Two-factor authentication for trusted third party free dispersed storage

Esiner, Ertem; Datta, Anwitaman

2018

Esiner, E., & Datta, A. (2019). Two-factor authentication for trusted third party free dispersed storage. *Future Generation Computer Systems*, 90, 291-306.
doi:10.1016/j.future.2018.08.001

<https://hdl.handle.net/10356/104249>

<https://doi.org/10.1016/j.future.2018.08.001>

© 2018 Elsevier B.V. All rights reserved. This paper was published in *Future Generation Computer Systems* and is made available with permission of Elsevier B.V.

Downloaded on 26 Apr 2023 17:52:08 SGT

Two-factor authentication for trusted third party free dispersed storage

Ertem Esiner, Anwitaman Datta

*Nanyang Technological University, 50 Nanyang Ave, 639798
Singapore*

Abstract

We propose a trusted third party free protocol for secure (in terms of content access, manipulation, and confidentiality) data storage and multi-user collaboration data dispersal, encryption as well as two-factor (knowledge and possession) based authentication and access control techniques so that unauthorized or a small set of colluding servers cannot gain access to the stored data. The protocol design takes into account usability issues as opposed to the closest prior work [1]. We explore the security implications of the proposed with event tree analysis and report on experiment results to demonstrate the practicality of the approach computational overheads. Given that the protocol does not rely on any trusted third party, and most operations including actual collaboration do not require users to be online simultaneously, it is suitable not only for traditional multi-cloud setups but also for edge/fog computing environments.

Keywords: Layered security, two-factor access control, data out-sourcing, edge computing, user controlled encryption, erasure codes

2017 MSC: 00-01, 99-00

1. Introduction

The overall premise of this work is that an end user wants to store data over autonomous storage service providers in a dispersed manner in order to ensure that neither any of

Email addresses: ertem001@ntu.edu.sg (Ertem Esiner), anwitaman@ntu.edu.sg (Anwitaman Datta)

these individual storage service providers nor any other unauthorized third party is
5 able to gain access to the stored authentication for legitimate access to the data. Such
a solution is desirable while storing data over untrusted storage services — an issue
that has gained considerable attention with the deployment of cloud services [2], and
the problems associated with untrusted storage are expected to be exacerbated in an
edge/fog computing set-up.

10 In the presented work, confidentiality of content from individual service providers
is achieved by augmentation of encryption with the dispersal of the content across mul-
tiple services — a technique which has been explored by several other works in recent
years [3, 4]. But in contrast to such existing works, our approach achieves an additional
layer of security by facilitating the data owner to autonomously set up a authentication
15 with each such service provider, without relying on any (trusted) third-party assistance.
A protocol to realize orthogonal factors (of knowledge and possession) in a decentral-
ized set-up without any dependence on any trusted third party is the main novelty of
this work.

The avoidance of a trusted third party has both security as well as usability ben-
20 efits. Specifically, in terms of usability, neither the storage service providers nor the
data owner need connectivity with such a third party facilitator — which makes our
approach particularly suitable for edge computing environments — even though it can
also be applied to (and improves upon existing) traditional multi-cloud storage setups
[3, 4, 5, 6]. In terms of security, it eliminates a single point of failure. Otherwise, the
25 trusted third party could be compromised — say, either by a hack, insider attack, or a
subpoena — to subvert the multi-factor access control.

Further to the ‘hard security’ benefits, our approach also (optionally) provides op-
portunities of additional obscurity — e.g., an end user can choose arbitrary set of stor-
age service providers, which are not readily identified by anyone (in contrast, a trusted
30 third party would have had by default the knowledge of the storage service providers
being used by a user).

In contrast to some recent works and our own prior work closest to this work [7, 1,
8], we disentangle the possession factor for authentication from the clients a user uses
to access the data. The user can use any client device to access the data, without the

35 client device needing access to the secret (knowledge factor) information that is used
to generate the possession factor based one time passwords. So, any device can be used
as a client. Furthermore, in case a client machine is under the control of an adversary,
it still cannot be used by the adversary to re-authenticate subsequently. We, however,
note that such a compromised client will give the adversary access to fetch any data
40 during the session that the user has authenticated himself, and thus, provide protection
against compromised clients is not achieved in our approach.

The rest of this paper is organized as follows. We introduce the background as
well as the related works in Section 2. In Section 3, we first present the setup where
decentralized access control is achieved for just a data owner, then discuss extensions
45 on how collaboration with other users is supported. Section 4 consists of a variation of
event tree analysis to explore the security/limitations of the proposed design. Finally,
we present the performance evaluation of our scheme in Section 5 to demonstrate that
the proposed approach is practical. We draw our conclusions in Section 6.

2. Background and Related Work

50 Our focus is on confidentiality of data/content stored at third party storage services.
Encryption is the obvious and immediate option to achieve this goal. There are ser-
vices that natively handle encryption [10, 11], while a user can also otherwise use tools
such as TrueCrypt [12] or Boxcryptor [13]. However, the eventual protection achieved
by the end user (where the adversary could be a third party, but even the storage ser-
vice provider) in such approaches is hinged on the confidentiality of (a password to
55 generate) the secret encryption key.

In this work, we explore a portable technique to achieve layered protection and two
factor authentication, but without the need of any trusted facilitator.

In the rest of this section, we broadly classify the related work across four topics
60 which are directly relevant to the proposed approach: (i) factors of authentication, (ii)
multi-server storage, (iii) proxy re-encryption and (iv) homomorphic hash function.

2.1. Factors of Authentication

There are many ways to realize multi-factor authentication [2]. The first and the most insecure one is to ask the user a shared secret, something both the server and user
65 knows (e.g. prompting images for the user to choose a pre-decided one). The second is to require a token, which is a physical device that a user has (e.g. a USB token, smart card or a password generating token). The third is based on what the user is (e.g. biometric information such as fingerprints, iris). While all three can be added, mostly one of the above three are chosen as a second factor of authentication.

70 Most previous works realize such an additional factor by trusting a third party in some manner. e.g., using a smart card issued by the server [14], a mobile phone application where the server knows all shared information to generate the information needed for verification [15].

In our recent work [1] we proposed a mechanism where the client disperses the data
75 across multiple storage services and generates a second factor of authentication which is unique for each server autonomously, without the help of any trusted third party. Since the second factor information is unique to each server, but none of the individual servers have the whole data (recall that it is dispersed), it afforded a trusted-third party free authentication. The main drawback of that approach, however, was that the client
80 needed to store (or access all the information, e.g. using a USB storage medium) locally - which diminished the portability and hence usability of the approach.

In this work, we propose the first password generating token (we call it an authenticator in the rest of the work) that works in parallel with the client device and that doesn't require a trusted party (neither the server nor any trusted third party). This au-
85 thenticator could be a stand alone device, or something like a smart phone/watch that the user anyway typically has.

Since the client device does not need to store or process the secret directly, unlike the previous work, it allows portability — and one can use even untrusted client devices for access, without compromising the secrets. Overall, the protocol design in this work
90 is totally new and distinct from our prior work [1], even though the high-level objective and the high-level general idea of dispersing the content across multiple storage service

providers are same.

Specifically, in our approach, the authenticator helps generate time dependent one-time passwords.

95 There are two main ways to realize a password generating token. The first is to use a hardware token manufactured by a trusted organization, e.g., RSA's SecureID. The second is to run a software based solution such as Google's one-time password (OTP) [16].

In this case, our aim is to allow users to only trust their devices, and to let them use
100 a computing device they already have and trust, thus we suggest a software based solution. We follow the same principle to generate a password that depends on time (Time Based One-Time Password Algorithm), which has been defined in IETF's RFC 6238 [17] — a dedicated computing device that generates the secret randomly after the user has it in his possession. While one may argue that the software to be used by the authenticator itself may be compromised, we will like to note that a open-source software
105 can be investigated and audited by many, thus making this approach practicable.

Though not vital to our design (since there is also a knowledge factor involved, as we will elaborate in the discussion of the protocol), we will like to note that most smart phones provide their own security mechanisms to prevent illegitimate access.
110 Thus, even if an adversary gets hold of the device used to realize the authenticator, the adversary may not necessarily be able to use it (depending on how the user has configured the device in general).

2.2. Multi-server Storage

Dispersal of data with redundancy (e.g., using erasure codes) across independent storage services has been explored for the purposes of both fault-tolerance [5, 4, 3] and
115 confidentiality [3, 18, 19]. Our approach follows the same basic design as [3, 18] in terms of the application of encryption at the client device and dispersal of data to the servers.

The main difference of this work with respect to all the others is in providing access
120 control in a decentralized and portable manner. We present an orthogonal possession

factor (added to the knowledge factor, namely, the password) enabling us to create time bounded authentication tokens.

Furthermore, Li et al.’s approach [19] aims data security with deduplication as an added functionality, by augmenting a secret sharing scheme called convergent dispersal. It is worth noting that in our work, we do apply erasure coding to the data. however, we leave the inherent security aspect of the dispersal techniques (as explored in [19]) beyond the scope of this work.

The relationship between multiple servers in [18] is somehow recondite. To allow a user to share his data with another person, their approach requires the servers to cooperate, yet the claimed security partly depends on the distribution of data across independent servers. In contrast, we facilitate sharing the data with other users without requiring the storage services to coordinate among themselves.

2.3. Proxy Re-encryption

Proxy re-encryption (PRE) allows multiple parties to collaborate over encrypted data without having to share among themselves a common secret used for encryption/decryption.

A proxy server can transform a data encrypted for one party, to the one of another without decrypting it, therefore the data is not only readable by the data owner, but by the collaborators as well. The proxy re-encryption schemes are built on computationally expensive primitives (such as bilinear maps), hence performs much slower (nearly 4 order of magnitude slower) than symmetric encryption schemes - we have compared the bilinear scheme implementation [20] that realizes the idea from [21] vs the of OpenSSL library [22] on a portable computer with a dual-core Intel Core i5-4288U @ 2.60GHz with 3MB shared L3 cache CPU, 8GB RAM, 512GB SSD (with average 700MB/s read 600MB/s write speeds) and running Ubuntu 15.10 Wily. For encryption, AES takes 3ms, 6ms and 3 seconds for 256 bytes, 1MB and 1GB data respectively, PRE throughput is only 256 bytes/8ms. The disparity is similar with decryption, AES takes 3ms, 5ms and 1.5 seconds for the same data set, while PRE throughput is only 256 bytes/20ms. It may seem peculiar at first glance, but AES decryption is faster than its

150 encryption since the decryption is parallelized, but the encryption is not parallelizable
in CBC mode. While the precise performance figures are artefacts of many factors,
including the specific implementation used, which may be further optimized, it never-
theless illustrates how applying current realizations of PRE on the actual bulk of the
data would not be practical, and is thus best limited to encrypt only meta-content such
155 as the symmetric encryption keys and metadata.

This disparity of computational overheads of the two classes of encryption also
suggests that in all cases, for the approach to be practical, the actual content (which is a
huge volume of data) should be encrypted using symmetric encryption while deploying
PRE for meta-information (e.g., symmetric encryption key of data subsets, random salt,
160 checksum).

The first PRE scheme was proposed by Mambo and Okamoto [23], later followed
by [24, 25], where the proxy functions and their respective security guarantees are de-
fined. There are two types of proxy re-encryption schemes; unidirectional and bidirec-
tional. A PRE scheme is unidirectional if the scheme allows one-way re-encryption,
165 and bidirectional, if the re-encryption works both ways. Moreover, a PRE scheme
is said to be of multi-use if a re-encrypted cipher can be re-encrypted again to be
decrypted under a third private key. A canonical example of multi-use bidirectional
scheme is Blaze et al.’s El Gamal based scheme [24]. Other examples are [26, 27].
Further works patch certain security flaws of the original scheme, such as, when the
170 proxy server and another party collude to reveal the owner’s secret [26], chosen cipher-
text attack (cca-security) [28, 27, 29].

Ateniese et al. applied PRE to secure storage systems [21], where collaboration
and sharing of the dispersed (by erasure coding) data is the main focus. The protocol
employs a unidirectional PRE and requires a centralized party though, therefore we
175 follow [30] where the setting is fully distributed. We note that using PRE for data
sharing is not a contribution of this work per se, but we make use of it to enhance our
authentication scheme and support collaboration.

2.4. Homomorphic Hash Function

A hash function is a function that takes an arbitrary sized input and compresses it into a fixed length string [31]. In this work, collision resistance property of a cryptographic
180 hash function is crucial. A hash function is collision resistant if there is no polynomial time adversary that can find a collision in it [31], where a collision means that two different inputs result in the same output. Lastly, for our scheme, we need the hash function to support homomorphism. Homomorphism of a function allows com-
185 putations to be carried out over the outputs of the function, where a corresponding operation on the inputs would result with the computed output when the function is applied. Mathematically, a function F is homomorphic if for the message space M , $\forall x, y \in M$, there exists operations $\odot$ and $\oplus$, such that $F(x \odot y) = F(x) \oplus F(y)$ [31]. Specifically, we use the Pederson commitment scheme [32] and the work by Chaum et
190 al. [33] as the basis, in a manner similar to Krohn et al.'s usage [34].

3.

3.1. System and trust model

We consider the following setup. Figure 1 represents the various entities involved in the protocol. Table 1 summarizes the notations used throughout this work. Foremost,
195 we have a **user** (data owner). Later in Section 3.6, when we discuss collaboration, then the other users are called the collaborators. In our proposed system, we assume that all the users collaborating on a shared space trust each other with the data, and grant access to each other.

A user *knows* two secrets (namely, e and pw), and *possesses* a portable device which
200 supports storage and computation (e.g., a smart phone), and is used as an **authenticator**. The users don't trust anyone with these secrets and the authenticator device. It is possible and best to keep the authenticator device offline at all times. Or more reasonably, the application can be run in an isolated environment where no access is allowed in and out. The authenticator generates a one-time password upon password input of
205 pw by the user.

The user accesses the storage service using a **client** device which runs a publicly available client software (e.g. a publicly audited and signed client software possibly in the form of a browser plugin). A client device, however, is to be trusted only during the setup phase and does not need to store any secret essential for the specific user to
210 authenticate himself with the **storage service providers (Servers)** indicated as S_i with ($i = 1 \dots n$). The servers are assumed to be able to carry out arbitrary precision arithmetic calculations, and enforce access control based on the authentication challenges, in addition to providing the actual storage. The servers are partially trusted (curious but honest), in the sense that they will keep the data intact and follow the protocol when
215 queried (even though these are important concerns, they are addressed in a different lines of work such as proof of data possession [35, 36, 37], and are beyond the scope of the current work) but layered security is required in case they are compromised by or colludes with adversaries that are after access to the data content [38].

Lastly, the client is assumed to have access to a directory service that is identified
220 by the user. The directory service can be hosted on a public server, or a private server operated by the user, or even be run on the same device that is acting as the authenticator (even though these are logically distinct functionalities, and concealment of the servers' information provides only obscurity and not hard security guarantees, and thus there is no specific security threat for the proposed protocol even in case this information
225 is in fact in the public domain), and is used solely for the purpose of identifying and contacting the servers S_i . In case the directory service is malicious or infected, then a denial of service can be performed. However, a denial of service attack does not result with an unauthorized access to the data. Furthermore, and given that the information stored in the directory service does not contribute to the hard confidentiality guarantees
230 achieved by the proposed scheme (though, it may add a layer of obfuscation) there are many ways to achieve a reliable directory service, for instance, using a standard cloud based service, or using a verifiable directory service [39], or by storing the relevant information piggybacked on a public blockchain [40].

In our consideration of attack scenarios, we also consider that parts of the commu-
235 nication between a client and the servers may be compromised (specifically, an attacker may try to carry out variations of man-in-the-middle attacks) and demonstrate that the

proposed protocol is robust against the same.

Overall, the design of the protocol is based on the pragmatism that there are many possible attack vectors, thus there is a need to eliminate single point of failure, which is realized by decreasing the need for trusted elements in the system, and instead de-
 240 ploying security in layers. Accordingly, in Section 4, we discuss how a breach at any step is covered with other layers of security.

Figure 1: Interacting entities.

3.2. Two factors of access control

We propose a design that enforces access control over two factors; knowledge and possession. It does so by letting the user set up the system using a trusted computing device under user's possession (in our prototype implementation we have used an
 245 Android based smart phone, yet it can be any device such as a smart watch or even a stand-alone device dedicated for being the authenticator). The main novelty is that the device generates (offline) its own secret which is used by the user to setup and subsequently authenticate with the storage servers. There is no central trusted entity that
 250 coordinates the possession factor. Dispersal of the stored data across multiple entities, in addition to the application of encryption over the dispersed data, ensures that individual storage servers cannot access the content even if they act in a Byzantine manner and deviate from what is expected of well-behaved storage servers.

255 The possession factor is a randomized secret key sk that is locally generated and

kept at the authenticator. Therefore, the authenticator is extremely portable since moving the secret key from one device running authenticator software to another is enough to create another authenticator with the same capabilities. This is identical to the prior work [1]. Yet, in contrast to the prior work [1], the client device with which the user carries out the authentication and data access does not need to store the secret key sk , and instead the decoupled authenticator produces information that can be keyed in on the client device — which provides usability at parity with existing centralized solutions where the OTP generation is decoupled from the client accessing a service.

3.3. Homomorphic Hash Function

For our protocol, we need a hash function (*hhash*) with the associative property, i.e., for a hash function H and an operation $\odot$, $(H(a) \odot H(b)) \odot H(c) = H(a) \odot (H(b) \odot H(c))$. A naive example of such a hash function may be to choose a random number r and the hash value of the given input i is r^i . Then, the $\odot$ operation is a simple multiplication where the ordering of operations does not affect the final result. We employ a homomorphic hash function that leverages on this mathematical property [34], while ensuring desirable cryptographic properties - namely, it provides collision resistance [41] and is secure under the assumption that discrete log is computationally hard to compute. For details about its security properties, please refer to [34].

hhash parameters: First of all, the set of hash parameters $G = (p, q, g)$ are generated. p and q are large prime numbers, where q divides $p - 1$. g is a vector of random elements from $\mathbb{Z}_p$, all order q , which means $g_i^q \bmod p = 1$ but $g_i^j \bmod p \neq 1$ for any $0 < j < q$. When creating the hash function to be applied to input x (as in Equation 1) the length of vector g should at least be the length of the input, i.e., $|x|$.

hhash generation: For an input x , represented as a vector where x_i is the i^{th} element, its hash according to G is computed as follows.

$$hhash_G(x) = \prod_{i=1}^{|x|} g_i^{x_i} \bmod p \quad (1)$$

hhash combine function: We combine two hash values by multiplying them,

where $h_x = \text{hhash}_G(x)$, $h_y = \text{hhash}_G(y)$.

$$h_x \odot h_y = h_x \times h_y \mod p \quad (2)$$

Table 1: Notations.

Notation	Description
sk	the secret key for authentication (which forms the possession factor)
sek	the secret key for data encryption
e	memorized encryption password
pw	memorized authentication password
hhash	homomorphic hash function
p, q	two primes, where q divides $p - 1$
g	vector of random elements from Z_p , all order q
$G = (p, q, g)$	hash parameters
OWF	a One Way Function
OTP	a One-Time Password
AT	an Authentication Token

3.4. Setup

3.4.1. Authentication Setup

285 **Protocol:** In Figure 2, we show how the protocol works, step by step. First, the user sets the servers up. For that, the user needs to generate the authentication tokens (AT). To initialize this process the user generates the registration code h_{sk} by entering the password pw to the authenticator (Step 1).

$$h_{sk} = \text{hhash}(sk) \odot \text{hhash}(pw) \quad (3)$$

290 The registration code is to be differentiated per server (to form distinct and reasonably unrelated ATs) so that an adversary that can compromise one server does not get access to the others by default. This can be done at the authenticator but making the

user key a code in per server would conflict with the usability constraint. Therefore, the differentiation is to be done at the client. In this case, the registration code is used at the client device during the setup.

295 For differentiation, the client then generates a salt per server. We don't impose any security role over the selection of the salt values, but to decrease their predictability, a salt value can simply be calculated employing a Given a one-way function OWF , and let “|” be the concatenation operation;

$$salt_i = OWF(h_{sk}|i) \quad (4)$$

Next, the distinct authentication tokens (AT_i for Server i) are generated (Step 3).

$$AT_i = h_{sk} \odot hhash(salt_i) \quad (5)$$

300 As a side note, one can choose to do the above at the authenticator and don't expose the registration code at the client device even at the setup phase. But if the client is compromised during the setup phase, not the registration code itself but all other values that have been generated from it are exposed anyway. If the same authenticator with the same key is to be used for multiple setups, the added security may be desirable, yet
305 setting up a new key per setup makes more sense in any case.

The client at this step creates a directory service on a user-owned server and sets it up by sending (Step 4) chosen server IPs and an authentication token so that the directory service can verify a later authentication request in the same manner as the servers do. Alternatively, a shared directory service may be queried for the addresses
310 of the prospective storage nodes (Alternative Step 4). We leave it as an implementation decision. Next, the client uploads all authentication tokens to their respective servers (Step 5). Each server stores the authentication token for future access control.

3.4.2. Directory Service

The scheme enables any device that can run the client software to act as the client
315 without any secret information stored. Some metadata about the files are naturally required thus we suggest employing a directory service. The 4th and 9th steps in Figure

Figure 2: Authenticator in use.

2 are when the directory service is queried. During the registration phase (precisely, 3rd step for registration and 8th step for authentication), the client can simply act like the directory service is yet another server and create an authentication token for it as well. Both during registration and authentication, the client communicates with the directory service asking for the meta-data (that is the file name hierarchy) and the server addresses. Each time the service is required exactly after the authentication tokens are created by the client, thus no extra iteration is needed for using the same authentication protocol between the client and the directory service.

3.4.3. File Setup

When a user needs to store an object O , the client splits the object into m parts and applies a maximum distance separable erasure code such as Reed-Solomon code [43] to generate n encoded parts. These n parts are then encrypted using distinct symmetric encryption keys sek_i to generate encrypted data d_i which are shipped to respective servers S_i . Per server symmetric encryption key creation is described in Equation 6.

By differentiating the encryption keys per server, the client ensures if an adversary successfully guesses the key or breaks the encryption used for a piece stored at a specific server, the rest of the pieces are still protected.

$$sek_i = hash(e|salt_i) \quad (6)$$

We opted to bind the encryption key with the possession factor as shown below.

335 Hence, the randomness of the encryption key is strengthened compared to a memorizable, therefore relatively simple, password. As a consequence, guessing becomes an impractical option for an adversary. Note that the h_{sk} is not exposed during the decryption phase due to its being masked by the memorized encryption password e . It is also worth noting that the decryption phase follows the authentication phase. The authenticator can simply keep the calculated registration code h_{sk} until a timeout and
340 calculate the encryption password e_a as below, which then is to be used to calculate the encryption key sek_i .

$$e_a = hhash(h_{sk}|e) \quad (7)$$

$$sek_i = hash(e_a|salt_i) \quad (8)$$

We have employed 128 bit AES in CBC mode (salt enabled) as the encryption method in our experiments, yet the encryption technique and mode can easily be changed
345 according to the needs of the users. The meta information of the encrypted data (we call it “encryption meta-data” in the following sections) is to be kept at the server as well, for which we discuss the details in Section 3.6.1 and subsequent sections, along with collaboration.

3.5. Authentication

350 To retrieve her data back from the storage servers, the user first uses the authenticator to generate a one-time password. Even though it conflicts with usability, the user may furthermore choose to keep this secret to generate OTP encrypted. If she chooses to do so, then she would need to key in another password (not shown in our steps) to first

decrypt the secret required, then to generate the OTP, otherwise the authenticator just
 355 generates an OTP upon entry of the password pw (Step 6). Then, the user keys in the
 OTP to the client (Step 7). As far as the design of our protocol is considered, the OTP
 thus generated may be of arbitrary length, however, taking usability into consideration,
 we choose the range for it to constitute of 6 numbers or up to 16 printable characters
 (as discussed in Section 3.5.1 and Section 5).

$$OTP_a = h_{sk} \odot hhash(timestamp) \quad (9)$$

360 Finally, the client, adding the salts (as discussed in Section 3.4.1), generates the
 one-time passwords per server (Step 8). Note that unlike the setup phase, differentiating
 authentication tokens at the authenticator wouldn't add any value this time, since during
 the authentication phase the secret h_{sk} is not exposed. If the client doesn't possess the
 address of the servers (not in the cache for instance), then first it queries the directory
 365 service (Step 9) and creates the one-time passwords. In this step, the one-time password
 can be extended with file IDs of the parts queried by simply concatenating the IDs to
 one of the hash inputs, so that the same OTP cannot be used for different data pieces.

$$OTP_c = OTP_a \odot hhash(salt_i) \quad (10)$$

3.5.1. Authenticator details

Given the $hhash$ function, now we construct our authenticator as the second factor
 370 of authentication. In this construction (shown in Figure 2) the authenticator device
 keeps a secret key (sk) and the device should possess nothing but the processing power
 to compute the hash value once it is initialized and it should know the time (roughly
 synchronized with other devices in the system). Note that output of the authenticator is
 limited with the printable characters since the user needs to type the OTP to the client
 375 device.

Initialisation: Either the client or any trusted entity first generates hash parameters
 $G (= p, q, g)$ as in Section 3.3. After the generation, the hash parameters are public,
 and there is no need to keep them secret. The authenticator generates a secret key sk
 and stores it locally. Alternatively, the user may enter the secret key to the authenticator

380 or use a different device to generate the key and then copy it in the authenticator device.
Both the secret key and p size depend on the amount of information that can be passed
from authenticator to the client via the user.

The size of p, sk : The larger the p , the better the security, but for usability there is
a trade off. The output range (for OTP_a) depends on the choice of p due to $hhash$ and
385 $combine (\odot)$ functions. The size of sk can be bigger for sure, but since the usability
bottleneck is the length of printable characters a user can type, sk being bigger than p
doesn't make much sense.

While this is not an inherent constraint of the proposed protocol, taking into ac-
count the usability and security trade-offs, we choose 6 numeral digits for the OTP size
390 and discuss it below. We also discuss the security implications and mechanisms for
mitigation against brute-force attacks in a decentralized set-up as in ours. The subse-
quent experiment results will also make similar assumptions, even though this choice
is somewhat orthogonal to the key contributions of this paper, which is the design of
the overall protocol (rather than the parameter values with which it is instantiated).
395 Accordingly, for a 6 digit OTP, the size of the output space is 10^6 ($2^{19.9}$). We choose
the maximum for p as a close prime to this space size, therefore limiting the hash
value output space with it. Below, we discuss the implications of such a choice against
brute-force attacks. For instance;

$$p = 999983 \text{ and } q = 6329, \text{ where } q \times 158 = p - 1 \quad (11)$$

Authenticator's one-time password (OTP_a): Currently, a generic authenticator
400 output (used by banks or cloud services) is generally 4 to 6 numbers. So we construct
our scheme based on a 6 numbers output, though in fact, this is tunable according to the
security level requirement (at the expense of reduced usability, since longer sequence
will have to be typed by the user). Lets say a user can type in n characters, which are
either numbers (0 - 9) or the printable ASCII characters (between 32 and 127 of the
405 ASCII table, there are 95 of them). Therefore the output space sizes are 10^n or 95^n in
turn, which determines the upper bound for p . On the other hand, the value chosen for
 p shouldn't be a small prime which would reduce the search space for a brute-force

attack. We furthermore, explore mitigation of a brute-force attack (described below).

We show how n affects the effective key strength in Table 3.5.1. Theoretically,
 410 there is no limit in the size (thus, security level) of OTP_a , but it is limited by the
 human usability constraint. Even though the OTP is short and limited with numbers,
 the added security as a second factor may be sufficient for many use cases, given the
 key is chosen randomly. Usability wise, not using a long and complicated OTP for
 frequent transactions is desirable. The shortcoming of an easy to use OTP is that it
 415 is possible to find a verifiable OTP by brute force. In a centralized set-up, a simple
 solution to prevent brute-force attack is to limit the number of authentication trials
 per user by deactivating the account and using off-channel mechanisms (sending an
 e-mail, phone call, etc.) to re-activate it. Next we elaborate how we can mitigate brute-
 force attacks in a decentralized set-up as ours, while finding a sweet-spot for usability
 420 without compromising security.

OTP Length (n)	6_{num}	4	6	8	12	16
Effective key strength	$2^{19.9}$	$2^{26.8}$	$2^{39.4}$	$2^{52.6}$	$2^{78.8}$	$2^{105.1}$

Table 2: Deciding the size of p .

Limiting the number of unsuccessful authentication attempts: During the setup
 phase, a user may double the process by choosing 2 set of hash parameters (G_1 and
 G_2 , where $|p_{G_1}|$ close to $2^{19.9}$ and $|p_{G_2}|$ close to $2^{105.1}$) and initialise 2 keys, one small
 sized and one bigger, then prepare and send 2 authentication tokens per server accord-
 425 ingly. Former, smaller key with G_1 is to be used for frequent authentications, while the
 server will demand the latter with G_2 if wrong OTP is received consecutively (e.g. 3
 times) for a particular user, thereby limiting the wrong number of trials to brute-force
 the smaller search space, and finding a sweet-spot for security and usability. Since the
 values we store both at the authenticator and the server are quite small, we consider the
 430 overhead negligible. We further discuss it in Section 5.1.

Mapping function (Algorithm 1): This function is necessary when ASCII charac-
 ters are required for the OTP output. Due to printable characters being limited to 95,
 the output of the OTP generator should be in limited to the 95 printable characters (or

10 numeric characters for numeral output) so that the user can type the values that she
 435 sees on the authenticator device.

The output o of the $hhash$ or combine $\odot$ functions is to be generated in base 95 and be mapped to the characters accordingly as shown below, where o and OTP_a are represented as a vector where o_i is the i^{th} cell and OTP_{ai} is the i^{th} cell of OTP_a .

Data: o

Result: OTP_a

$i = 0;$

while *not at end of "o"* **do**

$OTP_{ai} = \text{cast_to_ASCII}(o_i + 31);$

$i++;$

end

Algorithm 1: OTP_a mapping

The mapping is to be reversed at the client device. For simplicity, hereinafter we
 440 do not explicitly show the mapping operation in our illustrations, and we use the term "OTP".

3.5.2. Verification

Each server stores an authentication token that is the combination of $hhash(sk)$, $hhash(pw)$ and $hhash(salt_i)$ (as in Equation 5). When a server receives a timestamped
 445 authentication token from a client (Step 10 of Figure 2), first it checks if the timestamp is within the predefined limits. Thereupon, it simply combines $hhash(timestamp)$ to the AT_i stored. If the OTP_c (as in Equation 10) is equal to the newly computed value, then the server returns the stored part of the data (Step 11). To prevent a replay attack within the time window, the server can simply keep an OTP list that expires after the
 450 time limit, and reject the OTPs that are in the list.

3.6. Collaboration

In our protocol all parts of the outsourced data (by Alice) are encrypted, yet if Bob is to collaborate he also needs to first, have access to the data, second, have the capability to

decrypt the data owned by (and encrypted under the secret of) Alice, and third, upload
455 the uploaded data back to the servers, encrypted in a manner that the original data
owner Alice or other legitimate collaborators can also read and further manipulate the
updated content. Additionally, the data owner preferably should be able to revert some
edits (more sophisticated version management may also be desirable, but we leave
that out of the scope of the current work – however, the underlying authenticated data
460 structure we use readily extends to more sophisticated version control [44]). In the
original protocol, none are possible, since it is not desirable for the data owner to just
share her secrets with a collaborator. In the following we elaborate on how Alice can
set Bob up to gain access to and decrypt the data, and update it, while also enforcing a
access control.

465 3.6.1. *Collaborating on encrypted data*

Proxy re-encryption is the key to collaborate on encrypted data [21, 30]. In a PRE
scheme, a third party can re-encrypt a cipher text that has been encrypted with one
private key into a cipher text that is encrypted with another private key and vice versa.
During the process, no part of the plain text is revealed to the third party.

470 It is worth noting that our proposed protocol doesn’t contribute on the use of proxy
re-encryption per se. Instead, the contribution in this section is on how the access
control is set-up and enforced on a distributed and collaborated data. In the next sub-
section, we give a brief definition of a generic proxy re-encryption scheme. We use the
mentioned operational primitives in the protocol.

475 For the encryption, the data owner sets a symmetric encryption key as in Section
3.4.3. As a result, there are meta-data associated per encrypted file and a key that is
needed to decrypt the data. This is where the proxy re-encryption comes in handy. The
data owner (Alice) encrypts these values with her proxy re-encryption key and stores
them concatenated to the files they are associated with.

480 3.6.2. *Proxy Re-encryption*

For the next sections, we employ a generic proxy re-encryption scheme. A proxy
re-encryption scheme between Alice and Bob consists of the following 5 algorithms;

- Key generation **KeyGen**(sr) $\rightarrow (pub_k^{PRE}, priv_k^{PRE})$, where it takes the security requirements (such as key length, protocol specifications, etc.) as input and returns a public and a private key.
- Re-encryption key generation **ReKeyGen**($priv_{k-A}^{PRE}, priv_{k-B}^{PRE}$) $\rightarrow RK_{A \leftrightarrow B}$ that takes two secrets of two parties as input and returns a re-encryption key that works both ways between Alice and Bob.
- Encryption **Enc**(pub_{k-A}^{PRE}, m) $\rightarrow d_A$ that encrypts the data with public key and returns the ciphertext.
- Re-encryption **ReEnc**($RK_{A \leftrightarrow B}, d_A$) $\rightarrow d_B$ that transforms ciphertext for Alice with the re-encryption key and outputs ciphertext for Bob.
- Decryption **Dec**($priv_k^{PRE}, d_B$) $\rightarrow m$ that takes private key and encrypted data and returns the plaintext.

3.6.3. Access control for Bob

Figure 3: Bob (with a separate authenticator) collaborating on data owned by Alice.

In Figure 3, we present both the registration and later authentication procedures for Bob. In this presentation, we assume Alice has already sent an invitation to Bob and she

is authenticated to the servers when she accepts Bob's request. Then, she informs the servers and the directory service to serve Bob if he meets the requirements. Alice via
500 her client does so by uploading the corresponding authentication tokens to the servers. The client then uploads the meta-data and the respective servers' IP addresses (or IDs, since the service knows the ips already - we leave these as implementation decisions) to the directory service. The IP addresses can be uploaded directly if Alice wants to encrypt them with Bob's public key, or she can simply tell IDs to the directory service.

505 3.6.4. Setup: Bob's Registration

The Steps between 1 and 8 of Figure 3 shows Bob's registration steps. Bob, entering his password to his authenticator device, generates the registration code h_{sk_Bob} as in Section 3.4.1 and keys it in to the client (Step 1). The client then generates the salts and computes the authentication tokens per server (2). The client also generates
510 a private/public key pair (2) that depends on the output of the authenticator given the encryption password of Bob so that later the private key can be generated at the client given the same value. Then Bob's client uploads all authentication tokens and the private key to Alice's client (3) where a re-encryption key is computed together with Alice's private key (5) as above. Note that Alice encrypted the symmetric encryption
515 keys (which are different per server) with her private proxy re-encryption key. Since Alice's client is authenticated to servers already (meaning the client has Alice's information - if not, it may prompt Alice to acquire it), at this step, it can compute Alice's private key as well to create the re-encryption key. Then it distributes the meta-data, the authentication tokens and the re-encryption key (6,7) as in Figure 3. As an imple-
520 mentation decision, the servers may re-encrypt the keys for Bob just after they receive the re-encryption key (8) or they may do the re-encryption when a request from Bob arrives. While the former option shortens the time to serve Bob, the latter one saves storage space and computation power (to re-encrypt the encryption meta-data) for the files that Bob may never request.

525 3.6.5. Bob's Authentication

The Steps between 9 to 16 in Figure 3 shows the authentication and data acquiring steps. After Bob has the meta-data (such as directory information) for the files and the address of the directory service, he is ready to authenticate and receive the files in the shared directory. After the registration is done, Bob can receive the necessary inputs
530 from the directory service alone. Bob first keys in his password to the authenticator and create a $OTP_{a,B}$ for himself and keys it in to the client (9). The client then differentiates the OTP_a as $OTP_{c,B,DS}$ (10) and queries the directory service with the files' meta-data that he wants to receive (11). If the client initially doesn't have any metadata, then the directory service is the place to access it. In the meantime it can create the one-
535 time password per server $OTP_{c,i,B}$ (13) since he may know the details of the servers to be queried, having the meta-data (or not, if it is the first login from a new client). Then, having the IPs from the directory service (12) Bob's client sends authentication tokens to the servers (14). Verification is exactly is as in Section 3.5.2. When verified, the servers return each part to Bob's client (16) where it can decrypt the keys and the
540 encryption meta-data associated with the downloaded files with Bob's own private key $priv_{k,Bob}^{PRE}$, then decrypt the data and rebuild it by decoding it (16).

3.6.6. Bob's Updates

When a party downloads the data, naturally he can only decrypt it with the secret he knows. If a symmetric encryption is employed alone, Bob has no option but
545 to receive the data (encrypted for Bob) or the key at least (and encryption meta-data per file) through Alice. This causes a problem where for each new file addition Alice needs to communicate with all the collaborators to distribute the encryption meta-data per file and possibly a new key if the data is encrypted under a different key (to support partial sharing, this would be the case). Therefore, it would be desirable that the
550 servers re-encrypt necessary data and updates so that they can be decrypted by the requester's secret. But (re-)encrypting the whole data with a PRE scheme is not tractable. Both the encryption and decryption are around four orders of magnitude slower than a symmetric en/decryption, therefore the overhead to the server would be tremendous. When there are multiple parties collaborating, keeping all the data encrypted under a

555 (per item) symmetric encryption key set by Alice and encrypting the key and the encryption meta-data with Alice’s public key seems to be the best solution. Therefore all collaborating parties authorized by Alice can reach the associated keys and join the shared space even if Alice is not present. With this design choice the security shifts from a trusted server to a provably-secure cryptosystem, because the server can only
560 re-encrypt a key (and the encryption meta-data) to be decrypted by another private key only if the user owning that private key has been authorized by the data owner. An adversary cannot get the keys re-encrypted to his private key even by colluding with the servers, unless he has previously stolen a legitimate user’s private key.

Another important functionality is to give the data owner full control on the data
565 updates, since otherwise, one of the collaborators may corrupt the data owner should possess the ability to approve or disapprove each update and recover an older version of the data if needed. Yet, forcing Alice to check all the updates before they are applied wouldn’t be practical, especially if Alice is not present/online. We suggest applying an auditable versioning mechanism that has recently been proposed [44] and that specif-
570 ically addresses the issue. The scheme, other than providing the above-mentioned functionality, also allows both parties to check the integrity of all the versions of the data.

3.6.7. Auditable Versioning

We include a brief summary of the idea behind auditable versioning [44] for the
575 sake of self-containedness and completeness of our proposed collaboration framework. The main idea is to check data integrity of a dynamic (read/write enabled) data against system faults or malicious parties while providing the functionality of collaboration. The basic idea and backbone data structure, FlexList is presented in Figure 4. In the figure, d_1 to d_5 represents the chunks of the data, where d_B represents the chunk
580 Bob updates. As shown in the figure, the chunk d_B is added to the data structure by only creating new nodes on the path to the updated chunk d_4 . Therefore, persistence is achieved, and the previous version remains untouched. When Alice wants to go through approval/rejection, she can simply merge the update, or reject it by marking node A as the root node. Additionally, authenticating the data structure [45, 46, 44, 37]

Figure 4: Bob's Update.

585 also provides the function to check the integrity of the data. Since our setting is based on untrusted storage devices, we note that this functionality is crucial. The overheads are marginal, thus using it does not hurt performance, while enhancing functionality.

Given that the erasure coding and the encryption are applied at the block level, the data structure perfectly gets along, and translating an update to be applied to the data structure is trivial [37, 44]. In provable data possession schemes, the data structure is 590 held in the RAM of the server (thanks to its marginal size) to serve faster. One can simply store the required meta-data for auditing, along with the symmetric encryption keys and encryption meta-data, as discussed in Sections 3.6.1 and 3.6.6. Observing the data structure, one can learn how many versions there are, or which collaborator 595 applied how many updates, the size of the updates, etc. If it is desirable to mask such information, encrypting the data structure itself may be considered. This is a somewhat tangential issue with respect to the focus of this work, and we don't discuss it any further.

3.6.8. Server Storage Summary

As a result of the refinements discussed in the Sections 3.6.6 and 3.6.7, the server content looks like in Figure 5. One authentication token per collaborator is stored at the server. Data parts that are stored on that particular server are assembled under a persistent authenticated data structure, where each part is encrypted with a symmetric key. The symmetric key and the encryption meta-data are illustrated as a key in the figure where the key is encrypted with the public key (for PRE) of the data owner. We will like to note that the servers still continue to have symmetric roles, and any k out of n property holds also for each block of the authenticated data structure.

Figure 5: A server storage.

4. Security Discussion

So far we have discussed how the layered security can be achieved. In this section, we discuss how it is actually useful. We list possible breaches and investigate further (using event tree analysis) what an adversary needs to do even further to actually get hold on the stored objects after an initial breach.

We first enumerate the attack surface. Then in table 3, we show the outcomes of each shortest combination of successful attacks. The numbers in the cells indicate a possible ordering, but alternate ordering for a successful attack may also exist. The analysis demonstrates that, generally, breaching individual attack surfaces is not enough in itself to compromise the system. It is worth noting that, if the adversary compromises the client during the setup phase then the scheme breaks down (pretty much like all systems). With a keylogger or a cuckoo attack [47] the adversary could get hands on the passwords themselves, yet this is partially prevented by the use of the separate authenticator for the password key in. As also expected, for the protocol to function, the secret h_{sk} must not be compromised.

4.1. Possible attacks

An adversary may initiate an attack on several points of the system. It can focus its attacks on the server (we denote as SR), a client device (CL), on the communications channel (COMM), on the authenticator device (AUTH), or the user (USR) or directly to the data itself (DT). The adversary may perform its attack any either one or more of these. We list possible atomic attacks below, before analyzing how our system is affected in case of such successful atomic attacks.

Adversary...

- learns user's password pw (USR:PW)
- learns user's encryption password e (USR:E)
- compromises the authenticator (AUTH:COMP)
- breaks the encryption of the data parts by bruteforce (DT:BF)
- colludes with or compromises a server (SR:COMP)
- compromises the client being used for authentication (CL:COMP:AUTH)
- convinces the client that he runs a legitimate server - impersonation (CL:IMP)
- eavesdrop on the communication between the client and the servers (COMM:EAV)
- replays the authentication request - replays OTPs (SR:RPLY)
- performs man in the middle attack during authentication (COMM:MITM:AUTH)
- performs man in the middle attack during setup (CL:MITM:ST)
- compromises the client during setup (CL:COMP:ST)

4.2. Analysis

In Table 3, **the cases a to d** cover the situation where user's password and authenticator are both compromised. Without any one of these two, the adversary cannot authenticate to the servers even though he has found the servers. Trying all possible OTPs is an option that has been thwarted as discussed in 3.5.1. It is worth noting that the authenticator is harder to be compromised in our scheme compared to the previous work [1] since it is not attached to the client device during the authentication phase. This is

CASE NAME	USR: PW	AUTH: COMP	USR: E	DT: (k times)	BF	SR: COMP (s times)	CL: COMP: AUTH	CL:IMP (c times)	COMM: EAV	SR: RPLY	COMM: MITM: AUTH	CL: MITM: ST	CL: COMP: ST	OUTCOME
a	1	2	3											ACCESS TO THE DATA CONTENT
b	1	2		3 (if $k \geq m$)										ACCESS TO THE DATA CONTENT
c	1	2		3 (if $k < m$)										PARTIAL ACCESS TO THE DATA CONTENT
d	2	3	4			1								ACCESS TO THE DATA CONTENT
e				2 (if $k \geq m$)	1 (if $s \geq m$)									ACCESS TO THE DATA CONTENT
f				2	1 (if $s < m$)									PARTIAL ACCESS TO THE DATA CONTENT
g				2 (if $k < m$)	1 (if $s \geq m$)									PARTIAL ACCESS TO THE DATA CONTENT
h					1 (if $s \geq m$)	2								ACCESS TO THE DATA CONTENT
i					1 (if $s < m$)	2								PARTIAL ACCESS TO THE DATA CONTENT
j							1			2				ACCESS TO THE QUERIED CONTENT
k				2 (if $k \geq m$)				1 (if $c \geq m$)						ACCESS TO THE DATA CONTENT
l				2 (if $k < m$)				1 (if $c \geq m$)						PARTIAL ACCESS TO THE DATA CONTENT
m				2				1 (if $c < m$)						PARTIAL ACCESS TO THE DATA CONTENT
n				3 (if $k \geq m$)					1	2 (before the client)				ACCESS TO THE QUERIED CONTENT
o				3 (if $k < m$)					1	2 (before the client)				PARTIAL ACCESS TO THE QUERIED CONTENT
p				3 (if $k \geq m$)						2 (before the client)	1			ACCESS TO THE QUERIED CONTENT
q				3 (if $k < m$)						2 (before the client)	1			PARTIAL ACCESS TO THE QUERIED CONTENT
r				3 (if $k \geq m$)				2				1		ACCESS TO THE QUERIED CONTENT
s				3 (if $k < m$)				2				1		PARTIAL ACCESS TO THE QUERIED CONTENT
t													1	ADVERSARY LEARNS ALL THE SECRETS AND HAS FUTURE ACCESS TO THE SYSTEM

Table 3: Possible attack combinations to breach security.

655 a crucial differentiator of our work since in the previous works the possession factor is
 vulnerable to the attacks to the client device. Even though the adversary has access to
 both the passwords, to gain access to the data content, the adversary still needs to deal
 with all the pieces and break their encryptions since he needs the value h_{sk} from the
 authenticator for decryption as discussed in Section 3.4.3. In **the cases d to i**, we see
 660 that the adversary compromises some servers. Compromising a server doesn't reveal
 information to authenticate with the other servers, nor the information to decrypt the
 parts of the data that become accessible from the compromised servers. The adversary
 can try to break the encryption of the parts that are revealed by compromising a number
 of servers. Even if s or k is less than m (where m is a parameter to the erasure code
 665 employed) some fragmentary information about the data can be revealed to the adver-
 sary provided they can be decrypted, given that the dispersal is achieved with erasure
 coding. To assuage this later problem, threshold (secret sharing [48]) cryptography
 could be applied to the data instead of erasure coding. We considered it but did not
 use threshold cryptography. It is worth noting that threshold cryptography brings about
 670 significantly higher storage overheads, but for what would be a marginal benefit in our
 setup, over the proper use of encryption along with erasure coding based dispersal. To
 decrypt the data, in that case, the adversary either still needs the passwords and the
 authenticator of the data owner as **in case d** since the encryption password depends on
 them or needs to break the encryptions by brute-force as in **the cases e to g**. Colluding
 675 with or compromising a client doesn't give the adversary the opportunity to get the se-
 cret key re-encrypted for its private key. Therefore to acquire the encryption key from
 the servers, the adversary needs to have one of the legitimate collaborators private key.
 Which can be recalculated with the attacks **in case a** or the adversary needs to com-
 promise a collaborator while the client is decrypting the downloaded content (we call
 680 it authentication phase still). **The case j** takes the client device being compromised
 during the authentication into account. The server then can replay the authentication
 request to the servers. Whoever uses the generated one-time passwords first acquires
 the data from the servers. Therefore, the adversary can acquire the requested data.
 But note that a compromised client already implies that the requested data is exposed,
 685 therefore the adversary doesn't require to spend any further effort. **The cases k to m**

cover the cases where the adversary impersonates a server thus receiving the parts that are uploaded. The requirement of collecting and breaking the encryption for more than m parts remains, otherwise, the access is partial. **The cases n and o** start with the adversary eavesdropping on the communication channel, where the **cases p and q** start with the man in the middle attack during authentication. All these cases are followed by a replay attack within the time period allowed by the one-time passwords. If the replay arrives the servers before the actual query, which is easier with the man in the middle attack, then the queried content can be acquired by the adversary. In fact, this doesn't add much value since in both cases the adversary already has access to the returned data. To decrypt the data, the adversary either needs access to the passwords and the authenticator (yet having access to these already makes the attacks discussed in these cases redundant) or needs to decrypt (by breaking either encryption or acquiring a private PRE key of a legitimate collaborator) the data as discussed above. Note that the one-time passwords generated per server are not useful in the future and as discussed in 3.5 - the adversary cannot use them to acquire files other than the requested ones. **In cases r and s**, we see the case where the adversary impersonates the servers by a man in the middle attack during the setup phase. If this is achieved then the adversary gets all the data parts to himself which then needs to be decrypted as discussed in other cases above.

5. Evaluation

We have implemented the individual building blocks of the protocol, and they work in tandem. For testing, we used a laptop computer with average specifications instead of one of our servers to better simulate a client machine. We have conducted our experiments on a 64-bit computer (manufactured in 2013) with a mobile dual-core Intel Core i5-4288U @ 2.60GHz with 3MB shared L3 cache CPU, 8GB RAM, 512GB SSD (with average 700MB/s read 600MB/s write speeds) and running Ubuntu 15.10 Wily to benchmark the computations for the client, server as well as the authenticator. In practice, the different devices running the various operations would have different computational resources, and the presented benchmarks are solely meant to be rep-

715 resentative of the qualitative nature of overheads involved. We employed GMP[49],
 openssl[22], cashlib[50], ceph[51] and crypto++[52] libraries and C++, Java (Android
 SDK) as the programming languages for our implementation. For Proxy re-encryption,
 we have employed the implementation in Java by NICS [20], that realizes the scheme
 by Ateniese et al. [21]. We employed proxy encryption with modulus size 2048 bits,
 720 and we generated random strings of size s for the secret keys depending on the OTP
 size, where $s = |p|$. For the experiments, we assume the client has a total of 1GB of
 data to store, . All reported results are the average of 10 runs.

5.1. Storage

We first investigate the storage needs of the protocol. There is only one value that
 725 needs to be stored at the authenticator, which is the secret key. And each server keeps
 an authentication token per user, a re-encryption key per collaborator and the encrypted
 data. In table 5.1, we show the overheads for our protocol (excluding the encoded data)
 via a case where there are 5 users (1 owner, 4 collaborators), thus each server keeps
 four re-encryption keys and five authentication tokens. Lastly, it is worth noting that
 730 the client storage is 0 in all cases.

OTP Length (n)	ϕ_{num}	4	6	8	12	16
Authenticator	2.5	3.2	5	6.5	10	10.6
Server	1036.5	1040	1049	1056.5	1074	1077

Table 4: Storage needs of the protocol for the above example case (in Bytes).

When the protocol is initialized twice to limit the number of wrong trials (as dis-
 cussed in Section 3.5.1), the storage overhead adds up to the sum of the storage over-
 heads of only the authentication protocol for both OTP sizes used. Therefore, instead
 of using OTP of size 16 alone, using OTP of size 6 (only numbers) for frequent log-in
 735 attempts and keeping a size 16 option for revocation would simply add the extra over-
 head of 25% to the authenticator devices' storage and for the above example, around
 1% (of the storage overhead, excluding the data) per user for the server storage. To
 sum up, it is essential to let users key shorter OTPs in to use the system. To do so while

preserving the high security level, the sacrifice is marginal.

Figure 6: Client processing requirements.

5.2. Processing

We investigate the processing requirements of the protocol and present the results in Figure 6 and Figure 7 for the client and the authenticator respectively. The operation to verify an authentication request at the server is almost the same with the authentication process at the client device. Though the server will have different hardware capabilities than the client, the evaluations with the client indicate qualitatively the overheads of the operations also at the server end.

The client device can handle all operations with ease (Figure 6) and all operations for all considered OTP sizes would be imperceptible to a human. The time requirement increases with the size of OTP as expected, since all the hash parameters ($G = (p, q, g)$) are larger, thus affecting the calculation times of the hash values. The figure reflects the time spent for one server, and for n servers the operation is repeated n times.

For the authenticator, we have set up the application on various handsets (varying from Samsung Galaxy S7 (2016) to Galaxy Nexus (2012)). We conclude that even with an outdated hardware from 2012, the initialization and OTP generation measures in milliseconds and are instantaneous from a human perspective. We present the benchmarking on our client device in Figure 7.

Verification process needs the same amount of computation as the authentication of the client in Figure 6. Our proposed scheme is practical not only when the storage de-

Figure 7: Authenticator processing requirements.

vices are well provisioned, but also in edge computing set-up, where the storage servers
 760 would require the processes to be light - and may not cope well with the computational
 overheads of techniques [1] employing signature schemes.

Additionally, the solution discussed in Section 3.5.1 to limit the number of wrong
 trials, other than the ease of use, also proves itself useful by means of processing times
 at both the client and the server. The processing times are naturally much lower for the
 765 6 numbers OTP compared to 16 character long OTP. At the client device the difference
 may not be noticeable, but at the server's end where there are many clients to be served,
 the difference may be useful to serve more clients or serve the existing ones faster.

5.3. Data processing

In terms of computation, the most time-consuming part of our protocol is to create the
 770 n pieces out of the data, to be dispersed across the n servers, and subsequently recre-
 ating the original data object out of m of these n dispersed pieces. Note that there will
 also be network usage and latency issues, which we have left out in the current discus-
 sions. We considered the application of both erasure coding as well as Shamir's secret
 sharing [53]. Specifically, for erasure coding, we applied Reed Solomon Vandermonde
 775 implementation from jerasure plug-in of ceph library [51]. We used the default 4KB
 stripe size. The crypto++ [52] library was used to implement Shamir's secret sharing
 algorithm. The reported times include the actual computation time, but also the time to

read/write to the buffer.

Figure 8: Times spent for Reed Solomon (RS) and Shamir's Secret Sharing (SS) algorithms.

Before we discuss the computation time for these two approaches, it is worth discussing qualitatively the implications of the choice of parameters m and n . The two-factor aspect of approach will work even with $m = 1$, however the additional level of security achieved through dispersal will be missing in that case. For a given choice of m , a larger n will provide a potential attacker a larger choice of subsets of servers to compromise, and may have a negative effect on security from the perspective of content confidentiality. At the same time, though fault-tolerance was out of the direct scope of this paper, a larger n for any particular choice of m also provides the system to better mitigate server failures vis-a-vis data availability and integrity. The storage (and communication) overhead increases by a factor of n/m when erasure coding is used, while it increases by a factor of n when using Shamir's secret sharing scheme. This also means that for the same choice of m and n , secret sharing uses m -folds more storage than erasure coding. Thus, an optimal choice of m and n is not obvious, and further systematic investigation to determine good choices, likely dependent on the deployment environment, will have to be carried out in future. For the time being, we instead report the performance for a few ad-hoc parameter choices, where we aimed for n/m within 2 (since that will also be the storage overhead if erasure coding is used), while still allowing for a margin of recovery from 1 or 2 faults, while exploring non-trivial m and

yet keeping n small (since the storage overhead worst case scenario is determined by n when using secret sharing). Specifically, we chose the following n/m configurations: 3/2, 4/3 and 6/4 to carry out our experiments. Figure 8 shows the time for dispersal and recreation of 1GB of data. In order to also determine the fault-tolerance implications, we simulated one randomly missing piece among the dispersed data for each of our experiment iterations.

Since both Reed Solomon erasure coding and Shamir’s secret sharing (indicated with RS and SS respectively in Figure 8) are well studied, the results are on expected lines, but nevertheless, there are several interesting things to note. Erasure coding/decoding is orders of magnitude faster. In fact, the dominant time for erasure coding is really the data read/write operations, and a larger n/m value leads to larger amount of data writes after coding. That’s why coding operations for $n/m = 4/3$ is faster than for $n/m = 3/2$ and $n/m = 6/4$. In contrast, for secret sharing, while read/write operations take much more time than erasure coding, the dominant time is in the computation. Decryption, however, is significantly faster than encryption. We further note that even though the performance results are reasonably acceptable when using erasure codes, by enabling SIMD parallelization [54] and increasing the stripe size (e.g. 1Mb) the coding can be made even faster. Overall, given the other layers of security in our approach, and the performance penalty that secret sharing imposes, we advocate the use of erasure coding as the preferred mechanism for data dispersal. AES-256 encryption and decryption took respectively 3.75 and 2.02 seconds per GB. Note that due to the redundancy added for the dispersal process, the volumes of data to be encrypted increase, and the time taken for encryption/decryption increase proportionally. The encryption/decryption time overheads are dominated by those of (de-)coding. Though we have not measured the network latency to ship in and out data in our experiments, all the computations cumulated together still is significantly small, and hence, the proposed approach for access control should readily extend existing works like [3, 18] without perceptible performance deterioration. Exploring a completely integrated system and benchmarking other system aspects (network usage, energy implications for mobile devices, etc.) are outstanding aspects for future.

5.4. Overheads

Figure 9:

Figure 10:

In this subsection, we investigate the time and space requirements of the proposed protocol.

830 In Tables 5 and 6, we present the raw data to show explicit overheads introduced (other than the standard IO times for storage) at each step of the scheme. Each value is marked with a symbol (legends for the symbols are provided below the table). The first row of the table represents the simple case where the data owner sends her data

	Setup						
	Comm. Cost	Authenticator Process	Client Process	Server Process	Authenticator Storage	Client Storage	Server Storage (each)
	Client sends						
Simple (Base case)	Data	0	0	0	-	0	Data
Authentication Overhead (OH) with SHA-1	Data + 20 ^a bytes	0	0.05 ^a ms	0	-	0	Data + 20 ^a bytes
2 factor Authentication (2fa) OH	Data + 2.5 ^b bytes	0.00414 ^a ms	0.00159 ^c ms	0	2.5-bytes	0	Data + 2.5 ^b bytes x c
2fa + revocation	Data + 2.5 ^b bytes + 10.6 ^c bytes	0.00414 ^a ms + 0.0173 ^d ms	0.00159 ^c ms + 0.01 ^e ms	0	2.5-bytes + 10.6 ^c bytes	0	Data + (2.5 ^b bytes + 10.6 ^c bytes) x c
2fa + revocation + Replication OH	(Data + 2.5 ^b bytes + 10.6 ^c bytes) x r	0.00414 ^a ms + 0.0173 ^d ms	(0.00159 ^c ms + 0.01 ^e ms) x r +	0	2.5-bytes + 10.6 ^c bytes	0	Data + (2.5 ^b bytes + 10.6 ^c bytes) x c
2fa + revocation + Erasure Coding (EC) OH	Data x (n/m) + (2.5 ^b bytes + 10.6 ^c bytes) x n	0.00414 ^a ms + 0.0173 ^d ms	(0.00159 ^c ms + 0.01 ^e ms) x n + 16 ^f sec for 1GB	0	2.5-bytes + 10.6 ^c bytes	0	Data/m + (2.5 ^b bytes + 10.6 ^c bytes) x c
2fa + revocation + EC + AES Encryption OH	Data + (2.5 ^b bytes + 10.6 ^c bytes) x n	0.00414 ^a ms + 0.0173 ^d ms	(0.00159 ^c ms + 0.01 ^e ms) x n + 16 ^f sec for 1GB + 3 ^g sec for 1GB	0	2.5-bytes + 10.6 ^c bytes	0	Data + (2.5 ^b bytes + 10.6 ^c bytes) x c
2fa + revocation + EC + AES Encryption OH + Proxy Re-Encryption (PRE) OH	Data + (2.5 ^b bytes + 10.6 ^c bytes + 256 ^h bytes) x n	0.00414 ^a ms + 0.0173 ^d ms	(0.00159 ^c ms + 0.01 ^e ms) x n + 3000 ⁱ ms + 0.2 ^j ms + 16 ^f sec for 1GB + 3 ^g sec for 1GB + 8 ^k ms per 256bytes	0	2.5-bytes + 10.6 ^c bytes	0	Data + (2.5 ^b bytes + 10.6 ^c bytes) x c

^a Authentication Token (AT) ^b 2fa AT ^c with a larger p for revocation ^d Proxy re-encryption (PRE) key ^e PRE global parameters
^f registration code ^gTP One-time Password ^h Verification ⁱ Update ^j 2fa key ^k previous authentication request not verified
^l AES Encryption ^m AES decryption ⁿ PRE Data Encryption ^o PRE Data Decryption ^p PRE Data Re-encryption (background process)
^q Data Encoding (EC) ^r Data Decoding (EC) Data: The data size c: number of clients r: replica amount n/m: Erasure coding parameters

Table 5: Overheads introduced at the setup phase of the scheme.

to a server without processing it anyhow. In the table, we consider the client has a
835 GB data (indicated as “Data”). Descending in the table, with each row we present
time and space requirements at each step of the protocol for a set of additional security
measures. We start with adding the overheads of authentication to the basic case, which
is sending a simple hash value of the memorized password to authenticate. Then, we
add our two-factor authentication scheme. In the next row, we present overheads for
840 revocation where a second setup is performed for allowing the client to authenticate if
he mistypes the password or if anything else goes wrong and the one-time password
gets rejected by the server at the first try. . With the next row, we show how overheads
are effected when replication is performed. In that case, we send identical data pieces
to several servers as is and setup two-factor authentication with all. In the next row, we

	Retrieve (full/partial)				Update				
	Comm. Cost	Authenticator Process	Client Process	Server Process	Comm. Cost	Authenticator Process	Client Process	Server Process	
	Client sends	Server (each) sends							
Simple (Base case)	-0 bytes	max the whole data	-	0	0	256 ^a bytes	-	0	0
Authentication Overhead (OH) with SHA-1	20 ^b bytes	max the whole data	-	0.05 ^a ms	-0 ^c ms	256 ^a bytes + 20 ^b bytes	-	0.05 ^a ms	-0 ^c ms
2 factor Authentication (2fa) OH	2.5 ^d bytes	max the whole data	0.0033 ^{otp} ms	0.00171 ^{ms}	0.0018 ^e ms	256 ^a bytes + 2.5 ^d bytes	0.0033 ^{otp} ms	0.00159 ^{ms}	0.0018 ^e ms
2fa + revocation	2.5 ^d bytes if(=) 10.6 ^{ec} bytes	max the whole data	0.0033 ^{otp} ms 0.0166 ^{otp} ms	0.00171 ^{ms} + if(=) 0.0096 ^{ms}	0.0018 ^e ms + if(=) 0.01 ^{ms}	256 ^a bytes + 2.5 ^d bytes if(=) 10.6 ^{ec} bytes	0.0033 ^{otp} ms 0.0166 ^{otp} ms	0.00159 ^{ms} + if(=) 0.01 ^{ms}	0.0018 ^e ms + if(=) 0.01 ^{ms}
2fa + revocation + Replication OH	2.5 ^d bytes if(=) 10.6 ^{ec} bytes	Data/n or max the whole data	0.0033 ^{otp} ms 0.0166 ^{otp} ms	(0.00171 ^{ms} + if(=) 0.0096 ^{ms}) x r	0.0018 ^e ms + if(=) 0.01 ^{ms}	(256 ^a bytes + 2.5 ^d bytes if(=) 10.6 ^{ec} bytes) x r	0.0033 ^{otp} ms 0.0166 ^{otp} ms	(0.00159 ^{ms} + if(=) 0.01 ^{ms}) x r + 20 ^b ms	0.0018 ^e ms + if(=) 0.01 ^{ms}
2fa + revocation + Erasure Coding (EC) OH	2.5 ^d bytes if(=) 10.6 ^{ec} bytes	max Data/m	0.0033 ^{otp} ms 0.0166 ^{otp} ms	(0.00171 ^{ms} + if(=) 0.0096 ^{ms}) x n + 10 ^{sec} for 1GB	0.0018 ^e ms + if(=) 0.01 ^{ms}	(256 ^a bytes + 2.5 ^d bytes if(=) 10.6 ^{ec} bytes) x n	0.0033 ^{otp} ms 0.0166 ^{otp} ms	(0.00159 ^{ms} + if(=) 0.01 ^{ms}) x n	0.0018 ^e ms + if(=) 0.01 ^{ms}
2fa + revocation + EC + AES Encryption OH	2.5 ^d bytes if(=) 10.6 ^{ec} bytes	max Data/m	0.0033 ^{otp} ms 0.0166 ^{otp} ms	0.00171 ^{ms} + if(=) 0.0096 ^{ms} + 10 ^{sec} for 1GB + 1.5 ^{ms} sec for 1GB	0.0018 ^e ms + if(=) 0.01 ^{ms}	(256 ^a bytes + 2.5 ^d bytes if(=) 10.6 ^{ec} bytes) x n	0.0033 ^{otp} ms 0.0166 ^{otp} ms	(0.00159 ^{ms} + if(=) 0.01 ^{ms}) x n	0.0018 ^e ms + if(=) 0.01 ^{ms}
2fa + revocation + EC + AES Encryption OH + Proxy Re-Encryption (PRE) OH	2.5 ^d bytes if(=) 10.6 ^{ec} bytes	max Data/m	0.0033 ^{otp} ms 0.0166 ^{otp} ms	0.00171 ^{ms} + if(=) 0.0096 ^{ms} + 10 ^{sec} for 1GB + 1.5 ^{ms} sec for 1GB + 20 ^{ms} per 256 bytes	0.0018 ^e ms + if(=) 0.01 ^{ms} + 40 ^{ms} per 256 bytes x c	(256 ^a bytes + 2.5 ^d bytes if(=) 10.6 ^{ec} bytes) x n	0.0033 ^{otp} ms 0.0166 ^{otp} ms	(0.00159 ^{ms} + if(=) 0.01 ^{ms}) x n + 8 ^{ms}	0.0018 ^e ms + if(=) 0.01 ^{ms}

^a Authentication Token (AT) ^{2fa} AT with a larger p for revocation ^{Proxy re-encryption (PRE) key} ^{PRE} global parameters
^o registration code ^{otp} One-time Password ^{Verification} ^{Update} ^{2fa} key = previous authentication request not verified
^{ec} AES Encryption and AES decryption ^{PRE} Data Encryption ^{PRE} Data Decryption ^{PRE} Data Re-encryption (background process)
^{ms} Data Encoding (EC) ^{Data Decoding (EC)} Data: The data size c: number of clients r: replica amount n/m: Erasure coding parameters

Table 6: Overheads introduced at the retrieve and update phases of the scheme.

show the overheads when erasure coding is applied instead of replication Next to last, we present the overheads for data encryption And lastly, we show the total overheads introduced if the system is used as we discuss in this work, including with proxy re-encryption for collaboration.

6. Conclusions

In this work, we have proposed a new authentication protocol in a decentralized set-up (without any trusted third party), while using untrusted storage servers, and while also supporting both online and offline collaboration among multiple users over a shared data set. Our protocol does not require any trusted third party and possess a separate authenticator that is initialized and used offline. While the proposed approach can be

855 applied to multi-cloud storage set-ups in general, the nuances of the protocol design
(such as the elimination of the need to communicate with any trusted third party) make
it particularly suitable for also edge/fog computing set-ups, which are gaining traction
with internet of things (IoT) becoming main stream.

Acknowledgement

860 E. Esiner’s research was supported by the Singapore International Graduate Award
grant TU-2013-R13145800000000.

References

- [1] E. Esiner, A. Datta, Layered security for storage at the edge: On decentralized
multi-factor access control, in: Proceedings of the 17th International Conference
865 on Distributed Computing and Networking — ICDCN’16, ACM, New York, NY,
USA, 2016, pp. 9:1–9:10. doi:10.1145/2833312.2833452.
URL <http://doi.acm.org/10.1145/2833312.2833452>
- [2] J. M. Johansson, D. E. Canavor, D. W. Hitchcock, B. K. Bhimanaik, Approaches
for providing multi-factor authentication credentials, US Patent 9,864,852 (Jan. 9
870 2018).
- [3] A. Bessani, M. Correia, B. Quaresma, F. André, P. Sousa, Depsky: Dependable
and secure storage in a cloud-of-clouds, Transactions on Storage 9 (4) (2013)
12:1–12:33. doi:10.1145/2535929.
URL <http://doi.acm.org/10.1145/2535929>
- 875 [4] C. W. Ling, A. Datta, Intercloud raider: A do-it-yourself multi-cloud private data
backup system, in: M. Chatterjee, J.-n. Cao, K. Kothapalli, S. Rajsbaum (Eds.),
Proceedings of the 15th International Conference on Distributed Computing and
Networking — ICDCN’14, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014,
pp. 453–468. doi:10.1007/978-3-642-45249-9_30.

- 880 [5] H. Abu-Libdeh, L. Princehouse, H. Weatherspoon, Racs: A case for cloud storage diversity, in: *Proceedings of the 1st ACM Symposium on Cloud Computing — SoCC’10*, ACM, New York, NY, USA, 2010, pp. 229–240. doi: 10.1145/1807128.1807165.
URL <http://doi.acm.org/10.1145/1807128.1807165>
- 885 [6] J. Kovács, P. Kacsuk, Occopus: a multi-cloud orchestrator to deploy and manage complex scientific infrastructures, *Journal of Grid Computing* 16 (1) (2018) 19–37. doi:10.1007/s10723-017-9421-3.
URL <https://doi.org/10.1007/s10723-017-9421-3>
- [7] F. Wu, X. Li, A. K. Sangaiah, L. Xu, S. Kumari, L. Wu, J. Shen, A lightweight
890 and robust two-factor authentication scheme for personalized healthcare systems using wireless medical sensor networks, *Future Generation Computer Systems* 82 (2018) 727 – 737. doi:<https://doi.org/10.1016/j.future.2017.08.042>.
URL <http://www.sciencedirect.com/science/article/pii/S0167739X1730523X>
- 895 [8] E. Esiner, S. H. Hanley, A. Datta, Dmztore: A dispersed data storage system with decentralized multi-factor access control (demo), in: *IEEE 36th International Conference on Distributed Computing Systems — ICDCS’16*, 2016, pp. 757–758. doi:10.1109/ICDCS.2016.42.
- [9] M. Satyanarayanan, P. Bahl, R. Caceres, N. Davies, The case for vm-based
900 cloudlets in mobile computing, *IEEE Pervasive Computing* 8 (4) (2009) 14–23. doi:10.1109/MPRV.2009.82.
- [10] The SpiderOak, <https://spideroak.com/>, accessed: 2018-02-16.
- [11] Wuala by Lacie, <https://spideroak.com/>, accessed: 2018-02-24.
- [12] Truecrypt, <https://truecrypt.ch/>, accessed: 2018-02-21.
- 905 [13] Boxcryptor, <https://www.boxcryptor.com/>, accessed: 2018-02-21.

- [14] G. Yang, D. S. Wong, H. Wang, X. Deng, Two-factor mutual authentication based on smart cards and passwords, *Journal of Computer and System Sciences* 74 (7) (2008) 1160–1172. doi:<https://doi.org/10.1016/j.jcss.2008.04.002>.
URL <http://www.sciencedirect.com/science/article/pii/S0022000008000482>
- [15] F. Aloul, S. Zahidi, W. El-Hajj, Two factor authentication using mobile phones, in: *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications — AICCSA'09, 2009*, pp. 641–644. doi:10.1109/AICCSA.2009.5069395.
- [16] [Google] About 2-Step Verification, https://support.google.com/accounts/answer/180744?hl=en&ref_topic=1099588/, accessed: 2018-02-19.
- [17] D. M'Raihi, S. Machani, M. Pei, J. Rydell, Totp: Time-based one-time password algorithm, Internet Requests for Comments, Internet Engineering Task Force (IETF), RFC 6238.
- [18] H. Y. Lin, W. G. Tzeng, A secure erasure code-based cloud storage system with secure data forwarding, *IEEE Transactions on Parallel and Distributed Systems* 23 (6) (2012) 995–1003. doi:10.1109/TPDS.2011.252.
- [19] M. Li, C. Qin, J. Li, P. P. C. Lee, Cdstore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal, *IEEE Internet Computing* 20 (3) (2016) 45–53. doi:10.1109/MIC.2016.45.
- [20] NICS Crypto Library, <https://www.nics.uma.es/dnunez/nics-crypto/>, accessed: 2018-02-20.
- [21] G. Ateniese, K. Fu, M. Green, S. Hohenberger, Improved proxy re-encryption schemes with applications to secure distributed storage, *ACM Transactions on Information Systems Security* 9 (1) (2006) 1–30. doi:10.1145/1127345.1127346.
URL <http://doi.acm.org/10.1145/1127345.1127346>

- [22] J. Viega, P. Chandra, M. Messier, Network Security with Openssl, 1st Edition,
935 O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.
- [23] M. MAMBO, E. OKAMOTO, Proxy cryptosystems: Delegation of the power to
decrypt ciphertexts, IEICE Transactions on Fundamentals of Electronics, Com-
munications and Computer Sciences 80 (1) (1997) 54–63.
URL <https://ci.nii.ac.jp/naid/110003207677/en/>
- 940 [24] M. Blaze, G. Bleumer, M. Strauss, Divertible protocols and atomic proxy cryp-
tography, in: K. Nyberg (Ed.), Proceedings of the International Conference on
the Theory and Application of Cryptographic Techniques — EUROCRYPT'98,
Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 127–144. doi:10.
1007/BFb0054122.
- 945 [25] A. IVAN, Proxy cryptography revisited, Proceedings of the Network and
Distributed System Security Symposium — NDSS'03.
URL [http://www.isoc.org/isoc/conferences/ndss/03/proceedings/
papers/14.pdf](http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/14.pdf)
- [26] M. Jakobsson, On quorum controlled asymmetric proxy re-encryption, in: Pro-
ceedings of the 19th IACR International Conference on Practice and Theory in
950 Public-Key Cryptography — PKC'99, Springer Berlin Heidelberg, Berlin, Hei-
delberg, 1999, pp. 112–121. doi:10.1007/3-540-49162-7_9.
- [27] B. Libert, D. Vergnaud, Unidirectional chosen-ciphertext secure proxy re-
encryption, in: R. Cramer (Ed.), Proceedings of the 11th International Work-
shop on Practice and Theory in Public-Key Cryptography — PKC'08, Springer
955 Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 360–379. doi:10.1007/
978-3-540-78440-1_21.
- [28] R. Canetti, S. Hohenberger, Chosen-ciphertext secure proxy re-encryption, in:
Proceedings of the 14th ACM Conference on Computer and Communications
960 Security — CCS'07, ACM, New York, NY, USA, 2007, pp. 185–194. doi:
10.1145/1315245.1315269.
URL <http://doi.acm.org/10.1145/1315245.1315269>

- [29] J. Shao, Z. Cao, Cca-secure proxy re-encryption without pairings, in: S. Jarecki, G. Tsudik (Eds.), Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography — PKC’09, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 357–376. doi:10.1007/978-3-642-00468-1_20.
- [30] H. Y. Lin, W. G. Tzeng, A secure erasure code-based cloud storage system with secure data forwarding, IEEE Transactions on Parallel and Distributed Systems 23 (6) (2012) 995–1003. doi:10.1109/TPDS.2011.252.
- [31] J. Katz, Y. Lindell, Introduction to Modern Cryptography, Second Edition, 2nd Edition, Chapman & Hall/CRC, 2014.
- [32] T. P. Pedersen, Non-interactive and information-theoretic secure verifiable secret sharing, in: J. Feigenbaum (Ed.), Proceedings of the 11th Annual International Cryptology Conference — CRYPTO’91, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 129–140. doi:10.1007/3-540-46766-1_9.
- [33] D. Chaum, E. van Heijst, B. Pfitzmann, Cryptographically strong undeniable signatures, unconditionally secure for the signer, in: J. Feigenbaum (Ed.), Proceedings of the 11th Annual International Cryptology Conference — CRYPTO’91, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992, pp. 470–484. doi:10.1007/3-540-46766-1_38.
- [34] M. N. Krohn, M. J. Freedman, D. Mazieres, On-the-fly verification of rateless erasure codes for efficient content distribution, in: Proceedings of the IEEE Symposium on Security and Privacy — S&P’04, 2004, pp. 226–240. doi:10.1109/SECPRI.2004.1301326.
- [35] K. Yang, X. Jia, Data storage auditing service in cloud computing: challenges, methods and opportunities, World Wide Web 15 (4) (2012) 409–428. doi:10.1007/s11280-011-0138-0.
URL <https://doi.org/10.1007/s11280-011-0138-0>

- 990 [36] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, D. Song,
Provable data possession at untrusted stores, in: *Proceedings of the 14th ACM
Conference on Computer and Communications Security — CCS'07*, ACM, New
York, NY, USA, 2007, pp. 598–609. doi:10.1145/1315245.1315318.
URL <http://doi.acm.org/10.1145/1315245.1315318>
- 995 [37] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, O. Özkasap, Flexdpdp: Flexlist-
based optimized dynamic provable data possession, *Transactions on Storage*
12 (4) (2016) 23:1–23:44. doi:10.1145/2943783.
URL <http://doi.acm.org/10.1145/2943783>
- [38] J. Shen, S. Chang, J. Shen, Q. Liu, X. Sun, A lightweight multi-
1000 layer authentication protocol for wireless body area networks, *Future
Generation Computer Systems* 78 (2018) 956 – 963. doi:https:
//doi.org/10.1016/j.future.2016.11.033.
URL [http://www.sciencedirect.com/science/article/pii/
S0167739X16306963](http://www.sciencedirect.com/science/article/pii/S0167739X16306963)
- 1005 [39] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, M. J. Freedman,
CONIKS: Bringing key transparency to end users, in: *Proceedings of the 24th
USENIX Security Symposium — USENIX Security 15*, USENIX Association,
Washington, D.C., 2015, pp. 383–398.
URL [https://www.usenix.org/conference/usenixsecurity15/
1010 technical-sessions/presentation/melara](https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara)
- [40] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008).
- [41] M. Bellare, O. Goldreich, S. Goldwasser, Incremental cryptography: The case of
hashing and signing, in: Y. G. Desmedt (Ed.), *Proceedings of the 14th Annual In-
ternational Cryptology Conference — CRYPTO'94*, Springer Berlin Heidelberg,
1015 Berlin, Heidelberg, 1994, pp. 216–233. doi:10.1007/3-540-48658-5_22.
- [42] W. Diffie, M. Hellman, New directions in cryptography, *IEEE Transactions on
Information Theory* 22 (6) (1976) 644–654. doi:10.1109/TIT.1976.1055638.

- [43] I. S. Reed, G. Solomon, Polynomial codes over certain finite fields, *Journal of the Society for Industrial and Applied Mathematics* 8 (2) (1960) 300–304. arXiv: <https://doi.org/10.1137/0108018>, doi:10.1137/0108018.
 1020 [URL https://doi.org/10.1137/0108018](https://doi.org/10.1137/0108018)
- [44] E. Esiner, A. Datta, Auditable versioned data storage outsourcing, *Future Generation Computer Systems* 55 (2016) 17–28. doi:<https://doi.org/10.1016/j.future.2015.08.001>.
 1025 [URL http://www.sciencedirect.com/science/article/pii/S0167739X15002538](http://www.sciencedirect.com/science/article/pii/S0167739X15002538)
- [45] R. Tamassia, Authenticated data structures, in: G. Di Battista, U. Zwick (Eds.), *Proceedings of the 11th Annual European Symposium on Algorithms - ESA 2003*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 2–5. doi: [10.1007/978-3-540-39658-1_2](https://doi.org/10.1007/978-3-540-39658-1_2).
 1030 [10.1007/978-3-540-39658-1_2](https://doi.org/10.1007/978-3-540-39658-1_2).
- [46] M. Etemad, A. Kp, Transparent, distributed, and replicated dynamic provable data possession, in: M. Jacobson, M. Locasto, P. Mohassel, R. Safavi-Naini (Eds.), *Proceedings of the 16th International Conference on Applied Cryptography and Network Security — ACNS’18*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–18. doi:978-3-642-38980-1_1.
 1035 [10.1007/978-3-642-38980-1_1](https://doi.org/10.1007/978-3-642-38980-1_1).
- [47] B. Parno, Bootstrapping trust in a “trusted” platform, in: *Proceedings of the 3rd Conference on Hot Topics in Security — HOTSEC’08*, USENIX Association, Berkeley, CA, USA, 2008, pp. 9:1–9:6.
 URL <http://dl.acm.org/citation.cfm?id=1496671.1496680>
- [48] A. Beimel, Secret-sharing schemes: A survey, in: Y. M. Chee, Z. Guo, S. Ling, F. Shao, Y. Tang, H. Wang, C. Xing (Eds.), *Proceedings of the third International Workshop on Coding and Cryptology — IWCC’11*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 11–46. doi:978-3-642-20901-7_2.
 1040 [10.1007/978-3-642-20901-7_2](https://doi.org/10.1007/978-3-642-20901-7_2).
- [49] [GMPLib] The GNU Multiple Precision Arithmetic Library, <https://gmplib.org/>, accessed: 2017-04-19.
 1045 <https://gmplib.org/>, accessed: 2017-04-19.

- [50] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, A. Lysyanskaya, Zkpdl: A language-based system for efficient zero-knowledge proofs and electronic cash, in: Proceedings of the 19th USENIX Conference on Security — USENIX Security’10, USENIX Association, Berkeley, CA, USA, 2010, pp. 193–206.
1050 URL <http://dl.acm.org/citation.cfm?id=1929820.1929838>
- [51] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation — OSDI’06, USENIX Association, Berkeley, CA, USA, 2006, pp. 307–320.
1055 URL <http://dl.acm.org/citation.cfm?id=1298455.1298485>
- [52] Crypto++, <http://www.cryptopp.com/>, accessed: 2018-02-22.
- [53] A. Shamir, How to share a secret, Commun. ACM 22 (11) (1979) 612–613. doi: 10.1145/359168.359176.
URL <http://doi.acm.org/10.1145/359168.359176>
- 1060 [54] D. A. Patterson, J. L. Hennessy, Computer Organization and Design MIPS Edition: The Hardware/Software Interface, Newnes, 2013.