updates · Hidden credentials

1 Introduction

Alice and her friends want to communicate asynchronously. To do this they want to use a publicly available file system and write their messages to different files, which the other party later can read. This is a possible architecture for a decentralized online social network (DOSN). We are interested in enforcing access-control policies in such a public file system which does not have any built-in access control mechanisms. Our approach is to introduce a layer of encryption as a logical reference monitor. Beyond the expected confidentiality, Alice wants some stronger privacy properties as well: her friends should not be able to monitor her activity, e.g. infer with whom else she communicates, even the fact that she communicates with others.

We will present an ideal model of communication and its desired properties in Sect. 2, this is what we want to achieve. Then we will present the building blocks that we will use, and their accompanying assumptions, in Sect. 3. We will also assume a simple file system with no built-in access control. This system is discussed and defined in Sect. 3.1. This includes our adversary model: we will let the adversary control the file system.

Then we give two constructions that implement the functionality using different message-passing models and analyse what properties their primitives must

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved

have in Sects. 4 and 5. The two message passing models are called the pull model and the push model. In the pull model Alice's friends pull new messages from Alice, whereas in the push model Alice pushes new messages to her friends' inboxes. The motivation is that the pull model is optimized for Alice and the push model for Alice's friends. In a decentralized file system the pull model yields few connections for Alice, but many for her friends (if they have more friends than Alice). Conversely the push model yields many connections for Alice, but only one for her friends (they check their inbox). In some situations connections can be expensive, cf. establishing many Tor [5] circuits to transfer little data and establishing one circuit to transfer more data.

After presenting our constructions we shortly analyse their algorithmic complexity in Sect. 6. Finally we compare our results to related work in Sect. 7 and conclude by summarizing the main contributions and future work in Sect. 8.

2 The Ideal Communication Model

There are several ways Alice can implement the message passing with her friends. We will start by presenting an ideal model of communication (Definition 1) whose properties Alice wants to achieve. Then we will proceed to the details of two alternative protocols (Definitions 4 and 6) that yields the properties of the ideal model.

Definition 1 (Communication Model). *Let $\mathcal{C}_{p,S}$ be a (p,S)*-communication model *with a* publisher *p and a* set of subscribers *S. Then we have the following operations defined on $\mathcal{C}_{p,S}$:*

- *The publisher first runs $\mathcal{C}_{p,S}[p].\mathsf{setup}(1^\lambda)$ and each subscriber $s \in S$ runs $\mathcal{C}_{p,S}[p].\mathsf{setup}(1^\lambda)$, where λ is the security parameter.*
- *The publisher p uses $\mathcal{C}_{p,S}[p].\mathsf{publish}(R, m)$ to publish the message m to the designated* recipient set *R by making it available to all recipients $r \in R \subseteq S$.*
- *Each subscriber $s \in S$ uses $\mathcal{C}_{p,S}[s].\mathsf{get}()$ to get the set M of published messages $m \in M$ for which s was in the recipient set.*

Whenever p, S are clear from the context, we will simply omit them, e.g. $\mathcal{C}[p]$.setup and $\mathcal{C}[s]$.setup.

There are two ways the adversary can gain information. The first is from corrupted subscribers and the second is from the public file system over which the communication model is implemented. Hence the second is not visible in Definition 1, but will be in Definitions 4 and 6.

In our desired scenario, Alice acts as a publisher and her friends as the subscribers. We want the following properties:

Message Privacy. No subscriber $s' \in S$ can use $\mathcal{C}[s]$.get for $s' \neq s$. I.e. no subscriber can read the messages of any other subscriber, we call this property *message privacy.*

Hidden Policies. No subscriber s_i must learn which $s_1, \ldots, s_n \in R$ beyond that $i \in \{1, \ldots, n\}$ and $s_i \in R$ for a given message m. I.e. no subscriber must know who else received a message, we call this property *hidden policy*. Thus if Alice publishes a message, none of her friends know who else received it and none can read the others' received messages to check if they have received the same message.

Hidden Policy-Updates. In addition to the hidden-policy property, we want something we call *hidden policy-updates*. Consider the following example: Bob might become jealous if he realizes that Alice no longer includes him in the recipient set for her messages. As such Bob should only be able to determine that Alice publishes content by being in the recipient set himself.

There are several ways to implement the message-passing protocol for the communication model in Definition 1. We will focus on two alternative protocols, one using the pull model and the other using the push model for communication. The push model is analogous to the subscription of magazines: a subscriber contacts the publisher and signs a subscription, whenever the publisher issues a new magazine it sends a copy to the subscriber's mailbox. The pull model is the converse of the push model. It is analogous to the selling of magazines in kiosks: the publisher issues magazines and the 'subscribers' come to the kiosk and buy them whenever they want. We can see that our ideal communication model allows both a pull and push strategy for implementation. We will describe and analyse the pull construction in Sect. 4 and the push construction in Sect. 5. But first we need to review our needed building blocks.

3 Building Blocks

We will now describe the primitives and our assumptions upon which we will base our constructions. We start with a general file system, which is publicly readable and has no built in access control. We then describe the general cryptographic primitives we need and finish with the ANOBE [9] scheme which we will use for part of our construction.

3.1 A General File System

We will model our system as an abstract file system with the operations append (which includes create) and read. The operations are defined as we intuitively expect: we can create an object, read it and also append to it. The file system itself provides no access control.

Definition 2 (Public File System). *A* file $f = (i, m_i)$ *consists of the* identifier i *and the associated* content m_i. *We define the set of files* $F = \{(i, m_i)\}$ *together with the following operations to be the* public file system $\mathcal{FS}$*:*

- $\mathcal{FS}.\mathsf{append}(i, m_i)$ *will set* $(i, m) \leftarrow (i, m \,||\, m')$ *for* $(i, m) \in F$. *If* $\{(i', m') \in F \mid i' = i\} = \emptyset$, *i.e. the file* i *does not yet exist, this operation will create it by setting* $F \leftarrow F \cup \{(i, m_i)\}$.
- $\mathcal{FS}.\mathsf{read}(i) = m_i$ *if* $(i, m_i) \in F$, *otherwise* $\mathcal{FS}.\mathsf{read}(i) = \perp$.

We can see in the definition that anyone can read any object in the file system. Anyone can also create new files and append to existing files.

We will let the adversary Eve operate the file system $\mathcal{FS}$ defined in Definition 2. By this we mean that Eve has access to the internal state of $\mathcal{FS}$ directly, i.e. the set of files F. She can thus read all files (same as everyone else), but she can also do arbitrary searches since she does not have to use $\mathcal{FS}.\mathsf{read}$.

Although Eve could potentially also modify or delete the files, we do not treat denial-of-service attacks by Eve deleting the files. However, as we will see in our constructions, modifications (but not deletions) made by Eve will be detected.

Eve cannot distinguish between Alice's and Bob's requests to the operations of $\mathcal{FS}$ — in the definition there is nothing to identify who used any of the operations, so this is consistent with the definition. What she can do, is to record the times at which the operations occurred, since she executes them.

This adversary would correspond to a centralized setting, e.g. having the file system hosted by a cloud operator, or a decentralized setting where the adversary can monitor the network at the file system side.

3.2 Cryptographic Primitives

We will use mainly standard public-key cryptography. We need a public-key encryption scheme and two signature schemes. Although we can use shared-key mechanisms in some places, we will maintain a public-key notation throughout this paper and merely point out in the places where a shared-key scheme would be possible.

We will use a public-key encryption scheme $\mathsf{E} = (\mathsf{E.Keygen}, \mathsf{E.Enc}, \mathsf{E.Dec})$. This scheme must be semantically secure under chosen-ciphertext attacks (IND-CCA) and key-private (AI-CCA) [3], i.e. the ciphertext does not leak under which key it was created. We also need an unforgeable signature scheme $\mathsf{S} = (\mathsf{S.Keygen}, \mathsf{S.Sign}, \mathsf{S.Verify})$.

We additionally need an anonymous and semantically secure (ANO-IND-CCA) [9] broadcast encryption (BE) [6] scheme $\mathsf{BE} = (\mathsf{BE.Setup}, \mathsf{BE.Keygen}, \mathsf{BE.Enc}, \mathsf{BE.Dec})$. Let $U = \{1, \ldots, n\}$ be the universe of users. $\mathsf{BE.Setup}(1^\lambda, n)$ generates a master public key MPK and master secret key MSK for n users. $\mathsf{BE.Keygen}(MPK, MSK, i)$ generates the secret key k_i^{Pri} for user $i \in U$. Then $\mathsf{BE.Enc}(MPK, m, R)$ will encrypt a message m to a ciphertext c for the set of users $R \subseteq U$ and $\mathsf{BE.Dec}(MPK, k_i^{\mathsf{Pri}}, c)$ will return m if $i \in R$. Formally we define ANO-IND-CCA by the following game.

Definition 3 (ANO-IND-CCA [9]). *Let $U = \{1, \ldots, n\}$ be the universe of users and BE be BE scheme. BE scheme is anonymous and adaptively IND-CCA (ANO-IND-CCA) if the adversary has negligible advantage in winning the following game.*

Setup. *The challenger runs $\mathsf{BE.Setup}(1^\lambda, n)$, where λ is the security parameter. This generates a master public key which is given to the adversary $\mathcal{A}$.*

Phase 1. *The adversary $\mathcal{A}$ may corrupt any $i \in U$, i.e. request its secret key $k_i = \mathsf{BE.Keygen}(MPK, MSK, i)$ through an oracle. Additionally $\mathcal{A}$ has access to a decryption oracle to decrypt arbitrary ciphertexts for any $i \in U$, upon request (c, i) the oracle will return $\mathsf{BE.Dec}(MPK, k_i, c)$.*

Challenge. *The adversary chooses messages m_0, m_1 such that $|m_0| = |m_1|$ and recipient sets R_0, R_1 such that $|R_0| = |R_1|$. For all corrupted $i \in U$ we have $i \notin R_0 \cup R_1 \setminus (R_0 \cap R_1)$. If there exists an $i \in R_0 \cap R_1$ we require $m_0 = m_1$. Then $\mathcal{A}$ gives them to the challenger. The challenger randomly chooses $b \in \{0, 1\}$ and runs $c^* \leftarrow \mathsf{BE.Enc}(MPK, m_b, R_b)$ and gives c^* to the adversary.*

Phase 2. *The adversary may continue to corrupt $i \notin R_0 \cup R_1 \setminus (R_0 \cap R_1)$. $\mathcal{A}$ may corrupt $i \in R_0 \cap R_1$ only if $m_0 = m_1$. The adversary is not allowed to use the decryption oracle on the challenge ciphertext c^*.*

Guess. *$\mathcal{A}$ outputs a bit $\hat{b}$ and wins the game if $\hat{b} = b$.*

We define the adversary's advantage $\mathbf{Adv}_{\mathcal{A},\mathsf{BE}}^{ANO\text{-}IND\text{-}CCA}(1^\lambda) = |\Pr[\hat{b} = b] - \frac{1}{2}|$.

Throughout BE can be any BE scheme with the ANO-IND-CCA property. However, we will use the ANOBE scheme by Libert et al. [9] as an example in our discussion and modify it for one of our constructions. For our description of this scheme, which follows, we need slight variants of the above encryption (E) and signature (S) schemes. We need an encryption scheme $\bar{\mathsf{E}} = (\bar{\mathsf{E}}.\text{Keygen}, \bar{\mathsf{E}}.\text{Enc}, \bar{\mathsf{E}}.\text{Dec})$ which is in addition to key-private also robust (ROB-CCA) [1]. For the signature scheme $\bar{\mathsf{S}} = (\bar{\mathsf{S}}.\text{Keygen}, \bar{\mathsf{S}}.\text{Sign}, \bar{\mathsf{S}}.\text{Verify})$, we only need it to be a strongly-unforgeable one-time signature scheme.

3.3 An Anonymous Broadcast Encryption Scheme

We will now describe the ANOBE scheme by Libert et al. [9]. We will use ANOBE to denote this scheme. The algorithms work as follows. $\mathsf{ANOBE.Setup}$ generates a master public key $MPK = \left(\bar{\mathsf{S}}, \left\{k_i^{\mathsf{Pub}}\right\}_{i \in U}\right)$ and the master secret key $MSK = \left\{k_i^{\mathsf{Pri}}\right\}_{i \in U}$, where $(k_i^{\mathsf{Pub}}, k_i^{\mathsf{Pri}}) \xleftarrow{\$} \bar{\mathsf{E}}.\mathsf{Keygen}(1^\lambda)$. $\mathsf{ANOBE.Keygen}(MPK, MSK, i)$ simply returns k_i^{Pri} from MSK. An overview of the encryption function is given in Fig. 1 and an overview of the decryption function in Fig. 2.

Encryption. We must first generate a one-time signature key-pair (s, v), then we choose a random permutation $\pi \colon R \to R$. Next we must encrypt the message m and the verification key (m, v) for every user $i \in R$ in the recipient set $R \subseteq U$ under their respective public key, $c_i = \bar{\mathsf{E}}.\mathsf{Enc}\left(k_i^{\mathsf{Pub}}, m||v\right)$. We let the ANOBE ciphertext be the tuple (v, C, σ), where $C = (c_{\pi(1)}, \ldots, c_{\pi(|S|)})$ and $\sigma = \bar{\mathsf{S}}.\mathsf{Sign}(s, C)$. Note that the signature does not authenticate the sender, it ties the ciphertext together and is needed for correctness.

Decryption. We now have data which would like to decrypt. We parse it as (v, C, σ). If $\bar{\mathsf{S}}.\text{Verify}(v, C, \sigma) = 0$, we return $\perp$ as the verification failed. For each

```
function $\mathsf{ANOBE.Enc}(MPK, m, R)$          ▷ Recipient set $R$, $m$ to be encrypted.
    $(s, v) \overset{\$}{\leftarrow} \overline{\mathsf{S}}.\mathsf{Keygen}(1^\lambda)$          ▷ Signature key-pair, security parameter $\lambda$
    Choose a random permutation $\pi\colon R \to R$.
    for $i \in R$ do
        $c_i \leftarrow \overline{\mathsf{E}}.\mathsf{Enc}(k_i^{\mathsf{Pub}}, m \parallel v)$
    $C \leftarrow (c_{\pi(1)}, \ldots, c_{\pi(|S|)})$
    $\sigma \leftarrow \overline{\mathsf{S}}.\mathsf{Sign}(s, C)$
    return $(v, C, \sigma)$
```

Fig. 1. An algorithmic overview of the encryption algorithm in the ANOBE scheme.

```
function $\mathsf{ANOBE.Dec}(MPK, k^{\mathsf{Pri}}, C_{\mathsf{ANOBE}})$
    if $\overline{\mathsf{S}}.\mathsf{Verify}(v, C, \sigma) = 0$ then
        return $\perp$
    for $c \in C$ do
        $M \leftarrow \overline{\mathsf{E}}.\mathsf{Dec}(k^{\mathsf{Pri}}, c)$          ▷ Try to decrypt
        if $M = \perp$ then
            return $\perp$
        else if $M = (m, v)$ then
            return m
    return $\perp$
```

Fig. 2. An algorithmic overview of the decryption algorithm in the ANOBE scheme.

c in C: Compute $M = \bar{\mathsf{E}}.\mathsf{Dec}\left(k^{\mathsf{Pri}}, c\right)$. If $M \neq \perp$ and $M = (m, v)$, then return m. Otherwise, try the next c. If there are no more c to try, then return $\perp$.

To decrypt an ANOBE ciphertext, we need a trial-and-error decryption procedure to decide if the ciphertext was indeed intended for us. This is costly as it makes the decryption function complexity $O(|S|)$. Libert et al. [9] presented a tag-hint system along with their ANOBE scheme. The tag-hint system reduced the complexity back to $O(1)$. As this is not relevant for our discussion, we refer the reader to [9] but note that it can be used.

4 Construction and Analysis of the Pull Protocol

In this construction, each publisher has an 'outbox' file in the file system. This is simply a file object with a randomly chosen identifier. The publisher adds new publications to the outbox and subscribers pull new content from the outbox. In our analysis we define the protocol for the pull model as follows.

Definition 4 (Pull Protocol). *Let* BE *be a BE scheme,* S *be a signature scheme and* $\mathcal{FS}$ *be a public file system as defined in* Sect. 3. *We denote by* $\mathcal{C}_{p,S}^{\mathsf{Pull}}$ *the* pull protocol *implementing a* (p, S)*-communication model through the operations in* Fig. 3.

```
function $\mathcal{C}^{\mathsf{Pull}}_{p,S}[p].\mathsf{setup}(1^\lambda)$
    $i_p \xleftarrow{\$} \{0,1\}^\lambda$
    $(s_p, v_p) \xleftarrow{\$} \mathsf{S.Keygen}(1^\lambda)$
    $(MPK, MSK) \xleftarrow{\$} \mathsf{BE.Setup}(1^\lambda, |S|)$
    for $s \in S$ do
        $k_s^{\mathsf{Pri}} \leftarrow \mathsf{BE.Keygen}(MPK, MSK, s)$
        Give $(k_s^{\mathsf{Pri}}, v_p, i_p)$ to $s$.

function $\mathcal{C}^{\mathsf{Pull}}_{p,S}[p].\mathsf{publish}(R, m)$
    $c \leftarrow \mathsf{BE.Enc}(MPK, R, m)$
    $\sigma \leftarrow \mathsf{S.Sign}(s_p, c)$
    $\mathcal{FS}.\mathsf{append}(i_p, c \,\|\, \sigma)$

function $\mathcal{C}^{\mathsf{Pull}}_{p,S}[s].\mathsf{setup}(1^\lambda)$
    Receive from $p$.

function $\mathcal{C}^{\mathsf{Pull}}_{p,S}[s].\mathsf{fetch}$
    $C = \mathcal{FS}.\mathsf{read}(i_p)$
    $M \leftarrow \emptyset$
    for $(c, \sigma) \in C$ do
        if $\mathsf{S.Verify}(v_p, c, \sigma) \neq 1$ then
            Continue with next.
        $m_c \leftarrow \mathsf{BE.Dec}(MPK, k_s^{\mathsf{Pri}}, c)$
        if $m_c \neq \bot$ then
            $M \leftarrow M \cup \{m_c\}$
    return $M$
```

Fig. 3. Functions implementing the communication model for the pull protocol. The publisher's interface is to the left and the subscribers' to the right.

When Alice executes $\mathcal{C}^{\mathsf{Pull}}[p].\mathsf{setup}$ she will create all needed keys. She also randomly chooses an identifier. Then each respective secret key, the verification key and identifier are given to all her friends.

When Alice wants to publish a message m to the recipient set $R \subseteq S$, she runs $\mathcal{C}^{\mathsf{Pull}}[p].\mathsf{publish}(R, m)$. This operation creates a BE ciphertext and Alice appends the ciphertext to her outbox file. Each friend can then use $\mathcal{C}^{\mathsf{Pull}}[s].\mathsf{fetch}$ to retrieve the ciphertexts from the file system and decrypt them.

Note that we can have symmetric authentication, e.g. replacing S with a message-authentication code (MAC) scheme, although we use the notation of asymmetric authentication. This would yield different privacy properties, i.e. we remove the non-repudiation which S brings. However, this is not at the centre of our discussion, it just illustrates the modularity of the scheme.

We wanted no recipient to know who else received the message, the ANO-IND-CCA property of the BE scheme gives us exactly this. So we achieve the hidden policy that we wanted.

However, we cannot immediately achieve the hidden policy-update property. Bob can read Alice's outbox, conclude that there are entries which he cannot decrypt and thus he can become jealous. In Fig. 3 we see that Bob can count the number of $\bot$ in the output of $\mathcal{C}^{\mathsf{Pull}}[\cdot].\mathsf{fetch}$. One approach to prevent this is that Alice uses a unique outbox per friend, so Bob has his own outbox. This actually reduces the problem to our push construction, which we will cover later. We will instead analyse a simpler solution for the pull protocol now.

4.1 Changing Recipient Set in the Pull Protocol

We will now propose and analyse a solution to the problem of hidden policy-updates, i.e. Jealous Bob. This solution, however, is based on the intricacies of the ANOBE construction, and thus require us to use ANOBE. In essence, the

ANOBE construction, described in Sect. 3.3, allows us to create a ciphertext which decrypts to different messages.[1] We will use this to encrypt a special message that updates the outbox identifier i_p. Half of the recipients will receive i'_p and the other half will receive i''_p. We summarize this algorithm in Fig. 4. Alice could divide her subscribers into an arbitrary number of parts, even $|S|$ for having each subscriber in their own group. But for simplicity we describe the algorithm for dividing the subscriber set into two parts.

```
function SplitGroup($MPK, S_0, S_1$)    ▷ Recipient sets $S_0, S_1$ such that $S = S_0 \cup S_1$ and $S_0 \cap S_1 = \emptyset$.
    $i_0 \xleftarrow{\$} \{0,1\}^\lambda$, $(s_0, v_0) \xleftarrow{\$} \overline{\mathsf{S}}.\mathsf{Keygen}(1^\lambda)$    ▷ Generate new outbox and key-pair.
    $i_1 \xleftarrow{\$} \{0,1\}^\lambda$, $(s_1, v_1) \xleftarrow{\$} \mathsf{S}.\mathsf{Keygen}(1^\lambda)$    ▷ One per new group.
    $(s, v) \leftarrow \overline{\mathsf{S}}.\mathsf{Keygen}(1^\lambda)$    ▷ One-time signature-verification key-pair.
    Choose a random permutation $\pi\colon S \to S$.
    for $s \in S$ do
        if $s \in S_0$ then
            $c_s \leftarrow \overline{\mathsf{E}}.\mathsf{Enc}(k_s^{\mathsf{Pub}}, i_0 \parallel v_0 \parallel v)$
        else    ▷ $s \in S_1$
            $c_s \leftarrow \overline{\mathsf{E}}.\mathsf{Enc}(k_s^{\mathsf{Pub}}, i_1 \parallel v_1 \parallel v)$
    $C \leftarrow (c_{\pi(s)})_{s \in S}$    ▷ Put all subciphertexts in random order.
    $\sigma \leftarrow \overline{\mathsf{S}}.\mathsf{Sign}(s, C)$
    return $(i_0, s_0), (i_1, s_1), (v, C, \sigma)$
```

Fig. 4. An algorithm splitting a subscriber set S into two new S_0, S_1.

We note that this solution does not hide the fact that Alice updated her policy, we just hide from Bob whether he was excluded or not. When Bob get jealous he goes to Eve to ask her help, this means that we must prevent Eve from learning how Alice changed her policy too — the construction is also secure against this strong adversary.

The property we want from this algorithm is that Bob cannot determine how the subscribers are divided. Thus he cannot know whether he is removed or not. This follows from the ANO-IND-CCA property of the ANOBE scheme. We refer the reader to the proof of Theorem 1 in the full version of [9].

Now we know that Bob cannot distinguish whether everyone in the recipient set received the same message or not. We will add the split-group algorithm (Fig. 4) as an extension to the pull protocol of Definition 4 and we summarize this as the Extended Pull Protocol in the following definition.

[1] If this is not possible for BE scheme under consideration, then there is always the possibility to create a new instance. However, this might be more costly, since it requires a separate secure channel. But we are not bound to the ANOBE scheme, it simply provides some extra convenience in this case.

```
function $\mathcal{C}^{\mathsf{Pull}}_{p,S}[p].\mathsf{split}(R_0, R_1)$
    $(i_0, s_0), (i_1, s_1), C$
      $\leftarrow \mathsf{SplitGroup}(MPK, R_0, R_1)$
    $\sigma \leftarrow \mathsf{S.Sign}(s_p, C)$
    $\mathcal{FS}.\mathsf{append}(i_p, C \parallel \sigma)$
    return $\mathcal{C}^{\mathsf{Pull}}_{p,R_0}, \mathcal{C}^{\mathsf{Pull}}_{p,R_1}$

function $\mathcal{C}^{\mathsf{Pull}}_{p,S}[s].\mathsf{fetch}$
    $C = \mathcal{FS}.\mathsf{read}(i_p)$
    $M \leftarrow \emptyset$
    for $(c \parallel \sigma) \in C$ do
        if $\bar{\mathsf{S}}.\mathsf{Verify}(v_p, c, \sigma) \neq 1$ then
            Continue with next.
        $m_c = \bar{\mathsf{E}}.\mathsf{Dec}\big(k^{\mathsf{Pri}}_s, c\big)$
        if $m_c$ is new identifier then
            $(i_p, v_p) \leftarrow m_c$
        else
            $M \leftarrow M \cup \{m_c\}$
    return $M$
```

Fig. 5. The additional and modified interfaces of the Extended Pull Protocol. We assume there exists a coding that can differentiate a file identifier from an ordinary message.

Definition 5 (Extended Pull Protocol). *Let $\mathcal{C}^{\mathsf{Pull}}_{p,S}$ be an instance of the Pull Model as in* Definition 4. *Then we define the* Extended Pull Model *to additionally provide the interfaces in* Fig. 5.

Note that the execution of $\mathcal{C}^{\mathsf{Pull}}_{p,S}[p].\mathsf{split}(R_0, R_1)$ results in two new instances of the pull protocol, namely $\mathcal{C}^{\mathsf{Pull}}_{p,R_0}$ and $\mathcal{C}^{\mathsf{Pull}}_{p,R_1}$. After this happens, Alice should no longer use $\mathcal{C}^{\mathsf{Pull}}_{p,S}$, instead she should only use these two new instances.

We could instead incorporate $\mathcal{C}^{\mathsf{Pull}}[p].\mathsf{split}$ into the $C^{Pull}[p].\mathsf{publish}$ interface and let it keep track of the different groups (instances) in the state, this would make it more similar to the definition of the ideal model (Definition 1). However, for presentation purposes, many simpler algorithms are better than fewer that are more complex.

On the subscribers' side, due to the construction of $\mathcal{C}^{\mathsf{Pull}}_{p,S}[s].\mathsf{fetch}$, their instance $\mathcal{C}^{\mathsf{Pull}}_{p,S}$ is automatically turned into $\mathcal{C}^{\mathsf{Pull}}_{p,R_i}$, for whichever $i \in \{0, 1\}$ Alice put them in.

4.2 Running Multiple Pull Instances in Parallel

Now it remains for us to convince ourselves that Bob cannot distinguish between different instances of the pull protocol: i.e. Alice posting to two both R_0 and R_1 or Alice posting to R_0 and Carol to R_1.

We will use an information-theoretic argument. We can see that $\mathcal{C}^{\mathsf{Pull}}_{p_0,S_0}$ is information-theoretically independent from both $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^0_1}$ and $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^1_1}$. Although $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^0_1}$ and $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^1_1}$ are coming from a split, we can see in Fig. 4 that the two instances only depend on the randomly chosen identifiers and verification keys, and the sets of users. The public keys can remain the same, the key-privacy property of $\bar{\mathsf{E}}$ will ensure this. It follows that $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^0_1}$ is as indistinguishable from $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^1_1}$ as $\mathcal{C}^{\mathsf{Pull}}_{p_0,S_0}$ is indistinguishable from $\mathcal{C}^{\mathsf{Pull}}_{p_1,S^i_1}$, for $i \in \{0, 1\}$. Thus Bob's only chance is by distinguishing who received what new outbox in the split message, but this is difficult given the ANO-IND-CCA property.

5 Construction and Analysis of the Push Protocol

The idea of the push model is for each subscriber to have an inbox in the file system — as opposed to the pull model, where the publisher has an outbox. This is simply a file object with a randomly chosen identifier. The publisher then puts all published material in the inbox of each subscriber.

We can see that if we simply put the broadcast ciphertext from the pull protocol in all inboxes, then Eve can relate them since they contain identical ciphertexts. We thus have to make some more modifications. We will use the protocol in the next definition in our analysis.

Definition 6 (Push Protocol). *Let* $\mathsf{E} = (\mathsf{E.Keygen}, \mathsf{E.Enc}, \mathsf{E.Dec})$ *be an AI-CCA encryption scheme,* $\mathsf{S} = (\mathsf{S.Keygen}, \mathsf{S.Sign}, \mathsf{S.Verify})$ *be a strongly unforgeable signature scheme, and* $\mathcal{FS}$ *be a public file system. We denote by* $\mathcal{C}^{\mathsf{Push}}_{p,S}$ *the* push model protocol *implementing a* (p, S)*-communication model through the operations in* Fig. 6.

When Alice executes $\mathcal{C}^{\mathsf{Push}}[p].\mathsf{setup}$ it generates a signature-verification key-pair (s_r, v_r) for every subscriber $r \in S$ and gives the respective verification keys to each subscriber. Each subscriber $r \in S$, when they execute $\mathcal{C}^{\mathsf{Push}}[s].\mathsf{setup}$ it generates a public-private key-pair. Additionally they randomly choose a string as an identifier for their inbox. They give the public key and the identifier to Alice.

```
function $\mathcal{C}^{\mathsf{Push}}_{p,S}[p].\mathsf{setup}(1^\lambda)$
    for $r \in S$ do
        $(s_r, v_r) \xleftarrow{\$} \mathsf{S.Keygen}(1^\lambda)$
        Give $v_r$ to $r$.

function $\mathcal{C}^{\mathsf{Push}}_{p,S}[p].\mathsf{publish}(R, m)$
    for $r \in R$ do
        $c_r \leftarrow \mathsf{E.Enc}(k_r^{\mathsf{Pub}}, m)$
        $\sigma_r \leftarrow \mathsf{S.Sign}(s_p, c_r)$
        $\mathcal{FS}.\mathsf{append}(i_r, c_r \parallel \sigma_r)$

function $\mathcal{C}^{\mathsf{Push}}_{p,S}[r].\mathsf{setup}(1^\lambda)$        $\triangleright\ r \in S$
    $(k_r^{\mathsf{Pub}}, k_r^{\mathsf{Pri}}) \xleftarrow{\$} \mathsf{E.Keygen}(1^\lambda)$
    $i_r \xleftarrow{\$} \{0,1\}^\lambda$
    Give $(k_r^{\mathsf{Pub}}, i_r)$ to $p$.

function $\mathcal{C}^{\mathsf{Push}}_{p,S}[r].\mathsf{fetch}$        $\triangleright\ r \in S$
    $C \leftarrow \mathcal{FS}.\mathsf{read}(i_r)$
    if $C = \bot$ then
        return $\emptyset$
    $M \leftarrow \emptyset$
    for $(c, \sigma) \in C$ do
        if $\mathsf{S.Verify}(v_r, c, \sigma) \neq 1$ then
            Continue with next.
        $m_c \leftarrow \mathsf{E.Dec}(k_r^{\mathsf{Pri}}, c)$
        if $m_c \neq \bot$ then
            $M \leftarrow M \cup \{m_c\}$
    return $M$
```

Fig. 6. Functions implementing the communication model for the push model protocol. The publisher's interface is to the left and the subscribers' to the right.

When Alice wants to send a message m to a subset $R \subseteq S$ of her subscribers, she uses $\mathcal{C}^{\mathsf{Push}}[p]$ to create a ciphertext c_r with signature σ_r for each recipient $r \in R$. Then she uses $\mathcal{FS}$ to append $c_r \,||\, \sigma_r$ to the file with identifier i_r. The operation $\mathcal{C}^{\mathsf{Pull}}[r].\mathsf{fetch}$ works similarly as in the pull protocol, however, it uses a different file. It starts by reading the inbox from the file system. Then it iterates through the list of entries, decrypting each entry. Each entry, if successfully decrypted, is a new message. It appends the message to the list of messages which it returns upon finishing.

Note that, similarly as for the pull protocol, the authentication scheme S can be symmetric. Here, the encryption scheme E can also be a symmetric-key scheme, provided it is key-private. We can simply let $k^{\mathsf{Pub}} = k^{\mathsf{Pri}}$ and $s = v$ to achieve this. This would yield private group communication with few key exchanges. However, we use the notation of public-key cryptography in our abstraction. This makes the push protocol modular as well.

Before we continue our analysis of what Eve can do against the push protocol, let us first look back at the pull protocol. Imagine that Eve gets access to a publication oracle for an instance of the pull protocol, and remember that she controls the entire file system. It is not hard to convince ourselves that Eve cannot learn anything more from the file system than she can from the ciphertext alone (when playing the ANO-IND-CCA game): each publication just appends the new ciphertext to the same file in the file system — no matter how she changes the recipient set.

Let us now look at what Eve can do in the push protocol. Here she can actually learn information by observing the file system. Whenever a message is published she learns which inboxes are used to get messages from the publisher, thus she can relate several inboxes. Hence, the security analysis of this protocol is only interesting when the protocol is run in parallel.

5.1 Running Multiple Push Instances in Parallel

There is one issue we must consider before continuing our discussion of the security of the push protocol. For the security discussion to make sense we must first, as for the pull protocol, convince ourselves that Eve cannot distinguish between parallel instances of the push protocol. Thereafter we can continue our discussion of the security of the push protocol run in parallel.

We will give an information-theoretic argument for the indistinguishability. Each encryption in Fig. 6 depends on the following: the public key of the recipient, the signature key for the authentication, and the message. Both the public key and the signature key is unique per recipient and instance. It follows that two encryptions of the same message are equally likely done within the same instance as in two different instances. This follows from the security of the encryption scheme E. Thus, Eve's only chance is to decide from the ciphertexts that they contain the same message, but since E provides IND-CCA security this is possible only with negligible advantage.

Now we know that it makes sense to talk of the security when running parallel instances. Assume that we run n instances of the push protocol in parallel. It is not too difficult for Eve to distinguish which inboxes are related. Eventually one instance will publish when no other instance is publishing. Then that instance can be distinguished from others when publishing at the same time, hence Eve can learn more and more over time — even without using anything like the publication oracle mentioned earlier. To deal with this problem, we will introduce a mix-net.

Definition 7 (Mix-Net). *Let* $m_1, \dots, m_k$ *be messages from senders* $1, \dots, k$. *A mix-net is a functionality* $\mathcal{M}$ *such that on inputs* $m_1, \dots, m_k$ *the outputs* $\mathcal{M}[1](m_1), \dots, \mathcal{M}[k](m_k)$ *are unlinkable to its senders. More specifically* $\Pr[i \mid \mathcal{M}[i](m_i)] = \Pr[i] = \frac{1}{k}$.

We note that by definition the mix-net will not output anything until the last input slot has received input.

We will now assume that the calls to $\mathcal{FS}$ in the push protocol (Fig. 6) are replaced with $\mathcal{M}_w[\cdot](\mathcal{FS}.append(\cdot,\cdot))$ and $\mathcal{M}_r[\cdot](\mathcal{FS}.read(\cdot))$, where $\mathcal{M}_w$ and $\mathcal{M}_r$ are two mix-nets for writing and reading operations, respectively. The input slots are unique to each instance of the push protocol, instance i has input slot $\mathcal{M}[i](\cdot)$.

Remember that the inbox identifier is randomly chosen, this means that the probability for an inbox being used by two instances is low ($\approx 1/2^{\lambda/2}$). Without the mix-net Eve can also distinguish which inboxes are related by looking at the distribution of messages in the files in the file system. It is unlikely in practice that the behaviour of all publishers will be uniform, so Eve will eventually learn the related inboxes. However, due to the assumed mix-net, it will enforce a uniform number of messages, n messages in yield n messages out. There are mix-nets that add dummy messages to improve throughput — which will be needed for efficiency reasons — but we refer to that literature for a discussion on its security.

Another technique to distort the distribution is that each subscriber can reuse the inbox across instances for different publishers — this is actually desirable for the push protocol also for efficiency reasons, the subscriber has only *one inbox for all* publishers. However, this allows the publisher to determine that a subscriber subscribes to other publishers, i.e. a scenario related to Jealous Bob, but this time a Jealous Alice. This in turn can be solved by adding noise to each inbox, thus making real messages indistinguishable from noise. A more efficient solution would be for Bob to share an inbox with an *unknown* other subscriber. Bob can pick an inbox which already contains messages and use the same identifier. This would make fetching new messages less efficient, but will reduce the amount of noise needed.

Finally we note that if Eve did not control the entire file system, she would not be able to relate inboxes that are not in the part she controls. She would be able to read them, but the accuracy of her timing attacks would be reduced to how often she could read the files to detect change. She could still do the message distribution attacks though. This change of the adversary takes us from a potentially centralized setting to a distributed setting where there is no network wide adversary, but an adversary controlling only a part of the network of the file system nodes.

6 Algorithmic Complexity

We will now summarize the algorithmic complexity for our constructions. The performance is interesting to evaluate from two perspectives: the publisher's (Alice in all examples) and the subscriber's (Bob in all examples). From the publisher's perspective, it is interesting to investigate the needed space for key storage, communication complexity for publication and time complexity for encryption of new material. From the subscriber's perspective, the complexity of key-storage size and the time-complexity of aggregating the newest published messages are the most interesting aspects. An overview of the results is presented in Table 1.

Table 1. The storage, communication and time complexities in the two models. S is the set of all subscribers, R is the set of recipients of a message. All values are $O(\cdot)$.

Publisher	**Pull**	**Push**	**Subscriber**	**Pull**	**Push**
Key-storage size	$\|S\|$	$\|S\|$	Key-storage size	1	1
Ciphertext size	$\|R\|$	$\|R\|$	Ciphertext size	$\|R\|$	1
Encryption	1	$\|R\|$	Decryption	1	1
Communication	1	$\|R\|$	Communication	1	1

The space complexity for the key management is the same for both the pull and push protocols. If we have $|S|$ subscribers, then we need to exchange and store $O(|S|)$ keys: we need one public key per friend.

The space complexity for the ciphertexts are $O(|R|)$ for both models. However, the pull protocol is slightly more space efficient since we need less signatures. In the push protocol we require one signature per ciphertext, i.e. $O(|R|)$, whereas for the pull protocol we only need one per message.

The time complexity for encryption depends on the underlying schemes. But we can see that in the push protocol we get a factor $|R|$ to that of the encryption scheme, whereas we have a constant factor in the pull protocol. The time complexity for decryption on the other hand has a constant factor for both models.

Finally, we look at the communication complexity for the different protocols, which differ slightly. If we look at one single instance, then we get a constant number of connections for the subscribers. However, most subscribers will have to pull from several publishers, this is the argued benefit of push model of communication — there is only one inbox to read.

7 Related Work

Harbach et al. [7] identified three desirable and (conjectured sufficient) properties for a privacy-preserving AC mechanism: hidden policies, hidden credentials, and hidden decisions. The work in [7] focused on fully homomorphic encryption and is thus not directly feasible for our purposes. However, the properties are still relevant in our setting.

Hidden policies means that the access policy remains hidden from anyone but the owner and the subjects learn at most if they have access or not. This is the same definition as we used above, and we arguably achieve this property in both of our constructions. Furthermore, we also achieve what we call 'hidden policy-updates' as well (Jealous Bob), which prevents previously-authorized subjects from determining whether they are no longer authorized to access newer versions.

Hidden credentials means that the subject never has to reveal the access credentials to anyone. In our case this is a cryptographic key, and as a consequence we allow the subject to anonymously read the ciphertext from the storage node. This means that the storage node cannot track which subjects are requesting access to which objects.

Hidden decisions means that no-one but the subject must learn the outcome of an access request. This means that no-one should learn whether or not a subject could decrypt the ciphertext or not. However, if everyone only requests ciphertexts that they know they can decrypt (which is the most efficient strategy), then the storage operator can easily guess the decision. Most constructions probably suffer from this, including ours. This decision together with non-anonymized users would allow the storage operator to infer parts of the policy, hence breaking the hidden policy property. To prevent this subjects could also request ciphertexts they cannot decrypt (dummy requests of PIR). However, anonymous requests makes this a less relevant problem.

There is also related work in the DOSN community. There are several proposals available for DOSNs, e.g. DECENT [8], Cachet [10] and Persona [2]. The AC mechanisms in these proposals focus on providing confidentiality for the data. E.g. Persona uses KP-ABE to implement the AC mechanism and unfortunately, this yields lacking privacy: as this is not policy-hiding, anyone can read the AC policies and see who may access what data. There are also general cryptographic AC schemes that focus on achieving policy-hiding ciphertexts, see the section of related work in [7]. E.g. Bodriagov et al. [4] adapted PE for the AC mechanism in DOSNs. Works in this area that have employed policy-hiding schemes for DOSNs have also focused on solving the problem of re-encryption of old data upon group changes. We do not solve this problem, but rather contribute the insight that it would violate our desired privacy properties. So besides being more efficient in some cases, less efficient in others, they do not require the same properties.

8 Conclusions

We achieve privacy-preserving access control enforcement in a public file system lacking built-in access control. We presented two alternatives: the pull and the push protocols. Both implements the model of a publisher distributing a message to a set of subscribers.

The pull protocol achieves strong privacy properties. It essentially inherits the ANO-IND-CCA property of its underlying BE scheme. The subscribers cannot learn who the other subscribers are even if they control the network of the file system or the whole file system. Further, if the publisher wishes to exclude any of the subscribers from future publications, then all subscribers learn that there was a policy update but no one learns what changed — not even those excluded!

The push model for communication is an interesting case. Conceptually, the only difference between the push and pull models is that we distribute the message, instead of everyone fetching it. This seems to yield better privacy properties at first glance, but it turns out that we had to make considerable changes before we could get any security for the push protocol. Hence the security guarantees are much weaker, we have to make some trade-offs. One alternative is that the adversary can only control a part of the file system, or monitor only part of the network, this would make Eve's timing attacks more difficult. Eve can still compute the distributions of messages over the inboxes and relate them, so we need some techniques to make this estimation more difficult. But it is difficult to achieve any guarantees against the distribution attacks, we have to resolve to techniques like differential privacy. However, there are benefits: unlike in the pull protocol, when the subscriber wants to make a policy update, none of the subscribers will even be notified that there has been a policy update — let alone determine if they have been excluded.

As was pointed out in Sect. 7, we do not treat group management (i.e. revoking the credentials for subjects) as is done in other schemes. If the publisher excludes a subscriber, it is only from future publications — not from past! In fact, we can conclude from our treatment above that such functionality would actually violate the privacy properties. Even if possible, any subject could have made a copy anyway, i.e. it is the problem of removing something from the Web. However, other group changes are easily done, the most expensive one is to give a new subscriber access to old publications — this requires re-encrypting or resending all publications.

An interesting future direction would be to explore Eve's limitations in the push protocol in more detail. For example, under what conditions can Eve estimate the message distributions over the inboxes, according to what distributions should the subscribers add noise? This would help us design an efficient scheme that can distort the distributions and guarantee security.

Both protocols are modular enough to provide better deniability, e.g. using a MAC scheme for authentication of messages would remove the non-repudiation property. To let all subscribers, and not only the publisher, publish messages. Another interesting direction would be the opposite: stronger accountability. An example of desired accountability would be that Bob wants to verify that Carol received the same message, if Alice told him that she sent a copy to Carol as well. Due to the privacy properties this is not possible in the current protocols.

Acknowledgements. This work was inspired by some work with Benjamin Greschbach and work of Oleksandr Bodriagov and Gunnar Kreitz. We would like to thank the Swedish Foundation for Strategic Research for grant SSF FFL09-0086 and the Swedish Research Council for grant VR 2009-3793, which funded this work. We would also like to thank the anonymous reviewers and especially Anja Lehmann for valuable feedback.

References

1. Abdalla, M., Bellare, M., Neven, G.: Robust encryption. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 480–497. Springer, Heidelberg (2010)
2. Baden, R., Bender, A., Spring, N., Bhattacharjee, B., Starin, D.: Persona: an online social network with user-defined privacy. In: SIGCOMM (2009)
3. Bellare, M., Boldyreva, A., Desai, A., Pointcheval, D.: Key-privacy in public-key encryption. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, p. 566. Springer, Heidelberg (2001)
4. Bodriagov, O., Kreitz, G., Buchegger, S.: Access control in decentralized online social networks: applying a policy-hiding cryptographic scheme and evaluating its performance. PERCOM Workshops **2014**, 622–628 (2014)
5. Dingledine, R., Mathewson, N., Syverson, P.F.: Tor: the second-generation onion router. In: USENIX Security Symposium, pp. 303–320 (2004)
6. Fiat, A., Naor, M.: Broadcast encryption. CRYPTO 1993. LNCS, vol. 773, pp. 480–491. Springer, Heidelberg (1994)
7. Harbach, M., Fahl, S., Brenner, M., Muders, T., Smith, M.: Towards privacy preserving access control with hidden policies, hidden credentials and hidden decisions. In: Privacy, Security and Trust (PST), pp. 17–24 (2012)
8. Jahid, S., Nilizadeh, S., Mittal, P., Borisov, N., Kapadia, A.: DECENT: a decentralized architecture for enforcing privacy in online social networks. PERCOM Workshops **2012**, 326–332 (2012)
9. Libert, B., Paterson, K.G., Quaglia, E.A.: Anonymous broadcast encryption: adaptive security and efficient constructions in the standard model. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) PKC 2012. LNCS, vol. 7293, pp. 206–224. Springer, Heidelberg (2012)
10. Nilizadeh, S., Jahid, S., Mittal, P., Borisov, N., Kapadia, A.: Cachet: a decentralized architecture for privacy preserving social networking with caching. In: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, pp. 337–348 (2012)