End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage

Long Chen, *Institute of Software, Chinese Academy of Sciences;* Ya-Nan Li and Qiang Tang, *The University of Sydney;* Moti Yung, *Google & Columbia University*

https://www.usenix.org/conference/usenixsecurity22/presentation/chen-long

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage

Long Chen
Institute of Software, Chinese Academy of Sciences

Ya-Nan Li
The University of Sydney

Qiang Tang
The University of Sydney

Moti Yung
Google & Columbia University

Abstract

The cloud has become pervasive, and we ask: how can we protect cloud data against the cloud itself? For messaging Apps, facilitating user-to-user private communication via a cloud server, security has been formulated and solved efficiently via End-to-End encryption, building on existing channels between end users via servers (i.e., exploiting TLS, and encryption, without the need to program new primitives). However, the analogous problem for Apps employing servers for storing and retrieving end-user data privately, solving the analogous "privacy from the server itself" (cloud-blind storage) where (1) based on existing infrastructure and (2) allowing user mobility, is, in fact, still open. Existing proposals, like password protected secret sharing (PPSS), target end-to-same-end encryption of storage, but need new protocols, whereas most popular commercial cloud storage services are not programmable. Namely they lack the simplicity needed for being portable over any cloud *storage* service.

Here, we propose a novel system for storing private data in the cloud with the help of a key server (necessary given the requirements). In our system, the user data will be secure from any of: the cloud server, the key server, or any illegitimate users, while the authenticated user can access the data on any devices just via a correct passphrase. The most attractive feature of our system is that it does not require the cloud storage server to support any newly programmable operations, except the existing client login and the data storing. Moreover, our system is simply built on top of the existing App login, and the user only needs one passphrase to login the App and access his secure storage. The security of our protocol, in turn, is proved under our rigorous models, and the efficiency is further demonstrated by real-world network experiments over Amazon S3. We remark that a preliminary variant, based on our principles, was deployed by Snapchat in their *My Eyes Only* module, serving hundreds of millions of users!

1 Introduction

Modern Apps increasingly leverage cloud services to store and manage their clients' data. Email providers, online document programs, and music libraries that synchronize across all kinds of devices are all storing users data in the cloud. Leveraging public cloud storage services such as Google Cloud Storage, Amazon S3, etc, helps companies to design their Apps with great flexibility and scalability without the need to invest in their own infrastructure. However, this also significantly increases the risk of leaking client's information [33]. Indeed, since the data are owned by the users but collected by the App and stored in the cloud, privacy has already been raised as a serious concern for these Apps. It is, thus, natural that users have the right to require that their data are protected, so that accessing either by illegitimate users, or the provider of the App, or even the operators of the cloud server storing the content, is prohibited. Unfortunately, popular cloud storage services either directly store the user data, or keep the decryption key so that the files get decrypted on the server side every time they are accessed.

Given the laudatory success of cryptography research, one may naturally assume that it is a solved problem. However (somewhat unexpectedly), taking different requirements from real-world considerations into account, the problem of augmenting an App with a *secure and usable* cloud storage is still out of reach.

Portability & cloud-blindness. Portability is a desired property since a user may access his content from multiple devices, such as mobile phones, laptops, and desktops. Moreover, there is a possibility that the user loses (or breaks) a device and/or has to move to a new one and recovers the data from storage. To protect user data against cloud, the data stored in cloud should be blind to the cloud. A naive attempt that encrypting data with a strong key and storing it on the user device hinders the basic requirement of user's mobility among devices (portability, which is a major advantage in utilizing the cloud, to begin with!). On the other hand, one may suggest to encrypt with a password (or any corresponding derived key) directly [42] so that a human can remember. Unfortunately, from security point of view, it is a very weak (low entropy) key, thus vulnerable to potential offline dictionary attacks from a corrupted cloud (or whoever breached the cloud).

Such an inherent dilemma between cloud-blindness and portability hints at the fact that the problem of secure (blind) and portable cloud storage is not solvable in the theoretic model with only one storage provider which might be corrupt.[1]However, if we pay a closer look at existing system settings, we see that besides renting public cloud storage servers, most App providers also deploy separate, independently-run servers (corporate or private cloud component) for routine administration such as user management. Therefore, a natural idea is to consider this real-world system model with more than one independent server.

Indeed, cryptographic tools such as password-hardened encryption (PHE) [34]) leverage external crypto services to strengthen the security of password protected storage. In a PHE, the user password will be sent to the storage server and then "hardened" to a strong key with an external server called "rate limiter" via a password hardening service [16, 35, 47]; then the storage server takes this strong key to do encryption and decryption. It is secure against external attackers that may breach the storage server. On the other hand, as an orthogonal concept, the main motivation of PHE (and password hardening) is to provide secure service to a "crypto-free" client thus the storage server does the cryptographic operations including encryption/decryption. Thus the *cloud-blindness* as we wish is beyond the security model of PHE.

Deployablity. From theoretical feasibility point of view, with multiple servers, several cryptographic primitives can satisfy both cloud-blindness and portability, such as general secure multiparty computation (MPC), and password protected secret sharing (PPSS) [5, 10–12, 27–29, 36, 56]. In particular, a (t,n)-PPSS protocol allows a user to store a secret among n servers so that he can later reconstruct it with a password by contacting t servers. While an attacker breaking into up to $t-1$ servers and controlling all communication channels learns nothing about the secret. However, these more theoretical solutions are difficult to be used by current Apps for deployablity issues.

All existing PPSS solutions require specific algebraic operations (on both servers) that bring a major hurdle for a "scalable" deployment. Those operations are not supported at all by the APIs of most commercial cloud *storage* services! Note that plain storage services like Dropbox and Amazon S3 normally provide extremely limited set of APIs. Take Amazon S3 for example, they only provide two types of APIs: storage, retrieval, deletion and access control related functionalities. Therefore, the App providers have no choice but to use cloud *computing* services (such as Amazon EC2), which is dramatically (and unnecessarily) more expensive than using plain cloud *storage* (such as Amazon S3). For example, besides the fixed-rate transfer cost, using Amazon EC2 instances for storage incurs a very large computation cost to keep the virtual machines continuously running, plus about 5 times the storage cost of the cost of a plain storage service like Amazon S3. According to a rough estimation, an App provider with only 1 million users will have to pay around *two million dollars more* if deploying the same secure storage on EC2 instead of S3 for just one month! (See Section. 6 for details of the estimation.)

In this paper, we will consider the architecture with one plain storage server and one App server (sometimes also called key server), shown in Figure 1.

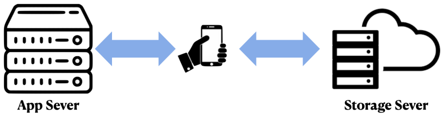

Figure 1: The architecture.

Built-in App login. We may let the client get his password "hardened" to a strong key K via the password hardening service [16, 35, 47] using the App server as the "rate limiter". Then the user uses K to encrypt his content and authenticate himself to the plain cloud server. However, there is one more issue to consider when augmenting an App with a secure and usable cloud storage: since the existing App login (usually password based) is already in place, and in real world, users often reuse their passwords or slight variations [46]. When the App server is corrupted, the login password may be learnt via offline attacks. Then the adversary can use this password to breach into the secure storage system! We thus requires a new design that can let the user reuse the existing App log-in without weakening security.

In a nutshell, while one can always design a protocol for the purpose of portable blind cloud storage, the deployability and built-in App login requirements in a real industrial setting make them, in fact, not as ready to use and augment existing systems. Note that the industrial-context constraints like the economic and engineering overhead are often ignored in the literature, but become particularly essential in upgrading an already heavily used App which cannot suffer down time and lose existing registered users (see Table 1). We, therefore, conclude that we still lack an industrial-system-oriented feasible solution for the basic question:

How to augment Apps with a cloud-secure (blind), portable, and highly efficient storage functionality, easily deploy-able on non-programmable cloud storage services?

[1] Indeed, several works, e.g., Boyen [8] formally showed that any password-protected portable storage between a user and *one* server is always vulnerable to an offline dictionary attack by the malicious server or hackers obtaining the server's database, who holds a "known content" can always offline play the user side.

	PPSS	PHE [34]	OKMS [30]	PBCS
Cloud-blindness	Yes	No	Yes	Yes
Plain cloud storage	No	*	Yes	Yes
Built-in App login	No	*	No	Yes
One pwd	Yes	Yes	No	Yes
Crypto-free client	No	Yes[1]	No	No
Oblivious key update	No	Yes	Yes	No[2]

Table 1: Comparison of different primitives. *Cloud-blindness* means that neither corrupted server can recover the user's data. *Plain cloud storage* means the system could deploy on the non-programmable cloud storage. * means that the property alone is not considered in PHE (beyond its model). *Build-in App login* means the existing App login mechanism for authentication is integrated. *One pwd* means that the user only needs to use one password for the secure storage. *Crypto-free client* means that the App client involves no cryptography except the basic TLS. *Oblivious key update* means that the system update key without client participation.

1.1 Our contributions

We model, design, analyze, implement, and experiment with a novel system, which we name Portable Blind Cloud Storage (PBCS). It aims to achieve the four goals simultaneously: *portability*, *cloud-blindness*, *deployability* and *built-in App login*. PBCS can realize the functionalities of both the App login and blind cloud storage modules with one protocol and be directly deployed on plain cloud storage services like Amazon S3. The user can reconstruct its data and login the App if and only if he contributes a correct passphrase (e.g., successfully log-in). At the core of our PBCS system is a "Give-and-Take protocol" that involves the user with its current device, a data server (plain storage) and a key server (See Fig. 2, Fig. 6 and Fig. 7). More detailed comparisons with previous works could be found in Section 1.4 and Table 1.

Portability: PBCS allows the users to securely access his content and login the App on any device using one passphrase. The security and the usability of the system will be unaffected when some of his devices are lost, as no secret is stored in the user's device. See Sec. 5 for detailed discussions.

Cloud-blindness: PBCS provides an "end-to-same-end" authenticated encryption for the data. Specifically, the data is semantically secure even against a compromised server and illegitimate users who do not follow the protocol. Furthermore, neither server can make the client accept tampered data. See security models and analysis in Sec. 4.2.

Deployability on plain cloud storage: PBCS minimizes the requirements of the data server, so that it can be deployed in existing plain storage services like Amazon S3 or Dropbox. In our protocol, the data server only needs to support password login and basic data storage/retrieve functions. We implement our protocol with simple optimizations. The simplicity and the essentially "negligible" overhead of our PBCS system are showcased by evaluations done in a real network environment with the data server deployed on Amazon S3. See Sec. 6.

Built-in App login: PBCS integrates the App login module, so the user does not need to enter two separate passphrases for the App login and the secure storage module. This not only saves the engineering overhead, but also minimizes the influence on the user experience after the secure storage modular has been augmented. Furthermore, it can prevent security issues when careless users set the two independent passphrases for the login and the storage to be same.

1.2 Technical overview.

To achieve the four goals simultaneously, there are several challenges to resolve. From security point of view, we need to be particularly careful about potential offline attacks. In a very high-level, we let client leverage each server to login another server, in a way that data server only needs to support login, while key/App server login can be reused. More importantly, these logins essentially "force" the adversary to do online attacks (which are easier to defend) instead of offline attacks.

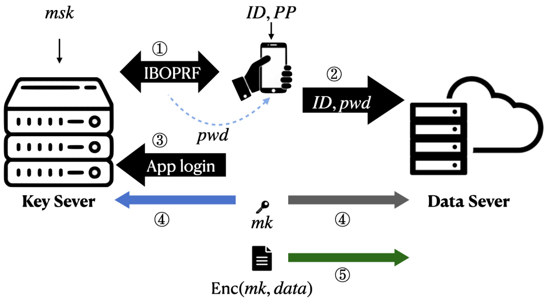

Figure 2: PBCS data flow. 1). the client derives his password by querying the key server with his passphrase. 2). the client logins the data server with his ID and the derived password. 3). the client logins the key server. 4). the client "shares" his master key. 5). the client encrypts his data with the master key and deposit the ciphertext to the data server.

Client authentication to a plain storage server. A computation-barren plain storage, like Dropbox, usually only supports a password login mechanism, it thus rules out PPSS that needs algebraic operations. Our key observation is that the user can leverage the key server to generate and maintain a high entropy "password" to login the data server. Inspired by the password hardening techniques [16, 35], we let the key server provide a service called *identity* based password hardening. This service enables the *client* (instead of server

[1] PHE let the server encrypt and decrypt the message.

[2] PBCS can also achieve the proactive security (updating the encryption key) by the client invoking the "Give-and-Take" protocol via new randomnesses. See Appendix B.

in PHE) to compute a unique pseudo-random string with sufficient length using a low entropy passphrase according to his identity, without revealing the passphrase (See Sec.3 for details). The "hardened" password is then used to register at and later login the data server (Step 1,2 in Fig. 2). Without the correct passphrase, it is impossible to login the data server.

Client authentication with a built-in App login. Since user will mostly do App login anyway, to use it without threatening the security of passphrase used in secure storage, we let the client use only one passphrase and ensure that the key server cannot learn the passphrase via App login. More precisely, the passphrase *PP* will be hardened, while the actual authentication token t for key server login will be replaced with another one that can be generated using *PP* and some randomness s_{id} that is stored in data server. Thus, anyone who can authenticate himself to the key server/login the App must login the data server first (whose authenticator was hardened using *PP* for the same *id* from the key server) and reconstructing the same t with the randomnesses s_{id}. In this way, the existing App login could be seamlessly integrated into our "Give-and-Take" protocol. More importantly, to login the App and access the secure storage is achieved by one protocol with only one passphrase user can remember.

Detecting malicious behaviors. Either corrupt server may not follow the protocol, hence we need to detect malicious behaviors of servers. But due to the deployment requirements, we do not have the luxury of involving non-interactive zero-knowledge (NIZK) proofs like most of the PPSS schemes [5, 10–12, 27, 36, 56] (either the proof or the verification) on the plain cloud storage. In PBCS, we want to provide a lightweight solution by leveraging authenticated encryption. Specifically, we let data be encrypted by an authenticated encryption scheme via a master key *mk*, hence the confidentiality and the integrity of the data is guaranteed if *mk* has not been leaked or tampered. However, we also need to protect *mk* itself via a KEM when a part of the KEM encryption key (some randomness r_{id}) must be barely stored on one of the servers. The security properties of the standard authenticated encryption will not trivially hold when the key is modified. To guarantee the integrity of *mk*, we introduce a *twisted Enc-then-Hash* paradigm, which enables the client (decryptor) to verify that the KEM encryption key has not been modified.

1.3 Extended applications

Moreover, instantiating PBCS in different settings could lead to different interesting new applications. They not only include various applications of PPSS such as *password manager* [5] and *portable cryptocurrency wallet* [28], but also more interesting scenarios, we give three examples below.

Secure personal repository. One of the most suitable settings for our PBCS is to keep personal multimedia data such as pictures and video albums secure in the cloud and available across devices; or upgrading a plain version of a social media App into one having secure storage at the back-end. Indeed, an early version of our protocol was implemented and deployed by Snapchat as their *My Eyes Only* module [7, 31, 49]. In the system, the smartphone transmits encrypted videos to a content server provided by Google Cloud, and sends corresponding tokens to a key server managed by the Snap. The client's videos are kept private to both Google and Snap. PBCS in this paper has significantly improved the preliminary version both on the usability and security. Particularly, we allow the data server to be *plain* cloud storage, and reuses the App log-in requiring only one single password after upgrade.

Secure sync-up system for E2EE messaging Apps. Encrypted messengers like WhatsApp routinely back up users' text messages and contact lists to cloud servers. These backups undo much of the strong security offered by E2EE — since they make it much easier for the companies and hackers to obtain users' plaintext content. Some Apps like Signal chose to provide a secure backup module relying on the trust hardware SGX [37], which may suffer from all kinds of side channel attacks [6, 9, 45, 52]. Instead, PBCS can help users synchronize their contact lists and chat history in an encrypted manner on a third-party cloud server chosen by themselves, while only use the App provider's server as the key management server of PBCS. Therefore the contact list can only be accessed by the user with correct passphrase.

Portable personal AI assistant with privacy guarantee. Nowadays Internet companies strive to provide personalized models for their customers based on the data collected overtime from user devices. Traditional methods involve extracting personal data to the cloud and doing the training on the data set to derive a model, then enabling the cloud server to assist whenever a user logs in to the server and provides new data input. However, at times this raises serious concerns about user privacy, and legal ones such as the EU General Data Protection Regulation (GDPR) [53]. Complying with such regulations while maintaining the utility of the AI assistant is important so learning on device has been suggested: Exemplified in "federated learning" [38]. However, once the user model is computed, restricting it to the single device is limiting. Using our solution, once a personal model is generated, the user can securely store her personal AI assistant model in the cloud, and get access to it via any other device or when changing devices. Note that in our solution, the user plain data never leaves the user devices and the user has full control of his own data and personal AI assistant model which remains private when passing across devices.

1.4 Other related works

Password hardening, introduced in [16] and modeled in [35, 47], aims at enhancing the security of password login with the assistance of an extra server not colluding with the server to be logged-in. A password management system, called SPHINX, was proposed in [48], which could protect the security of password even when password manager has been compromised. This line of works mainly focused on the security of password itself, while we are trying to improve the security of the overall cloud storage module for Apps. Nevertheless, we adopt the techniques in [48] to design an important building block IBOPRF of the PBCS system.

Password-hardened encryption (PHE) [34] is proposed to enhance the security of the password based cloud storage, although it does not achieve the cloud-blindness as we required. In PHE, the storage server takes the user password as input and interacts with an external server called "rate limiter" to "hardened" the password to a strong key, and use it to do encryption and decryption. One advantage of PHE is that the client does not need to process any cryptographic computations except transforming the username and password. Another feature of PHE is that both the crypto server and the storage server can rotate their secret keys to provide proactive security. However, in contrast to PBCS , the storage server in PHE learns both client's decrypted message and password, while in PBCS only the client encrypts and decrypts data.

Another related primitive is the Oblivious Key Management Service (OKMS) proposed in [30], which helps clients manage the secret key via an untrusted storage server. However, OKMS does not meet our deployability requirement, since the design of OKMS did not consider the authentication of the client. One may use two passwords to login into the two servers, which we aim to avoid. It is suggested [30] to instantiate OKMS within Hardware Security Modules (HSMs) [23] to prevent unauthorized queries, but the client-side storage for OKMS makes it non-portable (this also applies to end-to-end hardware modules [23]).

The PPSS [5, 10–12, 27–29, 56] can achieve the portability and the cloud-blindness as we required, since both systems let a user store secret information among multiple servers so that she can later recover the information solely on the basis of her password. However, most of the PPSS designers paid more attentions on the communication round optimization[2] instead of the deloyability, the compatibility and the concrete efficiency. When PPSS is used to augment an App with secure storage, the users need to remember one additional password to login the App. More importantly, PPSS cannot be deployed on a plain cloud storage which is the main stream (and cheaper) commercial cloud product. Another difference comes from the fact that the initiation phrase of PPSS requires authenticated channels between the user and the honest servers, but PBCS always allow the adversary to impersonate the legitimate user when depositing the content.

Very recently, Dauterman et al., [14] proposed a system for encrypted mobile-device backups, named SafetyPin. SafetyPin requires users to remember only a short PIN and defends against brute-force PIN-guessing attacks using hardware security protections. Since SafetyPin splits trust over a cluster of hardware security modules (HSMs), it can protect backed-up user data even against an attacker that can adaptively compromise many of the system's constituent HSMs. However, SafetyPin highly relies on the HSM cluster of the cloud, while our PBCS aims to be deployed on any commercial storage cloud. We summarize the detailed comparison in Table 1.

2 Preliminaries

Login mechanisms. Most cloud storage service providers deploy a login mechanism to authenticate their clients. In practice, such login mechanism can be instantiated via a password, a biometric like the fingerprint, a token from a third party authentication protocol like OAuth [20, 21], the recent Universal Second Factor protocol [51], or other types of authenticators from the user via her software to the cloud server. Without loss of generality, a login mechanism can be abstracted as the register algorithm $\mathsf{AuthReg}$ and the login algorithm Login:

- $\mathsf{AuthReg}(id, \alpha_{id})$: Client $\mathcal{U}$ with identity id registers to server $\mathcal{D}$ using an authenticator α_{id} which can be a password, a token, etc. After the registration, client $\mathcal{U}$ keeps α_{id}, and server $\mathcal{D}$ gets the stub β_{id} for later login verification.
- $\mathsf{Login}(id, \alpha_{id}, \beta_{id})$: After registration, client $\mathcal{U}$ authenticates himself to the server $\mathcal{D}$ by presenting his id and the authenticator α, and server $\mathcal{D}$ uses the stored stub β_{id} to verify the pair (id, α) and output a bit b to denote *success* or not.

For simplicity, we assume the communications in register and login are all protected by a secure channel, hence cannot be seen or altered by the adversary (in fact, pinned certificates and universal second factor plus password authenticators facilitate this situation).

```
Game $Bypass^{\mathcal{A}}$
    $id^* \leftarrow\$ \mathcal{A}$
    $\alpha_{id^*} \leftarrow\$ \chi$
        // Choose $\alpha^*$ from distribution $\chi$
    $\beta_{id^*} \leftarrow\$ \mathsf{AuthReg}(id^*, \alpha_{id^*})$
    $\alpha' \leftarrow\$ \mathcal{A}^{\mathcal{O}_{\mathsf{AuthReg}}, \mathcal{O}_{\mathsf{Login}}}$
    $\mathsf{Login}(id^*, \alpha', \beta_{id^*}) = b$
    return $b$

Oracle $\mathcal{O}_{\mathsf{AuthReg}}(id, \alpha)$
    $\beta \leftarrow\$ \mathsf{AuthReg}(id, \alpha)$
    return $\beta$

Oracle $\mathcal{O}_{\mathsf{Login}}(\alpha)$
    if $count > B$
        return $\perp$
    else
        $\mathsf{Login}(\alpha, \beta_{id^*}) = b'$
        $count = count + 1$
        return $b'$
```

Figure 3: The security game of the login scheme.

Here we define the formal properties of the Login mechanism. The login mechanism should satisfy:

[2] Although some PPSS [1, 27] may save 2 rounds than our PBCS .

- *Correctness:* If β_{id} is generated from the $\mathsf{AuthReg}$ algorithm with id and α_{id}, the Login procedure with id, α_{id} and β_{id} will always succeed.
- ε-*Security:* Without α_{id^*} or β_{id^*}, the probability that an adversary $\mathcal{A}$ outputs an authenticator α' that can pass the verification with β_{id^*} within B login attempts is less than ε. Here the upper bound B is specified by the scheme. Formally, a login scheme $(\mathsf{AuthReg}, \mathsf{Login})$ is ε-secure if the probability of the adversary wining the Bypass game in Fig. 3 is less than ε.

3 Identity Based Oblivious PRF

The identity based oblivious pusedorandom function (IBOPRF) is an important primitive underlying our PBCS protocol to insist on a single password. Here we formalize its syntax and properties to adapt to our PBCS design (though our construction adopts the techniques used in the password manager SPHINX [48]). We observe that most websites' and Apps' login systems enable the user to choose a really long password which is more than 16 characters, but the user usually cannot remember such a long password. Inspired by the oblivious pseudorandom function (OPRF) [17,27,44] and the password hardening service [16,35], we allow the user to leverage the IBOPRF service to generate a long and high-entropy login password from the short passphrase people can remember.

Different from the traditional OPRF, the IBOPRF is a protocol between one server and *multiple* clients. The server holds a master key msk while each client holds an unique identity id. Different from the password hardening service in [16,35], IBOPRF is a service that faces the end-users directly, instead of a three party protocol between the end users, the client (which may be a web server that performs password-based authentication of end users) and a hardening service provider. It is also different from the existing industrial password manager service like the Chrome [22] and the iCloud Keychain [24] ones, the IBOPRF server itself will not learn the user's passphrase or the login passwords.

The IBOPRF server computes a specific PRF key k_{id} for each identity id, and let the client obliviously compute the PRF value $\mathcal{F}_{k_{id}}(x)$ on his input x via interactions with the server. Later $\mathcal{F}_{k_{id}}(x)$ can be encoded into a long password. After the communication, on the one hand, the server cannot learn the client's input x nor predict the final output $\mathcal{F}_{k_{id}}(x)$. On the other hand, any client cannot compute the correct PRF value without communicating with the server.

The IBOPRF is a challenge-response protocol between a client and a server with the following syntax.

- $\mathsf{Setup}(1^\lambda) \to (pp, msk)$: Given the security parameter λ, generate the public parameter pp and the server's master secret key msk.

- $\mathsf{CEval}_1(pp, id, x) \to (ch, st)$: When inputs the public parameter pp, identity id and a secret input passphrase x, the client computes a challenge ch and an internal state st.

- $\mathsf{SEval}(pp, id, ch, msk) \to rp$: When the server receives an identity id and a challenge ch from the client, the server computes a response r according to the public parameter pp and his master secret key msk.

- $\mathsf{CEval}_2(pp, id, rp, st) \to y$: When the client receives the response rp from the server, he will retrieve the internal state st and compute the function output password y according to the public parameter pp and the identity id.

Properties. The IBOPRF should guarantee the following properties (Formal definitions can be found in Appendix A):

Uniqueness: If all parties follow the protocol, the client will learn a unique output $y = \mathcal{F}_k(id, x)$, i.e., the client will never output $y' \neq y$ for same id and x.

Pseudorandomness captures the security of IBOPRF against other malicious clients who did not corrupt the server but can arbitrarily query the server. It guarantees that those clients cannot distinguish $y = \mathcal{F}_k(id^*, x^*)$ from a random string r for a chosen identity id^* and a secret input x.

Obliviousness models the security of IBOPRF against the malicious server. It guarantees that a malicious server cannot predict y for a fixed id and an unknown input x, even if it can interact with the honest client holding id and x multiple times.

Construction. The IBOPRF could easily be constructed in the random oracle model. Our design is inspired by the 2HashDH protocol in [28]. In Appendix A, we give formal analysis for the above construction.

- $\mathsf{Setup}(1^\lambda)$: Choose a prime p which is λ bits large. The input passphrase space and the output password space are within $\{0,1\}^k$ and $\{0,1\}^\lambda$, respectively. The hash function H_1 is from $\{0,1\}^k$ to $\mathbb{Z}_p$. The hash function H_2 is from $ID \times \{0,1\}^\lambda$ to $\mathbb{Z}_p^*$. The hash function H_3 is from $\{0,1\}^k \times \mathbb{Z}_p$ to $\{0,1\}^\lambda$. Form the public parameter $pp = (p, \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3)$. Pick the master secret key $msk \leftarrow\$ \{0,1\}^\lambda$.

- $\mathsf{CEval}_1(pp, id, x)$: On input the identity id and the passphrase x, the user picks $st \leftarrow\$ \mathbb{Z}_p^*$ as the internal state and sends $ch = \mathsf{H}_1^{st}(x)$ to the server.

- $\mathsf{SEval}(id, ch, msk)$: Given the identity id, the server computes the client specific PRF key $k_{id} = \mathsf{H}_2(id, msk)$. Then the server generates the response $rp = ch^{k_{id}}$.

- $\mathsf{CEval}_2(id, rp, st)$: On message rp from the server, the client verifies $rp \in \langle g \rangle$. If the test passes, then the client retrieves the secret state st and returns $y = \mathsf{H}_3(x, rp^{1/st})$.

4 Architecture and Definition

As we explained in the introduction, we aim to upgrade an existing App (having a plain cloud storage) with a secure storage function while minimize the influence on usability. Therefore, the PBCS fully leverages the existing infrastructure of a typical App. Specifically, a typical App [25,31,40] (maybe without the secure storage) may use a manager server to administrate its service. To provide more storage space for each client, the App usually registers and maintains an account on the cloud server for each individual user. When the user wants to deposit or retrieve his large files, the App will help him to login to his cloud storage account via his identity and the password. After login, the client can freely deposit data to or retrieve data from the cloud. To be compatible with the existing App architecture, the PBCS system involves three parties: a client $\mathcal{U}$ (or user) who deposits/retrieves data, a data server $\mathcal{D}$ (cloud storage server) which stores the encrypted data, and a key server $\mathcal{K}$ (administrative server) which offers key management services (See Fig. 2).

As in standard practice, the PBCS system consists of two parts, i.e., the key encapsulation mechanism (KEM) part which lets a client distributively generate and store a strong master key *mk* with the help of the two servers; and the data encapsulation mechanism (DEM) part to encrypt the actual content with *mk*. The encryption of the DEM part can be easily instantiated through any standard authentication encryption; and the confidentiality and integrity of the content depends on the security of the master key. In the following, we will focus on the KEM part as the DEM part can be trivially augmented. For a concrete example walking through the whole system, we refer to Sec. 5.3.

4.1 Syntax of the KEM

Our PBCS fully leverages the login mechanism of the servers. Specifically, the KEM part of PBCS consists of three procedures: *Register*, *Give* and *Take*. The *Register* procedure enables a client with identity *id* and passphrase *PP* to register an account on the cloud server with the help of the key server. In the *Give* procedure, the client first logins his account on the cloud server, and chooses a master key *mk*, then distributively deposits *mk* to the cloud server and the key server. In the *Take* procedure, the client retrieves the master key *mk* by logining the cloud server and interacting with the two servers via *PP*. Note that the client only needs to remember the passphrase *PP* during all procedures. We first give a formal definition.

Definition 1 *A KEM part of our Portable Blind Cloud Storage system is a tuple of interactive procedures* $(\mathsf{Register}, \mathsf{Give}, \mathsf{Take})$ *after setup, each of which is meant to be run among three parties (modeled as interactive Turing machines): a user* $\mathcal{U}$*, a key server* $\mathcal{K}$ *and a data server* $\mathcal{D}$*. Each of them has three subroutines, i.e.,* $(\mathcal{U}_{\mathsf{Register}}, \mathcal{U}_{\mathsf{Give}}, \mathcal{U}_{\mathsf{Take}})$*,* $(\mathcal{K}_{\mathsf{Register}}, \mathcal{K}_{\mathsf{Give}}, \mathcal{K}_{\mathsf{Take}})$ *and* $(\mathcal{D}_{\mathsf{Register}}, \mathcal{D}_{\mathsf{Give}}, \mathcal{D}_{\mathsf{Take}})$ *for each procedure* Register*,* Give *and* Take*. In a PBCS system, the key server* $\mathcal{K}$ *and the data server* $\mathcal{D}$ *will maintain their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$*, respectively:*

Setup*: The key server and the data server generate their public parameters* $pp_{\mathcal{K}}$ *and* $pp_{\mathcal{D}}$*, and secret parameters* $sp_{\mathcal{K}}$ *and* $sp_{\mathcal{D}}$*, respectively. Moreover, the servers will initiate their internal states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$*.*

Register*: The client chooses his id and a passphrase PP. Given the public parameters* $pp_{\mathcal{K}}$ *and* $pp_{\mathcal{D}}$*, the client will interact with the two servers and create an account on the cloud server. If succeeds, the two servers will update their states* $s_{\mathcal{K}}, s_{\mathcal{D}}$*, accordingly.*

Give*: The client takes his id, the passphrase PP and the servers' public parameters* $pp_{\mathcal{K}}, pp_{\mathcal{D}}$ *as inputs. The servers take their states* $s_{\mathcal{K}}, s_{\mathcal{D}}$ *and secret parameters* $sp_{\mathcal{K}}, sp_D$ *as inputs, respectively. If succeeds, the client obtains a randomly generated mk and the servers update their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$ *incorporating the shares regarding mk respectively.*

Take*: In this procedure, the client retrieves the stored master key mk, which would be used later in the DEM part. The client inputs id, PP and the servers' public parameters* $pp_{\mathcal{K}}$ *and* $pp_{\mathcal{D}}$*, while the servers input their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$ *and secret parameters* $sp_{\mathcal{K}}$ *and* sp_D*, respectively. If succeeds, the servers update their states* $s_{\mathcal{K}}$ *and* $s_{\mathcal{D}}$ *respectively.*

4.2 Security Threats and Models

The PBCS system should guarantee both the confidentiality and integrity of the stored data against the illegitimate users, the corrupted key server, or the corrupted data server. Particularly, we require that :

- The illegitimate user without the correct passphrase can not learn the storage data, nor let the user accept forged data;

- If any one of the servers is malicious, he can not learn the storage data nor let the user accept forged data even if he does not follow the protocol;

- Even both servers are corrupted, the security of data falls back the best possible security in the single-server setting.

However, the denial of service attacks are out of scope of our security model. Furthermore, PBCS is designed for the running environment where only the servers have the certificates issued by PKI, but the clients do not. Accordingly, a sever-only authenticated and confidential channel [26, 32] can be established between the honest users and servers via TLS. So the adversary is allowed to impersonate any client in front of the server, but can not read or alter the communication between the honest client and the honest server.

Security Models. When the DEM part is instantiated via a standard authenticated encryption, the confidentiality and the integrity of the content in PBCS depend on the security of the master key, so here we only consider the formal security models for the KEM part. Now we will provide models to capture the master key's confidentiality and integrity.

We provide four security properties to formalize the confidentiality of the master key against illegitimate users, against either compromised key server (or data server) and even against two compromised servers. It is not hard to see that security against illegitimate users is straightforwardly implied by the security against a compromised key (or data) server since the server himself can disguise as an illegitimate user; also security in the case that both servers got compromised

```
IND-CKS Game
1 :  $b \leftarrow\$ \{0,1\}$
2 :  Generate the data server's $(pp_{\mathcal{D}}, sp_{\mathcal{D}})$
3 :  $pp_{\mathcal{K}} \leftarrow\$ \mathcal{A}$
4 :  $\mathcal{A}$ arbitrarily switches in two modes:
Mode 1: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{A}(), \mathcal{D}_{\mathsf{Register}}(sp_{\mathcal{D}}, s_{\mathcal{D}}) \rangle$
        / $\mathcal{A}$ can register on $\mathcal{D}$ with arbitrary identity $id$
Mode 2: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \left\langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \right\rangle$
        / $\mathcal{A}$ interacts with $\mathcal{D}$ while pretending as $\mathcal{K}$ or clients with any $id$
        / $\mathcal{A}$ assigns $\mathcal{D}$ to execute any procedures $(\mathsf{Give}, \mathsf{Take})$
5 :  $id^* \leftarrow\$ \mathcal{A}(\cdot)$    / The adversary chooses the challenge user $id^*$
6 :  $PP_{id^*} \leftarrow\$ \mathcal{C}$    / Choose the passphrase for the challenge user.
7 :  $\mathcal{A}$ arbitrarily switches in three modes:
Mode 1: $(\cdot, s_{\mathcal{D}}, \cdot) \leftarrow\$$
$\langle \mathcal{U}_{\mathsf{Give}}(id^*, PP_{id^*}, pp_{\mathcal{K}}, pp_{\mathcal{D}}), \mathcal{D}_{\mathsf{Give}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \rangle$
        / $\mathcal{A}$ interacts with $\mathcal{D}$ and $\mathcal{U}_{id^*}$ arbitrarily as $\mathcal{K}$ and views their responses
        / $\mathcal{D}$ and $\mathcal{U}(id^*, PP_{id^*})$ only execute the Give procedure
        / At this stage $\mathcal{U}$ has not generated $mk$ successfully yet
Mode 2: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \left\langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \right\rangle$
        / Similar to Mode 2 in Step 4
        / $\mathcal{A}$ may also pretend to be $\mathcal{U}_{id^*}$ but without knowing $PP_{id^*}$
Mode 3: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{A}() \leftrightarrows \mathcal{D}_{\mathsf{Register}}(sp_{\mathcal{D}}, s_{\mathcal{D}}) \rangle$
        / $\mathcal{A}$ can register $\mathcal{D}$ with arbitrary identity $id \neq id^*$
8 :  $(mk_0, s_{\mathcal{D}}, \cdot) \leftarrow\$$
$\langle \mathcal{U}_{\mathsf{Give}}(id^*, PP_{id^*}, pp_{\mathcal{K}}, pp_{\mathcal{D}}), \mathcal{D}_{\mathsf{Give}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \rangle$
9 :  $mk_1 \leftarrow\$ \{0,1\}^*$
10 : $\mathcal{A} \leftarrow mk_b$
11 : $\mathcal{A}$ arbitrarily switches in three modes:
Mode 1: $(mk', s_{\mathcal{D}}, \cdot) \leftarrow\$$
$\langle \mathcal{U}_{\mathsf{Take}}(id^*, PP_{id^*}, pp_{\mathcal{K}}, pp_{\mathcal{D}}), \mathcal{D}_{\mathsf{Take}}(s_{\mathcal{D}}), \mathcal{A}() \rangle$
            if $b = 0 \wedge mk' \neq \perp$    then $mk^* = mk'$
            if $b = 1 \wedge mk' \neq \perp$    then $mk^* = mk_1$
            if $mk' = \perp$    then $mk^* = \perp$
            $\mathcal{A} \leftarrow mk^*$    / The adversary learns $mk^*$
        / $\mathcal{A}$ interacts with $\mathcal{D}$ and $\mathcal{U}_{id^*}$ arbitrarily as $\mathcal{K}$ and views their responses
Mode 2: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \left\langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}() \right\rangle$
        / Similar to Mode 2 in Step 4
        / $\mathcal{A}$ may also pretend to be $\mathcal{U}_{id^*}$ but without knowing $PP_{id^*}$
Mode 3: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{A}() \leftrightarrows \mathcal{D}_{\mathsf{Register}}(sp_{\mathcal{D}}, s_{\mathcal{D}}) \rangle$
        / $\mathcal{A}$ can register $\mathcal{D}$ with arbitrary identity $id$
12 : return $b = b'$ where $b' \leftarrow\$ \mathcal{A}()$
```

Figure 4: The IND-CKS game. The challenger $\mathcal{C}$ simulates the data server $\mathcal{D}$ and the target client $\mathcal{U}_{id^*}$.

falls back to the single server case. We defer more security discussions to our full version. In the main body, we consider the situation that the adversary corrupts the key management server and other users (with identities different from the victim), and propose the IND-CKS security (here CKS denotes Compromised Key Server) in Def. 2. Similarly, we can model security against the compromise data server, which we call IND-CDS security (CDS denotes Compromised Data Server), by switching the role of $\mathcal{K}$ and $\mathcal{D}$. Note that our PBCS system provides a similar level of the security as PPSS, i.e., an attacker breaking into any one of these servers learns nothing about the secret (or the password).

Our IND-CKS experiment is similar to the game-based definition of PPSS in [27]. Intuitively, during the IND-CKS experiment (Fig.4), the challenger $\mathcal{C}$ simulates data server $\mathcal{D}$ and the challenge client (with identity id^*), while the adversary $\mathcal{A}$ plays the roles of the key server $\mathcal{K}$ as well as other clients. $\mathcal{A}$ can arbitrarily invoke the $(\mathsf{Register}, \mathsf{Give}, \mathsf{Take})$ procedures. Also she can adaptively register new accounts on the data server. The challenger simulates the procedure that the challenged client deposits a master key. The adversary, who controls the corrupted key server and deviates the protocol, aims to distinguish this master key from a random key. A slight difference of the IND-CKS experiment with the PPSS [27,28] is that PPSS requires the adversary cannot impersonate the honest client in front of the honest server in the initiation phrase, but the IND-CKS experiment the adversary can impersonate the target client in front of servers. More precisely, we have the following definition.

Definition 2 (Master key confidentiality against the compromised key server.) *Consider the following interactive game* $\mathit{Exp}_{\mathcal{A}}^{\mathsf{IND\text{-}CKS}}$ *in Fig. 4 between an adversary* $\mathcal{A}$ *and a challenger* $\mathcal{C}$*, parametrized by security parameter* λ *and a bit* b.[3] *Let* n *denote the min-entropy of the passphrase* PP_{id^*}.[4] *We say that a PBCS scheme is* secure against the compromised key management server *if there exists a negligible function s.t.* $\forall$ *P.P.T adversary* $\mathcal{A}$ *it holds that*

$$\Pr\left[\mathit{Exp}_{\mathcal{A}}^{\mathsf{IND\text{-}CKS}}\left(1^{\lambda}, 1^{n}\right) = 1\right] \leq \frac{1}{2} + \mathsf{Adv}(n) + \mathit{negl}(\lambda)$$

where $\mathsf{Adv}(n)$ *is the ideal security inherited for guessing* PP_{id^*}*, which is* $O\left(\frac{1}{2^n}\right)$*.To make the model meaningful, we assume that the adversary will cause a certain number (polynomially bounded) of failures, but the Give procedure of the challenge client will eventually succeed and a master key will finally be generated.*

[3] For two P.P.T interactive algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$, we denote by $(a,b) \leftarrow\$ \langle \mathcal{P}_1(x), \mathcal{P}_2(y) \rangle$ the event that $\mathcal{P}_1$ and $\mathcal{P}_2$ engage in an interactive protocol with $\mathcal{P}_1$'s input x and $\mathcal{P}_2$'s input y, and produce local outputs a and b, respectively. Similarly, we denote by $(a,b,c) \leftarrow\$ \langle \mathcal{P}_1(x), \mathcal{P}_2(y), \mathcal{P}_3(z) \rangle$ for the corresponding three-party interaction among $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$.

[4] Note that n may not directly equal to the length of the passwords, since the distribution of passwords is not uniform [54,55].

Our definition aims at capturing the *soundness* property as PPSS [27, 28] as well. The soundness means that one malicious server cannot make the user to accept a tampered master key in the Take procedure. Therefore, the adversary in our model not only can learn a master key mk_b after a successful Give (mk_b could be a real key mk_0 or a random key mk_1 according the flip coin b), but also is given a master key version mk^* after a successful Take procedure. If the scheme does not satisfy the soundness, the malicious adversary could make the client get a tempered master key which not equals to the one generated in the Give procedure, and then the adversary can distinguish his view is the real key or the random one by comparing mk_b and mk^*.

5 Construction

We now describe the details of the KEM part of our PBCS. Since the DEM part which carries the actual content can be augmented trivially, the KEM part becomes the most challenging design.The first challenge of designing our PBCS system lies on how to derive a simple yet client authentication solution which could be deployed on *non-programmable* cloud storage service. The second challenge is to integrate the App login mechanism in the secure storage module, so the user can leverage one password to login the App and access the secure content. Moreover, we also need to be careful that the malicious server may intentionally modify the storage data.

Simple cloud server authentication. To avoid heavy primitives that cloud storage APIs may not support, we need a simple way for the cloud server to authenticate the client. Note that we can not let the user directly use his passphrase to login the cloud server, since this passphrase will also be used to login the App and must be hidden to the cloud server. However, we observe that the existing login mechanism usually supports a very long password, which could be more than 16 characters and decoded into a 128 bits length string. Although it is crazy to require a person to remember such a long password, the client can generate the long password from a short passphrase via an IBOPRF service provided by the key server. Hence the user only needs to remember a short passphrase instead of the long password. Moreover, although the IBOPRF service is public to all clients as well as the data server (since to compute the password hardened value does not need to login in advance), the pseudorandomness guarantees that the low entropy passphrase is still kept secret to the (malicious) data server when he blindly queries the IBOPRF limited times with the same identity. The obliviousness of the IBOPRF guarantees that the key server also cannot learn the passphrase. Compromising one server does not help to authenticate to the other. Since our "Give-and-Take" protocol only involves the basic account register/login and TLS on the cloud server side, it can be directly deployed on a commercial cloud *storage* services like Dropbox or AWS S3.

Integrating App login into PBCS is not only the requirement of minimizing the influence on the user experience, but also important for the security. Indeed, users often reuse their passwords or use slight variations [46] on different services. To solve this problem, the user can not use the remembered passphrase to login the key server directly, since this passphrase is also used to login the cloud server and needed to be hidden to the key server. Our idea is to make the App login to be naturally accomplished in the meantime of that the "Give" or "Take" protocol is executed. Precisely, the client will generate an authentication token as the App login password from a high entropy randomness s_{id} and the passphrase PP. The high entropy randomness s_{id} is stored on the data server. In the "Give" or "Take" protocol, the client will authenticate himself to the key server by directly inputting this authentication token to the existing App login module[5], hence the client could simultaneously login the App. Moreover, since a compromised key server can not learn the passphrase, he can not login the data server and break the security of the private content.

Integrity guarantee. When the master key is fixed, the integrity of the content can be trivially protected by the authenticate encryption used in the DEM part. However, a malicious server may intentionally violate the protocol and let the client accept a modified master key. This is out of the scope of the security property of the traditional authentication encryption. To guarantee the integrity of the master key against a malicious key server, one may suggest to use authentication encryption to encrypt the master key mk. When the key server keeps the authentication ciphertext of mk instead of a piece of mk shares. However, the encryption key k of mk is stored on the data server and can be modified. To solve this problem, we let k be derived from the the randomness r_{id} stored by the data server and invent a novel method to verify that r_{id} is not modified by the malicious data server. Specifically, we tear the encryption key into two parts: k_1 and k_2. k_1 and k_2 are both derived from r_{id} and the passphrase PP but via two different random oracles KDF_2 and KDF_3. k_1 is used to encrypt mk and get the ciphertext ct, while k_2 is used to derive a HMAC with the form $\tau = \mathsf{H}_4(ct, k_2)$. Both ct and τ are stored on the key server side. If the data server provides a tampered r'_{id}, the client will generate a wrong k'_2. So when the key server sends back the tag τ, a false randomness r'_{id} will lead the tag verification fails, i.e., $\tau \neq \mathsf{H}_4(ct, k'_2)$. Note that the tuple of the IND-CPA secure ciphertext ct and the token τ can also be viewed as an authentication encryption ciphertext since it follows the Enc-then-Mac paradigm.

[5] The App login password is encoded from the authentication token which is a long bit string.

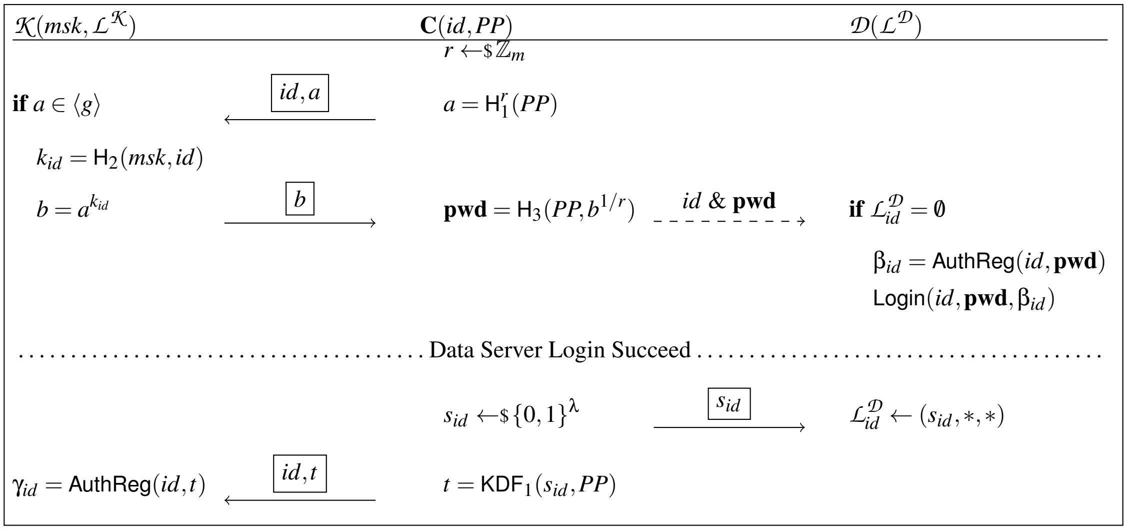
$\mathcal{K}(msk, \mathcal{L}^{\mathcal{K}})$
$\mathbf{C}(id, PP)$
$\mathcal{D}(\mathcal{L}^{\mathcal{D}})$
$r \leftarrow \$ \mathbb{Z}_m$
if $a \in \langle g \rangle$
$\boxed{id, a}$
$a = \mathsf{H}_1^r(PP)$
$k_{id} = \mathsf{H}_2(msk, id)$
$b = a^{k_{id}}$
$\boxed{b}$
$\mathbf{pwd} = \mathsf{H}_3(PP, b^{1/r})$
id & $\mathbf{pwd}$
if $\mathcal{L}_{id}^{\mathcal{D}} = \emptyset$
$\beta_{id} = \mathsf{AuthReg}(id, \mathbf{pwd})$
$\mathsf{Login}(id, \mathbf{pwd}, \beta_{id})$
Data Server Login Succeed
$s_{id} \leftarrow \$ \{0,1\}^{\lambda}$
$\boxed{s_{id}}$
$\mathcal{L}_{id}^{\mathcal{D}} \leftarrow (s_{id}, *, *)$
$\gamma_{id} = \mathsf{AuthReg}(id, t)$
$\boxed{id, t}$
$t = \mathsf{KDF}_1(s_{id}, PP)$

Figure 5: The Register Procedure. The client derives the password **pwd** from the IBOPRF with his passphrase *PP* and uses it to run AuthReg to register to the data server who obtains and stores the login sub β_{id} for the future login verification. Also the client derives the authentication token t from *PP* and a randomness seed s_{id}. Later s_{id} will be deposited to the data server. t, as the App login password, will be registered to the key server who obtains and stores the login stub γ_{id} for the future login verification. The boxed message means they are protected by TLS. The security requires the number of calling the IBOPRF service for one specific *id* on the key server side is bounded.

5.1 Construction details

Following the guidelines discussed above, we can design the three procedures of our "Give-and-Take" protocols as follows. During all procedures, the client will use TLS to protect the communications. Here we assume the cloud server and the App's login mechanisms have already deployed countermeasures to prevent the on-line dictionary attacks.

- **Register:** The key server holds a IBOPRF master secret key *msk*. The client chooses an identity *id* and a memorable passphrase *PP*. The key server and the data server will maintain a tuple $\mathcal{L}_{id}^{\mathcal{K}} \in \mathcal{L}^{\mathcal{K}}$ and $\mathcal{L}_{id}^{\mathcal{D}} \in \mathcal{L}^{\mathcal{D}}$ for each identity *id*, respectively. KDF_1 is a key derivation function as a random oracle. β_{id} and γ_{id} are login stubs for the data server and the key server, respectively. When the IBOPRF is instantiated via 2HashDH in Sec. 3, the register procedure is as Fig. 5.

- **Give:** When the protocol starts, the client with identity *id* holds the passphrase *PP*. The data storage server $\mathcal{D}$ holds the login stubs β_{id}, and maintains a list $\mathcal{L}^{\mathcal{D}}$ to record randomnesses. The key server $\mathcal{K}$ keeps a list $\mathcal{L}^{\mathcal{K}}$, the login stubs γ_{id} and his master secret key *msk*. Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be the algorithms of a symmetric encryption scheme with the ciphertext space $\mathcal{C}$. Let the function $y = \mathcal{F}_{msk}(id, x)$ be the IBOPRF computed by the key server as defined in Sec. 3. KDF_1, KDF_2 and KDF_3 are three key derivation functions which could be modeled as random oracles. H_4 is a hash function from $\mathcal{C} \times \{0,1\}^{\lambda} \to \{0,1\}^{\lambda}$ which could also be modeled as random oracles. The detailed Give procedure is demonstrated pictorially in Fig.6 when the IBOPRF is instantiated via 2HashDH as Sec. 3.

- **Take** is to let the client retrieve the master key from two servers. Take protocol will use the same primitives as Give. When Take starts, the client with identity *id* holds passphrase *PP*. The servers $\mathcal{D}$ and $\mathcal{K}$ each holds a tuple $(\beta_{id}, \mathcal{L}^{\mathcal{D}})$ and $(msk, \gamma_{id}, \mathcal{L}^{\mathcal{K}})$ respectively. The detailed Take procedure is demonstrated pictorially in Fig. 7, where the IBOPRF is instantiated by the 2HashDH in Sec. 3. Note that the number of the call of the IBOPRF service for one specific *id* on the key server side must be limited to guarantee the security.

5.2 Security

The security of PBCS can be founded in following theorems. The analysis will appear in the full version.

To show the master key is confidential to the data server, we argue that the only approach for the adversary to access ct_{id^*} via $\mathcal{K}$ is to successfully recover the authentication token t_{id^*}. Since the KDF_1 is a random oracle, the adversary must guess the passphrase *PP*. Due to the pseudorandomness of the IBOPRF, **pwd** will not leak the information about *PP* within limited time of invoking the IBOPRF service, so the adversary has to guess *PP* blindly. Moreover, the client will not accept a tampered master key, since the client could verify

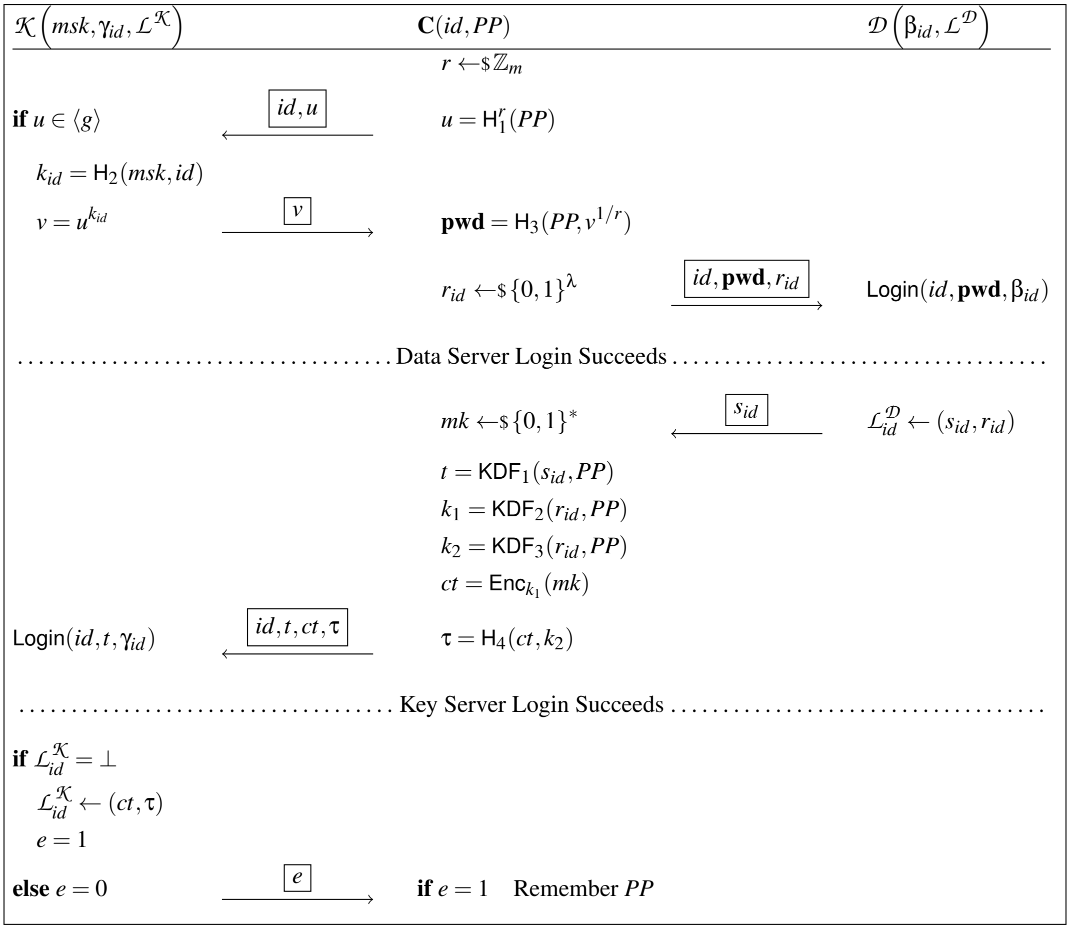
$\mathcal{K}\left(msk, \gamma_{id}, \mathcal{L}^{\mathcal{K}}\right)$
$\mathbf{C}(id, PP)$
$\mathcal{D}\left(\beta_{id}, \mathcal{L}^{\mathcal{D}}\right)$
$r \leftarrow\$ \mathbb{Z}_m$
if $u \in \langle g \rangle$
$\boxed{id, u}$
$u = \mathsf{H}_1^r(PP)$
$k_{id} = \mathsf{H}_2(msk, id)$
$v = u^{k_{id}}$
$\boxed{v}$
$\mathbf{pwd} = \mathsf{H}_3(PP, v^{1/r})$
$r_{id} \leftarrow\$ \{0,1\}^{\lambda}$
$\boxed{id, \mathbf{pwd}, r_{id}}$
$\mathsf{Login}(id, \mathbf{pwd}, \beta_{id})$
Data Server Login Succeeds
$mk \leftarrow\$ \{0,1\}^*$
$\boxed{s_{id}}$
$\mathcal{L}_{id}^{\mathcal{D}} \leftarrow (s_{id}, r_{id})$
$t = \mathsf{KDF}_1(s_{id}, PP)$
$k_1 = \mathsf{KDF}_2(r_{id}, PP)$
$k_2 = \mathsf{KDF}_3(r_{id}, PP)$
$ct = \mathsf{Enc}_{k_1}(mk)$
$\mathsf{Login}(id, t, \gamma_{id})$
$\boxed{id, t, ct, \tau}$
$\tau = \mathsf{H}_4(ct, k_2)$
Key Server Login Succeeds
if $\mathcal{L}_{id}^{\mathcal{K}} = \perp$
$\mathcal{L}_{id}^{\mathcal{K}} \leftarrow (ct, \tau)$
$e = 1$
else $e = 0$
$\boxed{e}$
if $e = 1$ Remember PP

Figure 6: The Give Procedure. The client logins the data server via the password **pwd** derived with the key server, where the data server stores the login stub β_{id} for the login verification. Then the client reconstructs the authentication token t and the key of the authentication encryption k_1 and k_2. The client uses id, t to login the key server who stores the login stub γ_{id} for login verification, and deposits the ciphertext of the master key ct and the corresponding tag τ to the key server. The boxed message means that they are protected by TLS. H_1, H_2 and H_3 are hash functions used in the 2HashDH IBOPRF defined in Sec. 3. The security requires the number of the call of the IBOPRF service for one specific id on the key server side is bounded.

the randomness r_{id} replied by the data server via the hash function H_4 and KDF_3. Formally, we have:

Theorem 1 *Let KDF_1 be a random oracle. The min-entropy of the passphrase PP is d. The IBOPRF is (ε, d, B)-pseudorandomness. The total number of the adversary calling IBOPRF service is bounded by B. The total number of the invalid APP login is bounded by $B_{\mathcal{K}}$. PBCS is secure against compromised data server, i.e., the probability for any adversary to win the IND-CDS game is less than $1/2 + \varepsilon + B_{\mathcal{K}}/2^d + \mathsf{negl}(\lambda)$.*

To get access to the encryption key k_{id^*} on the data server, the adversary has only two choices. The first is to authenticate to the data server, which means to successfully recover the password **pwd**. However, pseudorandomness of IBOPRF guarantees **pwd** cannot be guessed by the key server. The second is to blindly guess r_{id^*}, whose probability is negligible. Hence the master key is confidential to the malicious key server. Moreover, since the IND-CPA ciphertext ct and the tag τ from an authenticated encryption of the master key mk. Hence the malicious key server can not make the forged tuple (ct', τ') to be accepted by the client. Formally, we have:

Theorem 2 *Let $(\mathsf{Register}, \mathsf{Login})$ be a Adv_{Auth}-secure login mechanism (as in Sec. 2). Let χ be the distribution of input x, d be its min-entropy. The adversary makes q attempts to login the data server. The IBOPRF is (ε, d, q)-obliviousness. Let KDF_2 be a random oracle. Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a secure authenticated encryption scheme. PBCS is secure against compromised data server, i.e., the probability for any adversary to win the IND-CKS game (Fig. 4) is less than $1/2 + \varepsilon + \mathsf{negl}(\lambda)$, where λ is the security parameter.*

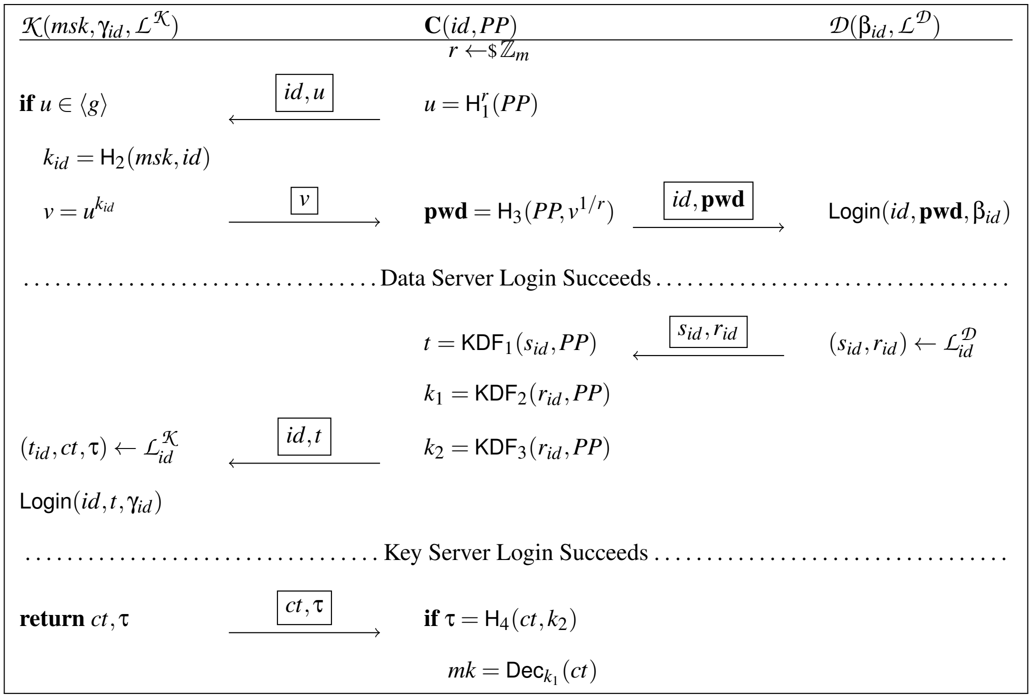
$\mathcal{K}(msk, \gamma_{id}, \mathcal{L}^{\mathcal{K}})$
$\mathbf{C}(id, PP)$
$\mathcal{D}(\beta_{id}, \mathcal{L}^{\mathcal{D}})$
$r \leftarrow \$ \mathbb{Z}_m$
$\mathbf{if}\ u \in \langle g \rangle$
$\boxed{id, u}$
$u = \mathsf{H}_1^r(PP)$
$k_{id} = \mathsf{H}_2(msk, id)$
$v = u^{k_{id}}$
$\boxed{v}$
$\mathbf{pwd} = \mathsf{H}_3(PP, v^{1/r})$
$\boxed{id, \mathbf{pwd}}$
$\mathsf{Login}(id, \mathbf{pwd}, \beta_{id})$
Data Server Login Succeeds
$t = \mathsf{KDF}_1(s_{id}, PP)$
$\boxed{s_{id}, r_{id}}$
$(s_{id}, r_{id}) \leftarrow \mathcal{L}_{id}^{\mathcal{D}}$
$k_1 = \mathsf{KDF}_2(r_{id}, PP)$
$(t_{id}, ct, \tau) \leftarrow \mathcal{L}_{id}^{\mathcal{K}}$
$\boxed{id, t}$
$k_2 = \mathsf{KDF}_3(r_{id}, PP)$
$\mathsf{Login}(id, t, \gamma_{id})$
Key Server Login Succeeds
$\mathbf{return}\ ct, \tau$
$\boxed{ct, \tau}$
$\mathbf{if}\ \tau = \mathsf{H}_4(ct, k_2)$
$mk = \mathsf{Dec}_{k_1}(ct)$

Figure 7: The Take Procedure. The client logins the data server via the password **pwd** derived with the key server, reconstructs the App login password t and the key of the authentication encryption k_1 and k_2. Then the client logins the App and authenticates himself to the key server with t, retrieves the ciphertext of the master key ct and the corresponding tag τ from the key server and then decrypts it. The tag τ is used to verify the integrity of the ciphertext ct. The boxed messages means they are protected by the TLS channel. β_{id}, γ_{id} are the login stubs stored in the data server and the key server, respectively, for login verification. H_1, H_2 and H_3 are hash functions used in the 2HashDH IBOPRF defined in Sec. 3. The security requires the number of the call of the IBOPRF service for one specific id on the key server side is bounded.

5.3 Deployment considerations

In practice, most of the above PBCS operations are run "under the hood". Here we describe how to leverage PBCS to smoothly integrate the secure storage module into the App and modify the App login mechanism while not affecting the user's experience.

When the user registers an account to the App with an identity by choosing a passphrase PP, in the back-end the client will launch a Register procedure of PBCS and help the user to register a cloud account with password **pwd** and an App account with password t using the identity id.[6]

When the user logins his App account using the passphrase PP at the first time, in the back-end the client and two servers run the Give procedure to complete the App login and automatically initialize the secure storage services as well. Afterward, the system will generate a distributively stored master key mk and store it securely.

When the user logins his App account using the passphrase PP from then on, the client's device will invoke the Take to login the App as well as retrieving the master key mk by communicating with the two servers at the back end. To upload the actual content, the client will encrypt the actual content using mk to generate ciphertext CT, and uploads CT to the data server. To retrieve the content, the client downloads the ciphertexts CT from the data server, uses mk to decrypt it and displays the results in the interface.

6 Experimental Evaluations

In this section, we demonstrate the deployability, efficiency, scalability and economical cost of our PBCS system via experiments carried out in Amazon Web Service (AWS for short).

Deployability. We did a survey of current popular cloud storage services, and found that popular storage services, including but not limited to Amazon S3 [2], Google Cloud Storage [18], Azure Storage [41], and Dropbox [15] provide the required API to act as the data server in PBCS to support secure storage. In our experiment, we only use the create/put/get API for java from Amazon S3 for deposit and retrieve. Other storage services provide such APIs supporting similar functions, like Dropbox's file upload/download API.

[6] For the users who already have App accounts before the update to PBCS, we highly suggest them to choose a new login passphrase, because the previous one may already be learned by the App server.

Moreover, our PBCS system is easy to be adopted to existing Apps. Our code can be packaged into a keyServerAPI and a clientAPI to provide service for App. Simply run the keyServerAPI on the App provider's administrative server and call the clientAPI on the App's client side to provide secure deposit and retrieve.

Efficiency. We implement a prototype in Java including instantiating all cryptographic primitives with the standard Java API and Bouncy Castle library. The IBOPRF is implemented with "secp256r1" curve. We use `TLS 1.2` with the `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256` ciphersuite for client to authenticate and securely connect key server, HTTPS protocol to communicate with the data server, and `AES/CTR/NoPadding` as the storage encryption, and `AES/GCM/NoPadding` as the key encryption, and `PBKDF2WithHmacSHA256` with one iteration with different salts to implement the $\mathsf{KDF}_{1,2,3}$. We use the build-in login of cloud storage service to implement the client authentication.

We do experiments on AWS. The data server is deployed on Amazon S3 in Tokyo. The key server is deployed on AWS EC2 using t2.micro instance in Osaka. The client is deployed in Seoul using AWS EC2 t3.xlarge instance, which has similar configuration with popular PCs[7]. The operating system is ubuntu 18.04 LTS. All the three are located in different regions to simulate remote clients and physically separated servers. We measured the round trip time (RTT) using `ping` from the client to the key server of 28.9ms and to the data server of 33.0ms, and the network upload/download speed of 700Mbps/699Mbps (using `iperf3`).

To show our design is efficient and practical, we measure the overhead of client and key server. We measure the time cost of each procedure over 100 iterations and get the average, where the IBOPRF, Give and Take costs 0.145s, 0.53s, and 0.48s, respectively. The key server overhead makes up less than 1% of IBOPRF including one hash to group element operation costing 4.5s per million iterations, and one elliptic curve scalar multiplication operation which costs 261.83s per million iterations. So TLS handshake and the network latency dominate the IBOPRF cost.

We do experiment on files with various sizes (from 10MB to 300MB) and run 25 iterations for both depositing and retrieval to get average cost. The results are displayed in Fig. 8, 9, where the red plain deposit denotes insecure deposit, the blue secure deposit includes one run of the Take protocol, encrypting file, and uploading encrypted file to the data server. For the client overhead, besides the above procedures, compared to insecure depositing and retrieving plain data without PBCS, secure data depositing and retrieval via PBCS need to encrypt and decrypt the data, which bring extra overhead. From Fig. 8, the showed overheads (secure deposits costs larger than plain deposits), are very small, increasing from 0 to 1.5 seconds as the file size increases to 300MB. From Fig. 9, the secure retrieval overhead is even fewer, near to zero. The detailed breakdown data could be found in Tables 2.

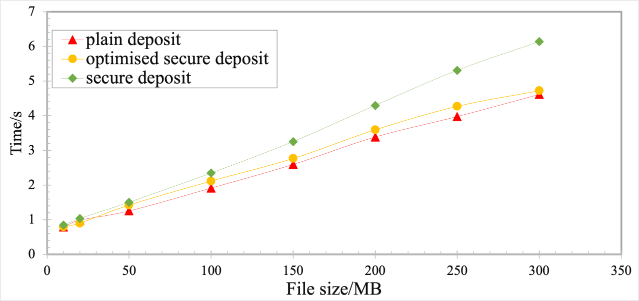

Figure 8: Time for depositing files

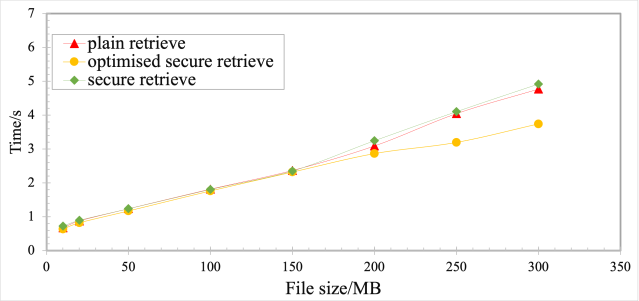

Figure 9: Time for retrieving files

Further optimization. We observe that our PBCS's overheads mainly come from the Enc/Dec, that increase (though still small) as the file gets larger, so it can be optimized by encrypting/decrypting data while uploading/downloading. If we cut the large file into smaller blocks, and encrypt/decrypt each of them while uploading/downloading another block, the total latency could be reduced to the encryption/decryption of only one block and the upload/download of all the blocks. We show the optimization details in Appendix C. From the yellow line in Fig.8,9 and the columns "OOvd" of Tables 2, one can see that the optimised overhead is extremely close to 0. More impressively, the time cost of downloading a plaintext file are even larger than securely downloading using the optimized implementation. This is because our optimization for retrieval folds both decryption and disk write operation with downloading data process[8].

Scalability. The main obstacle of PBCS to deploy in a popular App may be its influence on key server (the App server) scalability. So we test the key server throughput in the IBOPRF. As the reference , we also test the throughput of key server for static page.

[7] At the time of writing, the t2.micro instances were equipped with 1GB memory and 1 vCPU of Intel Xeon processors. The t3.xlarge instances were equipped with 16GB memory and 4 vCPU of Intel Xeon processors.

[8] We do not split the plaintext file into blocks, since it incurs more time cost than treating a plain file as a whole when a deposit/retrieve request needs to be made for each block at the server end.

The key server as a web server is equipped with nginx+tomcat framework. The IBOPRF requests are HTTPS GET requests. We use Siege as the throughput test benchmark running on AWS EC2 t2.2xlarge instance with 8vCPU and 32GB memory in Seoul, the same region as the client. We test 400 parallel requests with 250 iterations. For key server throughput with 1 vCPU, static page is 666.09 req/s and IBOPRF is 464.64 req/s. In IBOPRF, key server computes one hash-to-curve operation and one scalar multiplication of ECC point to deal with each request, which makes its throughput a bit lower than fetching static pages. The throughput of both increase linearly with the number of vCPU, shown in Fig. 10.

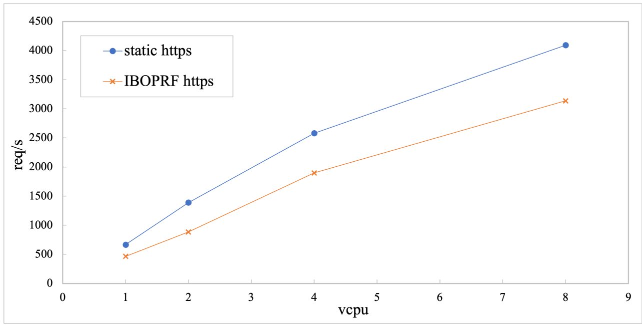

Figure 10: Key server throughput

Cost savings. Here we use Amazon S3 and EC2 as example to give some estimation of cost savings by insisting on deploying our system on cloud storage. Imaging an App with one million users (popular Apps have way more), it is augmented with secure storage using our PBCS: We first assume each user consumes about 1GB/month space (which could be much larger if the App involves videos). Considering the default file open soft limit of 1024 connections for each virtual machine, at least 1000 instances are required to keep one million connections open.

Besides the same cost of data transfer, using EC2 for same use, one needs to pay for extra computation cost [4] to keep VMs running, and 5 times cost of the storage [3]. Particularly, if we choose EC2 instance ("d2.4xlarge") with storage optimization to provide relatively robust service, it costs \$2.76 per hour (while S3 posts only a negligible request cost). Then the App provider mentioned above will need to pay around $2.76 * 24 * 30 * 1000 + 0.08 * 1,000,000$, which is already about *two million dollars more* if deploying the same secure storage on EC2 for just one month! Supporting more users, each user uses more data storage on average, or for a longer period would incur even larger costs proportionally.

7 Conclusion

We model, design, analyze, implement, and experiment with a novel system, which we name Portable Blind Cloud Storage (PBCS). It aims to for a secure and usable cloud storage system that satisfies multiple goals simultaneously.

Table 2: Cost breakdown for the deposit (time/s) and retrieve (time/s), where the size unit is MB. DPT/DCT denote depositing plaintext/ciphertext, RPT/RCT denote retrieving plaintext/ciphertext, Opt denotes the cost of optimized deposit time cost, Ovd/OOvd denote the overhead of the basic/optimized implementation.

Size	DPT	Enc	DCT	Opt	Ovd	OOvd
10	0.79	0.07	0.77	0.79	0.06	0
20	0.98	0.16	0.88	0.90	0.05	-0.08
50	1.25	0.39	1.11	1.42	0.25	0.17
100	1.91	0.78	1.56	2.12	0.43	0.21
150	2.82	1.18	2.08	2.77	0.44	-0.05
200	3.39	1.57	2.73	3.60	0.91	0.21
250	3.98	1.97	3.34	4.27	1.33	0.29
300	4.61	2.50	3.64	4.72	1.52	0.11

Size	RPT	Dec	RCT	Opt	Ovd	OOvd
10	0.67	0.07	0.65	0.64	0.05	-0.03
20	0.88	0.15	0.74	0.82	0.01	-0.06
50	1.24	0.39	0.85	1.18	0	-0.06
100	1.82	0.78	1.02	1.77	-0.02	-0.05
150	2.37	1.17	1.18	2.32	-0.02	-0.05
200	3.09	1.56	1.68	2.87	0.15	-0.22
250	4.05	2.07	2.03	3.20	0.05	-0.85
300	4.77	2.67	2.25	3.74	0.15	-1.03

Our entire design boils down to the preferred design principle of *"Constructivism"* (build on parts) over *"Gestalt"* (design it all) in deploying a large scale secure system. It is recommended that the modern software development [13, 19, 39, 43, 50] should start simple and only add components once really necessary, following the philosophical principle of *Occam's razor*, which essentially states that "*simpler solutions are more likely to be correct than complex ones*". In retrospect, we wish that designed system like our PBCS system that has been smoothly embedded in the existing architecture, fully exploits already existing standardized and existing components and tools, (this is in contrast with theoretical primitives holistically designed from scratch) such as login mechanism, TLS, and plain cloud storage. As previously discussed, our approach enables sound system development and a security proof for the entire system, relying on available optimized implementations of secure components and inheriting tested robustness and high efficiency. It makes the development practical exploiting existing APIs with high compatibility, reduces duplication, and simplifies the engineering work and maintenance of overall systems. This is especially necessary for upgrading and updating a living popular App supporting an enormous amount of users.

Acknowledgement

Moti Yung thanks the Snap Inc.'s security team for initial discussions leading to this work. Long Chen was supported by the National Key R&D Program of China 2021YFB3100100. Qiang Tang is supported in part by gifts from Stellar Foundation, Ethereum Foundation and Protocol Labs. Qiang Tang and Ya-Nan Li were also previously supported in part by NSF grant CNS #1801492 when the work was partially done and they were in New Jersey Institute of Technology.

References

[1] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In *ESORICS*, pages 61–79. Springer, 2016.

[2] AWS. https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html. Accessed July 30, 2020.

[3] AWS. Amazon EBS pricing. https://aws.amazon.com/ebs/pricing. Accessed July 31, 2020.

[4] AWS. Amazon EC2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand. Accessed July 30, 2020.

[5] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In *CCS*, pages 433–444. ACM, 2011.

[6] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *USENIX Security*, pages 1213–1227, 2018.

[7] Jad S Boutros, Jiayuan Ma, Filipe Jorge Marques de Almeida, and Marcel M Yung. Device independent encrypted content access system, July 2 2019. US Patent 10,341,304.

[8] Xavier Boyen. Hidden credential retrieval from a reusable password. In *ASIACCS*, pages 228–238, 2009.

[9] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure:SGX cache attacks are practical. In *WOOT*, 2017.

[10] Jan Camenisch, Robert R Enderlein, and Gregory Neven. Two-server password-authenticated secret sharing UC-secure against transient corruptions. In *PKC*, pages 283–307. Springer, 2015.

[11] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In *CRYPTO*, pages 256–275. Springer, 2014.

[12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *CCS*, pages 525–536. ACM, 2012.

[13] Kaizen Coder. Occam's razor in software development. https://www.cirdangroup.com/cirdan-blog/occams-razor-in-software-development. Accessed January 12, 2020.

[14] Emma Dauterman, Henry Corrigan-Gibbs, and David Mazières. {SafetyPin}: Encrypted backups with {Human-Memorable} secrets. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1121–1138, 2020.

[15] Dropbox. Dropbox for java developers. https://www.dropbox.com/developers/documentation/java. Accessed July 30, 2020.

[16] Adam Everspaugh, Rahul Chaterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The Pythia PRF service. In *USENIX Security*, pages 547–562, 2015.

[17] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324. Springer, 2005.

[18] Google. Cloud storage client libraries. https://cloud.google.com/storage/docs/reference/libraries. Accessed July 30, 2020.

[19] Cirdan Group. The role of occam's razor in agile software development. https://www.cirdangroup.com/cirdan-blog/occams-razor-in-software-development. Accessed January 12, 2020.

[20] Eran Hammer-Lahav. The OAuth 1.0 protocol. 2010.

[21] Dick Hardt. The OAuth 2.0 authorization framework. 2012.

[22] Google Chrome Help. Manage passwords. Website, 2021. https://support.google.com/chrome/answer/95606?co=GENIE.Platform.

[23] IBM Security. IBM HSM products. https://www.ibm.com/security/cryptocards, May 2018.

[24] Apple Inc. Set up icloud keychain. Website, 2021. SetupiCloudKeychain.

[25] Intricately. Everything you want to know about TikTok. https://my.intricately.com/companies/tiktok, 2017. Online; accessed 6 January 2020.

[26] Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In *CRYPTO*, pages 273–293. Springer, 2012.

[27] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT*, pages 233–253. Springer, 2014.

[28] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In *EuroS&P*, pages 276–291. IEEE, 2016.

[29] Stanisław Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *ACNS*, pages 39–58. Springer, 2017.

[30] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Updatable oblivious key management for storage systems. In *CCS*, pages 379–393, 2019.

[31] Brandon Jones. Do Snapchat memories take up space on your phone? https://www.psafe.com/en/blog/snapchat-memories-take-space-phone/, 2017. Online; accessed 6 January 2020.

[32] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, pages 429–448. Springer, 2013.

[33] Sarah Kuranda. How private is your public cloud? stacking up google, microsoft and aws data privacy. https://www.crn.com/news/cloud/300081714/how-private-is-your-public-cloud-stacking-up-google-microsoft-and-aws-data-privacy.htm, 2016. Online; accessed 6 January 2020.

[34] Russell WF Lai, Christoph Egger, Manuel Reinert, Sherman SM Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In *USENIX Security*, pages 1405–1421, 2018.

[35] Russell WF Lai, Christoph Egger, Dominique Schröder, and Sherman SM Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In *USENIX Security*, pages 899–916, 2017.

[36] Philip MacKenzie, Thomas Shrimpton, and Markus Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO*, pages 385–400. Springer, 2002.

[37] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *HASPICSA*, page 10. ACM, 2016.

[38] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data. *Google Research Blog*, 3, 2017.

[39] Tim Menzies. *Occam's Razor and Simple Software Project Management*, pages 447–472. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[40] Cade Metz. How Facebook moved 20 billion Instagram photos without you noticing. https://www.wired.com/2014/06/facebook-instagram/, 2014. Online; accessed 6 January 2020.

[41] Microsoft. Azure storage libraries for java. https://docs.microsoft.com/en-us/java/api/overview/azure/storage. Published February 13, 2020.

[42] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS# 5: password-based cryptography specification version 2.1. Technical report, 2017.

[43] Rick Mugridge. Test driven development and the scientific method. In *ADC*, pages 47–52. IEEE, 2003.

[44] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *FOCS*, pages 458–467. IEEE, 1997.

[45] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX ATC*, pages 227–240, 2018.

[46] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *CCS*, pages 295–310, 2017.

[47] Jonas Schneider, Nils Fleischhacker, Dominique Schröder, and Michael Backes. Efficient cryptographic password hardening services from partially oblivious commitments. In *CCS*, pages 1192–1203. ACM, 2016.

[48] Maliheh Shirvanian, Stanislaw Jareckiy, Hugo Krawczykz, and Nitesh Saxena. Sphinx: A password store that perfectly hides passwords from itself. In *ICDCS*, pages 1094–1104. IEEE, 2017.

[49] Snapchat. How to use my eyes only. https://support.snapchat.com/en-US/a/my-eyes-only. Accessed July 22, 2020.

[50] Mads Soegaard. Occam's razor: The simplest solution is always the best. https://www.cirdangroup.com/cirdan-blog/occams-razor-in-software-development. Accessed January 12, 2020.

[51] Sampath Srinivas, Dirk Balfanz, Eric Tiffany, FIDO Alliance, and Alexei Czeskis. Universal 2nd factor (U2F) overview. *FIDO Alliance Proposed Standard*, pages 1–5, 2015.

[52] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *SysTEXSOSP*, pages 1–6, 2017.

[53] Paul Voigt and Axel Von dem Bussche. The EU general data protection regulation (GDPR). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

[54] Ding Wang and Ping Wang. The emperor's new password creation policies: An evaluation of leading web services and the effect of role in resisting against online guessing. In *ESORICS*, pages 456–477. Springer, 2015.

[55] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *CCS*, pages 1242–1254. ACM, 2016.

[56] Lin Zhang, Zhenfeng Zhang, and Xuexian Hu. UC-secure two-server password-based authentication protocol and its applications. In *AsiaCCS*, pages 153–164. ACM, 2016.

A Security and Analysis of IBOPRF

Here we provide the formal definition of IBOPRF as well as the corresponding security analysis.

A.1 Security definition

Pseudorandomness: Intuitively, the pseudorandomness captures the security of IBOPRF against other malicious clients. It guarantees that other clients who did not corrupt the server can not distinguish $y = \mathcal{F}_k(id^*, x^*)$ with a random string r for a chosen identity id^* and a secret input x, even though he can arbitrarily query the server. More precisely, we have the following definition.

Definition 3 *let $\mathcal{D}$ be the distribution of input x and B be the maximum number of the adversary to query the server with identity id^*. If the minimum entropy of the distribution is d, we call an IBOPRF with (ε, d, B)-pseudorandomness if the probability of a probabilistic polynomial time adversary to win the game in the left side of Fig. 11 is* ε.

Obliviousness: Intuitively, the obliviousness captures the security of IBOPRF against the malicious server. It guarantees that a malicious server can not predict y for a fixed id and an unknown input x, even he can interact with the honest client holding id and x multiple times. More precisely, we have the following definition.

Definition 4 *Let $\mathcal{D}$ be the distribution of input x, where d is the min-entropy of the input distribution $\mathcal{D}$. We call an IBOPRF with (ε, d, k)-obliviousness if the probability of the successful guess in the game in the right side of Fig. 11 is less than ε when the adversary could make k different guesses for y.*

A.2 Analysis

In this section, we will show the IBOPRF construction in Sec.3 satisfies the uniqueness, pseudorandomness and obliviousness.

Uniqueness. The uniqueness is obvious, since the output of the function is $y = \mathsf{H}_3\left(x, \mathsf{H}_1(x)^{\mathsf{H}_2(msk,id)}\right)$.

Pseudorandomness. The pseudorandomness comes from the (N,Q) one-more Diffie-Hellman assumption [27, 28], which states that for any polynomial time adversary $\mathcal{A}$, $\Pr_{k \leftarrow\$ \mathbb{Z}_m, g_i \leftarrow\$ \mathbb{G}}[\mathcal{A}^{(\cdot)^k, \mathsf{DDH}(\cdot)}(g, g^k, g_1, \ldots, g_n) = S]$ is negligible, where $S = \{(g_{j_s}, g^k_{j_s}) | S = 1, \ldots, Q+1\}$, Q is the number of $\mathcal{A}$'s queries to the $(\cdot)^k$ oracle, and $j_s \in [N]$ for $s \in [Q+1]$. In other words, suppose $\mathcal{A}$ is allowed to query with a "kth power" oracle with Q times and a DDH oracle with polynomial queries. The assumption claims that although the $\mathcal{A}$ is allowed to compute the kth power of any Q of the N elements via quering $(\cdot)^k$ oracle, $\mathcal{A}$ computes the kth power of any $Q+1$ of the N elements (i.e. computes the kth power of "one more" element) is negligible.

Note that H_1, H_2 and H_3 could be modeled as random oracles. If the adversary has not been queried (msk, id^*) on the random oracle H_2, k_{id^*} is completely random to the adversary. Moreover, if the adversary has not queried $(x^*, \mathsf{H}_1(x^*)^{k_{id^*}})$ to the random oracle H_3, the value y will be truly random to the adversary. So we could conclude that a successful adversary must have queried x^* to H_1 and be able to compute $\mathsf{H}_1(x^*)^{k_{id^*}}$. Assume that the adversary has queried H_1 with q times. Since the adversary is only allowed to query the oracle $S()$ with id^* with B times, according to the (q,B) one-more Diffie-Hellman assumption the adversary can at most get B tuples with the form $(x, H_1(x)^{k_{id}})$. That means the adversary must get the correct x^* within B guesses. So the probability is $O(B/d)$.

Obliviousness. The obliviousness is easy to get when we model H_1 and H_3 as the random oracle. To get the correct y, the adversary must guess the correct input x^*. The probability to get the correct x^* within k guesses is less than $O(k/d)$.

```
Pseudorandom IBOPRF^{$\mathcal{A}$}
$b \leftarrow\$ \{0,1\}$
$(pp,msk) \leftarrow\$ \mathsf{Setup}(1^\lambda)$
$id^* \leftarrow\$ \{0,1\}^\lambda$
$x^* \leftarrow\$ \mathcal{D}$
$(ch,st) \leftarrow\$ \mathsf{CEval}_1(pp,id,x)$
$rp \leftarrow\$ \mathsf{SEval}(pp,id,ch,msk)$
$y_0 \leftarrow\$ \mathsf{CEval}_2(pp,id,rp,st)$
$y_1 \leftarrow\$ \{0,1\}^\lambda$
$b' \leftarrow\$ \mathcal{A}^{S()}(id^*,y_b)$
return $b = b'$
```

```
Oracle $S(id,ch)$
if $id \neq id^*$
    $rp \leftarrow\$ \mathsf{SEval}(pp,id,ch,msk)$
    return $rp$
else
    if $i < B$
        $rp \leftarrow\$ \mathsf{SEval}(pp,id^*,ch,msk)$
        $i = i+1$
        return $rp$
    else return $\perp$
```

```
Oracle $C(\cdot)$
$(ch,st) \leftarrow\$ \mathsf{CEval}_1(pp,id^*,x^*)$
return $ch$
```

```
Oblivious IBOPRF^{$\mathcal{A}$}
$b \leftarrow\$ \{0,1\}$
$(pp,msk) \leftarrow\$ \mathsf{Setup}(1^\lambda)$
$id^* \leftarrow\$ \{0,1\}^\lambda$
$x^* \leftarrow\$ \mathcal{D}$
$(ch,st) \leftarrow\$ \mathsf{CEval}_1(pp,id,x^*)$
$rp \leftarrow\$ \mathcal{A}$
$y_0 \leftarrow\$ \mathsf{CEval}_2(pp,id,rp,st)$
$y_1,\ldots,y_k \leftarrow\$ \mathcal{A}^C(msk)$
if for $i = 1,\ldots,k$
    $y_i = y_0$
    return 1
else return 0
```

Figure 11: The pseudorandomness and obliviousness of the IBOPRF. $\mathcal{D}$ is the distribution of input x and B is the maximum number of the adversary to query the server with identity id^*.

B Further Extensions

Due to our principle of minimal addition to an existing infrastructure and simplicity, our system could be easily extended to support further functionalities and more complicated settings.

Defending denial of service attacks. DOS attacks are a fundamental threat to Apps, not only to the PBCS system. In our protocol, the adversary can send an arbitrary number of Give requests to the key server. Since the key server does not authenticate the client in this stage, he will need to store everything it receives, and may give out his storage resources. A simple countermeasure is: one can make the key management server and the data server share one identity list, so only the client with the *id* on the list can "Give" the key.

Proactive security guarantees that an attacker has less time to compromise shares and as long as the attacker visits both server simultaneously, the system remains secure. PBCS can choose new randomness s'_{id} and r'_{id} stored under the cloud server and compute $t' = \mathsf{KDF}_1(s'_{id},PP)$, $k'_1 = \mathsf{KDF}_2(r'_{id},PP)$, $k'_2 = \mathsf{KDF}_3(r'_{id},PP)$, $ct' = \mathsf{Enc}(k'_1,mk)$, $\tau' = \mathsf{H}_4(k'_2,ct')$ and depositing (t',ct',τ') to the key management server.

Different authentication factors. The security of PBCS system relies on the security of the passphrase. The passphrase we described in the protocol does not have to be restricted to password. To achieve a higher level security, the user can choose the biometrics or a long PIN generated by a secure device or escrow system. Therefore, the passphrase can have a very high entropy, and PBCS is more robust against online dictionary attacks. Furthermore, if the user leverages different authentication factors to login the key server and the cloud server respectively, e.g., using the Face ID to login the key server and the password to login the cloud server, the leakage of one authentication factor will not be affected the security of PBCS and the IBOPRF module could be saved.

C Implementation Optimization

In the implementation of PBCS, we leverage a simple optimization idea: one can parallelly encrypt/decrypt data while upload/download, hence he does not need to wait the finish of the encryption/decryption before uploading/downloading the data. However, it is involved to determine how many blocks we should divide a file, since cloud storage needs to return an "ack" confirmation for each request on each block. The increased block number brings extra latency, which could be large when network delay is long.

Taking all those into account, we first model the time cost T as the function of file size s and total block number n, and then get a quick estimation $T(n,s) = k_2\frac{s}{n} + (2c_2 + c3)n + k_1 s + c_1$, where k_2 represents the encryption time (s/MB), c_2 is the network latency, and c_3 is the time for S3 processing one deposit instruction; while k_1 denotes the time needed for transferring and storing 1MB data to S3, c_1 denotes the TLS connection building time, which is constant for the same network. Uploading a plaintext will take $T(1,s)$. Thus the overhead can be easily derived as $\Delta T = T(1,s) - T(n,s)$, which we will minimize by finding the optimal number of blocks n^* as a function of s. We estimate the concrete parameters in our experiment environment, set n^* accordingly. In our concrete example $n^* = \sqrt{\frac{s}{20}}$, which could be easily adapted in different network conditions.