Secure Storage as a Service

YANAN LI

(M.Eng)

Supervisor: Qiang Tang

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

School of Computer Science
Faculty of Engineering
The University of Sydney
Australia

31 July 2025

Abstract

Cloud storage, as a storage infrastructure, offers many advantages over on-premises storage and even enhances other services, such as streaming services and gaming platforms, making them more efficient, scalable, and accessible. This aligns with the concept of storage as a service. However, it also raises significant security concerns on data privacy, as most stored data is accessible to the service provider, who could exploit it for profit, and is vulnerable to data breaches. In this dissertation, we systematically study secure storage as a service, including secure cloud storage services and secure cloud storage for other applications.

1. We first studied a secure cloud storage solution for other applications. Many applications provide services for users and rent cloud storage to store user's data. We modularly designed a secure storage solution for those applications, so that the user's data is only visible to user-self. Furthermore, the solution is fully compatible with existing cloud storage services such as AWS S3, and transparent to users who can still use the application via one password without concerns of offline attacks.
2. We studied how to enable the version control functionality and corresponding securities for secure storage. Most cloud storage services offer limited version control and access controls relying on the trust of storage providers. we got rid of the trust on server, designed secure storage with full version control functionalities. Moreover, the design is compatible with existing Git server that runs Git repository management program on top of cloud storage.
3. We studied secure storage with stronger security in terms of key compromise. Key rotation is an effective way to improve key compromise resilience via updating encrypted data under new key periodically, and *updatable encryption* enables data encryption with key rotation. We first studied updatable encryption with stronger security that is needed for secure storage. Then we further formalized secure storage system with key compromise resilience.

List of Publications

Authorship is in alphabetical order

1. Ya-Nan Li, Yaqing Song, Qiang Tang, Moti Yung: End-to-End Encrypted Git Services. The ACM Conference on Computer and Communications Security (**CCS**) 2025
2. Long Chen, Ya-Nan Li, Qiang Tang, Moti Yung: End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage. ACM Transactions on Privacy and Security **(TOPS)** 2025.
3. Ya-Nan Li, Tian Qiu, Qiang Tang: Pisces: Private and Compliable Cryptocurrency Exchange. Network and Distributed System Security Symposium **(NDSS)** 2024.
4. Long Chen, Hui Guo, Ya-Nan Li, Qiang Tang: Efficient Secure Storage with Version Control and Key Rotation. International Conference on the Theory and Application of Cryptology and Information Security **(ASIACRYPT)** 2023.
5. Long Chen, Ya-Nan Li, Qiang Tang, Moti Yung: End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage. **USENIX Security Symposium** 2022.
6. Long Chen, Ya-Nan Li, Qiang Tang: CCA Updatable Encryption Against Malicious Re-encryption Attacks. International Conference on the Theory and Application of Cryptology and Information Security **(ASIACRYPT)** 2020.

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Name: Yanan Li

Signature:

Date: 31 July 2025

Authorship Attribution Statement

This thesis contains published material in the following publications, in which the authorship is in alphabetical order and I am the lead author.

1. **Ya-Nan Li**, Yaqing Song, Qiang Tang, Moti Yung: End-to-End Encrypted Git Services. The ACM Conference on Computer and Communications Security (**CCS**) 2025
2. Long Chen, **Ya-Nan Li**, Qiang Tang, Moti Yung: End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage. ACM Transactions on Privacy and Security **(TOPS)** (2025).
3. Long Chen, **Ya-Nan Li**, Qiang Tang, Moti Yung: End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage. **USENIX Security Symposium** 2022.
4. Long Chen, **Ya-Nan Li**, Qiang Tang: CCA Updatable Encryption Against Malicious Re-encryption Attacks. International Conference on the Theory and Application of Cryptology and Information Security **(ASIACRYPT)** 2020.
5. Long Chen, Hui Guo, **Ya-Nan Li**, Qiang Tang: Efficient Secure Storage with Version Control and Key Rotation. International Conference on the Theory and Application of Cryptology and Information Security **(ASIACRYPT)** 2023.

My particular contributions to this have been:

- Chapter 3 of this thesis is published as [2, 3] in the list above.
  I designed the solution, analyzed the security, and wrote the manuscript.
- Chapter 4 of this thesis is published as [1].
  I designed the solution, analyzed the security, and wrote the manuscript.
- Chapter 5 of this thesis is published as [4] in the list above.
  I designed the solution, analyzed the security, and wrote the manuscript.

- Chapter 6 of this thesis is published as [5] in the list above.

  I designed the solution, analyzed the security, and wrote the manuscript.

Name: Yanan Li

Signature:

Date: 31 July 2025

Attesting Authorship Attribution Statement

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Supervisor Name: Qiang Tang

Signature:

Date: 31 December 2024

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Qiang Tang. Your taste, enthusiasm, wisdom, and kindness have not only guided me through my PhD journey but will also leave a lasting impact on my future career and life. From you, I learned what is good research and how to conduct it with integrity and excellence. You have been so much more than an advisor to me. I deeply cherish our conversations about research philosophy, literature, art, and the values that shape a meaningful life. You opened the door for me to explore and appreciate the greatest things in the world, unveil the mysteries of complex ideas, and build confidence in myself. I am truly fortunate to have had you as a mentor. Without your unwavering support and insightful guidance, this journey would not have been as rewarding, nor would I have grown into the person I am today.

I would like to extend my sincere thanks to my committee members Vanessa Teague and Serge Vaudenay, and my committee convenor Sri AravindaKrishnan Thyagarajan.

To my collaborators, I extend my heartfelt thanks for your invaluable contributions to this dissertation and for enriching my research with your expertise and perspectives. Moti Yung, for your invaluable guidance, vast expertise and profound insight that enriched my understanding and shaped the direction of this dissertation; Long Chen, for the collaborative guidance, fruitful discussions, shared ideas, and collective efforts; and Hui Guo, Tian Qiu, Yaqing Song and others, for collaborative efforts, discussions of technical details, and the exchange of ideas.

I'm also deeply thankful to all the members of our group Yuan Lu, Songlin He, Zhenliang Lu, Bo Pang, Erin Kenney, Xinrui Zhang, Tianyi Zhang, Tiancheng Mai, Sam Polgar, Quanhao Chen, Yuchen Ye, Omniyyah Ibrahim, Chengcong Hu, and others, for your talents in many aspects, helpful assistance, thoughtful discussion, and fostering such a collaborative and inspiring environment.

I am grateful to two universities NJIT and USYD, where I had the privilege to undertake my PhD journey, each offering supportive environment, unique experiences, and cultural perspectives. Both institutions have played an important role in shaping my development as a researcher and an individual. Special thanks to USYD for awarding me the faculty of engineering research scholarship and the faculty of engineering career advancement award.

I am profoundly grateful to my family for their unwavering support and endless love. To my parents and sister, your support and understanding made the journey possible. To my husband, Hanwen Feng, your love and companionship have been my anchor throughout this journey. Thank you for standing by me through every challenge and celebrating every milestone—it is because of you that this achievement is even more meaningful.

Finally, to everyone who has supported me along the way, whether academically, emotionally, or personally, my heartfelt thanks. This dissertation is a testament to your belief in me, and I am forever grateful.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

Over the past few decades, the way people store their data has undergone a dramatic transformation. Historically, data had been stored on premises in various physical forms, including turtle shells, paper notebooks, compact discs, etc., and ensuring the security of data relied heavily on the owner's ability to maintain these storage devices well. Since the 1990s, starting as the form of hosting services [1], the concept of Storage as a Service (StaaS) has gradually emerged and is realized by cloud storage services.

Nowadays, cloud storage services help users avoid the high upfront costs and technical expertise required to maintain physical storage devices and enjoy pay-as-you-go pricing models. Beyond cost-efficiency, cloud storage provides numerous advantages, such as elastic scalability, allowing storage capacity to expand with demand, and unparalleled convenience, enabling users to access their data anytime and anywhere without the need for physical storage devices. Moreover, cloud storage services, such as AWS S3 [2], Microsoft Azure Storage [3], and Google Cloud Storage [4], acting as storage infrastructure, provide storage services not only for individual users but also businesses, and significantly enhance other services, making them more efficient, scalable, and accessible. For example, with cloud storage service, streaming services such as Netflix and Spotify can store and deliver content efficiently; gaming platforms like Xbox Cloud Gaming can store game data and progress for seamless play with the help of cloud storage; and cloud storage facilitates the collection and processing of IoT data from devices worldwide.

While the benefits of cloud storage are evident, this paradigm shift has raised perennial security concerns. Users' data, such as medical records, credit card information, and business documents, is often highly sensitive and valuable. On the one hand, to address these concerns, service providers typically assure customers through product policies that their data is properly protected and that they retain control over it. On the other hand, from a technical standpoint, users must inherently trust the providers, as their data is physically stored on remote servers managed by these companies. However, providers are not always trustworthy due to various factors. Even without malicious intent, human or technical errors can lead to data breaches. Moreover, to ensure real-time access, these servers are often connected to open network environments, which increase their vulnerability to cyberattacks. The vast amount of data stored on such servers makes them attractive targets for malicious actors. Data breaches continue to occur frequently and leak billions of data, leading to significant consequences and fueling widespread societal concerns about data security [5, 6, 7]. Recently, U.S. officials even told Americans to use encrypted apps as the scope of cyberattack grows [8]. Additionally, some providers may exploit user data indirectly for their own benefit, such as using it to train artificial intelligence models, which may cause potential harm to users' intellectual property and privacy. Thus, protecting data in storage against service providers and unauthorized users with *end-to-end security* is pressing and critical.

To be clear, we emphasize that secure storage throughout this thesis considers end-to-end security, especially against service providers and all unauthorized users. Specifically, in the storage setting, end-to-end security protects data for the data owners and the shared users.

Encryption is one of the most fundamental tools in data protection. At a high level, encryption ensures that encrypted data does not reveal any information about the plaintext to anyone except the legitimate user possessing the decryption key. In theory, data owners could rely on storage services solely for storing encrypted data, ensuring security even in the event of a data breach, as long as the decryption key is kept securely.

Although the idea of enabling secure storage via storing encrypted data on cloud storage and letting only legitimate users have the decryption key to access the plain data looks straightforward, it is rarely securely implemented in practice. Many industrial products emerge

claiming to provide secure storage protection but turn out suffering various vulnerabilities [9, 10, 11]. The situation is reminiscent of secure communication. Transport Layer Security (TLS) protocols [12] have experienced a long history to achieve a secure communication channel that looks only requiring the combination of authentication and encryption. There is another similar long journey from TLS to end-to-end encrypted messaging, involving many related research works and producing widespread secure messaging applications such as Signal [13], WhatsApp [14], etc.

We found that the unique features of cloud storage, stemming from the very benefits that make cloud storage services appealing, bring inherent challenges of realizing end-to-end secure storage as a service:

- *Ubiquitous accessibility and plain storage infrastructure introduce challenges.* A key feature of cloud storage is its ubiquitous accessibility, enabling users to access their stored data anytime, from any device. However, this convenience introduces complexity in ensuring secure storage. Users may face limitations in carrying high-entropy secrets, while low-entropy secrets are susceptible to offline attacks. Furthermore, existing plain cloud storage services have become foundational infrastructure, serving a massive user base. Many App providers rely on these services to store data for their own users, as the need to leverage third-party cloud storage remains critical for operational efficiency and scalability. Ensuring compatibility with existing infrastructures in the end-to-end secure storage for Apps adds another layer of complexity, given that third-party plain cloud storage services typically offer only limited programmable APIs.
- *Realizing essential functionalities over encrypted data creates new challenges.* Many storage services provide functionalities beyond plain data storage, often relying on plaintext data to operate efficiently. For example, Git repositories support version control, which typically involves constant data update, tracks differences, and de-duplicate overlaps between versions for efficiency. The nature of collaboration and constant update in Git services brings new challenges to the security modeling, not only for confidentiality, but also for integrity and unforgeability, especially in the

software supply chain. Enhancing security via directly applying standard encryption complicates these functionalities. While advanced encryption tools like fully homomorphic encryption [15] can support arbitrary operations on ciphertexts, their computational costs are often orders of magnitude higher than equivalent plaintext operations. Innovative solutions are required to address these challenges to achieve practical and deployable end-to-end secure storage services.

- *Long-lived data in storage has issues against key compromise.* Data has a long life in storage before deletion, which is different from the case that a communication message is only protected before communicating parties see it. Securely storing a decryption key over a long period is a well-recognized challenge. Once the decryption key is compromised, all security guarantees provided by encryption collapse. In secure messaging, post-compromise security is defined to capture security against key compromise, which is achieved by frequently updating keys and encrypting new messages with new keys. Such practice is not easily transferable to the storage setting as previously encrypted data requires extra care.

While some of the aforementioned challenges have gained attention from the research community in recent years, as we will discuss shortly, significant gaps remain. In this dissertation, we aim to advance this field of study by addressing these challenges systematically, with the ultimate goal of realizing secure storage as a service.

1.2 Contributions of This Dissertation

In this dissertation, we conduct a systematic study on how to achieve end-to-end secure storage across three dimensions, which we elaborate on as follows:

Modularly augmenting an App with an efficient, portable, and blind cloud storage: Cloud storage provides widespread services for both individual users and businesses. Many application providers rely on third-party cloud storage to store data for their users. In this context, secure storage aims to protect the data security for end users, i.e., the data owners, rather than the App providers. Servers are often assumed to be potentially malicious, as

they may be vulnerable to internal compromises and external breach attacks. To ensure data security for end users while keeping it blind to servers, the underlying secrets need to have high entropy, which often makes them too long for users to remember. However, portability—which ensures user access to data anytime and anywhere—requires low-entropy secrets for convenience. Therefore, designing a secure storage system that balances strong security guarantees with portability is crucial. Additionally, secure cloud storage for Apps introduces new requirements for compatibility with existing cloud storage infrastructures. App providers significantly depend on third-party cloud storage services for efficiency and scalability, but these services typically offer only limited programmable APIs. This adds another layer of complexity to end-to-end secure storage design.

Solving the problem of "data privacy against servers" while (1) relying on existing infrastructure and (2) supporting portability remains open. Existing proposals, like password-protected secret sharing, target the same goal but are incompatible with existing cloud storage services. Specifically, they lack the simplicity needed to directly utilize existing cloud *storage* without requiring changes on the cloud side.

Here, we propose a novel system for securely storing private data in existing cloud storage with the help of a key server (necessary given the requirements). In our system, user data is secure against threats from the cloud server, the key server, and illegitimate users. Only the legitimate user can access the data on any device using a correct passphrase. Most importantly, our system does not require the storage server to support any newly programmable operations. Moreover, leveraging the existing App login, our system requires only one passphrase, which never leaves the user's device and remains hidden from both servers. End-to-end security with confidentiality and integrity is proved under formal models, and its efficiency is demonstrated by experiments conducted on AWS S3. Notably, a preliminary variant, based on our principles, was deployed by Snapchat in their *My Eyes Only* module, serving hundreds of millions of users.

Supporting Git services with full-fledged end-to-end security: Data update and versioning service for tracking file edit history are two common and essential functions that are offered by most plain cloud storage services including AWS S3, Microsoft Azure, and Google Cloud

and even many "secure" cloud services who claim providing data privacy protection including MEGA[16], pCloud[17], Proton Drive[18], etc. However, the only research paper [19] formalizing end-to-end encrypted cloud storage did not provide versioning service and simplified the data update with replacement, which cannot capture the efficient data update and version control function.

Compared with general storage services with very limited versioning services, such as Google Docs, Git services, which provide data storage with full version control, have been widely used to manage projects and enable collaborations among multiple entities. To better study secure storage with version control, we focus on the cloud storage deployed with the Git version control server (also called Git service).

Just as in messaging and cloud storage, end-to-end security has been gaining increased attention, as content in the repositories could be valuable (e.g., code for startups), while data breaches become routine nowadays. However, existing studies on Git service (mostly open-source tools and projects) provide very weak security guarantees and have large overheads. Moreover, we observed that *unforgeability* is also critical for end-to-end secure storage but is absent in existing secure storage studies [20, 19], which only captures two basic data security properties confidentiality and integrity.

We initiate the study of end-to-end encrypted Git services. Particularly, we formally define the syntax and full-fledged end-to-end security properties and then propose a construction called that provably meet those security properties. Moreover, our construction is compatible with current Git servers and thus can be directly tested and deployed on top of existing platforms, and the overhead is only proportional to the actual difference caused by each edit instead of the whole file (or even the whole repository) as in existing works. We implemented both constructions and tested them directly on several public Github repositories. Our evaluations show that our two constructions are both significantly more efficient than previous solutions that provide much weaker security.

Considering post-compromise security for long-lived data in storage: Variable cyberattacks bring frequent data breaches, which make data both in cloud storage and the user's

device vulnerable to be leaked. Standard encryption provides security on condition that the secret key is kept securely. Once the secure key gets compromised, attackers can decrypt the ciphertext to learn the plain data and even generate any valid ciphertext to break the integrity. To mitigate the damage of key compromise on encrypted data, Updatable encryption (UE) was proposed in CRYPTO'13 [21] that allows the secret key of the outsourced encrypted data to be updated to a fresh one periodically so that the security is reserved as long as attackers do not corrupt both the key and encrypted data within a short period.

Regarding post-compromise security of data in storage, we further study the cryptographic primitive UE itself about the insufficient securities and initiate the study of secure storage systems with post-compromise security.

(1) *UE with stronger security.* There are several elegant works about UE studying various security properties. We notice several major issues in existing security models of (ciphertext dependent) updatable encryption, in particular, integrity and CCA security. The adversary in the models is only allowed to request the server to re-encrypt honestly generated ciphertext, while in practice, an attacker could try to inject arbitrary ciphertexts into the server as she wishes.

We fill the gap and strengthen the previous security definitions in multiple aspects: most importantly, our integrity and CCA security models remove the restrictions in previous models and achieve standard notions of integrity and CCA security in the setting of updatable encryption. Along the way, we refine the security model to capture post-compromise security and enhance the re-encryption indistinguishability to the CCA style. Guided by the new models, we provide a novel construction **ReCrypt**$^+$, which satisfies our strengthened security definitions. The technical building block of homomorphic hash from a group may be of independent interest. We also provide examples that were secure in previous models but failed in our models; and interestingly, the folklore result in authenticated encryption that IND-CPA plus ciphertext integrity implies IND-CCA security does *not* hold in the ciphertext dependent setting of updatable encryption.

(2) *Secure storage system with post-compromise security.* Periodic key rotation is a widely used technique to enhance key compromise resilience. UE schemes provide an efficient approach to key rotation, ensuring post-compromise security for both confidentiality and integrity. However, these UE techniques cannot be directly applied to frequently updated databases due to the risk of a malicious server inducing the client to accept an outdated version of a file instead of the latest one.

To address this issue, we propose a scheme called Updatable Secure Storage (USS), which provides a secure and key updatable solution for dynamic databases. USS ensures both data confidentiality and integrity, even in the presence of key compromises. By using efficient key rotation and file update procedures, the communication costs of these operations are independent of the size of the database. This makes USS particularly well-suited for managing large and frequently updated databases with secure version control. Unlike existing UE schemes, the integrity provided by USS holds even when the server learns the current secret key and intentionally violates the key update protocol.

1.3 Organization

In this thesis, we present our systematic study on secure storage as a service. Before introducing details, we first overview the involved cryptographic preliminary in Chapter 2. We introduce the secure storage solution for other Apps called end-to-same-end encryption in Chapter 3. Furthermore, to support version control with full-fledged end-to-end security, we present end-to-end encrypted Git services in Chapter 4. Regarding key compromise resilience, we first present our theoretical study about UE with stronger security in Chapter 5; then we apply UE to secure storage setting and formalize the secure storage with post-compromise security in Chapter 6. Finally, we conclude and discuss future works in Chapter 7.

CHAPTER 2

Preliminaries

In this Chapter, we describe the underlying secure protocols, cryptographic primitives, and security assumptions in this thesis.

2.1 Login Mechanism

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 [22, 23], the recent Universal Second Factor protocol [24], 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 AuthReg and the login algorithm Login:

- AuthReg(id, α_{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.

- 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 $Bypass^{\mathcal{A}}$	Oracle $\mathcal{O}_{\mathsf{AuthReg}}(id, \alpha)$	Oracle $\mathcal{O}_{\mathsf{Login}}(\alpha)$
1 : $id^* \leftarrow\$\ \mathcal{A}$	1 : $\beta \leftarrow\$\ \mathsf{AuthReg}(id, \alpha)$	1 : **if** $count > B$
2 : $\alpha_{id^*} \leftarrow\$\ \chi$	2 : **return** β	2 : **return** $\perp$
// Choose α^* from distribution χ		3 : **else**
3 : $\beta_{id^*} \leftarrow\$\ \mathsf{AuthReg}(id^*, \alpha_{id^*})$		4 : $\mathsf{Login}(\alpha, \beta_{id^*}) = b'$
4 : $\alpha' \leftarrow\$\ \mathcal{A}^{\mathcal{O}_{\mathsf{AuthReg}}, \mathcal{O}_{\mathsf{Login}}}(id^*)$		5 : $count = count + 1$
5 : $\mathsf{Login}(id^*, \alpha', \beta_{id^*}) = b$		6 : **return** b'
6 : **return** b		

FIGURE 2.1. The security game of the login scheme.

Here we define the formal properties that the Login mechanism should satisfy:

- *Correctness:* If β_{id} is generated from the 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. 2.1 is less than ϵ.

2.2 Authenticated Encryption with Associated Data

Authenticated encryption with associated data (AEAD) is a variant of authenticated encryption (AE) that allows a recipient to check the integrity of both the encrypted and unencrypted information in a message. AEAD binds associated data (AD) to the ciphertext and to the context where it is supposed to appear so that attempts to "cut-and-paste" a valid ciphertext into a different context are detected and rejected. Specifically, an AEAD scheme consists of following three algorithms:

- $\mathsf{KeyGen}(1^\lambda)$ takes the security parameter λ as input, and outputs the secret key k.
- $\mathsf{Enc}(k, m, ad)$ takes the secret key k, a message m and the associated data ad as inputs, and outputs the ciphertext c. AE is a special case of AEAD where the

associated data ad is empty. For AE schemes we will abuse notation slightly and sometimes have ciphertexts, the output of $\mathsf{Enc}(k, m, ad)$, be a pair of bit strings (c, τ) (instead of a monolithic single string c), where τ denotes an authentication tag, c in the pair is a ciphertext except the tag.

- $\mathsf{Dec}(k, c, ad)$: take the secret key k, a ciphertext c and the associate data ad as inputs, and outputs the decrypted message m or the symbol $\perp$ to denote the decryption failure. In some cases, the associated data ad is a part of the ciphertext c, and there are only two parameters (k, c) as input.

Security Notions for Authenticated Encryption

MU-RoR-AE. Let $\pi = \{\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}\}$ be an AEAD scheme. The Multi-User Real or Random game for Authenticated Encryption MU-RoR-AE$^{\mathcal{A}}_{\pi}(\lambda, b)$ is defined in Fig. 2.2. This notion is derived from [25] and will be convenient for our reduction. It resembles the usual chosen-ciphertext-guessing game, in which the adversary has access to the encryption and decryption oracle using the same secret key, and is trying to guess which of its chosen two messages is contained in the challenge ciphertext. But in this game, the adversary can query the encryption and decryption oracles under multiple secret keys and must distinguish between a challenge ciphertext containing either a chosen message or a random one. The advantage of an MU-RoR-AE adversary $\mathcal{A}$ is measured by

$$\mathsf{Adv}^{\text{mu-ror-ae}}_{\pi,\mathcal{A}} = \left|\Pr[\text{MU-RoR-AE}^{\mathcal{A}}_{\pi}(\lambda, 1) = 1] - \Pr[\text{MU-RoR-AE}^{\mathcal{A}}_{\pi}(\lambda, 0) = 0]\right|$$

We say that π is MU-RoR-AE secure if and only if $\mathsf{Adv}^{\text{mu-ror-ae}}_{\pi,\mathcal{A}}$ is negligible for any PPT adversary $\mathcal{A}$.

We also will use a chosen-plaintext-only variant of MU-RoR-AE security for any symmetric key encryption, in which the adversary is disallowed to query the decryption oracle, i.e., Multi-User Real or Random (MU-RoR) security. Opposite to the multi-user scenario, we can similarly define One-Time Real or Random (OT-RoR) security for symmetric key encryption in the chosen-plaintext-only setting, where the adversary can query the oracle $\mathcal{O}_{\mathsf{ROR}}$ and $\mathcal{O}_{\mathsf{Enc}}$ at most once for each key index (also known as "one-time" security).

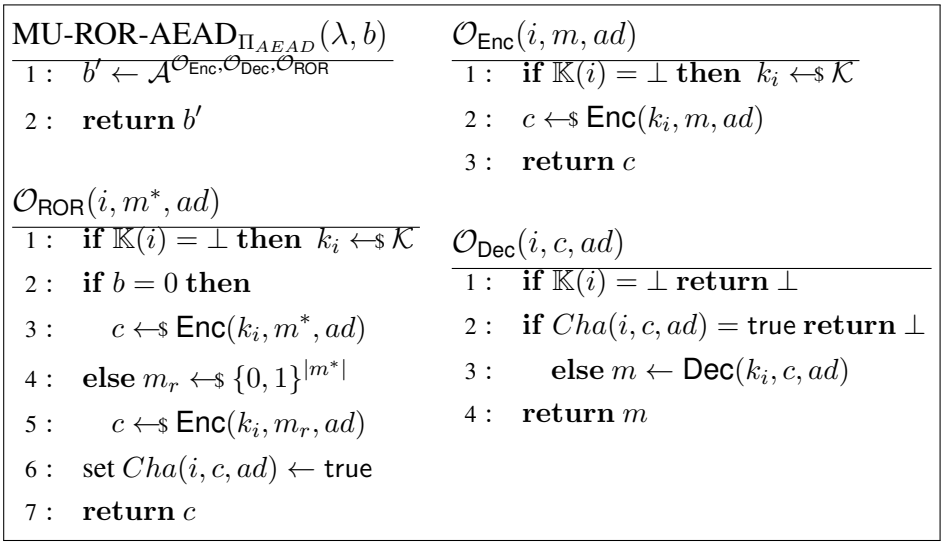


FIGURE 2.2. The experiment for MU-ROR-AEAD.

2.3 Commitment

A commitment scheme $\mathbf{Com} = \{\mathsf{Init}, \mathsf{Com}, \mathsf{Open}\}$ consists of three following algorithms: Init is used to generate the public parameter; Com outputs a commitment value com from a message m, while Open will check whether the commitment com is bound to the message m. A commitment scheme should satisfy both the *hiding* and *binding* properties. The hiding property requires the distributions of the commitment values for different messages can not be distinguished by the adversary, while the binding property requires the commitment value can not be opened to two different messages.

Some commitment schemes, such as the Pederson commitment [26], also satisfy the homomorphic property, which are called the homomorphic commitment. Specifically, the message space, the randomness opening space and the commitment values are all defined over additives group $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_3$ respect to the operations $\oplus$, $\odot$ and $\otimes$. The commitment scheme satisfies $\mathsf{Com}(m_1, open_1) \otimes \mathsf{Com}(m_2, open_2) = \mathsf{Com}(m_1 \oplus m_2, open_1 \odot open_2)$.

2.4 Key-Homomorphic PRF

The notion of key-homomorphic PRFs was proposed by Boneh et al. [21], and used in the UE constructions [25, 27]. Specifically, a key-homomorphic PRF $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a secure psedorandom function which satisfy the following property: for every $k_1, k_2 \in \mathcal{K}$, and every $x \in \mathcal{X}$: $\mathsf{F}(k_1, x) \otimes \mathsf{F}(k_2, x) = \mathsf{F}((k_1 \oplus k_2), x)$ where $\otimes$ and $\oplus$ are group operations respect to $\mathcal{K}$ and $\mathcal{Y}$. One example construction is to define as $y = H(x)^x$ where $H(\cdot)$ is a random oracle from a bit string to a group element.

2.5 Square-CDH Assumption

Recall the definition of bilinear groups. Let $\mathbb{G}$, $\mathbb{G}_T$ be bilinear groups of prime order p equipped with a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. Let $g \in \mathbb{G}$ be random generators. For an algorithm $\mathcal{B}$, define its advantage as

$$\mathrm{Adv}_{\mathcal{B}}^{\text{Square-CDH}}(\lambda) = |\Pr[\mathcal{B}(g, g^a) = g^{a^2}]$$

where $a \leftarrow\!\!\$\, \mathbb{Z}_p$ are randomly chosen. We say that the Square-CDH (Square Computational Diffie-Hellman) assumption holds, if for any probabilistic polynomial time (PPT) algorithm $\mathcal{B}$, its advantage $\mathrm{Adv}_{\mathcal{B}}^{\text{Square-CDH}}(\lambda)$ is negligible in λ, where λ is the security parameter.

2.6 Updatable Encryption

Updatable encryption (UE) is a cryptographic technique that allows periodic updates of the secret key of encrypted outsourced data. The syntax of UE is defined as follows:

DEFINITION 2.6.1 (Updatable Encryption). The ciphertext-independent updatable encryption (UE) consists of the following six algorithms

$$\mathsf{UE} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Next}, \mathsf{Upd}).$$

- $\mathsf{UE.Setup}(1^\lambda)$ is a randomized algorithm run by the client. It takes the security parameter λ as input and outputs the public parameter pp which will be shared with the server. Later all algorithms take pp as input implicitly.
- $\mathsf{UE.Keygen}(e)$ is a client-run randomized algorithm. It takes the epoch index e as input and outputs a secret key k_e for the epoch e.
- $\mathsf{UE.Enc}(k_e, m)$ is a client-run randomized algorithm. It takes the secret key k_e and the message m as inputs, and outputs the ciphertext C_e.
- $\mathsf{UE.Dec}(k_e, C_e)$ is a deterministic algorithm run by the client. It takes the secret key k_e and the ciphertext C_e as inputs, and outputs the message m or the symbol $\perp$.
- $\mathsf{UE.Next}(k_e, k_{e+1})$ is a randomized algorithm run by the client. It takes the old secret key k_e of the last epoch and the new secret key k_{e+1} of the current epoch as inputs and outputs a re-encrypt token Δ_e or the symbol $\perp$.
- $\mathsf{UE.Upd}(\Delta_e, C_e)$ is a deterministic algorithm run by the server. It takes the re-encrypt token Δ_e and the ciphertext C_e as inputs, and outputs a new ciphertext C_{e+1} under the secret key k_{e+1} or the symbol $\perp$.

The correctness of an updatable encryption scheme ensures that the update of a valid ciphertext C_e from epoch e to epoch $e + 1$ leads to a valid ciphertext C_{e+1} that can be decrypted under the new epoch key k_{e+1}. The UE security definitions of IND-ENC and IND-UPD can be found in [27] and are revisited as follows:

2.6.1 Security models for UE

Here, we present a revisit of the security models IND-ENC and IND-UPD for ciphertext-independent updatable encryption as proposed in [27]. In these models, the encryption key evolves with the epochs. In addition to the challenge ciphertext and the encryption oracle, the adversary is permitted to obtain keys from certain epochs. This is done to reflect the scenario where the client's keys are leaked. Furthermore, the adversary is capable of obtaining some previous versions of the challenge ciphertexts and update tokens, which captures the situation where previous storage on the server may not have been securely erased in time.

The challenger initializes a UE scheme with global state (k_e, Δ_e, S, e) where $k_0 \leftarrow \mathsf{UE.Setup}(1^\lambda)$, $\delta_0 \leftarrow \perp$, and $e \leftarrow 0$, and S consists of initially empty sets $\mathcal{L}, \tilde{\mathcal{L}}, \mathcal{C}, \mathcal{K}$ and $\mathcal{T}$. Furthermore, let $\tilde{e}$ denote the challenge epoch, and e_{end} denote the final epoch in the game.

- $\mathcal{L}$: List of non-challenge ciphertexts (C_e, e) produced by calls to the $\mathcal{O}_{\mathsf{enc}}$ or $\mathcal{O}_{\mathsf{upd}}$ oracle. $\mathcal{O}_{\mathsf{upd}}$ only updates ciphertexts contained in $\mathcal{L}$.
- $\tilde{\mathcal{L}}$: List of updated versions of the challenge ciphertext. $\tilde{\mathcal{L}}$ gets initialized with the challenge ciphertext $(\tilde{C}, \tilde{e})$. Any call to the oracle $\mathcal{O}_{\mathsf{next}}$ oracle automatically updates the challenge ciphertext into the new epoch, which $\mathcal{A}$ can fetch via a $\mathcal{O}_{\mathsf{updC}}$ call.
- $\mathcal{C}$: List of all epochs e in which $\mathcal{A}$ learned an updated version of the challenge ciphertext.
- $\mathcal{K}$: List of all epochs e in which $\mathcal{A}$ corrupted the secret key k_e.
- $\mathcal{T}$: List of all epochs e in which $\mathcal{A}$ corrupted the update token Δ_e.

Adversary is allowed to query encryption oracle $\mathcal{O}_{\mathsf{enc}}$ and ciphertext update oracle $\mathcal{O}_{\mathsf{upd}}$ to get data encryption and re-encryption of previous-epoch ciphertext in the current epoch. Adversary is also allowed to push the key evolving via $\mathcal{O}_{\mathsf{next}}$ oracle and corrupt the existing epoch key and token via $\mathcal{O}_{\mathsf{corrupt}}$ oracle. In the indistinguishability game, adversary is also allowed to get the update of challenge ciphertext via $\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}$ oracle.

$\mathcal{O}_{\mathsf{enc}}(m)$: On input a message $m \in \mathbb{M}$, compute $C \leftarrow \mathsf{UE.Enc}(\mathsf{k_e}, \mathsf{m})$ where k_e is the secret key of the current epoch e. Add C to the list of ciphertexts $\mathcal{L} \leftarrow \mathcal{L} \cup \{(C, e)\}$ and return the ciphertext to the adversary.

$\mathcal{O}_{\mathsf{next}}$: Upon triggering, the oracle $\mathcal{O}_{\mathsf{next}}$ generates new epoch key by running $k_{e+1} \leftarrow \mathsf{UE.Keygen}(e+1)$, and generates a new update token Δ_{e+1} by invoking the function $\mathsf{UE.Next}(k_e, k_{e+1})$. The global state is then updated to $(k_{e+1}, \Delta_{e+1}, S, e+1)$. If a challenge query has been previously made, this operation also updates the challenge ciphertext to the new epoch. Specifically, it executes $C_{e+1} \leftarrow \mathsf{UE.Upd}(\Delta_{e+1}, C_e)$ for each $(C_e, e) \in \tilde{\mathcal{L}}$ and adds the resulting pair $(C_{e+1}, e+1)$ to the set $\tilde{\mathcal{L}} \cup \{(C_{e+1}, e+1)\}$.

$\mathcal{O}_{\mathsf{upd}}(C_{e-1})$: On input ciphertext C_{e-1}, check that $(C_{e-1}, e-1) \in \mathcal{L}$ (i.e., it is a valid ciphertext from the previous epoch $e-1$), compute $C_e \leftarrow \mathsf{UE.Upd}(\Delta_e, C_{e-1})$, add (C_e, e) to the list $\mathcal{L}$, and output C_e to $\mathcal{A}$.

$\mathcal{O}_{\mathsf{corrupt}}(\{\text{token}, \text{key}\}, e^*)$: This oracle models adaptive corruption of the host and owner keys, respectively. The adversary can request a key or update token from the current epoch or any of the previous epochs. The oracle responds as follows:

(1) Upon input token, $e^* \leq e$, the oracle returns Δ_{e^*}, i.e., the update token is leaked. Calling the oracle in this mode sets $\mathcal{T} \leftarrow \mathcal{T} \cup \{e^*\}$.

(2) Upon input key, $e^* \leq e$, the oracle returns k_{e^*}, that is, the secret key is leaked. Calling the oracle in this mode sets $\mathcal{K} \leftarrow \mathcal{K} \cup \{e^*\}$.

$\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}$: Returns the current challenge ciphertext C_e from $\tilde{\mathcal{L}}$. Note that the challenge ciphertext gets updated to the new epoch by the O_{next} oracle, whenever a new key is generated. Calling this oracle sets $\mathcal{C} \leftarrow \mathcal{C} \cup \{e\}$.

Extended sets. Since in RISE both the key update and ciphertext update are bi-directional, we define the extended sets $\mathcal{C}^*$ and $\mathcal{K}^*$ as follows.

Recall that $\mathcal{K}^*$ denotes the set of epochs in which the adversary has obtained the secret key:

$$\mathcal{K}^* \leftarrow \{e \in \{0, \ldots, e_{\text{end}}\} \mid \text{ corrupt-key } (e) = \mathsf{true}\}$$

where corrupt-key $(e) = \mathsf{true}$ iff: $(e \in \mathcal{K}) \vee (\text{ corrupt-key } (e-1) \wedge e \in \mathcal{T}) \vee (\text{ corrupt-key } (e+1) \wedge e+1 \in \mathcal{T})$.

Define the set $\mathcal{C}^*$ containing all challenge-equal epochs:

$$\mathcal{C}^* \leftarrow \{e \in \{0, \ldots, e_{\text{end}}\} \mid \text{ challenge-equal } (e) = \text{ true }\}$$

where challenge-equal $(e) = \mathsf{true}$ iff: $(e = \tilde{e}) \vee (e \in \mathcal{C}) \vee (\text{ challenge-equal } (e-1) \wedge e \in \mathcal{T}^*) \vee (\text{challenge-equal } (e+1) \wedge e+1 \in \mathcal{T}^*)$.

IND-ENC. IND-ENC security ensures that ciphertexts obtained from the UE.Enc algorithm do not reveal any information about the underlying plaintexts even when $\mathcal{A}$ adaptively compromises a number of keys and tokens before and after the challenge epoch:

$\mathsf{Exp}^{\mathsf{ind\text{-}enc\text{-}cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, b)$

1 : $k_0 \leftarrow \mathsf{UE.Setup}(1^\lambda)$

2 : $e \leftarrow 0;\ \tilde{e} \leftarrow \perp;\ \mathcal{L} \leftarrow \emptyset$

3 : $(m_0, m_1, \text{state}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}}(1^\lambda)$

4 : proceed only if $|m_0| = |m_1|$

5 : $\tilde{e} \leftarrow e$

6 : $\tilde{C} \leftarrow \mathsf{UE.Enc}\,(k_{\tilde{e}}, m_b)\,,\ \tilde{\mathcal{L}} \leftarrow \{(\tilde{C}, \tilde{e})\}$

7 : $b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}, \mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}}(\text{state})$

8 : **return** b' **if** $\mathcal{C}^* \cap \mathcal{K}^* = \emptyset$

FIGURE 2.3. The game of IND-ENC for UE

DEFINITION 2.6.2 (IND-ENC). An updatable encryption scheme UE is said to be IND-ENC secure if for any P.P.T adversary $\mathcal{A}$ it holds that

$$\left|\Pr[\mathrm{Exp}^{\mathsf{ind\text{-}enc\text{-}cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Exp}^{\mathsf{ind\text{-}enc\text{-}cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, 1) = 1]\right| = negl(\lambda).$$

IND-UPD. IND-UPD security ensures that an updated ciphertext obtained from the UE.Upd algorithm does not reveal any information about the previous ciphertext, even when $\mathcal{A}$ adaptively compromises a number of keys and tokens before and after the challenge epoch adaptive manner:

DEFINITION 2.6.3 (IND-UPD). An updatable encryption scheme UE is said to be IND-UPD secure if for any probabilistic polynomial-time adversary $\mathcal{A}$ it holds that

$$\left|\Pr[\mathrm{Exp}^{\mathsf{ind-upd-cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Exp}^{\mathsf{ind-upd-cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, 1) = 1]\right| = negl(\lambda).$$

2.6.2 One secure construction -RISE.

In this paper, we leverage the homomorphic updatable encryption-RISE [27]. Recall the RISE construction as follows:

- $\mathsf{RISE.Setup}(1^\lambda)$: return pp as public parameter, also an implicit input of the following algorithms.

$\mathrm{Exp}^{\mathsf{ind\text{-}upd\text{-}cpa}}_{\mathsf{UE},\mathcal{A}}(1^\lambda, b)$

$k_0 \leftarrow \mathsf{UE.Setup}(1^\lambda)$

$e \leftarrow 0;\ \tilde{e} \leftarrow \perp;\ \mathcal{L} \leftarrow \emptyset$

$(C_0, C_1, \text{state}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}(1^\lambda)}$

proceed only if $(C_0, \tilde{e}-1) \in \mathcal{L}$ and $(C_1, \tilde{e}-1) \in \mathcal{L}$ and $|C_0| = |C_1|$

$\tilde{e} \leftarrow e$

$\tilde{C} \leftarrow \mathsf{UE.Upd}(\Delta_{\tilde{e}}, C_b),\ \tilde{\mathcal{L}} \leftarrow \{(\tilde{C}, \tilde{e})\}$

$b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{enc}}, \mathcal{O}_{\mathsf{next}}, \mathcal{O}_{\mathsf{upd}}, \mathcal{O}_{\mathsf{corrupt}}, \mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}}(\text{state})$

return b' **if** $\tilde{e} \notin \mathcal{T}^* \ \wedge\ \mathcal{C}^* \cap \mathcal{K}^* = \emptyset \wedge$ if UE.Upd is deterministic, then

$\mathcal{A}$ has neither queried $\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}(C_0)$ nor $\mathcal{O}_{\mathsf{upd}\tilde{\mathsf{C}}}(C_1)$ in epoch $\tilde{e}$

FIGURE 2.4. The game of IND-UPD for UE

- $\mathsf{RISE.Keygen}(e)$: $k_e \leftarrow\$\ \mathbb{Z}_p^*$.
- $\mathsf{RISE.Enc}(k_e, m)$: $y = g^{k_e}$, $r \leftarrow\$\ \mathbb{Z}_q$, return $C_e \leftarrow (y^r, g^r m)$.
- $\mathsf{RISE.Dec}(k_e, C_e)$: parse $C_e = (C_1, C_2)$, return $m \leftarrow C_2 \cdot C_1^{-1/k_e}$.
- $\mathsf{RISE.Next}(k_e, k_{e+1})$: $\Delta_{e+1} \leftarrow (k_{e+1}/k_e, g^{k_{e+1}})$, return Δ_{e+1}.
- $\mathsf{RISE.Upd}(\Delta_{e+1}, C_e)$: parse $\Delta_{e+1} = (\Delta, y')$ and $C_e = (C_1, C_2)$, $r' \leftarrow\$\ \mathbb{Z}_q$, $C_1' \leftarrow C_1^{\Delta} \cdot y'^{r'}$, $C_2' \leftarrow C_2 \cdot g^{r'}$, return $C_{e+1} \leftarrow (C_1', C_2')$.

The updatable RISE encryption scheme has been proven to be IND-ENC and IND-UPD secure under the decisional Diffie-Hellman (DDH) assumption [27]. Furthermore, it has been observed that RISE is homomorphic under its encryption algorithm Enc and decryption algorithm Dec. Specifically, given two plaintexts m and m', their respective RISE ciphertexts are $\mathsf{RISE.Enc}(k_e, m) = (C_1, C_2)$ and $\mathsf{RISE.Enc}(k_e, m') = (C_1', C_2')$. Then, their product is computed as $\mathsf{RISE.Enc}(k_e, m) \cdot \mathsf{RISE.Enc}(k_e, m') = (C_1 \cdot C_1', C_2 \cdot C_2')$. The decryption algorithm of RISE satisfies $\mathsf{RISE.Dec}(\mathsf{RISE.Enc}(k_e, m) \cdot \mathsf{RISE.Enc}(k_e, m')) = m \cdot m'$.

CHAPTER 3

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

---

3.1 Introduction

Modern Apps increasingly leverage cloud services to store and manage their clients' data. Email providers, online document platforms, and music libraries that synchronize across multiple devices are all storing users data in the cloud. Leveraging public cloud storage services such as Google Cloud Storage, Amazon S3, etc, enables companies to design their Apps with flexibility and scalability without the need to invest in their own infrastructure. However, this also significantly increases the risk of leaking client's information [28]. Indeed, since the data is owned by the users but collected by the App and stored in the cloud, privacy has become a serious concern. Naturally, users have the right to demand that their data are protected, preventing access by illegitimate users, the App provider, or even the operators of the cloud server storing the content. 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 remarkable success of cryptography research in areas, such as password-protected secret sharing [29, 30, 31, 32, 33, 34, 35, 36], password-hardened encryption [37], and oblivious key management [38], one might naturally assume that these problems have been solved. However, somewhat unexpectedly, when considering the requirements from real-world applications, the challenge of augmenting an app with secure and usable cloud storage remains unresolved.

Portability & cloud-blindness. Portability is an essential feature, as users frequently access their content from multiple devices, including mobile phones, laptops, and desktops. Furthermore, users may lose or break a device or need to transfer their data to a new one, necessitating data recovery from storage. The cloud and any server could be compromised or infected with viruses, potentially leaking internal states and inputs to attackers. To protect user data from a malicious cloud, both the stored data and the user's secret must remain hidden from the cloud, preventing any malicious cloud entity from using the user's secret to access their data.

A straightforward approach of encrypting data with a strong key and storing it on the user's device compromises the essential requirement of user mobility across devices, which is a primary advantage of utilizing the cloud. Conversely, encrypting data directly with a password (or a password derived key) that a human can remember, as suggested in [39], poses a significant security risk. From a security perspective, this method relies on a weak (low entropy) key, making it vulnerable to offline dictionary attacks by a corrupted cloud or any entity that breaches the cloud.

This inherent dilemma between cloud-blindness and portability suggests that blind and portable cloud storage cannot be achieved within the theoretical model that involves only a single, potentially corrupt, storage provider.[1] However, a closer examination of existing system settings reveals that, besides renting public cloud storage servers, most app providers such as slack also deploy separate, independently-run servers (corporate or private cloud components) for routine administration, such as user management. For instance[2], Netflix uses AWS for content delivery and its own infrastructure for user management. Slack employs AWS for hosting but maintains private servers for account management. Apple

[1]Indeed, several works, such as Boyen [40], have formally shown 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 who obtain the server's database. These attackers, holding "known content," can always perform offline attacks pretending to be the user.

[2]Netflix Case Study. AWS. https://aws.amazon.com/solutions/case-studies/netflix/; Slack is where work happens. AWS Case Study. https://aws.amazon.com/solutions/case-studies/slack/; Apple reportedly signed a deal with Google to use Google Cloud for iCloud services. The Verge. https://www.theverge.com/2018/2/26/17053926/apple-google-icloud-deal; Salesforce Now Runs on AWS Infrastructure. AWS Case Study. https://aws.amazon.com/solutions/case-studies/salesforce/.

iCloud combines its own data centers with third-party cloud services for storage and handles authentication internally. Similarly, Salesforce uses both its data centers and AWS for a hybrid infrastructure. Therefore, a natural idea is to consider this real-world system model with more than one independent server.[3]

Indeed, cryptographic tools like password-hardened encryption (PHE) [37] leverage external cryptographic services to enhance the security of password-protected storage against external attackers who might compromise server storage and secrets, limiting their attacks to online attempts on user passwords. However, PHE does not mitigate threats from internal attackers, such as a malicious server or server infected with malware leaking internal states to attackers. In PHE, the user's password is sent to the storage server and then strengthened into a robust key by an external "rate limiter" server through a password hardening service [41, 42, 43]. Subsequently, the storage server employs this strong key for encryption and decryption, thus thwarting external attackers breaching the storage server. Nevertheless, PHE does not achieve the desired level of cloud-blindness. Put it in another way, the primary objective of PHE (and password hardening) is to deliver secure services to clients without cryptographic capabilities, delegating cryptographic operations including password hardening, encryption, and decryption to the storage server. Therefore, achieving the level of *cloud-blindness* we desire surpasses the security model of PHE.

Deployability (compatibility). From a theoretical feasibility perspective, using multiple servers enables several cryptographic primitives to achieve both cloud-blindness and portability. Examples include general secure multiparty computation (MPC) such as secret sharing via mutually authenticated channels between the user and servers, and password-protected secret sharing (PPSS) [44, 29, 30, 31, 32, 36, 33, 34, 35] with an assumption of server-authenticated channels or server PKI model. Specifically, a (t, n)-PPSS protocol, a type of MPC protocol that achieves password-based user authentication and secret sharing, allows a user to distribute a secret among n servers and later reconstruct it with a password by contacting $t + 1$ of these servers. An attacker compromising up to t servers and controlling all communication channels

[3]Even with trusted hardware, a careful protocol design is needed, otherwise the vulnerability of trusted hardware can cause the protocol insecure, as we demonstrate in Sec 3.1.3 about secure backup.

gains no information about the secret. However, despite their theoretical strengths, these solutions face significant challenges in terms of deployability in current apps.

All existing PPSS solutions require specific algebraic operations on each server, creating a significant obstacle for scalable deployment. These operations are not supported by the APIs of most commercial cloud storage services, which typically offer only basic functionalities. For instance, services like Dropbox and Amazon S3 generally provide basic APIs for upload, retrieval, deletion, and access control. Consequently, app providers are forced to use cloud computing services, such as Amazon EC2, which are significantly more expensive than non-programmable cloud storage services like Amazon S3. For example, a rough estimation shows that an app provider with just one million users would have to pay around two million dollars more if deploying the same secure storage on EC2 instead of S3 for just one month (see Sec 3.6 for details of the estimation).

In this paper, we consider the architecture with one plain storage server and one App server (sometimes also called key server), shown in Figure 3.1.

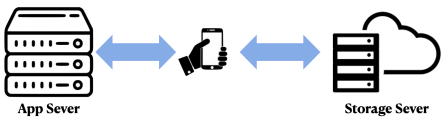


FIGURE 3.1. The architecture.

Built-in App login. Given the limited API support of existing storage services, where the only supported authentication is password login, it is natural to leverage the app server's power to harden the user's password, as in [45], and use the hardened password to log into the storage server. This ensures that the storage server cannot launch offline attacks if it becomes corrupted. However, due to the storage server's limited API support, this mechanism cannot be straightforwardly implemented by the storage server to help users authenticate to the app server without risking offline attacks from a malicious app server. Furthermore, app servers typically already have password-based login mechanisms in place to authenticate users for existing services before the secure storage augmentation. This built-in App login mechanism can weaken the security of the password hardening service because, in the real world, users often reuse their passwords or slight variations of them [46]. If the app server is corrupted,

the login password might be learned through offline attacks, allowing the adversary to use this password to breach the secure storage system. Therefore, a new design is required to enable users to reuse their existing app login credentials without compromising 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 such protocols, in fact, not as ready to use and augment existing systems. Note that the industrial-context constraints, such as economic and engineering overhead are often ignored in the literature, but become particularly essential in upgrading an already heavily used App which cannot afford down time or the loss of existing registered users (see Table 3.1). We, therefore, conclude that we still lack an industrial-system-oriented feasible solution for the basic question:

How can we efficiently augment apps with cloud-secure (blind) and portable storage deployable on non-programmable cloud storage services, while ensuring compatibility with the app's built-in login mechanisms?

TABLE 3.1. The comparison of different primitives. *Portability* means that password is the only secret users need for the system so that they can access the system at any device they want. *Cloud-blindness* means that neither corrupted server can recover the user's data. *Deployability* means the system could deploy on the non-programmable cloud storage. *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.

	PPSS	PHE [37]	OKMS [38]	PBCS
Portability	Yes	Yes	No	Yes
Cloud-blindness	Yes	No	Yes	Yes
Deployability	No	No	No	Yes
Built-in App login	No	No	No	Yes
One pwd	Yes	Yes	No	Yes

[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 Section.3.7.

3.1.1 Our results

We model, design, analyze, implement, and experiment with a novel system named Portable Blind Cloud Storage/ (PBCS). The system consists of three entities: the user, the cloud storage server (or data server), and the key server (or app server). It provides data security against a malicious data server and key server without collusion and aims to achieve four goals simultaneously: *portability*, *cloud-blindness*, *deployability*, and *built-in app login*. At the core of our PBCS system is a "Give-and-Take protocol" that involves the user with their current device, a data server (plain storage) and a key server (See Fig. 3.2, 3.9, 3.10). More detailed comparisons with previous works can be found in Section 3.1.4 and Table 3.1.

Portability: PBCS allows the users to securely access their content and log in to the App on any device using one passphrase. The security and the usability of the system remain unaffected when some of his devices are lost, as no secret is stored on the user's device. See Sec. 3.4 for detailed discussions.

Cloud-blindness: PBCS provides "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 cause the client to accept tampered data. See security models and analysis in Sec. 3.3.2.

Deployability on non-programmable cloud storage: PBCS minimizes the requirements of the data server, so that it can be deployed in existing storage services without programmable support like Amazon S3, Dropbox, etc. In our protocol, the data server only needs to support password login and basic data storage/retrieval functions. We implement our protocol with simple optimizations. The simplicity and the essentially "negligible" overhead of our PBCS system are demonstrated by evaluations conducted in a real network environment with the data server deployed on Amazon S3. See Sec. 3.6.

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 engineering overhead, but also minimizes the impact 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 the same.

3.1.2 Technical overview.

To achieve the four goals simultaneously, there are several challenges to address. From a security point of view, we need to be particularly careful about potential offline attacks. At a very high-level, we let client leverage each server to log in to 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 perform online attacks (which are easier to defend against) instead of offline attacks.

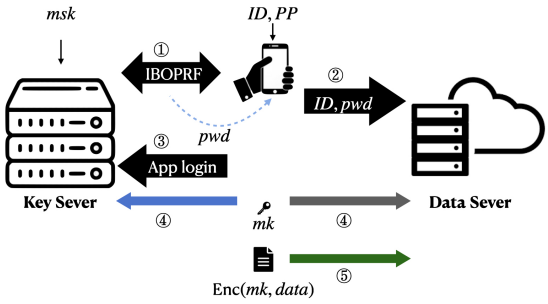


FIGURE 3.2. The data flow of PBCS. 1). the client derives his password by querying the key server with his passphrase. 2). the client logs in to the data server with his ID and the derived password. 3). the client logs in to the key server. 4). the client "shares" his master key. 5). the client encrypts his data with the master key and deposit the ciphertext to the data server.

User authentication to a plain storage server. A computation-barren storage service like Dropbox typically supports only a password login mechanism, which rules out PPSS since it requires the storage server to perform algebraic operations. Our key observation is that the user can leverage a key server to generate and maintain a high-entropy "password" for logging into the data server. Inspired by Oblivious PRF techniques such as those in [45, 41, 43], we enable the key server to provide a service called identity-based password hardening. This service enables the client to compute a unique pseudo-random string bound to a low-entropy passphrase and the user's identity without revealing the passphrase to any server (see Sec.3.2

for details). The hidden yet binding passphrase ensures cloud blindness and guarantees that only the correct passphrase can access user data. The clear yet binding identity enables the key server to implement rate limiting against brute force attacks on the user. The "hardened" password is then used to register with and later log into the data server (Steps 1 and 2 in Fig.3.2).

User authentication with a built-in App login. Since users will mostly perform App login anyway, to use it without threatening the security of passphrase used in secure storage, we allow the user to 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 themselves to the key server must log in to 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, logging in to the App and accessing secure storage are 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 [44, 29, 30, 31, 33, 32, 36] (either the proof or the verification) on the non-programmable 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 [4] where part of the KEM encryption key (the randomness r_{id}) is stored

[4]Please note that in this thesis, we slightly abuse the standard notion of KEM/DEM that refers to hybrid encryption in asymmetric setting. We refer to KEM/DEM as key encapsulation mechanism / data encapsulation mechanism in the symmetric setting, which is similar to [25]'s abuse using AE for both KEM and DEM, also called AE-hybrid.

on one of the servers. The security properties of standard authenticated encryption will not trivially hold when the encryption key is modified. To guarantee the integrity of mk, we introduce a *twisted Enc-then-Hash* paradigm, enabling the client (decryptor) to verify that the KEM encryption key has not been modified.

3.1.3 Extended applications

Moreover, instantiating PBCS in different settings can lead to various interesting new applications. These include not only various applications of PPSS such as *password manager* [29] and *portable cryptocurrency wallet* [34], but also other intriguing scenarios. Below, we provide three examples.

Secure personal repository. One of the most suitable settings for our PBCS is to keep personal data secure in the cloud while ensuring availability across devices. Two typical application examples are shown below: The first involves using PBCS to upgrade a plain storage service, essentially building a secure shell for the personal cloud storage to achieve private and portable data access. Concretely, one can deploy an independent key server to do the key management, leverage the personal storage server (where data is visible only to users and the server) like Dropbox and Github private repository to provide the storage service, and wrap the original storage app with a shell to provide access to a private "Dropbox" or "Github" ensuring that plain data is visible only to the user themselves.

The other example involves 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 [47, 48, 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 Snap. The client's videos are kept private to both Google and Snap. The PBCS described in this paper significantly improves upon the preliminary version in both usability and security. Particularly, we allow the data server to be *plain* cloud storage server, and reuse the App login, requiring only a single password after the upgrade.

Secure backup system for E2EE messaging Apps. Encrypted messengers like WhatsApp routinely back up users' text messages and contact lists to cloud servers. These backups undermine much of the strong security offered by E2EE as they make it much easier for companies and hackers to obtain users' plain content. The messaging service WhatsApp, provides a "secure" backup module based on hardware secure modules (HSMs)[50]. The protocol will fail to provide security when the HSMs are broken due to various side channel attacks [51, 52, 53, 54] or subversion of the HSM maker. Because the core of WhatsApp backup module is running a PAKE protocol between users and HSMs, where WhatsApp server forwards messages for them in the middle. So, the design security relies on HSM and users' password. Once HSM is broken, the backup suffers unlimited offline attack on users' password. Therefore, even the HSMs are deployed separately from WhatsApp server, the alone broken of HSM makes the protocol insecure. 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 using the App provider's server or HSM as the key management server of PBCS. Therefore the contact list can only be accessed by the user with the correct passphrase on any device. Particularly, even when the App server is corrupted, or the HSM maker maliciously embedded any backdoor, or any attacker steals the storage of HSM via a side-channel attack, PBCS can still protect the security of the backup data, unlike WhatsApp's backup solution.

Portable personal AI assistant with privacy guarantee. Internet companies increasingly strive to provide personalized models for their customers based on the data collected over time from user devices. Traditional methods involve transmitting personal data to the cloud and training on the dataset 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, as well as legal issues such as compliance with the EU General Data Protection Regulation (GDPR) [55]. Complying with such regulations while maintaining the utility of the AI assistant is important leading to the suggestion of on-device learning, as exemplified by "federated learning" [56]. However, restricting the user model to a single device can be limiting. Using our solution, once a personal model is generated, the user can securely store their 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 over their data and personal AI assistant model which remains private when passing across devices.

3.1.4 Other related works

Password hardening (PH), introduced in [41] and modeled in [42, 43], aims to enhance the security of password login with the help of an additional server that does not collude with the login server. PH focuses on improving security against external attackers and assumes the server interacting with users is honest. In contrast, we aim to improve the security of storage for apps where both servers can be malicious. Another password management system, SPHINX, proposed in [45], uses a local trusted device to protect password security even if the password manager is compromised. However, this trusted device does not meet our portability requirement. Nevertheless, we share a similar technical idea of using Oblivious PRF, and this paper designs an important building block, IBOPRF, for the PBCS system.

Password-hardened encryption (PHE) [37] 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 “harden” the password to a strong key, and use it to do encryption and decryption. One advantage of PHE is that the user side 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.

One may suggest to tune a bit PHE to achieve PBCS. For example, let the user take over the client in PHE to interact with rate limiter to finish the authentication to rate limiter and to derive an encryption key. Then the user retrieve ciphertext from other storage server and decrypt the ciphertext. This solution suffers two problems. One is that since user can

authenticate to the rate limiter with the only password, a malicious rate limiter can launch off-line dictionary attack to break user's password, so as to user's data privacy. The other is that user still needs to authenticate to the extra data server. Similarly, single password authentication between user and one server suffers from off-line dictionary attack of malicious server. So it is nontrivial to tune PHE or PH in a straightforward way to achieve PBCS.

Another related primitive is the Threshold Oblivious Key Management Service (Threshold OKMS) proposed in [38], which helps clients manage secret keys via multiple untrusted storage servers. However, Threshold OKMS assumes the user is authenticated by any method. Our portability requirement necessitates password-based authentication to two servers and password-based secret recovery, which can be achieved by two independent password-based authentications to two servers and a 2-out-of-2 Threshold OKMS for secret key management. This design, however, requires three independent passwords, which is inconvenient and vulnerable since users often reuse similar passwords across different mechanisms [46]. It is suggested [38] to instantiate OKMS within Hardware Security Modules (HSMs) [57] to prevent unauthorized queries, but the client-side storage for OKMS makes it non-portable (this limitation also applies to end-to-end hardware modules [57]).

The PPSS [29, 30, 31, 32, 33, 34, 35, 36] 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[5] 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 log in to the App. More importantly, PPSS cannot be deployed on a non-programmable cloud storage which is the main stream (and cheaper) commercial cloud product. Very recently, Dauterman et al., [59] 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

[5]Although some PPSS[33, 58] may save 2 rounds than our PBCS .

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 3.1.

3.2 Identity Based Oblivious PRF

The identity based oblivious pseudorandom 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 [45]). 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) [60, 61, 33] and the password hardening service [41, 43], 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.

3.2.1 Syntax

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 [41, 43], 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 [62] and the iCloud Keychain [63] 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.

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

- $\mathsf{CEval}_1(pp, id, x) \rightarrow (ch, st)$: 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.

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

- $\mathsf{CEval}_2(pp, id, rp, st) \rightarrow y$: 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.

3.2.2 Security properties

The IBOPRF should guarantee the following properties

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. Intuitively, 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, , even though he can arbitrarily query the server. More precisely, we have the following definition.

DEFINITION 3.2.1. 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. 3.3 is ϵ.

Pseudorandom $\mathsf{IBOPRF}^{\mathcal{A}}$

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

Oblivious $\mathsf{IBOPRF}^{\mathcal{A}}$

$b \leftarrow\$ \{0,1\}$
$(pp, msk) \leftarrow\$ \mathsf{Setup}(1^\lambda)$
$id^* \leftarrow\$ \{0,1\}^\lambda$
$x^* \leftarrow\$ \mathcal{D}$
$(ch^*, st^*) \leftarrow\$ \mathsf{CEval}_1(pp, id^*, x^*)$
$(rp, s) \leftarrow\$ \mathcal{A}(pp, msk, id^*, ch^*)$
  // s represents $\mathcal{A}$'s state
$y_0 \leftarrow\$ \mathsf{CEval}_2(pp, id^*, rp, st^*)$
$y_1, \ldots, y_t \leftarrow\$ \mathcal{A}^{C()}(pp, msk, s)$
if for $i = 1, \ldots, t$
  $y_i = y_0$
  return 1
else return 0

Oracle $C(\cdot)$

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

FIGURE 3.3. 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^*.

Obliviousness. It models the security of IBOPRF against the malicious server. Obliviousness 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. More precisely, we have the following definition.

DEFINITION 3.2.2. 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, t)-obliviousness if the probability of the successful guess in the game in the right side of Fig. 3.3 is less than ϵ when the adversary could make t different guesses for y.

3.2.3 Construction

The IBOPRF could easily be constructed in the random oracle model. Our design is inspired by the 2HashDH protocol in [34].

- $\mathsf{Setup}(1^\lambda)$: Choose a prime p which is λ-bit long. The input passphrase space and the output password space are within $\{0,1\}^l$ and $\{0,1\}^\lambda$, respectively. The hash function H_1 is from $\{0,1\}^l$ 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 $\{0,1\}^l \times \mathbb{Z}_p$ to $\{0,1\}^\lambda$. Form the public parameter $pp = (p, \mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_3)$. And 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 $y = \mathsf{H}_3(x, rp^{1/st})$.

3.2.4 Security analysis

We will show the IBOPRF construction in Sec.3.2.3 satisfies the security three properties: uniqueness, pseudorandomness and obliviousness.

Uniqueness. The uniqueness is obvious, since the function's output 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 [33, 34], which states that for any polynomial time adversary $\mathcal{A}$,

$\Pr_{k \leftarrow\$ \mathbb{Z}_p^*, g_i \leftarrow\$ \mathbb{G}}[\mathcal{A}^{(\cdot)^k, \mathsf{DDH}(\cdot)}(g, g^k, g_1, \ldots, g_n) = S]$ is negligible, where $S = \{(g_{j_s}, g_{j_s}^k) | S = 1, \ldots, Q+1\}$, 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.

Concretely, for our construction of IBOPRF, note that H_1, H_2 and H_3 are modeled as random oracles. If the adversary has never queried with $(id^*, \cdot)$ on the oracle $S()$, that is, no query of random oracle $\mathsf{H}_2(id^*, msk)$, so k_{id^*} is completely random. Otherwise, $\mathcal{A}$ has queried with $(id^*, \cdot)$ on the oracle $S()$ no more than B times and the challenger $\mathcal{C}$ responds using k_{id}, which is exactly the case that (N, Q) one-more Diffie-Hellman assumption captures, where k is represented with k_{id}. Moreover, in this case, if the adversary has not queried $(x^*, \mathsf{H}_1(x^*)^{k_{id^*}})$ to the random oracle H_3, the value y should be indistinguishable from a truly random value to the adversary, otherwise, adversary can leverage the advantage of distinguishing two values to break the (N, Q) one-more Diffie-Hellman assumption. 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 t guesses is less than $O(t/d)$.

Concretely, adversary can make at most t guesses on the IBOPRF output. Adversary can query $C()$ to get $ch = \mathsf{H}_1^{st}(x^*)$ where H_1 is modeled as random oracle to map x^* to a random new output and st is randomly selected from $\mathbb{Z}_p^*$, so ch does not leak any information of x^*. The target value $y_0 = \mathsf{H}_3(x^*, \mathsf{H}_1^{k_{id}}(x^*))$ is unpredictable to the adversary due to the random oracle property of H_3 if adversary does not guess x^* correctly. So, adversary can only randomly

guess x^*. Therefore, the probability of correct guess is less than $O(t/d)$ via at most t guesses on the all possible d values.

3.3 Architecture and Definition

As we explained in the introduction, we aim to upgrade an existing App (having a non-programmable cloud storage) with a secure storage function while minimizing the influence on usability. Therefore, the PBCS fully leverages the existing infrastructure of a typical App. Specifically, a typical App[64, 48, 65] (maybe without the secure storage) may use a management 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 log in to his cloud storage account via his identity and 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. 3.2).

As in standard practice, the PBCS system consists of two parts, i.e., the key encapsulation mechanism (KEM) 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) to encrypt the actual content with mk. The DEM part can be easily instantiated through any standard authentication encryption; and the confidentiality and integrity of the content depend 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 of walking through the whole system, we refer to Sec. 3.4.2.

3.3.1 Syntax of the KEM Part

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 logs in to 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 logging in to 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 3.3.1. 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.

3.3.2 Security Threats and Models

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

- 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 [66, 67] 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. We first consider the situation that the adversary corrupts some illegitimate users, and propose the IND-EXA security (here EXA denotes EXternal attackers) in Def. 3.3.2. Then we consider the situation in 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. 3.3.3. Similarly, we then

model security against the compromised data server, which we call IND-CDS security (CDS denotes Compromised Data Server), by switching the role of $\mathcal{K}$ and $\mathcal{D}$. Finally, we consider the situation that two servers collude, and define the IND-CBS security against collusive servers (here CBS demotes Collusive Both Servers).

We note that the security against a compromised key (or data) server implies the security against illegitimate users, because the compromised server himself can disguise as an illegitimate user, so attackers who can compromise one server are more powerful than those only compromise illegitimate users, and the security under the same goal but against more powerful attackers is stronger; Also, the security in case that both servers got compromised falls back to the single server case. We defer more security discussions to the end of this section.

Modeling illegitimate users We first provide an experiment in Fig. 3.4 to capture the security against illegitimate users. During the experiment, the challenger simulates the data server $\mathcal{D}$, the key management server $\mathcal{K}$ and one client $\mathcal{U}_{id^*}$ with the challenge identity id^*, while the adversary can register as clients with arbitrary id except id^*, that captures the illegitimate users setting. During the experiment, the adversary $\mathcal{A}$ can arbitrarily communicate with two servers for arbitrary times, but he cannot obtain the internal state of the challenge client id^*. The goal of the adversary $\mathcal{A}$ is to distinguish the master key mk_0 generated through the protocol between $\mathcal{D}$, $\mathcal{K}$ and $\mathcal{U}_{id^*}$, or a random master key mk_1. The formally definition should be as follows.

DEFINITION 3.3.2 (Secure against an External Attacker). Let PBCS be a Portable Blind Cloud Storage scheme under security parameter λ. Consider the interactive game in Fig.3.4 between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, parameterized by a bit b. The challenger, over the course of the experiment, simulates a data server $\mathcal{D}$, a key management server $\mathcal{K}$ and a challenge client id^* according to the design of the protocol. Let d denote the min-entropy of the passphrase PP_{id^*}. We say that an PBCS is *secure against an external attacker* for the security parameter λ if there exists a negligible function such that for all P.P.T adversaries $\mathcal{A}$ it holds that

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

IND-EXA Game

1 : $b \leftarrow\$ \{0,1\}$

2 : Generate the data servers' parameters $(pp_{\mathcal{D}}, sp_{\mathcal{D}})$ and the key servers' parameters $(pp_{\mathcal{K}}, sp_{\mathcal{K}})$

3 : $\mathcal{A}$ arbitrarily switches in the following two modes:

Mode 1: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{A}() \leftrightarrows \mathcal{D}_{\mathsf{Register}}(s_{\mathcal{D}}) \rangle$

/ $\mathcal{A}$ can register $\mathcal{D}$ with arbitrary identity id and authenticator α_{id}

/ After interaction, the challenger $\mathcal{C}$ updates $\mathcal{D}$'s global state $s_{\mathcal{D}}$

Mode 2: $(s_{\mathcal{K}}, s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{K}_{\mathsf{Give/Take}}(s_{\mathcal{K}}), \mathcal{D}_{\mathsf{Give/Take}}(s_{\mathcal{D}}), \mathcal{A}(\cdot) \rangle$

/ $\mathcal{A}$ interacts with $\mathcal{K}$ and $\mathcal{D}$ by sending any messages as a client

/ $\mathcal{A}$ assign $\mathcal{K}$ and $\mathcal{D}$ to execute any procedures (Give, Take)

/ $\mathcal{A}$ views all the responses from $\mathcal{K}$ and $\mathcal{D}$ back to the client

/ The challenger $\mathcal{C}$ updates $\mathcal{K}$ and $\mathcal{D}$'s global state $s_{\mathcal{K}}$, $s_{\mathcal{D}}$

4 : $id^* \leftarrow\$ \mathcal{A}$

5 : $s_{id^*} \leftarrow\$ \mathcal{C}$

6 : $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{U}_{\mathsf{Register}}(id^*) \leftrightarrows \mathcal{D}_{\mathsf{Register}}(s_{\mathcal{D}}) \rangle$

7 : $\mathcal{A}$ arbitrarily switches in the following two modes:

Mode 1: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{A}() \leftrightarrows \mathcal{D}_{\mathsf{Register}}(s_{\mathcal{D}}) \rangle$

Mode 2: $(s_{\mathcal{K}}, s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{K}_{\mathsf{Give/Take}}(s_{\mathcal{K}}), \mathcal{D}_{\mathsf{Give/Take}}(s_{\mathcal{D}}), \mathcal{A}(\cdot) \rangle$

/ Similar to Step 2

8 : $(s_{\mathcal{K}}, s_{\mathcal{D}}, mk_0) \leftarrow\$ \langle \mathcal{K}_{\mathsf{Give}}(s_{\mathcal{K}}), \mathcal{D}_{\mathsf{Give}}(s_{\mathcal{D}}), \mathcal{U}_{\mathsf{Give}}(id^*, PP) \rangle$

9 : $mk_1 \leftarrow\$ \{0,1\}^*$

10 : $\mathcal{A}$ arbitrarily switches in the following two modes:

Mode 1: $(\cdot, s_{\mathcal{D}}) \leftarrow\$ \langle \mathcal{A}(mk_b) \leftrightarrows \mathcal{D}_{\mathsf{Register}}(s_{\mathcal{D}}) \rangle$

Mode 2: $(s_{\mathcal{K}}, s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{K}_{\mathsf{Give}}(s_{\mathcal{K}}), \mathcal{D}_{\mathsf{Give}}(s_{\mathcal{D}}), \mathcal{A}(mk_b) \rangle$

Mode 3: $(s_{\mathcal{K}}, s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{K}_{\mathsf{Take}}(s_{\mathcal{K}}), \mathcal{D}_{\mathsf{Take}}(s_{\mathcal{D}}), \mathcal{A}(mk_b) \rangle$

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** $mk^* = \perp$

$\mathcal{A} \leftarrow mk^*$

/ Similar to Step 2

11 : $b' \leftarrow\$ \mathcal{A}$ / After interaction, $\mathcal{A}$ outputs the guess b'

12 : **return** $b = b'$

FIGURE 3.4. The IND-EXA game.

where $\mathsf{Adv}(d)$ is the ideal security inherited for guessing PP_{id^*}, which is $O\left(\frac{1}{2^d}\right)$ for a small constant.

Modeling the compromised key server. Our IND-CKS experiment is similar to the game-based definition of PPSS in [33]. Intuitively, during the IND-CKS experiment (Fig.3.5), 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 [33, 34] 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 3.3.3. *(Master key confidentiality against the compromised key server.)* Consider the following interactive game $\mathsf{Exp}_{\mathcal{A}}^{\text{IND-CKS}}$ in Fig. 3.5 between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, parametrized by security parameter λ and a bit b.[6] Let n denote the min-entropy of the passphrase PP_{id^*}.[7] 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

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

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.

[6]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 $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$.

[7]Note that n may not directly equal to the length of the passwords, since the distribution of passwords is not uniform [68, 69].

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: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \rangle$

// $\mathcal{A}$ interacts with $\mathcal{D}$ while pretending as $\mathcal{K}$ or clients with any id, and assigns $\mathcal{D}$ to execute any procedures (Give, 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: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}(\cdot) \rangle$ // 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: $(s_{\mathcal{D}}, \cdot) \leftarrow\$ \langle \mathcal{D}_{\mathsf{Give/Take}}(sp_{\mathcal{D}}, s_{\mathcal{D}}), \mathcal{A}() \rangle$ // 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 3.5. The IND-CKS game. The challenger $\mathcal{C}$ simulates the data server $\mathcal{D}$ and the target client $\mathcal{U}_{id^*}$.

Modeling the compromised data server The IND-CDS experiment is similar to the IND-CKS experiment in Fig. 3.5. $\mathcal{C}$, over the course of the experiment, simulates a key server $\mathcal{D}$ and the target client with identity id^* according to the protocol. $\mathcal{A}$ can choose to play the role of a client with any identity except id^* or the data server $\mathcal{K}$. $\mathcal{A}$'s capability captures the setting that the data server and other illegitimate users are compromised. The main difference with IND-CKS presented in the main body is that the adversary can register any identity by himself without interacting with the challenger, since he plays both the roles of the data server and illegitimate clients.

DEFINITION 3.3.4. *(Master key confidentiality against the compromised data server.)* Consider the following interactive game $\mathsf{Exp}_{\mathcal{A}}^{\text{IND-CDS}}$ in Fig. 3.6 between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, parametrized by security parameter λ and a bit b. Let the min-entropy of the passphrase PP_{id^*} and the authenticator α_{id^*} be denoted by n and m respectively. We say that an PBCS is *secure against the compromised key management server* if $\forall$ P.P.T adversary $\mathcal{A}$ it holds that

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

where $\mathsf{Adv}(n)$ is the ideal security inherited for guessing PP_{id^*}, which is $O\left(\frac{1}{2^n}\right)$.

Modeling colluding servers. Similarly we have the following definition for the security against the adversary who compromise both servers. The challenger only acts as the target client with identity id^* to interact with two compromised servers taken by the $\mathcal{A}$ who also acts as illegitimate users, which models the security against collusive servers.

DEFINITION 3.3.5 (Secure against collusive servers). Let PBCS be a Portable Blind Cloud Storage scheme under security parameter λ. Consider the following interactive game in Fig. 3.7 between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, parameterized by a bit b. The challenger, over the course of the experiment only simulates a challenge client id^*. Let d denote the mini-entropy of the passphrase PP_{id^*}. We say that an PBCS is *secure against collusive servers* for the security parameter λ if there exists a negligible function such that for all P.P.T

$\text{IND-CDS}^{\mathcal{A}}_{\mathsf{Enc}}$

$b \leftarrow\$ \{0,1\}$

$msk \leftarrow\$ \mathcal{C}$

$\left(\cdot, \mathcal{L}^{\mathcal{K}}\right) \leftarrow\$ \langle \mathcal{A}, \mathcal{K}_{\mathsf{Take/Give}}(\mathcal{L}^{\mathcal{K}}) \rangle$

$id^* \leftarrow\$ \mathcal{A}, \quad PP_{id^*} \leftarrow\$ \chi$

$\mathbf{pwd}_{id^*} = \mathcal{F}_{msk}(id^*, PP_{id^*})$

$\mathcal{A}$ arbitrary switches between the following modes

Mode 1: $\left(\cdot, \mathcal{L}^{\mathcal{K}}, \perp\right) \leftarrow\$ \left\langle \mathcal{A}(\mathbf{pwd}_{id^*}), \mathcal{K}_{\mathsf{Give}}(\mathcal{L}^{\mathcal{K}}), \mathbf{C}_{\mathsf{Give}}(id^*, \mathbf{pwd}_{id^*}) \right\rangle$

Mode 2: $\left(\cdot, \mathcal{L}^{\mathcal{K}}\right) \leftarrow\$ \left\langle \mathcal{A}(\mathbf{pwd}_{id^*}), \mathcal{K}_{\mathsf{Take/Give}}(\mathcal{L}^{\mathcal{K}}) \right\rangle$

$(s_{id^*}, r_{id^*}, h_{id^*}) \leftarrow\$ \{0,1\}^{\lambda}$

$mk_0 \leftarrow\$ \{0,1\}^*$

$t_{id^*} = \mathsf{KDF}_1(s_{id^*}, PP_{id^*})$

$k_{1,id^*} = \mathsf{KDF}_2(r_{id^*}, PP_{id^*})$

$k_{2,id^*} = \mathsf{KDF}_3(r_{id^*}, PP_{id^*})$

$ct = \mathsf{Enc}_{k_{id^*}}(mk_0)$

$\left(\mathcal{L}^{\mathcal{K}}, 1\right) \leftarrow\$ \left\langle \mathcal{K}_{\mathsf{Give}}(\mathcal{L}^{\mathcal{K}}) \leftarrow (id^*, t_{id^*}, ct, h_{id^*}) \right\rangle$

$mk_1 \leftarrow\$ \{0,1\}^*$

$\mathcal{A} \leftarrow (s_{id^*}, r_{id^*}, h_{id^*})$

$\mathcal{A}$ arbitrary switches between the following modes

Mode 1: $(\cdot, \mathcal{L}^{\mathcal{K}}, \cdot) \leftarrow\$ \left\langle \mathcal{A}(mk_b, \mathbf{pwd}_{id^*}), \mathcal{K}_{\mathsf{Take}}(\mathcal{L}^{\mathcal{K}}), \mathbf{C}_{\mathsf{Take}}(id^*, \alpha_{id^*}, PP_{id^*}) \right\rangle$

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** $mk^* = \perp$

$\mathcal{A} \leftarrow mk^*$

Mode 2: $\left(\cdot, \mathcal{L}^{\mathcal{K}}\right) \leftarrow\$ \left\langle \mathcal{A}(mk_b, \mathbf{pwd}_{id^*}), \mathcal{K}_{\mathsf{Take/Give}}(\mathcal{L}^{\mathcal{K}}) \right\rangle$

$b' \leftarrow\$ \mathcal{A}$

return $b = b'$

FIGURE 3.6. The security game against the compromised data sever.

adversaries $\mathcal{A}$ it holds that

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

where $\mathsf{Adv}(d)$ is the ideal security inherited for guessing PP_{id^*}, which is $O\left(\frac{1}{2^d}\right)$ for a small constant.

IND-CBS Game

1 : $b \leftarrow\$ \{0,1\}$
2 : $id^* \leftarrow\$ \mathcal{A}$
3 : $(s_{id^*}, \cdot) \leftarrow\$ \langle \mathcal{U}_{\mathsf{Register}}(id^*), \mathcal{A} \rangle$
4 : $PP \leftarrow\$ \mathcal{C}$
5 : $(\cdot, mk_0) \leftarrow\$ \langle \mathcal{A}, \mathcal{U}_{\mathsf{Give}}(id^*, s_{id^*}, PP) \rangle$
6 : $mk_1 \leftarrow\$ \{0,1\}^*$
7 : $(\cdot, \cdot) \leftarrow\$ \langle \mathcal{A}, \mathcal{U}_{\mathsf{Take}}(id^*, s_{id^*}, PP) \rangle$
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** $mk^* = \perp$
$\mathcal{A} \leftarrow mk^*$
8 : $b' \leftarrow\$ \mathcal{A}$
9 : **return** $b = b'$

FIGURE 3.7. The IND-CBS game.

Some security discussions. 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 definition aims at capturing the *soundness* property as PPSS [33, 34] 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^*.

Specifically, when the challenge client $\mathcal{U}_{id^*}$ successfully retrieved a master key mk' in the Take procedure (even mk' may not equal to the master key mk_0 generated in the Give procedure), our model allow the adversary to learn mk' when $b = 0$ and give him mk_1 when $b = 1$. If there exists $\mathcal{A}$ who successfully attack soundness then $\mathcal{B}$ can run $\mathcal{A}$ and output 0 if

the challenge client outputs mk_b in Give and mk^* in Take are the same, and 1 otherwise: If $\mathcal{A}$ breaks soundness then $\mathcal{B}$ breaks the IND-CKS security.

3.4 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 user 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 log in to 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 log in to the cloud server, since this passphrase will also be used to log in to 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 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 log in to the key server directly, since this passphrase is also used to log in to 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[8], hence the client could simultaneously login the App. Moreover, since a compromised key server can not learn the passphrase, he can not log in to 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

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

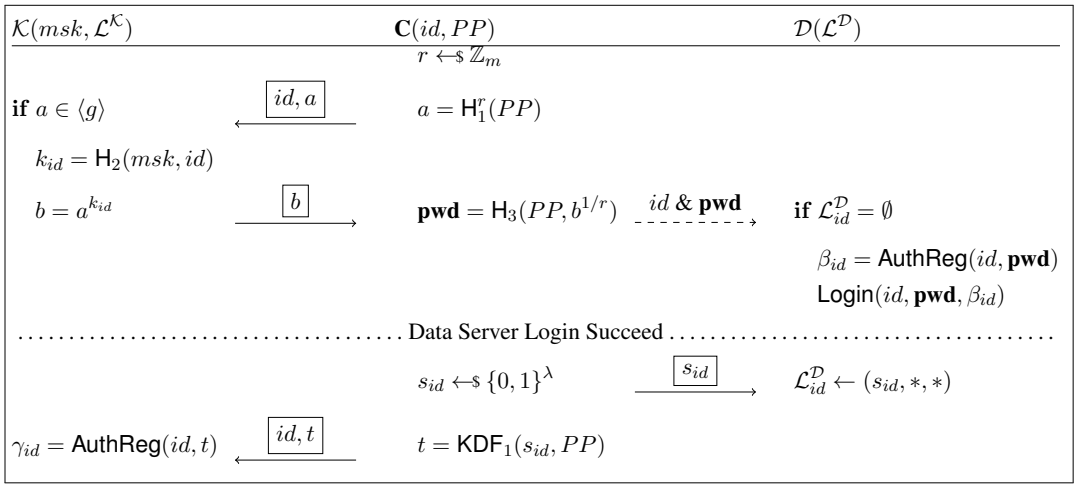


FIGURE 3.8. The Register procedure. The client derives the password **pwd** from the IBOPRF with his passphrase PP and uses it to run AuthReg to register to the data server who obtains and stores the login sub β_{id} for the future login verification. Also the client derives the authentication token 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.

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.

3.4.1 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

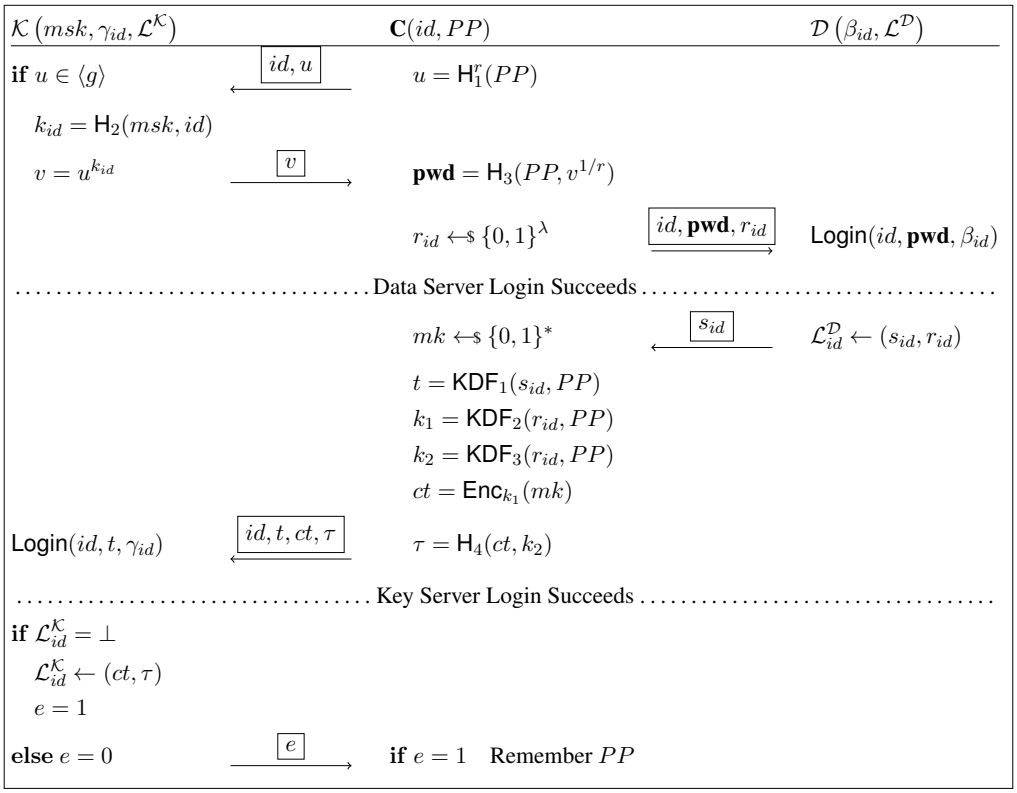


FIGURE 3.9. The Give procedure. The client logs in to the data server via the password **pwd** derived with the key server, where the data server stores the login stub β_{id} for verification. Then the client reconstructs the authentication token t and the key of the authentication encryption k_1 and k_2. The client uses id, t to log in to 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. 3.2.3. The security requires the number of the call of the IBOPRF service for one specific id on the key server side is bounded.

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. 3.2.3, the register procedure is as Fig. 3.8.

- **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. 3.2. 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.3.9 when the IBOPRF is instantiated via 2HashDH as Sec. 3.2.3.

- **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. 3.10, where the IBOPRF is instantiated by the 2HashDH in Sec. 3.2.3. Note that the number of calls of the IBOPRF service for one specific id on the key server side must be limited to guarantee security.

3.4.2 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.

Here we take one App deployment situation as an example, where there is an App server running for user management, and the storage is outsourced to the third-party cloud storage server. When integrating PBCS, the key server implementation is run in App server, and the data server is stilled deployed on the third-party cloud storage server. The App client is augmented with PBCS's client implementation so that the operation flow is as follows:

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.[9]

[9]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.

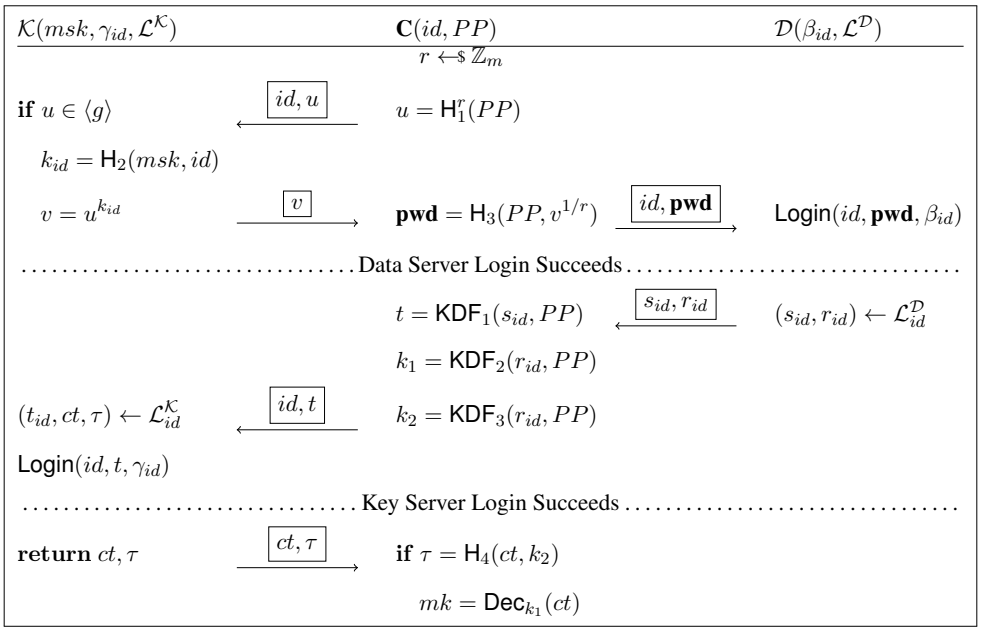


FIGURE 3.10. The Take procedure. The client logs in to the data server via the password **pwd** derived with the key server, reconstructs the App login password t and the key of the authentication encryption k_1 and k_2. Then the client logs in to the App and authenticates himself to the key server with 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. 3.2.3. The security requires the number of the call of the IBOPRF service for one specific id on the key server side is bounded.

When the user logs in to 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 log in to 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.

3.5 Security Analysis

In this section, we provide a formal security analysis to show our "Give-and-Take" protocol can guarantee the security of the master key according to the models defined in Sec. 3.3.2. We will mainly focus on security against a compromised data (or key) server since the collusion of two servers is a rare event in the real world. Nevertheless, the PBCS can still provide a minimal security in the extreme case that two servers collude (See 3.5.3).

3.5.1 Compromising the data server.

In the IND-CDS game, an adversary can compromise the data server as well as registering as users with any id except the challenge identity id^*. By default, the adversary can learn the randomness s_{id} and r_{id} as well as the password **pwd** from the data server side. According to the security definition in Sec. 3.3.2, we need to show that the integrity and the confidentiality of the master key.

To show the master key is confidential to the data server, we argue that the authenticated and confidential channel between the challenge client and the key management server can guarantee that the authentication token t_{id^*} and the ciphertext ct_{id^*} (both are on $\mathcal{K}$) are kept secure. When the adversary can only make $B_{\mathcal{K}}$ tries to log in to $\mathcal{K}$, 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, otherwise he can not compute the correct authentication token t_{id^*}. Due to the pseudorandomness of the IBPH, **pwd** will not leak the information about PP within limited time of invoking the IBPH service, so the adversary has to guess PP blindly. The probability to get passphrase PP within $B_{\mathcal{K}}$ guesses is less than $B_{\mathcal{K}}/2^d$ where d is the min-entropy of the passphrase.

To argue that the client will not accept a tampered master key, we show that the client could verify the randomness r_{id} replied by the data server has not been tampered. Since the hash function H_4 and KDF_3 are random oracles, a tampered r_{id} can not get a $k_3 = \mathsf{KDF}_3(PP, r_{id})$ which satisfy $\tau = \mathsf{H}_4(ct, k_3)$. Hence the client can make certain that the recovered $k_2 =$

$\mathsf{KDF}_2(PP, r_{id})$ is correct and the decryption result mk is not influenced by the malicious data server.

More formally, we have the following theorem.

THEOREM 3.5.1. Let KDF_1 be a random oracle. The min-entropy of the passphrase PP is d. The IBPH is (ϵ, d, B)-pseudorandomness. The total number of the adversary calling IBPH service is bounded by B. The total number of the invalid APP login is bounded by $B_{\mathcal{K}}$. Our scheme is secure against compromised data server, i.e., the probability for any adversary to win the IND-CDS game is less than $1/2 + \epsilon + B_{\mathcal{K}}/2^d + negl(\lambda)$.

Proof: We can take the IND-CDS game in Fig. 3.6. During this game, the challenger plays the roles of the key management server as well as a honest client with the challenge identity id^*. The adversary ($\mathcal{A}$) can impersonate the data server as well as other clients. The goal of the adversary is to distinguish the deposited master key mk_0 from a randomly generated key mk_1.

Specifically, in IND-CDS game in Fig. 3.6, after arbitrary communicating with the key management server, the adversary chooses the challenge identity id^*. The challenger simulates the procedure that an honest client with id^* registers to the adversary who plays the role of the data server. After that, the adversary can communicate with the key management server using arbitrary messages as the Give procedure or the Take procedure. Also the adversary can invoke the Give procedure and communicate with the challenge client id^* using any massages. Then the challenger simulates the procedure that the honest client with id^* generates a master key mk_0 and a passphrase PP as well as computing t_{id^*} and k_{id^*}. However, the randomnesses s_{id^*}, r_{id^*} and h_{id^*} are known to the adversary. The challenger will use k_{id^*} to encrypt mk_0 and get ct. The challenger uses $(id^*, t_{id^*}, ct, h_{id^*})$ to update the list of the key management server $\mathcal{L}^{\mathcal{K}}$ according to the protocol. Then the challenger will generate another random key mk_1, and give one of the two master keys to the adversary according to the random coin b. The adversary can continue to communicate with the key management server $\mathcal{K}$ using arbitrary messages in the Give or Take procedure. Also the adversary can invoke the Take procedure to

communicate with the challenge client id^* using any massages and learn the reconstructed master key mk' when $b = 0$. After that, the adversary will output her guess b'.

To show the security of the master key, first of all we need to make sure that ct in the list $\mathcal{L}^{\mathcal{K}}_{id^*}$ is encrypted by the client (challenger). If not, it means that the $\mathcal{L}^{\mathcal{K}}_{id^*}$ is previously generated by the adversary and deposited before the honest challenge client id^* does. So when the honest client id^* runs the Give procedure, the reply message from $\mathcal{K}$ is always $e = 0$ since $\mathcal{K}$ will check whether every id is used before. With the help of the authenticated and confidential channel between the key management server and the challenge client, the honest challenge client id^* will never generate the valid master key mk_0 since he receives $e = 0$.

Then we move the case that $ct_{id^*} = \mathsf{Enc}_{k_{id^*}}(mk_0)$ in the list $\mathcal{L}^{\mathcal{K}}_{id^*}$ is indeed encrypted by the honest challenge client id^*. In our model, the adversary can not get the real ciphertext ct in $\mathcal{L}^{\mathcal{K}}_{id^*}$ unless he pretends to be the client id^* and communicates with the key management server. According to our protocol, the adversary who can get ct from the key management server must provide the authentication token t_{id^*}. So there are two possible cases for the adversary. The first is to recover the passphrase PP_{id^*} from the given **pwd** and the IBPH service, the second is to generate the t_{id^*} without the passphrase PP_{id^*}. In the first case, when the adversary makes less than B queries to the IBPH service, the probability of the adversary getting the correct PP_{id^*} is less than ϵ since the IBPH is (ϵ, d, B) pseudorandomness. In the second case, the adversary must generate the t_{id^*} without the passphrase PP_{id^*}. Since the adversary is only allowed to make $B_{\mathcal{K}}$ guesses for the right t_{id^*}, the probability is no more than $B_{\mathcal{K}}/2^d$. To conclude, the probability of the adversary getting the ct_{id^*} is less than $\epsilon + B_{\mathcal{K}}/2^d$.

If the adversary can not get ct_{id^*} from the key server, the only way for him to win the game is to make the client to accept a tampered master key. Since ct is stored on the key server side which he can not corrupt, what the adversary can do is to make the client to accept a forged decryption key k'_1 which is derived from a forged randomness r'_{id}. However, a forged r'_{id} will also derive a tampered tag key k'_2, and with overwhelming probability that $\tau \neq \mathsf{H}_4(ct, k'_2)$.

Hence the probability for $\mathcal{A}$ to win the IND-CDS game is less than $1/2 + \epsilon + B_{\mathcal{K}}/2^n + negl(\lambda)$. □

3.5.2 Compromising the key server.

In the IND-CKS game, the adversary can corrupt the key management server as well as register as users with any identity except the challenge identity id^*.

To show the master key is confidential to the malicious key server, we argue that the authenticated and confidential channel between the challenge client and the data server can guarantee that the randomness r_{id^*} and s_{id^*} on the data server are indeed confidential to the adversary after successful user registration. To get access to the encryption key k_{id^*} on the data server side, the adversary has only two choices. The first is to successfully authenticate to the data server, which means successfully recovering the password **pwd**. However, the pseudorandomness of IBPH guarantees **pwd** can not be guessed by the key server. The second is to blindly guess r_{id^*}. Since KDF_2 and KDF_3 are modeled as a random oracle, the encryption key k_1 and the tag key k_2 are independent random strings unless the adversary queried r_{id^*} to these two oracles. But the probability of blindly guessing r_{id^*} is negligible. With a random key, the IND-CPA security of the symmetric key encryption ensures that no information about mk leaked by the ciphertext ct.

To show the integrity of the master key, we can easily see that 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.

Specifically, we have the following theorem.

THEOREM 3.5.2. Let $(\mathsf{Register}, \mathsf{Login})$ be a Adv_{Auth}-secure login mechanism (as defined in Sec. 2.1). Let χ be the distribution of input x, where d is its min-entropy. The adversary makes maximally k attempts to log in to the data server. The IBPH is (ϵ, d, k)-obliviousness. Let KDF_2 be a random oracle. Let $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a secure authenticated encryption scheme. Our scheme is secure against a compromised data server, i.e., the probability for any

adversary to win the IND-CKS game (see Fig. 3.5) is less than $1/2 + \epsilon + negl(\lambda)$, where λ is the security parameter.

Proof: We take the IND-CKS game in Fig. 3.5 as Game 0. During this game, the challenger plays the roles of the key management server as well as an honest client with the challenge identity id^*. Similarly, the goal of the adversary is to distinguish the real deposited master key mk_0 or a randomly generated key mk_1.

In the IND-CKS game in Fig. 3.5, after arbitrarily communicating with the data server in the Give or Take procedure and arbitrarily registering new clients, the adversary firstly chooses the challenge identity id^* which has not registered before. Then the challenger chooses a passphrase PP_{id^*} for the challenge client. The challenger simulates the procedure that an honest client with id^* registers to the data server. When instantiated by the 2HashDH, the challenger will choose a random element r from $\mathbb{Z}_m$ and send $u = \mathcal{H}_1^r(PP_{id^*})$ to the adversary. The adversary will repeat v to the challenger. The challenger will register with the password $\mathbf{pwd} = \mathcal{H}_3(PP_{id^*}, v^{1/r})$ and the identity id to the data server. After that, the adversary can communicate with the data server using arbitrary message as the Give or Take procedure and register new client accounts. Also the adversary can invoke the Give procedure and communicate with the challenge client id^* using any massages. Then the challenger simulates the procedure that the honest client id^* generates a master key mk_0 as well as computing t_{id^*} and k_{id^*} based on the randomnesses s_{id^*} and r_{id^*}. The challenger will use k_{id^*} to encrypt mk_0 and get ct_{id^*}. The adversary will get $(h_{id^*}, t_{id^*}, ct_{id^*})$, then output the bit e to denote whether the Give procedure succeeds. If $e = 0$, that means the Give procedure fails and the adversary can continue to communicate with the data server using arbitrary message as the Give or Take procedure as well as registering new accounts, or with the challenge client id^* as the Give procedure. The challenger will regenerate a new master key mk_0 for anther time. This procedure will repeat until the adversary outputs $e = 1$, which means the Give procedure succeeds. Then the challenger will generate another random key mk_1, and give one of the two master keys to the adversary according to the random coin b. The adversary can continue to communicate with the data server $\mathcal{K}$ using arbitrary message in the Give or Take procedure, as well as to register any new identity $id \neq id^*$. Also the adversary can invoke the

Take procedure for the challenge client id^* and communicate with it using any massages. The adversary can also learn that the master key mk^* is reconstructed by the challenge client id^*. After that, the adversary will output her guess b'.

First of all, we will argue that the adversary can not see or alter the randomness r_{id^*}, s_{id^*} and h_{id^*} on the data server. Since the communication between the client and the data server is protected by the TLS, the adversary can not see r_{id^*}, s_{id^*} and h_{id^*} unless he can log in to the data server by pretending the honest client. However, the **pwd** is unknown to the adversary due to the obliviousness of the IBOPRF (When instantiated by the 2HashDH scheme, since the password **pwd** is computed from $\textbf{pwd} = \mathcal{H}_3(PP_{id^*}, v^{1/r})$, the adversary can not compute **pwd** unless he can guess the passphrase PP_{id^*}.). Since the number of query $B_{\mathcal{D}}$ are limited on the challenge identity id^*, the adversary can not guess **pwd**. So in Game 1 we can replace the $\mathcal{D}_{\mathsf{Take}}$ and $\mathcal{D}_{\mathsf{Give}}$ with $\mathcal{D}'_{\mathsf{Take}}$ and $\mathcal{D}'_{\mathsf{Give}}$ that always return $\perp$ when the received message from the adversary is $(id^*, \textbf{pwd})$ for any α. For other $id \neq id^*$, $\mathcal{D}'_{\mathsf{Take}}$ and $\mathcal{D}'_{\mathsf{Give}}$ behave as same as $\mathcal{D}_{\mathsf{Take}}$ and $\mathcal{D}_{\mathsf{Give}}$. Hence, if $\Pr(\mathsf{Game\ 0})$ and $\Pr(\mathsf{Game\ 1})$ denote the probability for the adversary to win Game 0 and Game 1 respectively, we have $\Pr(\mathsf{Game\ 0}) \leq \Pr(\mathsf{Game\ 1}) + \epsilon$ when the IBPH is (ϵ, d, k)-obliviousness.

In Game 2, we replace k_{1,id^*} and k_{2,id^*} with random strings k_1, k_2. Also, we replace ct_{id^*} with a ciphertext which is encrypted under k_1. Since KDF_2 can be model as a random oracle, to distinguish Game 2 from Game 1 needs to successfully guess r_{id^*}. However, in Game 1, we already replace the $\mathcal{D}_{\mathsf{Take}}$ and $\mathcal{D}_{\mathsf{Give}}$ with $\mathcal{D}'_{\mathsf{Take}}$ and $\mathcal{D}'_{\mathsf{Give}}$, the adversary can not get the information related to r_{id^*} except blindly guessing. The length of r_{id^*} is λ, so the probability to guess r_{id^*} is less than $negl(\lambda)$. If $\Pr(\mathsf{Game\ 2})$ denotes the probability to win Game 2, $\Pr_1$ is less than $\Pr(\mathsf{Game\ 2}) + negl(\lambda)$.

In Game 3, we replace ct_{id^*} with a random ciphertext ct, so the probability for the adversary to learn the stored master key is negligible. However, the adversary may still win the game by making the client to accept a tampered master key. Note that the tag $\tau = \mathsf{H}_4(ct, k_2)$, hence the adversary can not tamper the ciphertext ct and pass the verification of the tag τ. This is because of the security of the symmetric key encryption as k is a random string to adversary.

Finally, we have that the probability for the adversary to win the IND-CKS game is less than $1/2 + \epsilon + negl(\lambda)$. □

3.5.3 Best possible security when two servers collude

In the real world, the collusion event is really rare: The App service provider can choose different cloud services. Or more typically, one server is run by the cloud company and one is internal to the App company. In practice, the covert collaboration will be easily noticed by insider engineers in the App company. A respectful cloud company wishing to stay in business cannot afford the reputation lost for open violation of security requirements and agreements. Let alone the key server may be instantiated in isolated secure hardware modules as extra protection.

Even in the real world, the probability of collusion is very small, PBCS still provides the basic security of the master key in those extreme cases. Intuitively, the master key is encrypted by the key $k_1 = \mathsf{KDF}_2(r_{id}, PP)$. Even if the adversary obtained both r_{id} and $ct = \mathsf{Enc}_k(mk)$, to recover k still needs to guess the correct PP. However, thanks to our strong authentication method, no servers can learn the passphrase directly. That means when we model KDF_1, KDF_2 and KDF_3 as random oracles, the adversary does not have a strategy to recover k_1 better than guessing PP by brute force. More importantly, since verifying a candidate guess requires computing a slow key drive function KDF_1 or KDF_2, a brute force search for the correct PP still takes considerable time.

A similar situation happens when we consider the integrity of the content after collusion. To forge a ciphertext, the adversary must generate the correct $\tau = \mathsf{H}_4(ct, k_2)$ while $k_2 = \mathsf{KDF}_3(r_{id}, PP)$. Since verifying a forged ciphertext requires computing a slow key drive function KDF_3, a brute force search for the correct PP still takes considerable time.

3.6 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). The prototype implementation is available in https://github.com/yananli117/E2SE, which got all three badges in the Usenix artifact evaluation, including the artifact available badge, the artifact functional badge, and the results reproduced badge.

Deployability. We did a survey of current popular cloud storage services, and found that popular storage services, including but not limited to Amazon S3 [70], Google Cloud Storage [71], Azure Storage [72], and Dropbox [73] 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.

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 the 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 built-in login of the cloud storage service to implement the user 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 the t2.micro instance in Osaka. The client is deployed

in Seoul using AWS EC2 t3.xlarge instance, which has a similar configuration with popular PCs[10]. The operating system is ubuntu 18.04 LTS. All 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 efficiency, we not only measure the time cost of each procedure but also test and compare the cost of depositing and retrieving plain data and securely with PBCS.

Running IBOPRF protocol to get the high-entropy password in the register, the total Give and Take procedure are the overhead of PBCS KEM part. We run each procedure over 100 iterations and average the time cost. Concretely, the Give and Take procedures cost about 0.53s and 0.48s, respectively. Running IBOPRF costs around 0.145s, where the key server overhead makes up less than 1%, 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, which is unavoidable in the normal use of TLS communication.

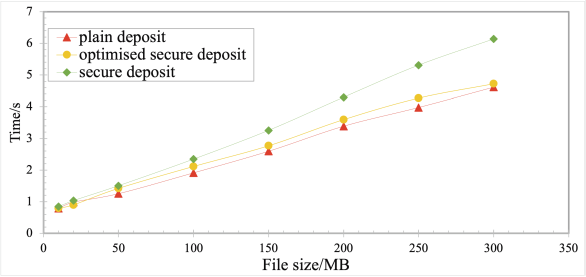


FIGURE 3.11. The time for depositing files

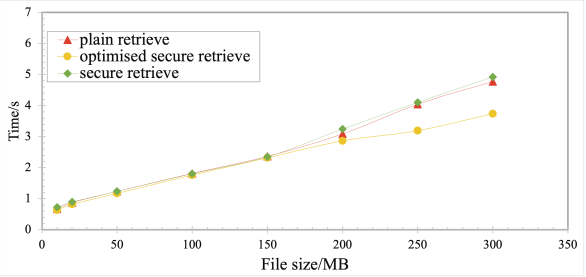


FIGURE 3.12. The time for retrieving files

We evaluate the performance of secure data deposit and retrieval and compare it with insecure operations. We test on files with 8 different sizes (from 10MB to 300MB) and run 25 iterations for each experiment to get the average cost. The results are plotted in Fig. 3.11, 3.12, where the red plain deposit/retrieve denotes insecure plain deposit/retrieve, the blue secure deposit/retrieve includes one run of the Take protocol, encrypting/decrypting the file, and

[10]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.

uploading the encrypted file to the data server or downloading an encrypted file from the data server. Secure deposit costs are larger than the plain deposit, and the overhead of secure deposit increase with the file size increase. See Fig. 3.11.

To have a better understanding of our design's overhead, we also measure the cost breakdown for each phase, shown in Tables. 3.2, 3.3. For the client, 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. The showed overheads of our basic implementation are already very small, from 0 to about 1.5 seconds, as the file size increases to 300MB. . More impressively, the overhead for retrieve is quite small and keeps less than 0.2s. See the column "Ovd-SD" in Table 3.2 and column "Ovd-SR" in Table 3.3.

TABLE 3.2. The time cost breakdown for depositing files, where DPT/DCT denote depositing plaintext/ciphertext, Opt-SD denotes the time cost of optimized secure deposit, Ovd-SD/Ovd-Opt-SD denote the overhead of the basic/optimized secure deposit.

Size/MB	DPT/s	Enc/s	DCT/s	Opt-SD/s	Ovd-SD/s	Ovd-Opt-SD/s
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

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. We have a simple optimization idea: one can parallelly encrypt/decrypt data while uploading/downloading, hence he does not need to wait for the finish of the encryption/decryption before uploading/downloading the data. If we cut the large file into smaller blocks, and encrypt/decrypt each of them while uploading/downloading another block, ideally the total latency could be reduced to the encryption/decryption of only one small block plus the uploading/downloading of all the blocks, and the smaller the block, the shorter the latency. However, the cloud storage needs to

TABLE 3.3. The time cost breakdown for retrieving files, where RPT/RCT denote retrieving plaintext/ciphertext, Opt-SR denotes the time cost of optimized secure retrieve, Ovd-SR/Ovd-Opt-SR denote the overhead of the basic/optimized secure retrieve.

Size/MB	RPT/s	Dec/s	RCT/s	Opt-SR/s	Ovd-SR/s	Ovd-Opt-SR/s
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

return an “ack" confirmation for each request on each block. The increased block number brings extra time costs, especially to the block uploading process, which could be large when the network delay is long. So it is involved to determine how many blocks we should divide a file to minimize the time cost.

Taking all above factors into account, we first model the time cost T as the function of file size s and number of blocks n, and then get a quick estimation $T(n,s) = k_2\frac{s}{n}+(2c_2+c3)n+k_1s+c_1$, 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”; 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.

For the optimized implementation, the performance of secure deposit/retrieve is almost identical to and even less than deposit/retrieve the file itself! See the yellow linses in Fig. 3.11, 3.12; also the optimised overhead is small and indeed close to 0 and even 0! See the column “Ovd-Opt-SD” in Table 3.2 and the column “Ovd-Opt-SR” in Table 3.3. 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[11].

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 a reference, we also test the throughput of the key server for the static page.

The key server as a web server is equipped with the 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, the static page is 666.09 req/s and IBOPRF is 464.64 req/s. In IBOPRF, the 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 increases almost linearly with the number of vCPU, shown in Fig. 3.13.

Remark. Please not that the throughput should linearly increase as more vcpus are used, which is shown from 1-4 vcpus, that is to grow in a consistent speed. The increase of throughput in 8-vcpus case is not as fast as before, because the client is run on 8vcpus machine to build TLS connection and send requests, and TLS connection cost both requesting and responding devices' computation power, which cause the client cannot provide adequate requests to test responding device's throughput. So we can see the throughput grows slow in 8 vcpus case.

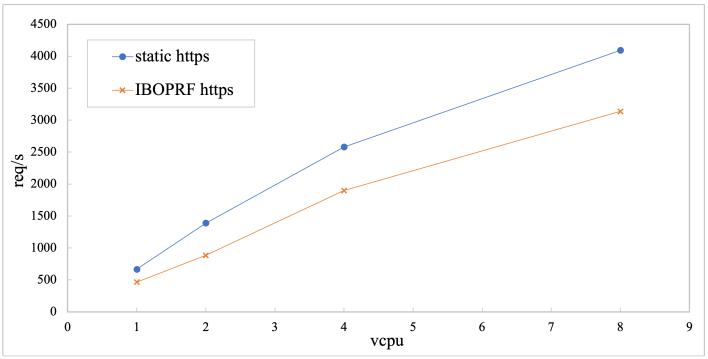


FIGURE 3.13. The throughput of key server

[11]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.

Cost savings. Here we use Amazon S3 and EC2 as examples 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 the same use, one needs to pay for extra computation cost [74] to keep VMs running, and 5 times the cost of the storage [75]. 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.

3.7 Further Extensions

Due to our principle of minimal additions to existing infrastructure and simplicity, our system can be easily extended to apply to additional scenarios.

Augment App without outsourcing storage to a third-party cloud server. Our PBCS system was initially proposed to augment an App with an efficient, portable, and blind cloud storage, where the App provider managed one server and outsourced the storage to the third-party cloud server. However, PBCS can also be deployed in other scenarios. Many App providers do not rent third-party cloud services but instead manage all services themselves including the user management and storage. In such case, to make storage blind to anyone but users, PBCS's data server needs to be handled by another cloud storage server. Considering that stored data, such as videos and pictures, can be very large and unaffordable for users to maintain themselves, users could rent personal cloud storage, such as Google Drive or

DropBox, to store only the randomness part. The encrypted data would still be stored by the App server. In PBCS, the encrypted data can be public, but the secret key used for encryption must be securely protected. Therefore, by storing only the randomness part separately, secret sharing of the secret key is achieved and the design remains secure.

Support distributed data servers. To provide robust service and reduce the impact of occasional server corruption, distributed cloud servers are often used. For example, WhatsApp employs 5 data centers to back up users' messaging data [50]. Our PBCS can support distributed data servers as well. For instance, in a setup of PBCSwith 3 data servers, users can interact with the key server to derive 3 high-entropy passwords to log in to 3 data servers. The randomness can be secretly shared among 3 data servers with 2-out-of-3 secret sharing. Each server can store a replica of the encrypted data an a share of the encrypted data, depending on the secret sharing method.

Compatible with any App authentication method in a black-box way. In PBCS design, both the data server and App server use a password login mechanism for user authentication, and the user is required to remember one password which is easier than remembering two independent passwords. Actually, the App server authentication mechanism does not have to be restricted to password login. PBCS can be compatible in a black-box way with any App authentication method including biometric information like a fingerprint, iris, etc. Specifically, in the "Register", "Give", and "Take" protocols, the key server password login mechanism can be replaced with App original authentication mechanism. When upgrading the App with PBCS, PBCS's key server module is deployed in the App server, and App's original authentication module is used. The user inputs their passphrase, runs the IBOPRF with the key server to get a high entropy password to log in to the data server, and follows the App original authentication instruction to authenticate to the App server. Then the user could run the "Give" and "Take" protocols to share and reconstruct the master key mk via interacting with two servers using the passphrase. As we claim that PBCS is compatible with any App authentication method, using another passphrase to authenticate to App server is also acceptable. However, the two passphrases must be different and independent, otherwise the leakage of one passphrase can compromise the security of the other. If users insist on using

two passwords, simpler solutions could be implemented, which would require modification to the PBCS design.

Provide proactive security. Proactive security ensure that the system remains secure by reducing the time available for an attacker to compromise shares and visit both servers simultaneously. PBCS can achieve this by simply redoing the "Give" protocol to update the shares of mk with new randomness s'_{id} and r'_{id} stored under the cloud server. the updated values are computed as follows: $t' = \mathsf{KDF}_1(s'_{id}, PP)$, $k'_1 = \mathsf{KDF}_2(r'_{id}, PP)$, $k'_2 = \mathsf{KDF}_3(r'_{id}, PP)$, $ct' = \mathsf{Enc}(k'_1, mk)$, and $\tau' = \mathsf{H}_4(k'_2, ct')$ with (t', ct', τ') deposited to the key management server. Moreover, PBCS can support periodic key rotation for mk via running the "Take" protocol to retrieve mk, running the "Give" protocol to deposit new mk', and uploading the key update token $\Delta_{mk,mk'}$ to the data server. The data server re-encrypts the user storage with $\Delta_{mk,mk'}$ to ensure encryption under the new key mk'.

Password hashing functions. The modular design of PBCS enables us to independently create, modify, replace, or exchange cryptography primitives. For example, the key derivation function `PBKDF2WithHmacSHA256` can be replaced with memory-hard functions such as scrypt [76], Balloon hashing[77] or Argon2 [78]. These advanced password hashing functions could enhance the security of PBCS even when two severs collude, as they significantly increase the cost of attempting many possible passwords against a leaked database of hashed passwords.

Post-quantum security. Regulations and standards for cryptography products often vary across countries and evolve over time. Recently, NIST has recommended migrating the cipher suit in use to post-quantum ones [79]. Consequently, any added secure cloud storage must accommodate changes to these regulations and standards, especially if the App is designed for global use and intended to have a long lifecycle. Thanks to the modular design of PBCS, post quantum secure OPRF [80, 81] can be used to design the IBOPRF module, post-quantum TLS can be leveraged to ensure communication security [82, 83]. These allow PBCS to be upgraded in post-quantum era.

Furthermore, PBCS system can support more complex settings and additional functionalities.

Enable the secure sharing of private data. The PBCS system ensures that private data is accessible only to the user. In some scenarios, a sharing function is often desired. For example, one might need to share personal photos with friends or family members. A simple way to achieve the secure sharing of private data is to combine an E2EE messaging app with a storage service that supports access sharing. Concretely, the E2EE messaging module can securely share the encryption key of the shared data, which can then be further protected using the PBCS system. Authorized users can be granted access to the shared storage via a shared link or other access control methods.

Improve security for pure personal cloud storage service. Personal cloud storage services, such as Google Drive, DropBox, and OneDrive, are widely popular. The stored data is encrypted, but the encryption key is often either accessible to the cloud server or unavailable on another device. To enhance the security of personal cloud storage, an independent app service can be introduced by deploying the PBCS's key server and client as the App server and App client, respectively. To support a broader range of personal storage services, the client can be further developed to be compatible with all kinds of cloud APIs, and provides users with multiple storage options to choose.

3.8 Concluding Remarks and Open Problems

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.

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 [84, 85, 86, 87, 88] 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 a 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 mechanisms, TLS, and non-programmable cloud storage. As previously discussed, our approach enables sound system development and 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.

The area of designing a cryptographic function to a suitable API is both challenging and useful, but is rare: typically a general cryptographic protocol for an abstract primitive is embedded in a general environment and not the opposite. This work took the challenge of an area that was neglected before and is "Augmenting a cloud storage service to a Secure from the cloud storage service". Existing solutions for the abstract problem require much more than just a storage interface from the cloud storage and hence are not applicable (due to cost, development needed, etc.). By casting the problem as an End-to-the-same-end solution, and building on the logic of the active area of end-to-end encryption which is about communication security, a solution for "just storage" in the cloud was found. Requiring simple storage API from the cloud makes future adoption and enhances settings for applications of the solution possible. This is a classical case: Application derives new theory, and new theory, in turn, derives new application.

Open problems. Our End-to-Same-End encryption can be viewed as a general framework that enables many potential applications. We gave some examples in the paper. It would be interesting questions to explore other non-trivial End-to-End secure applications or build non-trivial systems on top of our E2SE. For the framework itself, we achieved using only one passphrase for PBCS, and showed an extension to combine one passphrase and any other authentication factor for PBCS. It is interesting but still open to make PBCS compatible with any other single-factor strong authentication method without a passphrase as assistance.

Moreover, more advanced properties such as post-compromise security by integrating our E2SE with updatable encryption [25, 89] propertly would also be interesting to explore. Last, but not least, an interesting open problem is about enabling search in PBCS but not weakening the semantic security of stored data.

CHAPTER 4

End-to-End Encrypted Git Services

4.1 Introduction

Git services have become indispensable in the IT industry, facilitating project management and collaboration among multiple (potentially a large number of) entities via hosting platforms like GitHub, Bitbucket, GitLab, Azure Repos services, and many others. In these platforms, the entirety of a project's data, including files (such as code and documentation) and directory structures, constitutes a repository. Moreover, in a Git repository, the file data includes each version of all tracked files and their corresponding directory structure. Authorized users can access and edit the shared repository data in a local Git client and then synchronize to the Git server via pull/push operations. Repositories can be public or private, while private ones allow project owners to manage visibility and keep data hidden from the public.

The rising demand for *end-to-end* security. Privacy is undoubtedly a great concern for both individual users and enterprises that collaborate over hosting Git platforms for projects that may contain sensitive information and/ or trade secrets. The situation becomes particularly more alarming in the AI era when repositories become very powerful, containing AI models that are trained on code and data stored in Git repositories or even that directly provide coding assistance (e.g., GitHub Copilot).

Unfortunately, in existing Git hosting platforms, the data, even in private repositories, is visible to the server itself. Even if Git servers may not actively disclose users' data (e.g., for compliance and reputation) and have taken actions to protect data against external attackers

(e.g., encrypting the data using the server's own key), the usual risk of data breach (due to external attacks or internal misbehavior of staff) is paramount nowadays.

Moreover, the collaborations on Git services essentially form a supply chain of software development (open source or not, and more broadly online collaboration): any unauthorized modification could have detrimental impacts on the final "product". The potential issues are partially solved by a few Git platforms, such as GitHub [90], GitLab [91], Gitea [92], and Bitbucket [93] that started to support an optional verified commit signature to authenticate the author of each edit but most versions are not verified. It follows that ensuring integrity, proper write access control, and even authenticity of each edit in Git services is also of utmost importance. Unfortunately, current practice mainly relies on the honesty of the Git servers/ platforms, which might even have conflicts of interest on certain projects, to ensure that each repository version is integrated and written by the shown author.

The above situation highlights the need for *end-to-end* (E2E) security [1] in Git services that guarantees critical security properties, even against possibly *corrupted* servers. Indeed, a few industrial projects have been introduced with the aim of moving toward an E2E secure system supporting Git service and online collaborations such as [94, 110, 95, 96, 18].

We note that similar security concerns were widely recognized in *secure messaging*, where there is a long line of research work [97, 98, 99, 100, 101, 102, 103, 104]. These works attempt to rigorously realize E2E security in different settings and analyze potential vulnerabilities in widely deployed tools. Recently, a sequence of work has also emerged [20, 19], initiating the study of E2E security for *cloud storage*. In the latter setting, further complications arise due to some features that cloud storage possesses, such as sharing among multiple users, portability via password-based authentication, and subtle vulnerabilities that led to attacks on real-world products, such as [9, 105, 106].

[1] We will use the standard terminology "end-to-end encrypted" Git services; however, it goes beyond confidentiality and includes integrity and more security properties.

E2E secure Git service is not yet available. Despite the recent progress in relevant applications and strong demand, E2E secure Git service is currently out of reach of current techniques and methods.

Insufficiency of using E2E encrypted cloud storage directly. First, one may wonder whether deploying Git servers on E2E secure cloud storage immediately solves the problem. Unfortunately, the situation is more complex than we thought. These recent E2E encrypted storage solutions [20, 19] are at a very early stage of their own development and are insufficient in both functionality and security.

In terms of functionality, the most common operations in Git services are "push" and "pull," used to upload/retrieve missing versions to the server/client. Unlike E2E encrypted cloud storage systems, which return only the content specified in the request without computation, the Git "pull" operation requires the server to compute and return the minimal missing parts, as the client cannot determine which parts are missing or minimal.

In terms of security, E2E encrypted cloud storage only considered basic security properties for *static* data. While for Git services, storage exhibits a feature that data is constantly *updated*. This not only further complicates the already complicated security properties, such as confidentiality (and basic integrity), but also raises new (yet natural) security requirements on "access control." As Git service is a distributed collaborative environment, some basic access control actions for managing authorized users (without relying on an honest Git server), called unforgeability – which we will discuss below soon, are not only required for identifying the author of edits but also has direct impacts on confidentiality. We note that this type of operation might also be needed for cloud storage but has not yet been studied in the literature.

Furthermore, efficiency is a very important consideration. Unlike traditional cloud storage systems, which often store only a limited number of file versions (for example, Google Drive keeps only 100 versions), Git servers keep the whole chain of edits for users to track the history. This creates a strong incentive to deduplicate data across versions to minimize storage costs. Naively adopting encryption as in E2E encrypted cloud storage to Git service, i.e., treating each file within each repository version as a new file to encrypt, transfer, and store,

may incur very high costs for users on computation in file encryption, on communication in data transfer, and on storage in local client and remote servers (note that usually free cloud storage is only provided with limited space).

Security risks in existing ad hoc secure Git service designs. Numerous industrial products and some research papers exist, attempting to address the data protection needs of Git services, including Keybase [95], Git-secret [110], Git-crypt [94], and others. Despite the fact that some of these tools have received thousands of stars (being saved with stars) in GitHub, none of them have been rigorously analyzed. Jumping ahead, we can easily see (in Table 4.1) important security properties that actually fail, *especially when we do not place blind trust in the storage server*. We elaborate on this below.

First, even very basic confidentiality requires care when efficiency improvements are considered. For example, Git-crypt [94] and Gringotts [107] tried to save storage costs by utilizing deterministic encryption schemes to enable data compression on the ciphertext. This is at the (obvious) cost of privacy. It is well known that deterministic encryption offers weak protection, as it trivially leaks pattern information that the same data has the same ciphertext. Indeed, recent research has demonstrated how to abuse such leakage via "injection attacks" on E2E encrypted applications, such as backup, to, in fact, expose the content [108].

Furthermore, the conventional integrity of the *repository* (jumping ahead, actually a weaker form of unforgeability that is only against a corrupted server and users without legitimate access) may easily fail, too. Several systems simply employ the hybrid encryption paradigm as seen in Git services like Git-Secret [110] and Keybase [95]. To reduce storage costs, Git-secret [110] even allows the users to choose which files to encrypt (e.g., sensitive data only), while leaving others still in the clear. It is easy to see that in either case, repository integrity cannot be achieved, as a corrupted server could attack via injection and deletion. For example, a malicious server injects a new data encryption key using the receiver's public key and adds the corresponding encryption of new data. Attackers can also simply delete certain files of a repository version without being detected. What is worse, once the maliciously

inserted data encryption key is used by the receiver to encrypt the next version, it further breaks the data confidentiality.

Additionally, existing Git services' features for tracking version history and authorship, as well as read-write access separation, are vulnerable to attacks, highlighting the lack of truly secure Git services. Currently, author tracking and read-write access separation mainly rely on a trusted Git server, allowing malicious users and a corrupted server to falsify authorship, write on behalf of others, frame honest users, or break the read-only limit to write. To prevent such attacks, E2E encrypted Git services require an "unforgeability" property related to edit-source authenticity and read-write access separation (defined later). This ensures that attackers, including corrupted servers or users with write access, cannot misattribute edits without detection and that read-only attackers cannot perform writes. [2]

We stress that these security vulnerabilities in ad hoc designs are not just of theoretical interest. Similar situations exist in cloud storage and secure messaging, where multiple practical attacks were identified [9, 108]. These highlight the need to design a formally analyzed E2E encrypted Git service.

Dealing with overhead and compatibility. If we are to ignore performance, designing an E2E secure Git service is easy via "trivial-enc-sign". That is, for each edit, a user simply performs the following commands: fetch the latest ciphertext version, verify and decrypt to get the plaintext version, edit the files of the plaintext version to get the new plaintext version, then completely re-encrypt the new plaintext version, sign, and upload. Instead of this trivial operation (which can be followed on plaintext Git services without the cryptography), the repetitive nature of Git updates (with significant overlaps across versions) has been carefully leveraged in systems and led to the current implementation of plain Git adopting various measures for reducing complexity and enhancing performance.

We note that despite several existing systems (still with various security issues shown in Table 4.1) that have tried to provide security for Git services with reduced complexity (compared

[2] Unforgeability is also essential in secure cloud storage systems. Unfortunately formal treatments [20, 19] have not captured this security, assuming an all-or-nothing type of access (either no access or full access), with no guarantees on the source of edits.

to the trivial secure solution), overheads for each edit are still significant. This is because they still operate on files, thus making the overhead relevant to edited files or even the whole repository, even if only one character was changed. A more careful and fine-grained treatment may give us the opportunity to minimize the overhead (while maintaining E2E security), e.g., it makes sense to require operations to only be proportional to each actual edit.

Another important practical property (also recognized in [20]) is *platform compatibility* with existing infrastructure; in our setting, existing Git services include GitHub and Bitbucket servers. This is an important property often ignored in many theoretical works. In fact, with this compatibility, current users can employ E2E encrypted Git services by simply installing a new secure Git client and directly using Git servers that current Git services provide to do Git operations (in most cases, services are not accessible except based on the existing defined queries).

We summarize detailed comparisons in Table 4.1 and Section 4.1.3.

The discussion above showed that secure Git services of the E2E nature are really beyond the current state of the art. Hence, in this paper, we tackle the following challenge: *identify and formalize critical security properties of an E2E encrypted Git service, and give provably secure constructions that are both with minimal overhead and platform-compatible with existing Git servers.* [3]

4.1.1 Our results

- We present formal syntax and security models for E2E encrypted [4] Git service. Particularly, we propose two main security properties of *data confidentiality* and *repository unforgeability*, each with a weaker variant. All properties are against a malicious Git server and unauthorized users, while *unforgeability* is even further against malicious insiders.

[3] We note that there might be potential risks of misuse with E2E security. Addressing these risks while maintaining security for the majority of the innocent users requires additional countermeasures, which seem to require what we advocate herein, but which are beyond the scope of this paper.

[4] The term "encrypted" is used in a broader sense to refer to protection through cryptographic tools, not just encryption alone.

TABLE 4.1. Comparison with the state-of-the-art "encrypted" Git services.

Schemes	Confidentiality	Integrity#	Unforgeability	Storage increase per version†	Client enc cost per update‡	Comm cost per update§	Compatibility**
Git-crypt* [94]	✗	✗	✗	$n_f L$	$n_f L$	$n_f L$	✓
Gringotts* [107]	✗	✗	✗	$n_f \ell_1$	$n_f L$	$n_f \ell_1$	✗
Git-secret [110]	✓*	✗	✗	$n_f L$	$n_f L$	$n_f L$	✓
Git-re-gcrypt [96]	✓	?	?	$n_f L$	$n_f L$	$n_f L$	✓̸
Disac [109]	✓	?	?	$n_f L$	$n_f L$	$n_f L$	✓
Keybase-Git [95]	✓	?	?	$n_f L$	$n_f L$	$n_f L$	✗
Trivial-enc-sign	✓	✓	✓	R	R	R	✓
Our SGitLine	✓**	✓	✓	$n_f \ell_1$	$n_f \ell_1$	$n_f \ell_1$	✓
Our SGitChar	✓	✓	✓	$n_f \ell_2$	$n_f \ell_2$	$n_f \ell_2$	✓

n_f denotes the number of changed files per repository version, and L, R denotes the (average) size of files and repository, respectively. ℓ_1, ℓ_2 denote the average size of line change and character change per file update. Usually, each update works on a small portion of some files; thus, usually, $\ell_2 \ll \ell_1 \ll L$ and $n_f L \ll R$.
Integrity denotes the conventional integrity of repository data, which is also a weaker version of unforgeability.
† Repo storage increment per version measures the average storage increases for updating to a new version.
‡ Client enc cost for update measures the rough computation cost of a client in each version update.
§ Comm cost per update measures the user-dominating communication cost of each version update.
* means the storage cost of the scheme may be slightly smaller due to compression on deterministic encryption.
** Compatibility asks if the Git service is compatible with existing Git hosting platforms, e.g., GitHub, Bitbucket, etc., and supports all basic Git operations, including commit, push, pull, fetch versions and objects, merge, etc. ✗: not compatible with Git server; ✓: compatible with Git server with full Git operations; ✓̸: compatible with Git server with limited Git operations.
$\star$ means the corresponding security is conditional, as it is left to users to decide which part to encrypt.
$\star\star$ means a weaker version of our confidentiality definition.
? means that the security is unclear at the moment since there is no formal security analysis or obvious attack.

- We give two constructions that provably meet data confidentiality and repository unforgeability (with the caveat that the first construction satisfies only a weaker confidentiality), and both are fully compatible with existing Git servers (including all Git hosting platforms like GitHub server) as shown in Figure 4.1 and standard cryptographic libraries. Moreover, these two constructions achieve security with minimal overhead that is relevant only to the edits (instead of the whole repository or files), and have different efficiency performances for different edit patterns on the managed projects.

- We implement our two constructions and carry out extensive experiments on popular GitHub repositories to evaluate computation, communication, and storage costs. Our experiment results show that our constructions perform better than the "naive" solution and those using deterministic encryption.

4.1.2 Technical overview

Git service architecture and defining security properly. We illustrate the architecture of plain and our E2E secure Git services in Figure 4.1, highlighting the syntax definition and our general principle of *platform compatibility* with Git services.

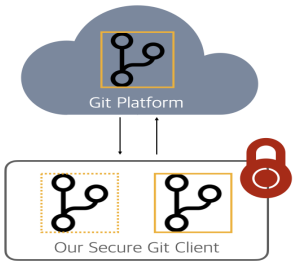


FIGURE 4.1. The architecture of plain/secure Git service

Abstracting out authentication. Unlike recent works on E2E secure cloud storage [19, 20] that blend password authentication into the overall security model, we abstract out authentication as a standalone black-box module. This approach simplifies the analysis by focusing on the core security properties of E2E secure Git services, which are already complex. Furthermore, authentication methods vary widely, including password login, two-factor authentication, device-based authentication, token-based authentication, and more. By treating authentication as a parameter, our framework can naturally inherit its security, enabling straightforward generalization.

Formalizing security properties. The basic intuition of E2E secure Git services is to "emulate" an ideal access control as if it is enforced by a trusted server (which we do not have now) to capture real-world attacks from malicious servers and unauthorized users.

Data confidentiality mimics the conventional IND-CPA security. However, additional subtleties arise when considering multi-user data sharing and frequent data updates. Both require us to provide the adversary with extra capabilities to flexibly interact with honest users via sharing, updating, and more. Moreover, certain "metadata" regarding updates should also remain concealed from the malicious server. Specifically, the update/edit operations, e.g., insertion or deletion, as well as the precise edit positions, including which lines or words are altered, should remain hidden. Essentially, while the server is aware that users are performing operations on specific files and sizes, it remains oblivious to the update details. We remark that only our SGitChar satisfies this strong confidentiality, while the SGitLine construction only satisfies a weaker version that only hides the data (not the "metadata") of the edit.

Repository integrity is a second natural property that ensures users can verify whether each version of a repository is intact and has not been modified by outsiders, including malicious servers and unauthorized users. This is clearly stronger than file-wise integrity, as a malicious server may delete a set of files to save storage without violating the latter, weaker form of integrity. We may also easily upgrade the notion to make sure the whole repository history remains intact. To capture the natural need for edit-source authenticity and read-write access separation, we further strengthen integrity to be against *malicious insiders* who may indeed have write permission. We call the strong integrity *repository unforgeability*. We remark that while repository integrity is weaker, it captures the existing integrity guarantees commonly provided in secure cloud storage. This weaker property is reasonable for single-user repositories or settings with all-or-nothing access control. The stronger property is needed in a multi-user collaborative setting like Git service, while currently it is not discussed in E2E encrypted storage literature where read-write access comes as an all-or-nothing flavor (but it exists in some secure group messaging work [102, 103, 104]).

Secure and efficient constructions with compatibility. As mentioned above, there are naive solutions that always download (`pull`) the whole repository before editing, then apply all needed cryptographic operations on the *whole repository* and upload (`push`). It is clear that this comes with a high cost. Namely, editing even a single character in one file results in the increment of a full-size repository (for both server storage and client

communication/computation). Meanwhile, in conventional (plain) Git services, the cost could be minimized, as simple compression tricks can be applied. For example, during the `push` execution, the differences between two versions would be computed by a `Diff` operation on the client, and only the resulting "delta" (difference) induced by edits is uploaded.

File-wise treatment for secure Git services is possible (indeed adopted in most of the existing systems [110, 96]). However, they still mostly incur large overhead (proportional to file size instead of the difference size); further, optimizations to reduce overhead already cause various security vulnerabilities, as we briefly discussed above (and also in Table 4.1). The reason lies in the dilemma that if naively encrypting (say using standard authenticated encryption) the updated file as a whole, then during `push`, the two encrypted files would look completely irrelevant; the actual difference cannot be computed. On the other hand, if encryption is applied in a finer granularity, e.g., line-wise or character-wise, security risks increase due to greater leakage on operation type, position, and edit length. Moreover, the requirement of repository integrity and the enforcement of read-write access separation and edit-source authenticity add complexity.

To address the security vulnerabilities in existing schemes caused by overhead optimizations and summarized with three security properties in Table 4.1, our first construction, SGitLine, applies standard encryption at a finer granularity within files, specifically at the line level. Encrypting by line, a natural structural unit of files, helps preserve data confidentiality while reducing the update overhead, simplifying version tracking, and enabling more compact storage.

To achieve unforgeability (thus also integrity), enforcing read-write access separation, and tracking the source of each edit, we simply need some publicly verifiable mechanism for each repository version. We leverage digital signatures to sign on a whole version of the encrypted repository following the hash-and-sign paradigm. Since the hash of the whole version is generated by Git commit anyway, the signing cost is constant and small. Then, the encryption we apply could be a standard IND-CPA secure cipher instead of authenticated encryption. This fairly simple encrypt-then-sign paradigm was recently proven for secure "symmetric signcryption" [102]. And SGitLine may offer better efficiency in certain scenarios because

each single version is history-free and not relevant to previous updates. However, line-wise encryption suffers from a drawback in confidentiality: it exposes information about update operations and positions during line insertions and deletions (thus only achieves weaker confidentiality). In addition, the communication costs of updating one version depend on the number and length of the modified lines, even if only one character is modified in each line.

To deal with the efficiency-confidentiality dilemma, we dig deeper into Git. When users `push` a new version after edits, current systems basically send a whole new encrypted file because the built-in compression in both Git client and Git server cannot properly work on ciphertext. We make a simple observation that we may create the ciphertext in a form that helps the deduplication (on ciphertext instead of plaintext, thus not influencing confidentiality).

In our main construction SGitChar, we propose a "Diff-then-Enc-then-Sign" paradigm that, after editing the pulled file, we let the client run a version of `Diff` algorithm that identifies the differences Δ at the character level (e.g., which position, which operation, on which character) with the previous version. Then the client encrypts Δ to obtain C^*, while he pushes $C||C^*$, where C is the ciphertext just pulled before modification. Of course, the whole version is always signed before `push`. In this way, the built-in deduplication mechanism will remove C and only upload C^* when executing `push`. Since all details, including update operations and content, are encrypted, SGitChar satisfies our data confidentiality, while unforgeability holds as before. Moreover, since the update only sends C^*, the overhead is still only relevant to the difference Δ (independent of the file).

Obviously, combining the various cryptographic techniques requires us to prove the security properties as defined in our model.

4.1.3 Other related works

Several open-source projects [110, 96, 95] focus on improving confidentiality by using standard CPA secure encryption to do file-wise encryption. So the storage cost is linear to the product of version numbers and the size of changed files in a repository. For each update operation, a user needs to re-run encryption on the changed files of the repository.

So the encryption time and communication size are related to the size of all changed files. Moreover, Keybase Git [95] is designed to work with Keybase server and is not compatible with existing Git Servers such as GitHub. To save the storage cost, Git-crypt [94] sacrifices the CPA confidentiality by using the hash value of the file as the initialization vector of AES encryption. For those unchanged files, the hash values keep unchanged, so as to the ciphertexts of the files. But any tiny change of the file can produce a totally different ciphertext from the previous version. So the storage saving method does not apply to minor changes spreading most files of the repository. Git-remote-gcrypt [96] applies delta compression on the entire new plaintext version of the repository to generate a packfile before encryption. Thus, in the Git server, each update will add a new encrypted packfile, compressing all files (including objects for the new version) together. Since the packfile is encrypted, the Git server cannot interpret it to parse a new version, and all versions are treated as a single version. As a result, some Git operations, such as fetching a specified version or object from the Git server, and merging two versions on the Git Server are not supported in Git-remote-gcrypt.

Gingotts [107] uses another deterministic encryption to save storage. They fix the IV of AES and do line-wise encryption, so that the data compression can be done cross files and a tiny change within a line only brings a new line of ciphertext, which saves the storage cost more than doing file-wise encryption. [107, 109] further enhances the access control of VCS via attribute-based encryption. Gringotts [107] considered a weaker model in terms of unforgeability, where the remote server is assumed to be honest. Disc [109] applies attribute-based signature to force write access control without formal model analysis. [111] studied the auditable integrity of VCS to ensure that each version of the repository is retrievable in the malicious server setting.

4.2 Syntax

We abstract seven core operations for E2E encrypted Git services below, and each operation is formalized as an interactive protocol between the user and server (e.g., Git server).

– *Registration.* Users register to the Git server.

– *Authentication.* Users authenticate to the Git server to open an active session, within which users can interact with the Git server to do the following repository operations.
– *Initialization.* Users set up the Git repository structure locally and remotely, which is mapped to `git init` command of Git and initialize the first version of the repository with tracking files.
– *Update.* Users update the repository with new files or new versions of existing files, which are mapped to a series of Git commands `git add, git commit, git push`.
– *Pull.* Users fetch the local repository's missing part from the server to sync with the server's repository, which is mapped to Git commands `git pull, git fetch`.
– *Share.* Users share the repository with others. There are two sub-protocols, denoted as $share_I$ and $share_{II}$, where the sender interacts with the Git server to request, and the receiver interacts with the Git server to accept, respectively.

Syntax. Formally, an end-to-end encrypted Git service is composed of a tuple of interactive protocols $\mathsf{SGit} := (\Pi_{reg}, \Pi_{auth}, \Pi_{init}, \Pi_{update}, \Pi_{pull}, \Pi_{share_I}, \Pi_{share_{II}})$ outlined above, where each is run by a user $\mathcal{U}$ and a server $\mathcal{S}$ via a subroutine, e.g., $\Pi_{reg} = \langle \mathcal{U}_{reg}, \mathcal{S}_{reg} \rangle$. Users and the server maintain their states st_U, st_S, respectively.

In the following, we will describe the protocols with compulsory inputs and outputs and omit other optional ones. In each protocol, each party has a bit of implicit output indicating the execution state, where one indicates it succeeds, otherwise fails.

$\Pi_{reg}\langle uid; st_S \rangle \rightarrow \langle (cred, km); (st'_S) \rangle$**:** the registration protocol creates a new user, where the user takes the unique user ID uid as input and gets the authentication credential $cred$ and key materials km as output. Server updates state st_S with the new user record.

A user record in st_S includes all data related to the user uid. After registration, it has at least two attributes: uid and necessary material for verifying user authentication, e.g., $cred$ or the corresponding public key if $cred$ is a private key. Π_{reg} must be run once on behalf of that user before any other protocols can be run. It does not involve any persistent state of the user yet (i.e., the user state is empty $st_U.s = \epsilon$).

$\Pi_{auth}\langle (uid, cred); st_S \rangle \rightarrow \langle (st_U); (st'_S) \rangle$**:** the authentication protocol authenticates a user to the server and initiates a new active user session. The user takes $uid, cred$ as input.

After passing the authentication, the two parties update their states with the new session state.

This user session state $st_U.s$ is shared among all following protocols run within this session. A user can initiate multiple user sessions in parallel (each holding their own state $st_U.s$), which can concurrently access the user's repositories in the Git server. [5]

$\Pi_{init}\langle(st_U, km, rid, \mathbf{f}^{pt}); st_S\rangle \rightarrow \langle(repo); (st'_S)\rangle$**:** the initialization protocol runs within an active session and initiates a new repository locally and remotely. A user takes as input st_U, km, a globally unique identifier rid, and plain tracking files $\mathbf{f}^{pt}$ including file path, name, and contents. The user outputs Git repository $repo$, including a ciphertext repository $repo^{ct}$. The server updates st_S by adding $(rid, repo^{ct})$ to the user's record.

$\Pi_{update}\langle(st_U, km, rid, repo_{old}, \mathbf{f}^{pt}_{new}); st_S\rangle \rightarrow \langle(repo_{new}); (st'_S)\rangle$**:** the update protocol runs within an active session, and updates the contents locally and remotely. $repo_{old}$ denotes the latest committed repositories locally and $\mathbf{f}^{pt}_{new}$ is the new plain files to be updated. The user's output is the updated repositories $repo_{new}$, including the updated ciphertext repository $repo^{ct}_{new}$. The server updates st_S with an updated user record $(uid, rid, repo^{ct}_{new})$.

$\Pi_{pull}\langle(st_U, km, rid, repo_{old}); st_S\rangle \rightarrow \langle(repo_{new}); (st_S)\rangle$**:** the pull protocol runs within an active session, fetch the missing part from the remote repository, and get the plain contents. The user outputs the new repository $repo_{new} = (repo^{pt}_{new}, repo^{ct}_{new})$, where $repo^{ct}_{new}$ is the latest ciphertext repository in st_S's user record with (uid, rid).

$\Pi_{share_I}\langle(st_U, km, rid, acs, repo_{old}, uid_{re}); st_S\rangle \rightarrow \langle(repo_{new}, oob); (st'_S)\rangle$**:** the repository sharing protocol runs within an active session and enables users to share the access defined in acs of the repository rid with the receiver uid_{re}. The protocol updates the repository to a new version with a new access list. We consider two types of access: read-only and write access, and only the repository owner can share it with others. The user outputs the new repository $repo_{new}$ and the out-of-band message

[5] All remaining protocols operating on repositories can only be called with non-empty user session state $st_U.s$ (i.e., we implicitly require them to fail otherwise). Overall, this enforces that user registration must be run before authentication, and authentication must be run before any repository operation protocol.

oob, which can be communicated via the out-of-band secure channel. The server states that st'_S has one more pending record for managing the receiver's access acs and gets updated with new ciphertext repository $repo^{ct}_{new}$.

$\Pi_{share_{II}}\langle(st_U, rid, oob); st_S\rangle \rightarrow \langle(st'_U); (st'_S)\rangle$**:** the repository accepting protocol runs within an active session and enables the receiver to accept the repository sharing. The server removes the pending access of uid and adds it to the access list of the repository rid. As such, the user uid can access it when logged in.

Remark on revocation. In this paper, we do not consider access revocation. Once a user gets access to a repository, it lasts until the repository gets deleted. To enable access revocation, one possible method is to revoke the user's access to future versions of the repository. It can be done by changing the repository encryption key for future versions, not sharing the encryption key with revoked users, and removing the revoked user's signing key from the repository. However, revoking access to previous versions of the repository is more challenging. A trivial method is to delete the whole repository and re-initialize it from scratch, which is inefficient and results in the loss of all history. Better methods for revocation is an interesting open problem.[6]

Notations for modification. We follow [114] to define document modifications. A document D is denoted as a sequence of blocks $m_1, \ldots, m_n$, where the block size may depend on the security parameter k, and n denotes an integer since a document can always be padded using standard padding methods if the original size is not a multiple of the block size. We use $O = (op, idx, m)$ to denote a generic modification operation, where op is the operation type, idx is the operation position, m is the new message, and $|O.m|$ is the message length. $O(D)$ denotes the effect of O on document D. So, a sequential modification operations $\{O\}_n$ on document D can be denoted as $O_n(\ldots(O_3(O_2(O_1(D)))))$. In this paper, we consider the basic two operation types $O.op \in \{delete, insert\}$ that essentially can capture all modifications on the document, including replace, copy-and-paste, cut-and-paste, etc.

[6]And in general, there are many interesting questions to be studied, including systematic treatment on post-compromise security, e.g., via updatable encryption, e.g., [25, 89], better cryptographic group management [112], full metadata protection [113], accountability, enabling AI assistance while maintaining E2E security, and many more.

$O = (insert, i, m_i)$**:** insert m_i as the i-th block of the document.
$O = (delete, i)$**:** deletes the i-th data block.

Data update. In the update protocol $\mathsf{SGit}.\Pi_{update}$, the modification operations between the two versions f, f' of each tracked file are calculated. We use ComDiff algorithm that takes f, f' as input and outputs a sequential set of modification operations $\{O\}_n$ in the form we defined before. We do not specify the specific construction for ComDiff algorithm. The correctness of ComDiff requires that $f' = O_n(O_{n-1}(\ldots(O_1(f))))$ where $\{O\}_n \rightarrow \mathsf{ComDiff}(f, f')$.

Correctness. When the Git server and all users who have access to the repository act honestly, users can always pull the repository with the same contents as the last push.

Correctness captures that: (1) an honest user registered to the service can authenticate with the same user ID and credentials used during registration; (2) a repository initialized, updated, or shared by an honest user can be retrieved with its original contents.

$$
\begin{aligned}
&\Pr[\Pi_{auth}(uid, cred;) = (1;1) | \Pi_{reg}(uid;) = (1, cred;1)] = 1 \\
\wedge\, &\Pr[\Pi_{pull}(st_U, rid;) = (repo;) | \Pi_{init}(st_U, rid, repo;) = (1;1)] = 1 \\
\wedge\, &\Pr[\Pi_{pull}(st_U, rid;) = (repo';) | \Pi_{update}(st_U, rid, repo';) = (1;1)] = 1 \\
\wedge\, &\Pr[\Pi_{pull}(st_{U_{uid_{re}}}, rid;) = (repo;) | \\
&\Pi_{share_I}(st_U, rid, uid_{re};) = (1;1) \wedge \Pi_{pull}(st_U, rid;) = (repo;)] = 1
\end{aligned}
$$

4.3 Security Models

We will formally define security properties for an E2E encrypted Git service. Intuitively, in a plain Git service, if one fully trusts the Git server, the server can enforce access control policies. End-to-end secure Git services try to "emulate" this ideal setting via algorithm/protocol design. Naturally, it has to satisfy the security requirements of confidentiality (w.r.t the *read* access) and integrity (we consider a stronger version, called unforgeability, that is w.r.t the *write*

access). However, modeling these properties becomes significantly more complex in practice due to the functionality of data *updates* and the multi-user *sharing* setting; these allow adversaries to interact with honest users in a dynamic and complex manner.

Setup assumption: First, we will assume a plain PKI model; that is, users can know others' public keys via an out-of-band channel. This can be achieved via the PGP mechanism in practice.

Data confidentiality, captures not only the content in the repository but also the update details, even against a corrupted Git server, which can interact with honest users. The subtle point is that Git services allow version updates. The ciphertexts of multiple versions can give more power to adversaries than purely exposing the ciphertext of a single version (which is the case in the standard confidentiality model). More specifically, the ciphertexts of multiple versions may not all be generated directly from their plaintexts, and the ciphertext of a later version might be generated based on the ciphertext of its former version. This complicates the modeling and analysis: In the conventional confidentiality model (with CPA flavor), adversaries can obtain a ciphertext by querying a chosen plaintext. But now, adversaries are allowed to additionally obtain ciphertexts by "updating" with a previous ciphertext and a new chosen plaintext. Our confidentiality is already stronger than CCA security.

We also give a slightly weaker version of confidentiality by allowing adversaries to learn the update locations.

Repository unforgeability, tries to capture verifiable write access, which is necessary for Git services of version control among multiple users. Unforgeability guarantees that even if a Git server gets corrupted, each user can only edit on their own behalf and cannot forge other users' edits or frame other honest users. We thus consider that attackers have strong capabilities and could corrupt the server and legitimate users who have read and/or write permission to the target repository (malicious insiders), pretend to be other honest users, and try to forge a new version of the repository on behalf of honest users. For example, a user who has read-only access to the repository may get compromised. In this case, attackers can pull the contents of

the repository but should not be able to break the access restriction (e.g., push) or pretend to any honest user to write even if the attacker corrupts some other users with write access.

Interestingly, by simply restricting adversaries to corrupt only the server and users who do not have read permission to the target repository (can be viewed as external attackers and slightly adapt security games), we can easily get a weaker notion of repository unforgeability called *repository integrity*. This notion may also be useful to ensure repository integrity so that an honest repository will remain complete (no file deletion/insertion without being noticed). [7]

4.3.1 Modeling preparations: oracles & states

To prepare for the security modeling, we first introduce eight oracles $\mathcal{O} = \{\mathcal{O}_{reg}, \mathcal{O}_{auth}, \mathcal{O}_{init}, \mathcal{O}_{pull}, \mathcal{O}_{upd}, \mathcal{O}_{share_I}, \mathcal{O}_{share_{II}}, \mathcal{O}_{corrupt}\}$ to capture the adversary's capability. Since each protocol in SGit is run between the user and the Git server, which is corrupted in all security models. Oracles mainly run user-side algorithms and provide interfaces for adversaries to interact with honest users, except that $\mathcal{O}_{corrupt}$ is provided to corrupt honest users.

Our models consider the single server setting and selective user corruption. So in each security game, adversary $\mathcal{A}$ first specifies the list of corrupted users $U_{corrupt}$, which means later $\mathcal{A}$ can only query $\mathcal{O}_{corrupt}$ with user id $uid \in U_{corrupt}$.

In our security games, $\mathcal{A}$ has the same access to all oracles in $\mathcal{O}$ except different restrictions on the user corruption oracle $\mathcal{O}_{corrupt}$. In the data confidentiality and weak repository unforgeability model, adversaries are not allowed to corrupt users who have legitimate access to the challenge/target repository since the two models only capture security against outsiders of the challenge/target repository. In the repository unforgeability model, adversaries are allowed to query $\mathcal{O}_{corrupt}$ to corrupt insiders with read or write access to the target repository as long as the target user is not corrupted.

We define several global states maintained by the challenger and oracles for security games.

[7] There is an intermediate security notion between repository integrity and unforgeability, which captures integrity against malicious read-only insiders, ensuring that read-only users cannot modify the repository. We leave the systematic exploration of this case, which supports more fine-grained access control, for future work.

U: a set of uid recording registered users.

C: a credential dictionary mapping user id uid to authentication credential $cred$.

K: a key material dictionary mapping user id uid to a tuple of key materials $km = (mk, sk_e, pk_e, sk_s, pk_s)$.

R: a set of rid recording existing repositories.

S: a user session state dictionary mapping a tuple of user id uid and session id sid to the session state st.

RP: a dictionary mapping repository id rid to its latest local repository $repo$.

O: a dictionary mapping repository id rid to its owner id uid.

$A[rid]$: the set of accessible users uid for the repository id rid.

$W[rid]$: the set of users uid with write access to the repository rid.

rid^*: the challenge repository identifier.

fid^*: the challenge file id.

f_b^*: the two challenge related plain files for $b \in \{0, 1\}$.

$repo^*$: the challenge repository.

The formal oracle description is shown in Figure 4.2. For clarity, each oracle has an implicit output indicating the procedure succeeds or fails and is specified for other output. The details of oracle description are described as follows:

$\mathcal{O}_{reg}$: allows $\mathcal{A}$ to initiate a user registration with user id uid. The generated credential and key materials are hidden from $\mathcal{A}$ and can be corrupted via $\mathcal{O}_{corrupt}$. Each uid is globally unique and can only be registered once with a successful record.

$\mathcal{O}_{auth}$: allows $\mathcal{A}$ to initiate the user authentication with given uid. A successful authentication starts a new session with id sid and persistent state $S[uid, sid]$.

$\mathcal{O}_{init}$: allows $\mathcal{A}$ to initialize the repository with repository id rid, the plain files $\mathbf{f}^{pt}$ including each file path $fid \in \mathbf{f}^{pt}.Fid$ and corresponding contents $\mathbf{f}^{pt}[fid]$.

$\mathcal{O}_{pull}$: retrieves the latest repository rid in the specified active session sid on behalf of user uid. It returns the specified repository $repo_{new}$. To avoid $\mathcal{A}$'s trivial win of the confidentiality game, the retrieval of the challenge repository only returns plaintext versions of non-challenge files.

$\mathcal{O}_{reg}(uid)$

1 : **if** $uid \in U$ **return** "Registered!"
2 : **else** $\langle \mathcal{U}_{reg}(uid), \mathcal{A} \rangle \rightarrow (cred, km)$ // only show user-side output
3 : add $uid \rightarrow U, km \rightarrow K[uid], cred \rightarrow C[uid]$
4 : **return** (uid, pk_e, pk_s). // km includes $(mk, sk_e, pk_e, sk_s, pk_s)$

$\mathcal{O}_{auth}(uid)$

1 : **if** $uid \notin U$, **return** $\perp$ // only for registered users via $\mathcal{O}_{reg}$
2 : **else** $cred \leftarrow C[uid],\ sid \leftarrow\$,\ \langle \mathcal{U}_{auth}(uid, cred), \mathcal{A} \rangle \rightarrow st,$ set $st \rightarrow S[uid, sid],$ **return** sid

$\mathcal{O}_{init}(uid, sid, rid, \mathbf{f}^{pt})$

1 : **if** $S[uid, sid] = \epsilon \vee rid \in R$ **return** $\perp$ // only for active sessions of $\mathcal{O}_{auth}$
2 : **else** $\langle \mathcal{U}_{init}(S[uid, sid], K[uid], rid, \mathbf{f}^{pt}), \mathcal{A} \rangle \rightarrow repo_{new}$
3 : add $rid \rightarrow R,\ uid \rightarrow A[rid],\ uid \rightarrow W[rid],$ set $repo_{new} \rightarrow RP[rid],\ uid \rightarrow O[rid]$

$\mathcal{O}_{upd}(uid, sid, rid, \mathbf{f}^{pt}_{new})$

1 : **if** $S[uid, sid] = \epsilon \vee uid \notin W[rid]$ **return** $\perp$
2 : **else** parse $RP[rid] = repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$
3 : **if** $rid = rid^* \wedge fid^* \in \mathbf{f}^{pt}_{new}.Fid$ **then** set $f \leftarrow \mathbf{f}^{pt}_{new}[fid^*]$
run $O_0 \leftarrow \mathsf{ComDiff}(f^*_0, f), O_1 \leftarrow \mathsf{ComDiff}(f^*_1, f)$
4 : require $O_0.op = O_1.op \wedge |O_0.m| = |O_1.m|$
// require same type and length of modification for challenges f^*_0, f^*_1
5 : $\boxed{\text{require } O_0.idx = O_1.idx}$ // require modification on same position
6 : run $\langle \mathcal{U}_{upd}(S[uid, sid], rid, repo_{old}, \mathbf{f}^{pt}_{new}), \mathcal{A} \rangle \rightarrow repo_{new},$
7 : set $repo_{new} \rightarrow RP[rid]$, **if** $rid = rid^*$, set $RP[rid] \rightarrow repo^*$

$\mathcal{O}_{pull}(uid, sid, rid, v)$

1 : **if** $S[uid, sid] \rightarrow st = \epsilon$ **return** $\perp$
2 : **else** $\langle \mathcal{U}_{pull}(st, rid, RP[rid]), \mathcal{A} \rangle \rightarrow repo_{new} = (repo^{pt}_{new}, repo^{ct}_{new}),$ set $repo_{new} \rightarrow RP[rid]$
3 : **if** $rid \notin R$ **then** add $rid \rightarrow R, repo^{ct}_{new}.owner \rightarrow O[rid]$
4 : **for** each $f_{acs} \in repo^{ct}_{new}$
5 : set $A[rid] \cup f_{acs}.R \rightarrow A[rid], W[rid] \cup f_{acs}.W \rightarrow W[rid]$
6 : **if** $rid \neq rid^*$ **return** $repo_{new}$
7 : **else** set $repo_{new} \rightarrow repo^*$ **return** $\mathbf{f} \in repo^{pt}_{new}$, where $fid^* \notin \mathbf{f}.Fid$ // no challenge file return

$\mathcal{O}_{share_I}(uid, sid, rid, uid_{re}, acs)$

1 : **if** $S[uid, sid] \rightarrow st = \epsilon \vee uid \notin O[rid]$ **return** $\perp$
2 : **else** $\langle \mathcal{U}_{shr}(st, rid, uid_{re}, K[uid], RP[rid]), \mathcal{A} \rangle \rightarrow (repo_{new}, oob)$
3 : set $repo_{new} \rightarrow RP[rid],$ update $A[rid], W[rid]$ per $acs,$ **return** oob

$\mathcal{O}_{share_{II}}(uid, sid, rid, oob)$

1 : **if** $S[uid, sid] = \epsilon$ **return** $\perp$ **else** $\langle \mathcal{U}_{acp}(S[uid, sid], rid, oob), \mathcal{A} \rangle$

$\mathcal{O}_{corrupt}(uid)$

1 : **if** $uid \notin U \vee uid \notin U_{corrupt}$ **return** $\perp$ **else return** $(C[uid], K[uid])$

FIGURE 4.2. The oracles of data confidentiality. $\boxed{Boxed}$ is for weaker confidentiality.

$\mathcal{O}_{upd}$**:** updates the repository rid with new files $\mathbf{f}^{pt}$ on behalf of user uid in session sid. For update queries on the challenge repository, further checks are needed to avoid $\mathcal{A}$'s trivial wins via differences of update operation and content length or update position to get an advantage in the confidentiality game.

$\mathcal{O}_{share_I}$**:** initiates the repository sharing. $\mathcal{A}$ specifies uid, sid, rid, and the receiver id uid_{re}. It returns the out-of-band message oob.

$\mathcal{O}_{share_{II}}$**:** initiates the receiver acceptance process of sharing within an active session. $\mathcal{A}$ specifies uid, sid, rid.

$\mathcal{O}_{corrupt}$**:** allows $\mathcal{A}$ to corrupt honest users and returns their secrets.

4.3.2 Data confidentiality

Data confidentiality captures both the file content confidentiality and *update* confidentiality against outsiders (who do not have legitimate access to the target repository), including the malicious server, except the length of the initial file and update metadata.

The confidentiality game is defined in Figure 4.3. In the game, $\mathcal{A}$ has access to all eight oracles in $\mathcal{O}$. In the challenge submission phase, $\mathcal{A}$ submits a targeted registered honest user uid^* and his repository identified by rid^*, and specifies the file id fid^* and two challenge files f_0^*, f_1^*. The challenger $\mathcal{C}$ randomly selects one file f_b^* to initiate or update the repository via interacting with $\mathcal{A}$. $\mathcal{A}$'s goal is to distinguish which file (the bit b) was chosen. To avoid trivial wins, $\mathcal{A}$ is not allowed to corrupt any user who has legitimate access to the challenged repository.

DEFINITION 4.3.1 (Data Confidentiality). Let SGit be a Git service, and $G^{\texttt{CONF}}_{\mathsf{SGit},\mathcal{A},q}$ be the data confidentiality game defined in Figure 4.3 with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of $\mathcal{A}$ playing this game as

$$\mathrm{Adv}^{\texttt{CONF}}_{\mathsf{SGit},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\texttt{CONF}}_{\mathsf{SGit},\mathcal{A},q} = 1] - 1/2.$$

Data Confidentiality Game $G^{\mathtt{CONF}}_{\mathsf{SGit},\mathcal{A},q}$

$b \leftarrow\$ \{0,1\}$
Global $U, C, K, R, S, RP, O, \mathbf{A}, \mathbf{W}, rid^*, fid^*, f_0^*, f_1^*, repo^*$
$U_{corrupt} \leftarrow \mathcal{A}$ // specify user corruption
$(uid^*, sid^*, rid^*, fid^*, f_0^*, f_1^*) \leftarrow \mathcal{A}^{\mathcal{O}}$ // submit challenge after queries
update global state $(rid^*, fid^*, f_0^*, f_1^*)$
if $uid^* \notin U \vee S[uid^*, sid^*] \rightarrow st = \epsilon \vee (rid^* \in R \wedge uid^* \notin W[rid^*])$
  return $\perp$ // exclude invalid challenge
if $uid^* \in U_{corrupt} \vee U_{corrupt} \cap A[rid^*] \neq \emptyset$
  return $\perp$ // exclude insider corruption
if $rid^* \notin R$ **then** set $f_{ori} = \emptyset$
else parse $repo_{old} \leftarrow RP[rid^*]$to get $\mathbf{f}_l^{pt}$ // $\mathbf{f}_l^{pt}$ is the latest version of plain files
  set $f_{ori} \leftarrow \mathbf{f}_l^{pt}[fid^*]$
$O_0 \leftarrow \mathsf{ComDiff}(f_{ori}, f_0^*), O_1 \leftarrow \mathsf{ComDiff}(f_{ori}, f_1^*)$
if $|O_0| \neq |O_1| \vee O_0.op \neq O_1.op$ **return** $\perp$
// exclude trivial win with different update operation or length
$\boxed{\textbf{if } O_0.idx \neq O_1.idx \textbf{ return } \perp}$ // exclude trivial win via different update locations
if $rid^* \notin R : \langle \mathcal{U}_{init}(st, rid^*, K[uid^*], fid^*, f_b^*), \mathcal{A} \rangle \rightarrow repo_{new}$
else $\langle \mathcal{U}_{upd}(st, K[uid^*], rid^*, repo_{old}), \mathcal{A} \rangle \rightarrow repo_{new}$
// challenge enc or update
set $repo_{new} \rightarrow repo^*, repo_{new} \rightarrow RP[rid^*]$
$b' \leftarrow \mathcal{A}^{\mathcal{O}}$ // $\mathcal{A}$ guess after challenge and queries
if $U_{corrupt} \cap A[rid^*] \neq \emptyset$, **return** $(b = b')$; **else return** $\perp$

FIGURE 4.3. The data confidentiality game. $\boxed{boxed}$ is for weaker confidentiality $G^{\mathtt{CONF_w}}_{\mathsf{SGit},\mathcal{A},q}$, where the update position is protected.

Remark. We define a weaker version of data confidentiality, called weak data confidentiality, as follows. It is formalized via the game $G^{\mathtt{CONF}_w}_{\mathsf{SGit},\mathcal{A},q}$, in which the attacker is allowed to learn the update positions.

DEFINITION 4.3.2 (Weak Data Confidentiality). Let SGit be a Git service, and $G^{\mathtt{CONF_w}}_{\mathsf{SGit},\mathcal{A},q}$ be the weak data confidentiality game defined in Figure 4.3 including additional boxed restriction, with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of the adversary playing this game as

$$\mathrm{Adv}^{\mathtt{CONF_w}}_{\mathsf{SGit},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{CONF_w}}_{\mathsf{SGit},\mathcal{A},q} = 1] - 1/2.$$

4.3.3 Repository unforgeability

Repository unforgeability captures that an adversary cannot forge a new version of a valid ciphertext repository on behalf of honest users, even if the adversary has the capability to corrupt users with write access to the repository (weak repository unforgeability restricts the adversary's capability to access the repository). Moreover, this unforgeability inherently inherently enforces both edit-source authenticity and read-write access separation. Edit-source authenticity ensures that users cannot impersonate others when editing the repository, preventing any user from writing a new version on behalf of another without detection. Read-write access separation guarantees that read-only users are cryptographically prevented from performing write operations. A weaker form of unforgeability still ensures write access control against attackers with no access privileges.

The unforgeability game is defined in Figure 4.4, where $\mathcal{A}$ has access to all eight oracles in $\mathcal{O}$. To capture the security against malicious insiders, $\mathcal{A}$ is allowed to corrupt users who have legitimate access to the challenge repository except for the challenged honest user, which may cause a trivial win. In this game, $\mathcal{A}$'s goal is to impersonate an honest user by forging a new version of the repository on behalf of the target honest user. The weaker unforgeability game imposes an additional restriction, boxed in Figure 4.4, which prohibits the adversary $\mathcal{A}$ from corrupting any user who has access (read or write) to the target repository.

DEFINITION 4.3.3 (Repository Unforgeability). Let SGit be a Git service, and $G^{\mathtt{UNF}}_{\mathsf{SGit},\mathcal{A},q}$ be the repository unforgeability game defined in Figure 4.4 with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of an adversary playing this game as

$$\mathrm{Adv}^{\mathtt{UNF}}_{\mathsf{SGit},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{UNF}}_{\mathsf{SGit},\mathcal{A},q} = 1].$$

Repository integrity. We define repository integrity (also called weak repository unforgeability) against malicious repository outsiders, including the malicious server, except users who have legitimate access to the repository. It captures that attackers who have no access to the target repository cannot forge a new version of the target repository, even given many

Repository Unforgeability Game $G^{\mathtt{UNF}}_{\mathsf{SGit},\mathcal{A},q}$

1 : Global U, C, K, R, S, RP, O, A
2 : $U_{corrupt} \leftarrow \mathcal{A}$ // specify user corruption
3 : $(uid^*, sid^*, rid^*, repo^*_{ct}) \leftarrow \mathcal{A}^{\mathcal{O}}$ // submit challenge after queries
4 : **if** $repo^*_{ct} \subseteq RP[rid^*] \vee uid^* \in U_{corrupt} \vee uid^* \notin U \vee$
5 : $S[uid^*, sid^*] \rightarrow st = \epsilon$ **return** $\perp$
// exclude trivial win and invalid challenge
6 : $\langle \mathcal{U}_{pull}(st, rid^*, K[uid^*], RP[rid^*]), \mathcal{A} \rangle \rightarrow (repo^*;)$
7 : **require** $repo^* = (repo^*_{pt}, repo^*_{ct}) \neq \perp$ // check valid repo
8 : $\boxed{\textbf{require } A[rid^*] \cap U_{corrupt} = \emptyset}$ // exclude trivial win via user corruption
9 : **if** $\exists\ repo^*_v, s.t., repo^*_v \in repo^*_{ct} \wedge repo^*_v \notin RP[rid^*] \wedge f_{tag}.au = uid^*$
// $repo^*_v = (\mathbf{f}^{ct}, f_{acs}, f_{tag})$ is a version of repository, where $f_{tag}.au$ is the author
10 : **return** 1 // $\mathcal{A}$ win if exist one untracking version edited by honest user
11 : **else return** 0

FIGURE 4.4. The repository unforgeability game. $\boxed{boxed}$ is for weaker unforgeability $G^{\mathtt{UNF_w}}_{\mathsf{SGit},\mathcal{A},q}$ (called repository integrity), in which corrupted users do not have any legitimate access to the target repository.

versions of the repository. This guarantees that even a malicious server cannot cheat users with an incomplete version of the target repository where partial files are deleted or lost.

The Integrity game is shown in Figure 4.4 with additional restrictions in the box to the user corruption. During the game, $\mathcal{A}$ has access to the eight oracles, and the goal is to provide a new version of the ciphertext repository, which is valid but is different from all existing versions. The trivial win is that $\mathcal{A}$ corrupts a user who has legitimate access to the target repository.

DEFINITION 4.3.4 (Repository Integrity). Let SGit be a Git service, and $G^{\mathtt{INT}}_{\mathsf{SGit},\mathcal{A},q}$ be the repository integrity game, which is also weak repository unforgeability game $G^{\mathtt{UNF_w}}_{\mathsf{SGit},\mathcal{A},q}$ shown in Figure4.4 including the boxed restriction with any probabilistic polynomial-time adversary $\mathcal{A}$ querying at most q times. We define the advantage of the adversary playing this game as

$$\mathrm{Adv}^{\mathtt{INT}}_{\mathsf{SGit},\mathcal{A},q}(\mathcal{A}) = \mathrm{Adv}^{\mathtt{UNF_w}}_{\mathsf{SGit},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{UNF_w}}_{\mathsf{SGit},\mathcal{A},q} = 1].$$

Repository unforgeability and integrity. We defined repository unforgeability and integrity (the weaker version of unforgeability) to capture different attackers. In both security modelings, adversaries share the same goal of forging a valid ciphertext. But they have different capabilities. The integrity adversary can be seen as an outside attacker who has no access to the repository. However, the unforgeability adversary acts as an inside attacker who has legitimate access to the repository, including read and write access. It is easy to conclude that the unforgeability against insiders is stronger than the integrity against outsiders. For this reason, we will only prove the unforgeability against insiders for our constructions.

Further modeling discussion: *strengthening integrity and unforgeability*. In this paper, we consider integrity and unforgeability where malicious attackers can not forge a different one from existing versions of the repository. The malicious server may delete or lose an entire version of a repository. A stronger security notion captures that it can be caught if the malicious server cannot provide an old version. A promising solution could be to use a hash chain to link the previous versions with the next version and sign it so that, as long as the hash chain is signed, malicious attackers cannot forge one from an internal point. *Remarks.* Regarding defending denial-of-service (DoS) attacks, we always assume the server is semi-honest, which blocks illegitimate users and provides available service to legitimate users. Also, besides the data confidentiality, there could also be metadata privacy protecting the file name, file directory, etc, which we leave for further study.

4.4 Provably Secure Constructions

We propose two constructions of E2E encrypted Git services, SGitChar and SGitLine, which are fully compatible with existing Git servers, including GitHub, and formally analyze their security. In this section, we first take SGitChar as an example to show the main workflow. Then we introduce our two constructions: SGitLine which we briefly describe and achives weak data confidentiality, and SGitChar which we describe in detail and which satisfies both data confidentiality and repository unforgeability.

Overview of E2E encrypted Git workflow. In E2E encrypted Git services, both a client (a user's device) and the Git server maintain a repository but in the form of ciphertext. To enable the user to efficiently read and edit the repository, the user's device maintains one more repository with the corresponding plaintext. As in conventional Git services, users register at the very beginning and authenticate to the Git server before doing repository-related operations. After authentication, the initialization protocol enables the user to create a new repository and synchronize the first version with the Git server. Later, users can update the repository and synchronize with the Git server, share the repository with other users, and pull new versions from the Git server to synchronize the local repository.

Concretely, (1) for secure initialization, a user initiates two repositories for plain data and ciphertext first, configures the remote ciphertext repository, and connects it with the local ciphertext one. Then, the user takes the first version of the plaintext repository as input, generates a ciphertext version by applying encryption on each file, commits it to the local ciphertext repository with a signature, and pushes it to the remote ciphertext repository in the Git server to finish the initialization. (2) Regarding the secure update, the user has the previous and new plaintext repository and the previous ciphertext repository, forms a new version of ciphertext repository locally, and pushes it to the Git server. The methods to form a new version of the ciphertext repository are different for the two constructions. (3) Regarding secure pull, the user pulls local absent versions from the Git server. The workflow is that the user first pulls the remote ciphertext version to the local ciphertext repository and then verifies and recovers the plaintext version of the repository. The ways of recovering the plaintext version correspond to the methods to form the ciphertext version, which are different in the two schemes. (4) To share a repository with others, the sender needs to send a request to the Git server to give access permission to the receiver. The sender also needs to update the ciphertext repository with a new access control file, which includes key material encrypted under the receiver's public key and the sender's authorization via a signature. The receiver needs to accept the access via interacting with the Git server.

We assume each user has two key pairs, for digital signature and public key encryption, which are bound to the user's identity. The distribution of public keys is via an out-of-band channel

so that each user knows other users' public keys. Each user has a small, constant size of secure storage, e.g., hundreds of bits, for keeping secrets locally. To be compatible with the most widely deployed user authentication, we support users in authenticationng to the server usingwith general credentials such as a passwords and a tokens. So, each users may need to remember a passwords and keep all private secrets locally.

A diagram describing the main workflow of the pullupdate and updatepull procedure with SGitChar as the specific construction is shown in Figure 4.5.

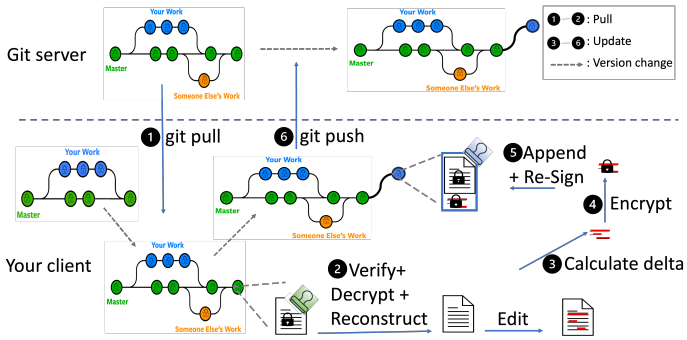


FIGURE 4.5. The main workflow of SGitChar

Preparation of construction. An important component of Git we will leverage for efficient construction is the Diff computation functionality. $\mathsf{ComDiff}(repo, repo') \to \delta$**:** The ComDiff algorithm takes two versions of a repository as input, and generates the difference δ. The difference δ makes sure that $repo'$ can be reconstructed from $repo$ and δ. The reverse reconstruction is not compulsory but could be useful for optimizing reconstruction efficiency. With different implementations of ComDiff, the size of difference δ is also different. One direction of optimizing the storage cost is to make δ as small as possible. We have two schemes $\mathsf{ComDiff}_{char}$ and $\mathsf{ComDiff}_{line}$ with different granularities to compute the difference. $\mathsf{ComDiff}_{char}$ compares difference between two characters and $\mathsf{ComDiff}_{line}$ for two lines. For the two repositories, the size of $\mathsf{ComDiff}_{char}$ should be no larger than $\mathsf{ComDiff}_{line}$.

4.4.1 Construction: SGitLine

A secure construction should get rid of deterministic encryption that leaks data patterns and only provides a weak security guarantee. To reduce cost, the encryption is not trivially applied on all files so that the computation and storage cost is not linear to the product of the version number and file size. One of our goals is to balance confidentiality and efficiency. The other is to achieve desired integrity and unforgeability while keeping confidentiality and efficiency.

After a careful investigation of version control systems, we observe that they have a more fine-grained data partition and location method, so that they can reduce storage and communication for repetitive data. We can apply standard encryption on a smaller unit of data, aligning with the version control system's common data processing unit, so that any changes can be located in a more fine-grained way, not as a whole file or repository.

Git already treats lines as essential units, organizing data based on line, and utilizing line indexes to display differences. So we apply symmetric encryption to the repository in a line-wise way. For each data update, only those changed lines are re-encrypted, and unchanged lines remain unchanged in terms of ciphertexts. We propose using lines as the treatment unit, as the name SGitLine indicates.

Leveraging this existing organization eliminates the need for partitioning and reconstruction. Additionally, the flexibility of line length allows users to customize it, potentially mitigating the significance of IV storage, especially in average cases with longer lines.

At a high level, SGitLine involves encrypting data line by line, maintaining the ciphertext unchanged if the plaintext remains the same. In the repository initialization procedure, the user first encrypts every line for each file and then takes the entire ciphertext version as a whole together with the repository id and the user's id to sign using the user's private key. Digital signature helps for the integrity and unforgeability. The repository id and user id bind the repository version to the specified repository and the user who writes this version. In subsequent update operations, the user compares the line-wise differences between two versions of the plaintext repository before and after the update. These differences

indicate insert and delete modifications, such as which lines are deleted, and where to insert a line of content. With the modifications, the user can encrypt the contents line by line in each modification, and operate each modification on the ciphertext repository with the corresponding ciphertext. Then, the user follows the same way to sign the entire version of the ciphertext repository. It is evident that in the ciphertext repository, unchanged lines remain unchanged. Consequently, the computation, communication, and storage cost of one more version is only linear to the size of changed lines, not the entire version of the repository. In the pull procedure, the user first interacts with the Git server to retrieve the entire version of the ciphertext. Then, the user checks the signatures to make sure the pulled version is written by a valid user with write access to the repository. After passing all checks, the user decrypts ciphertexts line-by-line for each file to form the plaintext repository. The share procedure includes requesting access permission to the server (which relies on the Git server api is), appending the encrypted key material to the access control file, and authorizing the sharing via signing the access file on the repository. The receiver accepts the sharing access via the server share api. Later, when reading or writing the repository, the receiver first decrypts the encrypted key file to get the data encryption key.

Storage drawback. While SGitLine offers substantial storage savings compared to simply applying encryption to the entire repository, it may incur higher storage costs for code repositories. This is particularly true for repositories where each line tends to be very short. Additionally, in the case of minor patch updates involving only a few word changes on certain lines (e.g., correcting typos), the storage cost could be significantly larger compared to the plaintext repository.

Security drawback: We will prove that SGitLine satisfies the weak data confidentiality in Section 4.4.4.3. Regarding weak confidentiality, it does not protect the updated position. While SGitLine ensures IND-CPA security for the repository data, it does not conceal the update operation itself. Specifically, the position information about which lines get deleted and which lines get newly inserted remains unprotected. Also, the length of each line of the file is unprotected.

4.4.2 Construction: SGitChar

A secure construction should avoid deterministic encryption, which leaks data patterns. To reduce cost, encryption cannot be trivially applied to all files, as this would make the total computation and storage cost scale linearly with the number of repository versions n and the size of each version f. Our goal is to achieve both security (at least standard semantic security) and high efficiency where the update cost is independent of the whole repository size, ideally only depending on the size of the modified content.

The high-level idea of SGitChar is that by shifting our perspective on repository updates, we can consolidate various update operations within a file into a single operation encapsulating a set of *differences*. With this approach, a single insert operation containing a line of content encompassing all differences can record all the modifications with minimal storage cost. More importantly, by aggregating all differences into one operation and storing them in one position (at the end of the file), which is independent of any modification position, we effectively conceal the position of each internal update operation. The encryption cost is only related to the size of the difference, not the entire version of the repository, which reduces the computation cost. For similar reasons, the communication cost of push/pull is also minimal and depends only on the size of the difference between the two consecutive versions.

Furthermore, the straightforward method to achieve unforgeability is signing all files of one repository version, which costs linear time to the size of the version. Inspired by the Git tree graph to organize the repository directory and compute the root hash of all files, we do not directly sign on all files but sign on the root hash of all files and their hierarchical organization structure, which is known as a tree structure. In this way, a single file change only needs hash computation on the changed file as a leaf node and on the intermediate nodes to the root of the tree that depends only on the height of the leaf node (not related to the size of the entire version of files).

Detailed construction. Let $\Pi_{\mathsf{authenticate}} = (\Pi_{\mathsf{AuthReg}}, \Pi_{\mathsf{Auth}})$ be any authentication mechanism. Let $\Pi_{SE} = \{\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}\}$ be a symmetric key encryption scheme, KDF be a secure key derivation function, $Hash$ be a collision resistant hash function, and `MerkleDAG` be a directed

acyclic graph structured collision resistant hash function [115]. Let $\Pi_{PKE} = \{\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}\}$ be public key encryption, and $\Pi_{DS} = \{\mathsf{KG}, \mathsf{Sign}, \mathsf{Vrfy}\}$ be digital signature. The detailed constructions of SGitChar and SGitLine are described as follows and shown in Figure 4.8.

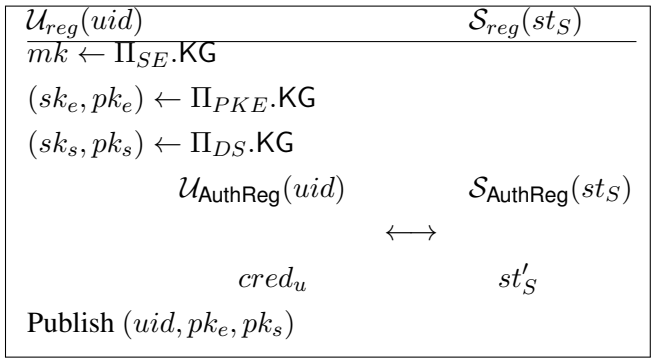


FIGURE 4.6. The registration protocol

$\Pi_{reg}(uid; st_S) \rightarrow (cred, km; st'_S)$**:** a user registers a new account with user id uid, formally shown in Figure 4.6. In the registration, the user first runs the key generation algorithm KeyGen includes running symmetric key encryption's key generation algorithm $\Pi_{SE}.\mathsf{KG}$, the key generation of public key encryption $\Pi_{PKE}.\mathsf{KG}$, and digital signature's key generation algorithm $\Pi_{DS}.\mathsf{KG}$ to generate the key material $km = (mk, pk_e, sk_e, pk_s, sk_s)$ including a master secret key mk, a key pair (pk_e, sk_e) for public key encryption, and a key pair (pk_s, sk_s) for digital signature. Then the user and the server run Π_{AuthReg} of an authentication mechanism to let the user get a credential $cred$ and the server update st_S for later authentication.

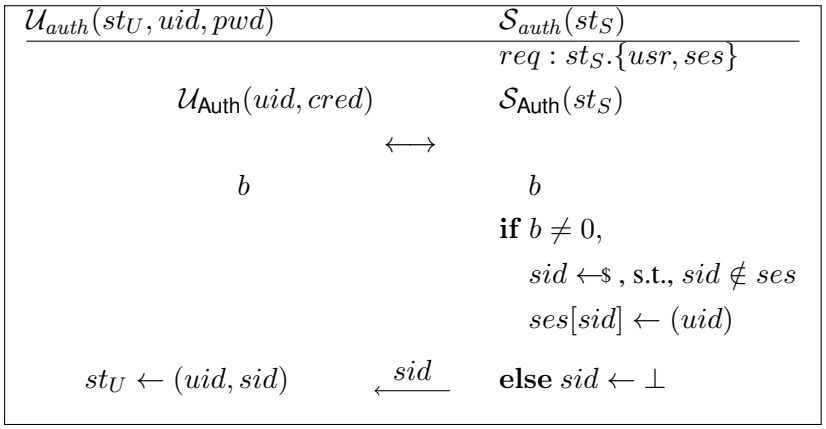


FIGURE 4.7. The authentication protocol

$\Pi_{auth}(uid, cred; st_S) \rightarrow (st'_U; st'_S)$**:** a user authenticates to the server by running Π_{Auth}, which is formally depicted in Figure 4.7. Concretely, the user interacts with the server to run the authentication procedure Π_{Auth} and sets the user session state $st_U \leftarrow (uid, sid)$. In the current programmable access to Git and our experiment, we use a token-based authentication method, i.e., in the registration phase, the token is randomly generated by the server and kept secret by the user for authentication. During the authentication procedure, the user shows the token to authenticate to the server.

$\Pi_{init}(st_U, km, rid, \mathbf{f}^{pt}; st_S) \rightarrow (repo_{new}, st'_U; st'_S)$**:** In an active session, a user first encrypts each file in $\mathbf{f}^{pt}$ with the key k derived from the master key mk and the repository id rid to get a ciphertext version $\mathbf{f}^{ct}$. For SGitChar, the user takes each file as a whole to run encryption and get the ciphertext as content. For SGitLine, the file is parsed by line, and users take each line as input to run encryption to get the ciphertext as the content of the corresponding line of the ciphertext file. Then, the user runs $rh \leftarrow \texttt{MerkleDAG}(\mathbf{f}^{ct})$ to hash on ciphertexts and the file structure, and gets the signature tag $\sigma \rightarrow \mathsf{Sign}(sk, Hash(rid \| uid \| rh))$, then creates a new repository with the repository id $rid = (uid, nonce)$, required to be globally unique and include the creator's uid and a random nonce. The new ciphertext repository of the client and Git server includes tracking ciphertext files $\mathbf{f}^{ct}$, an empty access file $f_{acs} = \emptyset$, and a tag message $f_{tag} = (uid, \sigma)$.

$\Pi_{update}(st_U, km, rid, repo_{old}, \mathbf{f}^{pt}_{new}; st_S) \rightarrow (st'_U, repo_{new}; st'_S)$**:** a user first takes latest committed plaintext and ciphertext repository $repo_{old} = ({repo_{old}}^{pt}, {repo_{old}}^{ct})$, and the new plaintext files $\mathbf{f}^{pt}_{new}$ as input to get ciphertext files $\mathbf{f}^{ct}_{new}$.

Concretely, for newly added files, the user encrypts files as in initialization to get the ciphertext version. For updated files, the user computes the differences with the last committed plain file and encrypts the differences to get the corresponding updated ciphertext file. SGitChar runs $\mathsf{ComDiff}_{char}$ on the new and last prior version of the plain file to get a set of modifications $\{O\}_z$, encrypts $\{O\}_z$, and appends the ciphertext at the end of last prior version of ciphertext file, to get the new version of corresponding ciphertext file. While SGitLine runs $\mathsf{ComDiff}_{line}$ to get $\{O\}_z$ and

only encrypts the contents of the insert operation to replace the plaintext content and leave the delete operation unmodified. Then, those operations are applied to the last prior ciphertext files to get new versions that correspond to them.

With new ciphertext files $\mathbf{f}_{new}^{ct}$ and unchanged last prior ciphertext files as the new version of the repository, the user then hashes and signs the new version to get a tag, then commits it with a message including the tag and push to the Git server.

$\Pi_{pull}(st_U, km, rid, repo_{old}; st_S) \rightarrow (st'_U, repo_{new}; st'_S)$**:** In an active session, a user interacts with the server to fetch the missing versions from the server. Then, the user runs the signature verification algorithm Vrfy to check the integrity of each missing version and access control file. If all checks pass, the user derives or decrypts to get the data encryption key. Then, do the following to get a new plaintext repository.

For each missing version from old to new, SGitChar runs the decryption algorithm on the differences compared with the last prior version to get a set of modification operations and applies modifications on the last prior plaintext version to get the next version of the plaintext repository. SGitLine can directly decrypt each file line-by-line to get the next new version or, based on the difference, decrypt the contents of insert operations as new content and leave the content of delete operation empty, then apply those operations on the last prior version of plaintext files to get the next plaintext version.

$\Pi_{share_I}(st_U, km, rid, uid_{re}, acs, repo_{old}; st_S) \rightarrow (st'_U, repo_{new}; st'_S)$**:** a user interacts with the server to add a collaborator with the id uid_{re} for the repository identified by rid so that the user uid_{re} also has access acs to the repository. The user generates ct_{shr}, which is an encryption of the repository encryption key and sender uid under a collaborator's public key $\mathsf{PKE.Enc}(pk_e^{uid_{re}}, uid \| k)$. Then the user updates the access control file accordingly by adding one entry (uid_{re}, ct_{shr}) to the read list $f_{acs}.R$ repository by inserting ct_{share} to the shared file. If acs is write access, then add uid_{re} to write list $f_{acs}.W$. Then, the user hashes and signs the repository according to the new ciphertext version, commits locally, and pushes it to the Git server.

$\Pi_{share_{II}}(st_U, rid, oob; st_S) \rightarrow (st'_U; st'_S)$**:** After receiving the out-of-band message from the Git server, the user can take this message as input to interact with the server to accept the shared repository access right via APIs of the Git server.

4.4.3 Construction extensions

We give three further extensions to support more functionalities, including the delete and merge function, to enable portability and to optimize retrieval efficiency.

More functionalities. Based on the syntax and Git supported operations, we can easily extend our E2E encrypted Git services to support more functions, such as file deletion and branch merge. Both are special cases of the update operation Π_{update}, which can be captured by our general construction, and do not affect the security.

To delete a file, the update protocol can achieve it with the input $\mathbf{f}^{pt}_{new}$, which includes a file with the same file name as the file to be deleted and empty content.

To merge two branches, user can run update protocol with specified inputs: $repo_{old}$ and $\mathbf{f}^{pt}_{new}$, where $repo_{old}$ is one branch of the repository (identified branch 1), and $\mathbf{f}^{pt}_{new}$ is the files in the other branch (identified as branch 2) which are different from branch 1. Git provides basic functions to find the needed $\mathbf{f}^{pt}_{new}$, such as `git diff`. The method works as it applies one branch's update to the other branch so that the updated version includes all updates of the two branches. The result is the correct merge version as it is independent of the order of branches. Please note that the Git merge function only merges two branches without conflict updates, i.e., different updates on the same file, so does our method.

Achieving semantic security and portability simultaneously. The above secure Git services require users to keep their secret keys by themselves. The trivial way is to store the secrets locally, which hinders the portability and brings extra inconvenience when users want to change devices or just access remote repositories via multiple devices. Local storage is vulnerable to all kinds of fishing attacks, viruses, ransomwares, etc. To get rid of the reliance

$< \mathcal{U}_{Init}(st_U, km, rid, \mathbf{f}^{pt}), \mathcal{S}_{Init}(st_S) >$

$\mathcal{U}$: $req : st_U.\{uid, sid\}$. $k \leftarrow \mathsf{KDF}(mk, rid)$

$\mathcal{U}$: **for** $f_i \in \mathbf{f}^{pt}, i \in [1, n]$ / n is # of content files in $\mathbf{f}^{pt}$

$\mathcal{U}$: **for** $j \in [1, f_i.l]$ / $f_i.l$ is # of lines in f_i

$\mathcal{U}$: $lct_j \leftarrow Enc(k, l_j)$ / encrypt by line

$\mathcal{U}$: $ct_i \leftarrow (lct_1, \ldots, lct_{f_i.l})$ $ct_i \leftarrow Enc(k, f_i)$

$\mathcal{U}$: $\mathbf{f}^{ct} \leftarrow (ct_1, \ldots, ct_n)$

$\mathcal{U}$: $rh \leftarrow \mathsf{MerkleDAG}(\mathbf{f}^{ct})$ / each ct_i is a leaf node, $i \in [1, n]$

$\mathcal{U}$: $h \leftarrow Hash(rid \| uid \| rh), \sigma \leftarrow \mathsf{Sign}(sk_s, h)$

$\mathcal{U}$: add commit message $f_{tag} \leftarrow (uid, \sigma)$

$\mathcal{U}$: $repo^{ct}_{new} \leftarrow (\mathbf{f}^{ct}, f_{acs} = \emptyset, f_{tag})$

$\mathcal{U}$: Send $uid, sid, rid, repo^{ct}_{new}$ to $\mathcal{S}$

$\mathcal{U}$: add $repo^{pt}_{vl} = (\mathbf{f}^{pt}, f^{pt}_{acs} = \emptyset) \rightarrow repo^{pt}_{new}$

$\mathcal{S}$: $req : st_S.\{usr, ses, Rid\}$

$\mathcal{S}$: **if** $ses[sid] \neq uid$ or $rid \in Rid, Fail$

$\mathcal{S}$: Add $repo^{ct} \leftarrow repo^{ct}_{new}$

$< \mathcal{U}_{update}(st_U, km, rid, repo_{old}, \mathbf{f}^{pt}_{new}), \mathcal{S}_{update}(st_S) >$

$\mathcal{U}$: $req : st_U.\{uid, sid\}$

$\mathcal{U}$: $repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$ / last committed local repositories

$\mathcal{U}$: parse $repo^{pt}_{vl} = (\mathbf{f}^{pt}_{ol}, f^{pt}_{acs}) \in repo^{pt}_{old}$

$\mathcal{U}$: parse $repo^{ct}_{vl} = (\mathbf{f}^{ct}_{ol}, f_{acs}, f_{tag}) \in repo^{ct}_{old}$ and $rid = uid_o \| nonce$

$\mathcal{U}$: **if** $uid = uid_o$ $k \leftarrow KDF(mk, rid)$

$\mathcal{U}$: **else** $c_k \leftarrow f_{acs}.R[uid]$, $k \leftarrow \mathsf{PKE.Dec}(sk^{uid}_e, c_k)$

$\mathcal{U}$: **for** $fid \in \mathbf{f}^{pt}_{ol}.Fid \cap \mathbf{f}^{pt}_{new}.Fid$

$\mathcal{U}$: $f \leftarrow \mathbf{f}^{pt}_{ol}[fid]$, $f' \leftarrow \mathbf{f}^{pt}_{new}[fid]$, $ct_f \leftarrow \mathbf{f}^{ct}_{ol}[fid]$

$\mathcal{U}$: $\{O\}_z \leftarrow \mathsf{ComDiff}_{line}(f, f')$

$\mathcal{U}$: **for** $i \in [1, z]$

$\mathcal{U}$: $ct_l \leftarrow Enc(k, O_i.m)$, $O'_i = (O_i.op.O_i.idx, ct_l)$

$\mathcal{U}$: $ct'_f \leftarrow O'_z(\ldots(O'_1(ct_f)))$, $\{O\}_z \leftarrow \mathsf{ComDiff}_{char}(f, f')$

$\mathcal{U}$: $ct_o \leftarrow Enc(k, \{O\}_z)$, $ct'_f \leftarrow (ct_f, ct_o)$

$\mathcal{U}$: add $ct'_f \rightarrow \mathbf{f}^{ct}_{new}$, $\mathbf{f}^{pt}_{new}[fid] \rightarrow \mathbf{f}^{pt}_{new}$

$\mathcal{U}$: **for** $fid \in \mathbf{f}^{pt}_{new}.Fid \backslash \mathbf{f}^{pt}_{ol}.Fid$

$\mathcal{U}$: $f' \leftarrow \mathbf{f}^{pt}_{new}[fid]$

$\mathcal{U}$: **for** $j \in [1, f'.l]$ / $f'.l$ is # of lines in f'

$\mathcal{U}$: $lct_j \leftarrow Enc(k, l_j)$ / encrypt by line

$\mathcal{U}$: $ct'_f \leftarrow (lct_1, \ldots, lct_{f'.l})$ $ct'_f \leftarrow Enc(k, f')$

$\mathcal{U}$: add $ct'_f \rightarrow \mathbf{f}^{ct}_{new}$, $\mathbf{f}^{pt}_{new}[fid] \rightarrow \mathbf{f}^{pt}_{new}$

$\mathcal{U}$: **for** $fid \in \mathbf{f}^{ct}_{ol}.Fid \backslash \mathbf{f}^{pt}_{new}.Fid$

$\mathcal{U}$: add $\mathbf{f}^{ct}_{ol}[fid] \rightarrow \mathbf{f}^{ct}_{new}$, $\mathbf{f}^{pt}_{ol}[fid] \rightarrow \mathbf{f}^{pt}_{new}$

$\mathcal{U}$: $h' \leftarrow \mathsf{MerkleDAG}(\mathbf{f}^{ct}_{new}, f_{acs})$ / each ct'_f and f_{acs} are leaf nodes

$\mathcal{U}$: $\sigma' \leftarrow \mathsf{Sign}(sk_s, rid \| uid \| h')$, update file $f'_{tag} \leftarrow (uid, \sigma')$

$\mathcal{U}$: $repo^{ct}_{new} \leftarrow (\mathbf{f}^{ct}_{new}, f_{acs}, f'_{tag})$, $repo^{pt}_{new} \leftarrow (\mathbf{f}^{pt}_{new}, f^{pt}_{acs})$

$\mathcal{U}$: Send $uid, sid, rid, repo^{ct}_{new}$ to $\mathcal{S}$

$\mathcal{S}$: $req : st_S.\{usr, ses, Rid, repo^{ct}\}$

$\mathcal{S}$: Update $repo^{ct} \leftarrow repo^{ct}_{new}$

$< \mathcal{U}_{pull}(st_U, km, rid, repo_{old}), \mathcal{S}_{pull}(st_S) >$

$\mathcal{U}$: $req : st_U.\{uid, sid\}$, send $uid, sid, rid, \mathbf{v}_{old}$ to $\mathcal{S}$

$\mathcal{S}$: $req : st_S.\{usr, ses, Rid, repo^{ct}_{new}\}$. **if** $ses[sid] \neq uid$, $Fail$

$\mathcal{S}$: **for** $vi \in repo^{ct}_{new}.\mathbf{v} \backslash \mathbf{v}_{old}$

$\mathcal{S}$: Send $repo^{ct}_{vi} = (\mathbf{f}^{ct}, f_{acs}, f_{tag})$ to $\mathcal{U}$

$\mathcal{U}$: parse $rid = uid_o \| nonce$, $repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$

$\mathcal{U}$: **if** $uid = uid_o$ $k \leftarrow KDF(mk, rid)$

$\mathcal{U}$: **else** $c_k \leftarrow f_{acs}.R[uid]$, $k \leftarrow \mathsf{PKE.Dec}(sk^{uid}_e, c_k)$

$\mathcal{U}$: set $repo^{pt}_{new} \leftarrow repo^{pt}_{old}$, $repo^{ct}_{new} \leftarrow repo^{ct}_{old}$

$\mathcal{U}$: **for** each $repo^{ct}_{vi} = (\mathbf{f}^{ct}, f_{acs}, f_{tag})$ sent from $\mathcal{S}$

$\mathcal{U}$: parse $f_{acs} = (R, W, \sigma_{acs})$ and $f_{tag} = (uid_w, \sigma)$

$\mathcal{U}$: $rh \leftarrow \mathsf{MerkleDAG}(\mathbf{f}^{ct}, f_{acs})$, $r \leftarrow Hash(rid \| uid_w \| rh)$

$\mathcal{U}$: **if** $\neg \mathsf{Vrfy}(pk^{uid_o}_s, f_{acs}) \vee uid_w \notin f_{acs}.W \vee$

$\mathcal{U}$: $\neg \mathsf{Vrfy}(pk^{uid_w}_s, r, \sigma)$ **then** $Fail$

$\mathcal{U}$: **for** each $ct_i \in \mathbf{f}^{ct}$

$\mathcal{U}$: **for** $j \in [1, ct_i.l]$: $l_j \leftarrow Dec(k, l_j)$

$\mathcal{U}$: $f_i \leftarrow (l_1, \ldots, l_{ct_i.l})$

$\mathcal{U}$: parse $ct_i = (ct_{i_0}, ct_{o_1} \ldots ct_{o_z})$ / z is # of update in ct_i

$\mathcal{U}$: $f_{i_0} \leftarrow Dec(k, ct_{i_0})$

$\mathcal{U}$: **for** $j \in [1, z]$ $O_j \leftarrow Dec(k, ct_{o_j})$

$\mathcal{U}$: $f_i \leftarrow O_z(\ldots(O_1(f_{i_0})))$

$\mathcal{U}$: add $f_i \rightarrow \mathbf{f}^{pt}$

$\mathcal{U}$: add $repo^{pt}_{new} \leftarrow repo^{pt}_{vi} = (\mathbf{f}^{pt}, f_{acs}), repo^{ct}_{new} \leftarrow repo^{ct}_{vi}$

$\mathcal{U}$: get $repo_{new} \leftarrow (repo^{ct}_{new}, repo^{ct}_{new})$

$< \mathcal{U}_{share_I}(st_U, km, rid, uid_{re}, acs, repo_{old}), \mathcal{S}_{share_I}(st_S) >$

$\mathcal{U}$: $req : st_U.\{uid, sid\}$, parse $repo_{old} = (repo^{pt}_{old}, repo^{ct}_{old})$

$\mathcal{U}$: parse $repo^{ct}_{vl} = (\mathbf{f}^{ct}, f_{acs}, f_{tag}) \in repo^{ct}_{old}$, $rid = (uid_o, nonce)$

$\mathcal{U}$: **if** $uid \neq uid_o, Fail$

$\mathcal{U}$: $k \leftarrow \mathsf{KDF}(mk, rid)$, $ct_{shr} \leftarrow \mathsf{PKE.Enc}(pk^{uid_{re}}_e, uid \| k)$

$\mathcal{U}$: $f'_{acs}.R \leftarrow f_{acs}.R \cup \{(uid_{re}, ct_{shr})\}$

$\mathcal{U}$: **if** $acs =$ write **then** $f'_{acs}.W \leftarrow f_{acs}.W \cup \{uid_{re}\}$

$\mathcal{U}$: $f'_{acs}.\sigma \leftarrow \mathsf{Sign}(sk_s, f_{acs}.W \| f_{acs}.R)$

$\mathcal{U}$: $f'_{acs} \leftarrow (f'_{acs}.R, f'_{acs}.W, f'_{acs}.\sigma)$

$\mathcal{U}$: $rh' \leftarrow \mathsf{MerkleDAG}(\mathbf{f}^{ct}, f'_{acs})$, $h' \leftarrow Hash(rid \| uid \| rh')$

$\mathcal{U}$: $\sigma' \leftarrow \mathsf{Sign}(sk^{uid}_s, h')$, $f'_{tag} = (uid, \sigma')$

$\mathcal{U}$: $repo^{ct}_{new} = repo^{ct}_{old} \cup \{(\mathbf{f}^{ct}, f'_{acs}, f'_{tag})\}$

$\mathcal{U}$: Send $uid, sid, rid, uid_{re}, acs, f'_{acs}, f'_{tag}$ to $\mathcal{S}$

$\mathcal{S}$: $req : st_S.\{usr, ses, repo^{ct}, shr\}$

$\mathcal{S}$: $oob \leftarrow\!\!\$$, add $shr[rid] \leftarrow (uid_{re}, oob, acs)$

$\mathcal{S}$: add new version $(\mathbf{f}^{ct}, f'_{acs}, f'_{tag})$ to $repo^{ct}$, send oob to $\mathcal{U}$

$< \mathcal{U}_{share_{II}}(st_U, rid, oob), \mathcal{S}_{share_{II}}(st_S) >$

$\mathcal{U}$: $req : st_U.\{uid, sid\}$, send uid, sid, rid, oob to $\mathcal{S}$

$\mathcal{S}$: $req : st_S.\{usr, ses, shr, A\}$

$\mathcal{S}$: **if** $ses[sid] \neq uid$ or $(uid, oob, acs) \notin shr[rid], Fail$

$\mathcal{S}$: add $(uid, acs) \rightarrow A[rid]$.

FIGURE 4.8. The constructions of SGit, where boxed purple part with solid line belongs to SGitLine, and boxed teal part with dashed line belongs to SGitChar.

on secure local storage and improve the portability, we propose a solution to integrate password-based key management into E2EE Git.

Currently, users have three secrets: password, private key, and secret key. Users use a password to authenticate to the Git server, use the private key to authenticate to other users, and use the secret key to derive a data encryption key for data encryption. Note that among the three secrets, only the password is easy for users to memorize and bring everywhere, and thus we consider using password-based key management to improve the portability. However, we know that a password has low entropy and is vulnerable to dictionary attacks when the Git server gets compromised or the server storage gets breached.

We utilize End-to-Same-End encryption (E2SE for short) design idea, where another server is introduced to increase the user's entropy for each server. We integrate E2SE into E2EE Git by introducing a new server acting as a key server and letting the GitHub server act as a storage server, so that users use passwords to authenticate to two servers and derive a master key for encrypting/decrypting the private key and secret key.

Further optimizations. For SGitChar, a single version of the ciphertext file includes the initial version of the ciphertext and a sequence of updates. It is efficient in terms of repository storage cost, update communication size, and encryption time, only related to the size of the difference. But in one case that client does not have the repository locally and only wants to fetch a single latest version of the repository, the communication cost and decryption time are linear to the versions of the repository, as the latest version includes all the previous updates. SGitLine does not have such an issue due to each ciphertext version is history independent. To mitigate the special case cost of SGitChar due to history dependence, we can have further optimization by setting a length of history dependence, e.g., 6 versions. Concretely, for every six updates of a file, users treat the file as the first version to directly encrypt an entire version from scratch, which is independent of the history. Even if users do not have any local repository and only fetch a specified version, the communication cost at most includes five updates, and the decryption overhead is at most linear to five update differences.

4.4.4 Security analysis

4.4.4.1 Data confidentiality of SGitChar

We give a formal proof of Thm.4.4.1 that SGitChar satisfies the data confidentiality defined in Def. 4.3.1 in the following.

THEOREM 4.4.1 (Data confidentiality). Let $\Pi_{SE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA secure symmetric encryption, $\Pi_{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA public-key encryption, $\Pi_{Sig} = (\mathsf{KG}, \mathsf{sign}, \mathsf{Vrfy})$ be a strongly existentially unforgeable digital signature, KDF be a random oracle. Let `MerkleDAG` be a directed acyclic graph structured collision-resistant hash function. SGitChar has data confidentiality, i.e.,

$$\mathrm{Adv}^{\mathtt{CONF}}_{\mathsf{SGitChar},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{CONF}}_{\mathsf{SGitChar},\mathcal{A},q} = 1] - 1/2 = negl(\lambda).$$

PROOF. In a high level, we use game hop to reduce the data confidentiality to the security of underlying schemes. When playing a game with SGitLine adversary $\mathcal{A}$, challenger $\mathcal{B}$ could act as an adversary of underlying schemes and interact with their respective challenger $\mathcal{C}$ to answer $\mathcal{A}$'s queries. If $\mathcal{A}$ can distinguish, $\mathcal{B}$ can leverage it to break underlying schemes. We use four game hops to deal with the decryption queries of $\mathcal{O}_{pull}$, the sharing queries of $\mathcal{O}_{share_I}$, the encryption queries of $\mathcal{O}_{init}$, and the update queries of $\mathcal{O}_{upd}$, gradually reducing the data confidentiality to the repository unforgeability of SGitChar proved in Theorem 4.4.2, the IND-CPA security of Π_{PKE}, the pseudorandomness of KDF, and the IND-CPA security of Π_{SE}.

Specifically, Game 1 ignores all decryption queries of $\mathcal{O}_{pull}$ that involve malformed ciphertexts. The adversary $\mathcal{A}$ cannot distinguish this change due to the unforgeability of SGitChar, which is further reduced to the security of Π_{Sig} and `MerkleDAG`. Game 2 replaces the key materials shared with honest users with random values, and encrypts these values when responding to $\mathcal{O}_{share_I}$ queries. This modification is indistinguishable from using real key materials due to the IND-CPA security of Π_{PKE}. In Game 3, the derivation of the repository encryption key is replaced with a random value in $\mathcal{O}_{init}$ queries. This change is hidden from $\mathcal{A}$ due to the pseudorandomness of the KDF output when given a random input. For challenge related

queries, Game 4 converts into challenges to the challenger in the IND-CPA security game of Π_{SE}, and uses the challenge responses as the corresponding ciphertexts to proceed, which is indistinguishable due to the IND-CPA security of Π_{SE}.

The details are as follows:

- Game 0: $\mathcal{B}$ acts the same as challenger in SGitLine confidentiality game.
- Game 1: we deal with the decryption oracle. For those pull queries to $\mathcal{O}_{pull}$ on those repositories which only honest users have access to, $\mathcal{B}$ refuses to respond if the queried ciphertext repository is not the previous queried one. $\mathcal{A}$ can distinguish Game 1 from Game 0 with negligible probability due to the repository unforgeability of SGitLine.
- Game 2: $\mathcal{B}$ first replaces all key materials shared with honest users with encryption on a random key, which is indistinguishable from Game 1 due to the IND-CPA security of PKE. When queried to the $\mathcal{O}_{pull}$ oracle with shared repository on shared honest user, $\mathcal{B}$ just looks up the table to find the real data encryption key instead of decryption to get the key. This ensures that the shared key material with honest users does not leak any information about the real data encryption key.
- Game 3: for each initialization query to $\mathcal{O}_{init}$ on honest users, $\mathcal{B}$ replaces repository encryption key with a random value. Even if the malicious user is shared by the honest user, the random value is indistinguishable from the correct output of key deviation function on honest user's master key mk and repository id rid as input. So that the honest user's mk is never leaked to corrupted users. Due to Game 2, the key material of the repository with only honest users is never leaked to $\mathcal{A}$.
- Game 4: given the two files as the challenge, if it is an initialization query, $\mathcal{B}$ directly forwards the challenge to the challenger of symmetric encryption $\mathcal{C}_{\Pi_{SE}}$. Later for the update queries on the challenge repository, $\mathcal{B}$ comparing the differences between update file f and two challenges by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{char}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{char}(f_1, f)$. $\mathcal{B}$ checks the validity of $\{O_0.m\}$ and $\{O_1.m\}$ based on different conditions for SGitChar and SGitLine. If the challenge is valid without trivial win, $\mathcal{B}$ forwards $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $\mathcal{C}_{\Pi_{SE}}$. If the challenge query is an

update query, $\mathcal{B}$ first calculates two sets of challenge modifications in terms of their prior file f by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{char}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{char}(f_1, f)$. Then $\mathcal{B}$ submits the different modification messages $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $\mathcal{C}_{\Pi_{SE}}$. $\mathcal{B}$ follows the same way to deal with later update queries on the challenge repository. Finally, $\mathcal{B}$ forwards $\mathcal{A}$'s guess to $\mathcal{C}_{\Pi_{SE}}$. If $\mathcal{A}$ has a non-negligible probability to distinguish the two challenges, then $\mathcal{B}$ can break Π_{SE}'s IND-CPA security.

As a result, SGitChar has data confidentiality. □

4.4.4.2 Repository unforgeability of SGit

We formally prove the Thm. 4.4.2 that two SGit constructions SGitLine and SGitChar satisfy the repository unforgeability defined in Def. 4.3.3.

THEOREM 4.4.2 (Repository unforgeability). Let $\Pi_{Sig} = (\mathsf{KG}, \mathsf{Sign}, \mathsf{Vrfy})$ be a strongly unforgeable digital signature scheme and `MerkleDAG` be a directed acyclic graph structured collision-resistant hash function. SGitLine and SGitChar have repository unforgeability, i.e., $\mathrm{Adv}^{\mathtt{UNF}}_{\mathsf{SGitLine},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{UNF}}_{\mathsf{SGitLine},\mathcal{A},q} = 1] = negl(\lambda)$ and $\mathrm{Adv}^{\mathtt{UNF}}_{\mathsf{SGitChar},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{UNF}}_{\mathsf{SGitChar},\mathcal{A},q} = 1] = negl(\lambda)$.

PROOF. Intuitively, we begin by assuming that $\mathcal{A}$ produces a successful forgery, which must fall into one of three possible cases: a new signature, a new hash root, or new repository contents. However, each case contradicts the original assumptions on the underlying building blocks—specifically, the strong unforgeability of Π_{Sig} and the collision resistance of `MerkleDAG`. Therefore, $\mathcal{A}$ cannot produce a successful forgery, and SGitChar satisfies repository unforgeability.

Since SGitLine and SGitChar use the same components to ensure unforgeability, namely, applying a structured hash function to an entire repository version and signing the resulting hash root, the proof of repository unforgeability applies equally to both constructions.

Concretely, a valid forgery $repo*_{ct} = repo^h_{ct}, f_{acs}, f_{tag}$ on target user uid^* and repository rid^* contains three parts where σ is signature on message $rid^*\|uid^*\|h$, $f_{tag} = (uid^*, \sigma)$, and $h = \texttt{MerkleDAG}(repo^h_{ct}, f_{acs})$. The forgery is a new message signature pair. So there are two cases.

- Case 1: The signature σ is new.
- Case 2: The signature is the previous one, but the message $rid^*\|uid^*\|h$ is new.
- Case 3: Both signature and message are previous ones, but $(repo^h_{ct}, f_{acs})$ is new.

Case 1 and Case 2 mean that $\mathcal{A}$ forges a new message signature pair, which is contradictory with the strong unforgeability of digital signatures. For case 3, since the message is old, the h is also the previous one. But $(repo^h_{ct}, f_{acs})$ is new. Previously, there exists one $(repo'^h_{ct}, f'_{acs})$ such that $h = \texttt{MerkleDAG}(repo'^h_{ct}, f'_{acs})$. We know that $h = \texttt{MerkleDAG}(repo^h_{ct}, f_{acs})$. So there is a collision which contradicts with the collision resistent property of MerkleDAG. As a result, SGitLine has unforgeability. □

4.4.4.3 Weak data confidentiality of SGitLine

We prove that the SGitLine construction satisfies weak data confidentiality defined in Def. 4.3.2.

THEOREM 4.4.3 (Weak data confidentiality). Let $\Pi_{SE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA secure symmetric encryption, $\Pi_{PKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA public-key encryption, $\Pi_{Sig} = (\mathsf{KG}, \mathsf{sign}, \mathsf{Vrfy})$ be an strong existing unforgeable digital signature, KDF be a random oracle. Let MerkleDAG be a directed acyclic graph structured collision resistant hash function. SGitLine has weak data confidentiality, i.e.,

$$\mathrm{Adv}^{\mathtt{CONF_w}}_{\mathsf{SGitLine},\mathcal{A},q}(\mathcal{A}) = \Pr[G^{\mathtt{CONF_w}}_{\mathsf{SGitLine},\mathcal{A},q} = 1] - 1/2 = negl(\lambda)$$

PROOF. Intuitively, we use game hop to reduce the data confidentiality to the security of the underlying schemes. When playing a game with SGitLine adversary $\mathcal{A}$, challenger $\mathcal{B}$ could act as adversary of underlying schemes and interact with their respective challenger $\mathcal{C}$ to answer $\mathcal{A}$'s queries. If $\mathcal{A}$ can distinguish, $\mathcal{B}$ can leverage it to break underlying schemes.

We use four game hops dealing with decryption query of $\mathcal{O}_{pull}$, sharing query of $\mathcal{O}_{share_I}$, encryption queries of $\mathcal{O}_{init}$, and update queries of $\mathcal{O}_{upd}$ to reduce the security. Compared with the proof of SGitChar, the only difference of the weak confidentiality proof of SGitLine is in Game 4 for dealing with challenges to the $\mathcal{O}_{init}$ or $\mathcal{O}_{upd}$ oracles and the following update queries to the $\mathcal{O}_{upd}$ oracle with one more restriction on the challenge update position. The details are as follows:

- Game 0, $\mathcal{B}$ acts the same as challenger in SGitLine confidentiality game.
- Game 1, we deal with the decryption oracle. For those pull queries to $\mathcal{O}_{pull}$ on those repositories which only honest users have access to, $\mathcal{B}$ refuses to respond if the queried ciphertext repository is not previously queried one. $\mathcal{A}$ can distinguish Game 1 from Game 0 with negligible probability due to the repository unforgeability of SGitLine.
- Game 2, $\mathcal{B}$ first replaces all key materials shared with honest users with encryption on a random key, which is indistinguishable from Game 1 due to the IND-CPA security of PKE. When queried to the $\mathcal{O}_{pull}$ oracle with shared repository on shared honest user, $\mathcal{B}$ just looks up the table to find the real data encryption key instead of decryption to get the key. This ensures that the shared key material with honest users does not leak any information about the real data encryption key.
- Game 3, for each initialization query to $\mathcal{O}_{init}$ on honest users, $\mathcal{B}$ replaces repository encryption key with a random value. Even if a malicious user is shared by the honest user, the random value is indistinguishable from the correct output of the key deviation function on the honest user's master key mk and repository id rid as input. So that the honest user's mk is never leaked to corrupted users. Due to Game 2, the key material of the repository with only honest users is never leaked to $\mathcal{A}$.
- Game 4, given the challenge two files, if it is an initialization query, $\mathcal{B}$ directly forwards the challenge to the challenger of symmetric encryption $\mathcal{C}_{\Pi_{SE}}$. Later for the update queries on the challenge repository, $\mathcal{B}$ comparing the differences between update file f and two challenges by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{line}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{line}(f_1, f)$. $\mathcal{B}$ checks the validity of $\{O_0.m\}$ and $\{O_1.m\}$ based on the

trivial condition for SGitLine including the restriction of the consistent update position. If the challenge is valid without trivial wins, $\mathcal{B}$ forwards $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $\mathcal{C}_{\Pi_{SE}}$. If the challenge query is an update query, $\mathcal{B}$ first calculates two sets of challenge modifications in terms of their prior file f by running $\{O_0\} \leftarrow \mathsf{ComDiff}_{line}(f_0, f), \{O_1\} \leftarrow \mathsf{ComDiff}_{line}(f_1, f)$. Then, $\mathcal{B}$ submits the different modification messages $\{O_0.m\}$ and $\{O_1.m\}$ as challenge to $\mathcal{C}_{\Pi_{SE}}$. $\mathcal{B}$ follows the same way to deal with later update queries on the challenge repository. Finally, $\mathcal{B}$ forwards $\mathcal{A}$'s guess to $\mathcal{C}_{\Pi_{SE}}$. If $\mathcal{A}$ has a non-negligible probability to distinguish the two challenges, then $\mathcal{B}$ can break the IND-CPA security of Π_{SE}.

As a result, SGitLine has weak data confidentiality. □

4.5 Implementation and Evaluation

We implemented our two schemes and presented the experimental results in this section.

Implementation. We implemented both SGitLine and SGitChar using Python and the pycryptodome library and used AES-CTR as the encryption algorithm, ECDSA as a signature scheme, SHA-256 as the hash function, and HKDF-SHA-256 as the key derivation function. We will open-source it soon. For a fair comparison, we re-implemented the deterministic encryption-based scheme adopted by Git-crypt [94] using Python, where AES-CTR encrypts each file with an initialization vector (IV) derived from the SHA-1 HMAC of the file. We also implemented Trivial-enc-sign, where the whole updated version of a repository would be re-encrypted before pushing.

For SGitLine, we utilize `git diff` to obtain the line-wise difference for each update. For SGitChar, we utilize the diff_match_patch package [116] to obtain the character-wise delta. To record the user's signature in the current commit, we take the following steps: 1) run the `git commit` command, 2) sign on the commit information including the parent commit hash value, the `MerkleDAG` hash value of the current commit, the author, timestamps, and the commit message; 3) use the `git --amend` command to update the commit message with

the signature. In this way, a user can obtain the updated ciphertext and the signature from one commit and then verify them. Git uses SHA-1 by default, while due to the insecurity of SHA-1, we recommend utilizing SHA-256 as the default hash function.

Experiments. The local Git repositories were hosted on a Windows laptop with an Intel Core i7 processor (2.1 GHz) and 32 GB RAM. We also carried out the experiments on Amazon Web Service (AWS for short), where the repositories are hosted on an AWS virtual machine with Ubuntu (64-bit), 1 vCPU, and 30 GB of disk storage. We used Git tools and the GitHub API to interact with GitHub, deploying a remote repository on the GitHub server. To compare our schemes with the other two in typical scenarios, we selected five of the top ten rated code repositories on GitHub, considering the variety in their scale and number of files. The selected repositories are awesome [117], free-programming-books (FPB) [118], bootstrap [119], react [120], and freeCodeCamp (FCC) [121]. Additionally, we included a paper repository (denoted as DecRepo), which mainly contains LaTeX files of an academic manuscript and has a different structure and pattern compared to conventional code repositories. The specific information is provided in Table 4.2.

TABLE 4.2. The repository information (as of April 2024). The .git folder includes all history versions. The size including the .git folder is the size of all versions, excluding the .git folder is the size of the current version.

Repository	size (MB)		# of files	# of lines
	include .git	exclude .git		
awesome [117]	2.1	0.37	25	2560
FPB [118]	23.2	2.5	217	30690
bootstrap [119]	295.2	20.3	755	174764
react [120]	474.2	30.5	2598	655335
FCC [121]	934.2	451.3	75438	11033103
DocRepo	3.7	2	67	18301

We evaluated the four schemes on these six repositories, comparing their performance in terms of communication, computation, end-to-end time, and local storage costs. In the initialization, the first ciphertext version of a repository is generated locally and pushed to the GitHub server. Regarding one version update, we randomly select ten commits from each repository and calculate the average computation costs for updating the ciphertext, as well as the average communication costs for pushing it. We also utilize the same commits to test the recovery costs, supposing that the client has the original version of a commit, another collaborator

makes a new commit to GitHub, and the client needs to update the local repository. We also test the average end-to-end time of the randomly selected ten commits, including local computation delay and communication delay of pushing to the GitHub server. For storage costs, we utilize the first commit of each repository as the initial version and record the storage costs after 10, 20, 30, 40, and 50 commits. The detailed description is provided as follows.

In the initialization phase, we measure the computation costs of running the initialization algorithm on the first version of a repository and the communication costs of pushing the ciphertext. To evaluate the update costs, we randomly select a commit from the repository, with each commit corresponding to two versions: the original and the updated version. We first run the initialization algorithm on the original version, then apply the update algorithm to generate the ciphertext for the updated version, and finally push the ciphertext to the GitHub server. To ensure generality, we randomly select ten commits from each repository and calculate the average computation costs for updating the ciphertext, as well as the average communication costs for pushing it. We also evaluate the end-to-end delay of one update. For Git, the end-to-end delay includes the time of pushing the updated version to the GitHub server. For SGitChar and SGitLine, it contains the time of delta computation, encryption, signing, and pushing, and the end-to-end time of Trivial-enc-sign contains the same components except for delta computation. For Git-crypt, it only includes the delay of encrypting the modified files and pushing.

Regarding recovery, we use the same ten commits to test the communication costs of pulling data. Specifically, we assume that the client has the original version of a commit, and another collaborator makes a new commit to GitHub. We then measure the communication costs of pulling the updated version from the GitHub server and the computation costs of recovering the updated version. We measure the costs of storing each encrypted repository using the four schemes and the costs of storing the plaintext repository using plain Git. We utilize the first commit of each repository as the initial version and record the storage costs after 10, 20, 30, 40, and 50 commits. We run the `git gc` command to pack the objects that have been generated after we commit a new version. This command allows us only to store the initial version and the delta generated by a new commit.

Evaluation summary. The communication costs of the four schemes are shown in Table 4.3. Regarding updates, both SGitLine and SGitChar perform much better than the other two schemes, especially for a large repository with minor updates. SGitChar takes fewer communication costs than SGitLine, except for some special cases. In terms of initialization, SGitChar achieves comparable performance to Git-crypt and Trivial-enc-sign.

TABLE 4.3. The communication costs of each operation on six repositories using different schemes.

Repository	Initialization (KB)					One Version Update/Recovery (KB)				
	Git	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign	Git	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign
awesome [117]	0.54	1.06	1.36	1.06	1.06	0.33	0.48	0.44	38.17	0.21 MB
FPB [118]	0.94	1.58	1.99	1.58	1.58	0.41	0.69	0.58	19.62	0.68 MB
bootstrap [119]	253.42	400.50	471.57	400.50	400.50	0.80	4.51	3.31	122.78	2.52 MB
react [120]	620.61	984.00	984.00	984.00	984.00	1.92	8.77	10.20	49.81	23.82 MB
FCC [121]	1.04	1.69	2.30	1.69	1.70	2.39	11.49	11.98	120.61	59.57 MB
DocRepo	445.93	729.38	847.20	719.89	729.45	2.23	10.41	11.01	74.50	0.83 MB

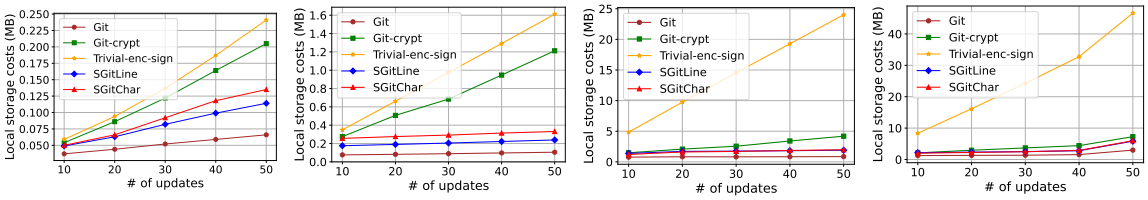
The computation costs of updates are provided in Table 4.4. Our schemes SGitChar and SGitLine generally perform better than Trivial-enc-sign, apart from a few special scenarios, such as updates throughout the entire repository. Regarding the end-to-end time cost shown in Figure 4.10, SGitLine consistently outperforms Trivial-enc-sign, as it may incur higher computation costs but achieves significantly lower communication overhead, resulting in overall greater efficiency.

TABLE 4.4. The computation overhead of updating six repositories under different schemes.

Repository	SGitChar (s)			SGitLine (s)			Git-crypt (s)	Trivial-enc-sign(s)
	Compare	Encrypt	Total	Compare	Enc+update	Total		
awesome [117]	0.0003	0.0001	0.0004	0.0277	0.0001	0.0278	0.0002	0.0008
FPB [118]	0.0003	0.0001	0.0004	0.0275	0.0001	0.0276	0.0001	0.0045
bootstrap [119]	0.1004	0.0001	0.1005	0.0287	0.0010	0.0297	0.0006	0.0229
react [120]	0.0888	0.0001	0.0889	0.0376	0.0010	0.0386	0.0003	0.1235
FCC [121]	0.0683	0.0002	0.0685	0.0340	0.0009	0.0349	0.0008	0.6045
DocRepo	0.3337	0.0002	0.3339	0.0336	0.0008	0.0344	0.0005	0.0033

In terms of storage cost, figure 4.9 shows that SGitChar and SGitLine take less storage costs compared with Git-crypt and Trivial-enc-sign as the number of updates increases, and SGitChar generally outperforms SGitLine, except for some special cases. The detailed analysis of special cases is provided later.

Communication costs. As the costs of initialization and recovery shown in Table 4.3, SGitChar, Git-crypt, and Trivial-enc-sign share similar costs, while SGitLine incurs a little



A Repo awesome [117] B Repo FPB [118] C Repo bootstrap [119] D Paper Repository

FIGURE 4.9. The costs of storing the repositories using different schemes.

bit more costs, since SGitLine needs to store a nonce for each line. We observe that the pull costs of recovery are almost as large as the push costs of updating a commit to the GitHub server, since the pull and push communication costs are all determined by the delta of the commit. For both push and pull operations, SGitLine and SGitChar have much fewer communication costs than Git-crypt and Trivial-enc-sign, especially for a large repository with minor updates. Our constructions are generally $2 \sim 3$ orders of magnitude more efficient than Trivial-enc-signin terms of update/recovery communication cost. In general, SGitChar spends fewer communication costs than SGitLine in the update phase, since the word-wise difference is usually shorter than the line-wise difference, except for some special cases (analyzed below). Particularly, the update costs of SGitChar are at most 5.6 times that of Git, which is acceptable given the strong security guarantees our scheme provides.

There are some special cases that SGitChar has slightly more communication than SGitLine, e.g., [117], [118], and [119]. The reason is besides recording the updated content itself, the character-wise delta needs to keep extra information, including the exact update position and update type (insert or delete). This extra information may cost more space than line-wise delta when a new line is inserted, a line is deleted, or multiple major modifications occur within one line.

TABLE 4.5. The computation costs of initializing and recovering six repositories under different schemes.

Repository	Initialization (s)				One version Recovery (s)			
	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign	SGitChar	SGitLine	Git-crypt	Trivial-enc-sign
awesome [117]	0.0001	0.0004	0.0001	0.0001	0.0001	0.0048	0.0001	0.0008
FPB [118]	0.0001	0.0005	0.0002	0.0001	0.0002	0.0029	0.0001	0.0058
bootstrap [119]	0.0021	0.0474	0.0033	0.0021	0.0013	0.0322	0.0003	0.0237
react [120]	0.0120	0.4917	0.0235	0.0150	0.0005	0.0140	0.0002	0.0976
FCC [121]	0.0001	0.0007	0.0002	0.0001	0.0021	0.0368	0.0005	1.328
DocRepo	0.0033	0.0717	0.0051	0.0032	0.0007	0.0074	0.0002	0.0030

Computation costs. The computation costs of the initialization and recovery phases are presented in Table 4.5. SGitChar performs as well as Trivial-enc-sign and outperforms Git-crypt and SGitLine in the initialization phase, since Git-crypt needs to compute SHA-1 HMAC from files and SGitLine needs to encrypt the file line-by-line, which costs much for files with many lines. Regarding recovery, SGitChar is more efficient than Trivial-enc-sign but slightly less efficient than Git-crypt, since SGitChar does not need to decrypt each entire file as Trivial-enc-sign, but has to decrypt the patch(es) and then apply them to the original content. We observe that there are two decryption methods of SGitLine. One is to directly decrypt files line by line. The other is to first compute the delta on the ciphertext repositories and then only decrypt the modified lines, as the client has the original version. We adopt the former because it does not need line-wise computation and is more efficient. Even with the more efficient method, SGitLine underperforms, as it has to decrypt files line by line, which is time-consuming.

The costs of the update phase are shown in Table 4.4. The costs of SGitChar include computing word-wise difference and encrypting it. The costs of SGitLine include obtaining line-wise difference using `git diff` and encrypting it as well as updating the ciphertext. SGitChar performs better than SGitLine when fewer modifications are made, where the costs for computing the difference and encrypting it in SGitChar are smaller than those of SGitLine, e.g., for repositories [117, 118]. When more modifications are made, i.e., computing word-wise difference costs much more than computing line-wise one, SGitLine performs better in [119, 120, 121] and DocRepo. Git-crypt outperforms, since it does not need to compute the difference. For the same reason, Trivial-enc-sign performs better than our two schemes for small repositories. As the size of the repository and the number of files increase, its advantage disappears.

The costs of generating and verifying an ECDSA signature are 0.93 ms and 0.69 ms, respectively. We directly obtain the MerkelDAG hash value from the output of `git commit`, and thus the costs of signing an update and verifying it are constant. Therefore, we omit these costs in Table 4.4 and 4.5.

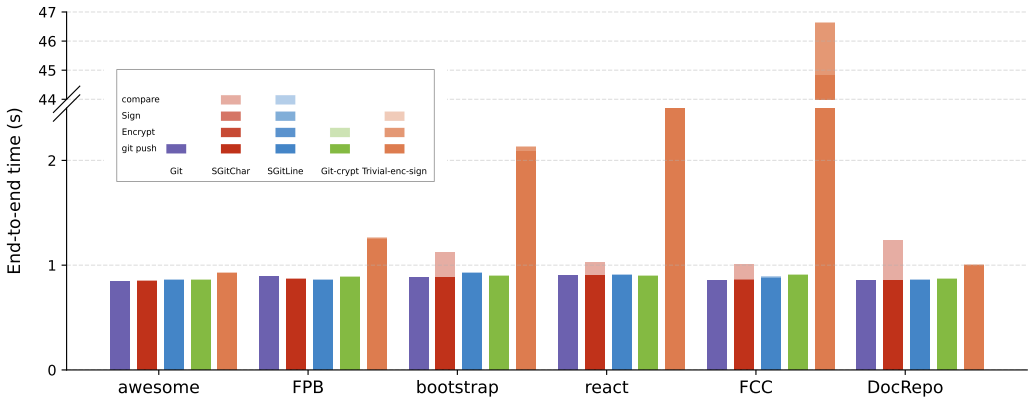
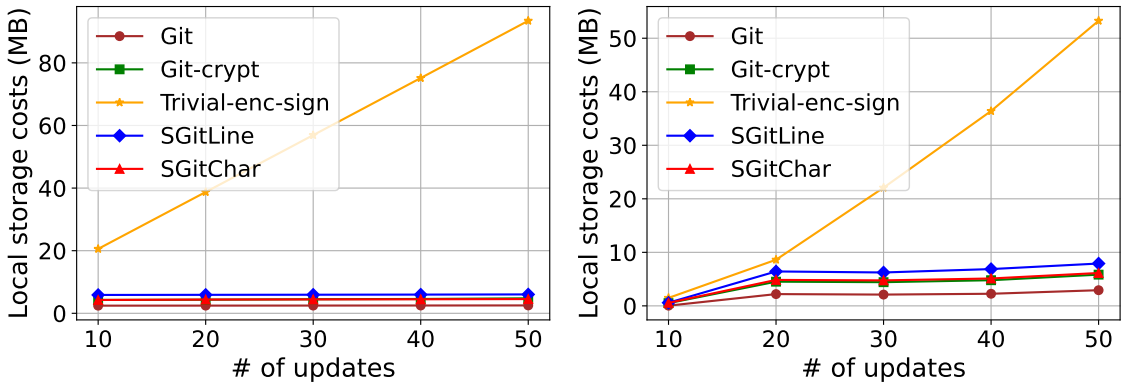


FIGURE 4.10. Average end-to-end client time cost of each version update and push to the server under different schemes

End-to-end delay. The experiments were conducted on an AWS virtual machine. We measured the round-trip time to the GitHub server using `ping github.com`, and the average latency was approximately 20 milliseconds. Figure 4.10 shows the end-to-end delay for each repository using different schemes. The end-to-end delay is primarily determined by communication delay, i.e., the time of running `git push`. The time spent on encryption (except for Trivial-enc-sign) and signing operations is negligible in comparison. The communication delay is primarily determined by the size of the transmitted data. According to Table 4.3, although Git-crypt incurs more communication costs compared to Git, SGitChar, and SGitLine, their communication costs remain at the KB-level. As a result, all schemes share similar communication delays in practice. Generally, SGitLine and SGitChar outperform Trivial-enc-sign, except for a special case, DocRepo.

For DocRepo, each commit is a large revision, and there are multiple changes to each `.tex` file, and computing the character-wise differences takes more time. Due to the small number of files and their small size, encrypting the whole version takes much less time than computing character-wise differences and then encrypting. Therefore, SGitChar needs more end-to-end time than Trivial-enc-sign, even though SGitChar takes less communication delay.

Storage costs. Figure 4.9 and Figure 4.11 show the storage costs of the six repositories using SGitLine and SGitChar, Git-crypt [94], and Trivial-enc-sign. Generally, SGitChar performs best among the four schemes. For example, as the storage costs of bootstrap [119] shown in



A Repo react [120] B Repo FCC [121]

FIGURE 4.11. The storage costs of repositories using different schemes.

Figure 4.9c, the costs of SGitChar and SGitLine are 1.98 MB and 2.06 MB after 50 commits, respectively, which are much smaller than 24.97 MB for Trivial-enc-sign and 4.19 MB for Git-crypt.

In general, SGitChar outperforms SGitLine, while there are two special cases, awesome [117] and FPB [118]. This is because modified lines have multiple changes, which would cause the character-wise delta to be larger than the length of the lines.

SGitChar and SGitLine take fewer local storage costs compared with Git-crypt and Trivial-enc-sign. However, there are two special cases for storing the repository react [120] and FCC [121] in Figure 4.11a and 4.11b, where SGitLine, SGitChar, and Git-crypt have similar storage costs since these tested versions mainly involve file-wise modifications.

For example, in FCC [121], we can see that Git-crypt takes fewer storage costs than SGitChar. This is because some updates removed a lot of text, which caused the size of the character-wise delta to be larger than that of the file. This result shows that when using SGitChar, if there is a significant version update, such as rewriting a large portion of the files or deleting most of the original content, the user can re-encrypt the updated repository instead of adding incremental ciphertext of the delta. We can also see that Git-crypt takes fewer costs than SGitLine. The reason is that some updates add many new files with lots of lines. For these files, the ciphertext generated by SGitLine is much larger than that generated by Git-crypt. Thus, the storage costs incurred by SGitLine are relatively higher than those of Git-crypt.

Further discussion. We notice that some files cannot be encrypted by line, e.g., images, and the character-wise delta computation method does not apply to these files. Therefore, when implementing SGitLine and SGitChar, we directly encrypt such files for initialization and re-encrypt them if they are modified. GitHub servers would check the format of files uploaded by users and may block the user who tries to upload files with the wrong format. Actually, encryption may destroy the file format, especially for images. To upload the ciphertext to GitHub servers, we use Base64, a binary-to-text encoding, to encode the ciphertext bytes into ASCII characters, since text files have no special format, which may enable the ciphertext to pass the format check. The drawback of this approach is that it results in a 30% increase in ciphertext size. Thus, how to more efficiently upload encrypted files needs further research.

Desynchronization problem due to the Git server's temporary unavailability. So far, end-to-end encrypted Git services do not introduce additional issues beyond those faced by existing plain Git services when the Git server is temporarily unavailable. If multiple collaborators update different files in the same repository, merging remains straightforward—just as in plain Git. However, if two users edit the same file, once the Git server becomes available again, only one user can push their changes normally. Others who modified the same file must first pull the latest version from the server and resolve the conflict locally.

In such cases, the user must pull the remote version, recover the corresponding plaintext, compare it with their recent local changes to identify the conflict, decide which modifications to keep, and then re-initiate the secure push process. This involves redoing the comparison between the local plaintext version and the newly pulled remote version, re-encrypting the differences, re-signing the updated file to produce a complete ciphertext, and finally pushing the result.

Currently, resolving merge conflicts directly on the Git platform (i.e., remotely) is not supported by end-to-end encrypted Git services, since the plaintext data is not visible on the server side. Supporting conflict resolution directly over ciphertext remains an open—but highly interesting and important—problem.

4.6 Conclusion and Open Problems

This work is the first formal systematic investigation of end to end encrypted Git services. We formalize security properties including confidentiality and integrity to capture real-world vulnerabilities of Git. Moreover, our proposed secure designs are compatible with existing Git servers, making it easy to be augmented. There are many interesting questions to be explored by the community.

Security-wise, our security models capture both the privacy considerations and software supply-chain security. The latter remains underexplored and could be used as a lens to analyze actual security or real-world attacks of products that claim to offer end-to-end security in multi-user setting. On the down side, our security models currently consider only static corruption. This is similar to relevant recent formal studies of E2E security in cloud storage (e.g., [19]), and messaging (e.g., [102]). A natural question is to extend our models to handle adaptive corruption in the multi-user setting. Moreover, achieving stronger metadata security, such as hiding users' access patterns and edit behaviors (particularly in systems like Git), remains an interesting and important open problem.

Functionality-wise, our current design focuses on the most critical operations of Git. Many advanced Git features remain unexplored and could be valuable directions for future work. Furthermore, it is important to also investigate how to support more flexible cryptographic group management (which defines access control policies), as well as features such as key rotation, revocation, accountability, and secure integration with web-based Git interfaces.

CHAPTER 5

CCA Updatable Encryption Against Malicious Re-encryption Attacks

---

5.1 Introduction

Increasingly number of companies, government bodies and personal users choose to store their data on the cloud instead of their local devices. As a public infrastructure, frequent data breaches from the cloud were reported. One potential mitigation is to let the user to upload encrypted data and keep the decryption key locally. However, even if these data are protected by encryption mechanisms, there are still risks that the users' decryption keys get compromised, especially after the key has been in use for a while. It is widely acknowledged (and implemented in industry) that a wiser strategy is to let the user periodically refresh the secret key which is used to protect the data (and update the corresponding ciphertext in the cloud). For instance, the Payment Card Industry Data Security Standard (PCI DSS) [122, 123] requires that the credit card data must be stored in encrypted form and mandates key rotation, i.e., encrypted data is regularly refreshed from an old to a newly generated key. The similar strategy has also been adopted by many cloud providers, such as Google and Amazon [25].

Though we have many standardized encryption tools to use, facilitating key rotation requires care. A naive solution is to let the client download all encrypted data, decrypt, choose a new key, encrypt the data, and upload the new ciphertext to the cloud server. This is obviously too inefficient (e.g., large communication for big data) to be useful. To efficiently and securely execute the key rotation, Boneh et al. [21] proposed a new primitive called updatable encryption (UE) for efficiently updating ciphertexts with a new key. In such a scheme, a client only needs to retrieve a very *short* piece (called header) of information, and

generates a *short* update token that allows the server to re-encrypt the data himself from existing ciphertext, while preserving the security of the encryption. Everspaugh et al [25] gave a systematic study of UE, especially on the key rotation on *authenticated encryption*, which is the standard practice for encryption. The seemingly paradoxical feature of modifying ciphertext while maintaining integrity is both necessary and conceptually intriguing; more importantly, integrity is as indispensable as confidentiality in secure storage. Very recently, Boneh et al [124] proposed strengthening on confidentiality and improved the efficiency of [25].

The security of updatable encryption in a nutshell. The security models of updatable encryption mimic those of authenticated encryption (AE) to capture both the data confidentiality and integrity. But a critical difference is that UE wishes to capture the survivability of the system after the server is briefly breached or the client is temporarily hacked. To characterize these attack scenarios, the adversary in the UE model is allowed to view the secret keys in the previous epochs and the current version of the continuously updating ciphertext. And also, other related information generated during the key rotations, such as the update tokens, headers, will also be leaked to the adversary. The only restriction is to rule out the trivial impossibility that the secret key and the ciphertext are both obtained by the attacker simultaneously. Since adversary's strategy could be very diverse, clearly defining the boundary so that the strategies leading to trivial break of the system are disallowed is complex.

In the pioneer work [25], Everspaugh et al. defined an IND-CPA analogous security called UP-IND and a ciphertext integrity (CTXT) analogous security called UP-INT-CTXT. CCA security was not considered at all in [25, 124], as in a standard AE scheme, it is well-known that IND-CPA and CTXT imply IND-CCA security. However, given that those security models are fairly complex, we first ask a question *whether such implication still holds in the general ciphertext dependent updatable encryption.* [1]).

[1] A very recent work [125] demonstrates this relationship still holds for UE in the ciphertext independent setting, which is a special case for updatable encryption that headers are not needed for update, Both settings have pros and cons [124], which we will discuss in detail in the section of related works. In this chapter, we focus on the general ciphertext dependent UE, as [21, 25, 124].

The security after the server being compromised. A more serious issue is related to those existing definitions themselves. Compared to the models for AE schemes, the UE models should fully consider the content security when the server is occasionally compromised. As noticed by [126], the previous integrity model UP-INT-CTXT is only against restricted attackers: the attacker is not allowed to ask the server to re-encrypt a *maliciously* formed ciphertexts that is of her choice. Instead, she can only query the re-encryption oracle with honestly generated ciphertext that was received from the challenger via related oracles (e.g., (re)encryption oracle). Clearly, an adversary could try to inject all kinds of ciphertext into the server and eventually got updated and mixed into the user-supplied ciphertext. Indeed, as Klooß, et al. concluded, both the confidentiality and integrity protections in [25] "*are only guaranteed against passive adversaries*".

Indeed, existing constructions of updatable encryption will become insecure if we allow the *malicious re-encryption queries*. In Sec 5.3.1, we provide a concrete example to show an active "attack" on the integrity of the KSS scheme proposed by Everspauph et al. [25]. It follows that the constructions are vulnerable against active adversaries who try to inject malformed ciphertext, which immediately violates the integrity; and what's worse, such capability could be leveraged to break confidentiality. The situation is the same in [124].

Having noticed the problem, some partial progresses have been made in the ciphertext independent setting [126].[2] In their first construction, they also have the same restriction in both ciphertext integrity and CCA security. In their second construction, they remove the restriction partially, that achieved plaintext integrity and RCCA security (Replayable CCA [127]). It is widely believed that PTXT does not provide a strong enough integrity guarantee for secure storage [128], as the adversary may still be able to generate a ciphertext that was mauled from a target ciphertext. While RCCA security has another restriction that a ciphertext generated by re-randomizing a challenge ciphertext is not allowed to query decryption oracle, thus clearly not CCA secure.

[2]As mentioned above and we will discuss further in related work, the security of ciphertext dependent UE are even more involved due to the extra headers and flexible generation of update tokens.

The security after the key being compromised. Besides characterizing the server breach scenario, how to precisely define the security when the breach occurs on the client side also needs to be crystal clear. The main motivation of updatable encryption is to enable the outsourced storage to "regain" security even the client got temporarily hacked, so long as the system later executes the update process (updating both secret key and ciphertext). However, it has been pointed out in [27] that the security model of [25] is ambiguous regarding whether the adversary is allowed to see a certain version of the challenge ciphertext, which is updated from a ciphertext that was encrypted under a leaked key.

If we look at the example for the model of UP-IND [25] in more detail: the keys are all generated once and there are no clearly defined epochs. Suppose the challenge ciphertext c_1^* is first encrypted under k_1. When the adversary queries c_1^*'s update under k_3 after the adversary queries k_2, the challenger will directly re-encrypt the challenge ciphertext c_1^* under k_1 to a ciphertext c_3^* under k_3. During this procedure, the challenge ciphertext has never been updated to some version under the key k_2. More generally, in the model of [25], for all the versions of exposed challenge ciphertext, their previous version were always encrypted under a safe key which has never been exposed. (This is the same in [124]).

But in reality, the server updates sequentially, all ciphertext have been updated from a previous version whose key may be leaked (that's why it is related to post-compromise security). It is possible that the updated ciphertext contains some private information accessible to the key of the prior ciphertext version. Also, the adversary likely pretends as the client to query the header she wants, even including that of challenge ciphertext encrypted under breached keys.

For those reasons, a model that aims to precisely capture post-compromise security was proposed in [27] for the ciphertext independent setting, in which the client generates one update token for all ciphertext. However, it is unclear whether we can adapt straightforwardly the security from ciphertext independent setting to the more general ciphertext dependent setting. In the former, there was no headers involved, and one update token will be used to update all ciphertext; while in the latter, a more careful treatment is needed to deal with those headers and ciphertext specific update tokens.

5.1.1 Our results

In this chapter, we give a systematic study of standard ciphertext integrity and security notions against CCA attacks, in the general setting of ciphertext dependent updatable encryption (CDUE). We summarize our results with comparison with previous work in Table. 5.1.

Scheme	Update Manner	Confidentiality	Integrity	ReEnc IND
BLMR [21]	CD	CPA	No	CPA
KSS [25]	CD	CPA	CTXT^{-}	$\perp$
ReCrypt [25]	CD	CPA	CTXT^{-}	CPA
Nested UAE[124]	CD	CPA	CTXT^{-}	CPA
KH-PRF UAE [124]	CD	CPA	CTXT^{-}	CPA
RISE [27]	CI	CPA	No	CPA
E&M [126]	CI	CCA^{-}	CTXT^{-}	CPA
NYUE [126]	CI	RCCA	PTXT	CPA
SHINE [125]	CI	CCA^{-}	CTXT^{-}	CCA^{-}
ReCrypt^{+}	CD	CCA	CTXT	CCA

TABLE 5.1. Comparison of properties of existing UE schemes. CD/CI means ciphertext dependent/independent respectively; CCA^{-} and CTXT^{-} means the models that disallow malicious re-encryption queries.

Security models and relations. We provide a new model combination *strengthened* UP-IND-CCA (sUP-IND-CCA) and *strengthened* UP-INT-CTXT (sUP-INT-CTXT) to characterize both the confidentiality and the integrity of CDUE. Comparing the combination of UP-IND and UP-INT suggested in [25, 124], our model strengthens the security in following aspects.

- We capture the active adversary who can query the re-encryption oracle with maliciously generated ciphertexts in confidentiality and ciphertext integrity models (CPA, CCA and CTXT). To demonstrate the practical security improvement in our models, we also show an "attack" on the KSS scheme [25] when facing malicious re-encryption threats in Sec 5.3.1.
- We use the notion of epoch from [27] in both the confidentiality and integrity models, to capture the post-compromise security. As noted before, we need to carefully deal with the headers, and flexibly generated update tokens in ciphertext dependent setting. We added two more oracles to give a more fine-grained characterization.

$\mathcal{O}_{\mathsf{Next}}(\cdot)$ is used to force the challenger to update, and $\mathcal{O}_{\mathsf{Header}}(i)$ is used to respond with the header of challenge ciphertext in epoch i (updated from previous epoches). In Sec 5.3.2, we provide a variation of KSS scheme from [25] which fails to achieve post-compromise security, but was proven secure in the existing model.

- Interestingly, after clearly defining the CPA, CCA and CTXT securities, we show that in contrast with the conventional wisdom in AE, IND-CPA security + CTXT security do *not* imply IND-CCA security in the setting of ciphertext dependent UE. Note that the CCA attack on our counter example holds with or without malicious re-encryption. That means we have to study both IND-CCA security and CTXT security in ciphertext dependent UE.
- As a byproduct, we also consider CCA style of re-encryption indistinguishability, which is to capture update unlinkability. We defer details regarding this part in Sec 5.4.4.

Construction. With the strengthened security models at hand, we set force to construct a (ciphertext dependent) updatable encryption named **ReCrypt**$^+$, which can be proven secure under our sUP-IND-CCA, sUP-INT-CTXT and sUP-REENC-CCA models. Our starting point is the **Recrypt** scheme in [25], which already has the basic confidentiality and integrity. The existing attacks reminded us several main challenges: first we need to ensure that the update procedure is as "independent" as possible so that post-compromise security can be achieved; next major challenge is how to mute the malicious re-encryption attacks. Intuitively, the validity of ciphertexts must be checked before updating. Here is the dilemma: the server does not have the secret key, thus have to rely on the assistance of the client to do the checking. But the client only sees the short header during the key rotation.

Let us walk through the subtleties and our ideas. **ReCrypt** uses the key encapsulation mechanism (KEM for short) + data encapsulation mechanism (DEM for short) [3] with secret sharing. Specifically, its header is a KEM $\mathsf{Kem}(k, x)$ for the DEM key share x under the master key k, and the body is with the form $(y, \mathsf{Dem}(x \oplus y, m))$ for the DEM key share y

[3] As noted earlier in footnote 4, this thesis considers a slightly abused notion of KEM/DEM only in the symmetric setting, which is also referred to as AE-hybrid in [25].

and the DEM of the message m. During the key rotation, the header (i.e. $\mathsf{Kem}(k, x)$) will be sent back to the client. We can instantiate the KEM via an authenticated encryption. Hence the validity of the header part can be directly verified by the client who holds the master key. However, the main challenge remains as validity check of the ciphertext body still has to be carried out on the server side.

A naive attempt. A naive suggestion is to hash all the ciphertext body can include the digest into the header plaintext. The client will use the AE to check whether the header is intact, and include the digest in the update token, so that the server can check the body. This has two major problems: first, it immediately kills the possibility for efficient update; moreover, such a method may not be sure: when the server notices the invalidity of the ciphertext after receiving the decrypted digest from the client, the update token has already been sent out. The server may stop re-encryption, but the adversary who obtains the update token may already be able to infer useful information.

Enable validity checking. To facilitate efficient update and checking, we would need a "hash" that satisfies the following: (1) it compresses the ciphertext body, otherwise the header would be too long; (2) it is "binding", so that the server can check the digest and ciphertext body; (3) it is partially hiding: as the secret key of previous epoch might be leaked, combining with part of the ciphertext may lead to the exposure of some master key; (4) it satisfies certain key homomorphism so that efficient update could be facilitated. Using a commitment scheme will not be compressing; while using a collision resistant hash may not be hiding. We proceeds in two steps: the key share y needs to be protected, thus it will be committed to c_y using a homomorphic commitment scheme; while the payload carrying the actual encrypted data will be compressed into a short digest h with a homomorphic collision resistant hash. $c_y||h$ will be the derived digest.

Avoid dangerous update token. Regarding the second problem, either the server or the client should be able to detect the invalidity of ciphertext *before* the update token has been generated! To facilitate such verifiability, we put $c_y||h$ as the associated data to encrypt them together with the key share in the header using authenticated encryption with associated data. We emphasize that encrypting the digest using AE directly (without putting them in plain as well)

will be problematic, as now the server cannot check first, adversary may inject a header which is not bound to the ciphertext body, e.g, taking from a previous ciphertext. Now the client cannot detect and will generate the update token.

Homomorphically hash from a group. One more subtlety remains, as the above verification ideas have not considered how to be compatible with the re-encryption. Specifically, **ReCrypt** updates the DEM part via the key homomorphic pseudorandom functions (KH-PHF) [21]. When the DEM part is updated by adding new KH-PRF values, we wish that the hash value of the DEM part, which is included in the header, can be updated by the client conveniently according to those KH-PRF values. Therefore, we design a new homomorphic collision resistant hash function, whose domain needs to match the range of the KH-PRF which is some particular groups instead of binary strings. Specifically, we construct such homomorphic hash functions from the asymmetric bilinear maps $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. The KH-PRF could be constructed over $\mathbb{G}_1$, where the DDH problem is hard.

5.1.2 Related works

Two flavors of updatable encryption. As we briefly mentioned above, in many of the updatable encryption schemes, during the key rotation, the client would first retrieve a small piece of the ciphertext (called header), and then generates a update token. Such kind of UE is called ciphertext dependent UE [21, 25, 124], (CDUE in short). On the other hand, one may insist that the client directly generates the update token. Such a UE scheme is called ciphertext independent UE [27, 126, 125] (CIUE in short).

Though ciphertext independent UE saves one round of communication, the header is normally extremely short in ciphertext dependent UE. More importantly, since in a ciphertext dependent UE, the client can generate update token based on each ciphertext header, this gives a fine-grained control over updating procedure and security: the client could choose to update only part of the ciphertext, and leakage of some token does not influence other ciphertext.

As discussed in detail in previous work [124], there are both pros and cons for these two flavors of UE, and the different updating paradigms yield different security definitions, applications

and construction strategies. In this article, we focus on ciphertext dependent schemes, and fill the gap exists in integrity and CCA security. We discussed more detailed comparisons in Sec 5.1.3.

Other related works. The first updatable encryption scheme (**BLMR**) is proposed by Boneh et al. [21]. However, only the confidentiality is considered in this work, and the other security notions have not been formalized. Later, Everspauph et al.[25] provided a systematic study of updatable encryption in the ciphertext dependent setting, as we discussed, they did not allow malicious re-encryption in integrity and CPA notions, which are the main objective of this chapter. Very recently, Boneh et.al [124] revisit the results of Everspauph et al. about CDUE. Their security notion is similar to [25], and they did not consider the post-compromise security and the malicious update resistance. Moreover, **Nested UAE** can only proceed the key rotation with bounded number of times.

Lehmann and Tackmann [27] point out the models UP-IND and UP-REENC in [25] are hard to capture the post compromise security. So they provide the models (IND-ENC and IND-UPD) and the construction (**RISE**) with the post-compromise security. Recently, Klooß et al. [126] add the integrity considerations to [27], and provide two constructions (**E&M** without malicious update resistance and **NYUE** with only plaintext integrity and the weaker RCCA security).

Boy et.al [125] first formally prove that for CIUE without malicious update, the folklore relationship in authenticated encryption that the combination of CPA and CTXT security yields CCA security still holds. However, the relationship for CDUE remains open.

5.1.3 Discussions on CDUE vs. CIUE

First of all, since every token is specifically generated for each ciphertext for CDUE, the leakage of one token for one ciphertext may not effect the security of other ciphertexts. However, since CIUE uses one token to upgrade all ciphertexts, the leakage of that token

could be a more serious threat to the security. Therefore, the security models for CDUE and CIUE about confidentiality and integrity must be completely different.

Secondly, the different features of two types of UE make them suitable for different applications. The communication cost of CIUE is comparatively cheaper, since the client does not need to download anything from the server for the key rotation, and the upload message is a single token whose size is independent of the number of the ciphertexts. However, the CDUE supports a more fine-grained control of which ciphertexts should be re-encrypted towards the new key. Since UE may not only be used to protect the confidentiality but also the integrity, the fine-grained control by the client side is critical for some special applications. For example, the Department of Motor Vehicle (DMV) may use the CDUE to encrypt the documents for each driving license, and outsource the ciphertexts to a cloud storage service. The DMV will update these ciphertexts periodical. But if one driving license is revoked, the DMV will refuse to generate a corresponding update token. As the consequence, all the valid licenses can be easily audited according to the valid ciphertexts of the current epoch on the server, even the server itself is not fully trusted.

Thirdly, the existing constructions of CIUE have an obvious restriction: their plaintext space is too small to protected the integrity of a large file. To our best knowledge, the only CIUE schemes that can protect the integrity are the two constructions in [126], whose plaintext space is a group element of security parameter size. To encrypt a large file, one has to divide the file into a sequence of blocks, and encrypt each block separately. However, in this case the adversary may be able to change the order of these blocks without been detected. Note that the scenario is different for CDUE [25, 124], since they can directly encrypt a plaintext with arbitrary length.

5.2 Formalization

In this section, we formalize the syntax of the ciphertext dependent updatable encryption scheme following [25].

Intuitively, the data flow of the outsource storage from CDUE can be seen in Fig. 5.1. With loss of generality, we divide the whole storage period into multiple time epochs. At the beginning of the storage, the client generates a secret key k_0 for the epoch 0, encrypts his file m with the key k_0, and outsources the initial ciphertext $C_0 = \left(\tilde{C}_0, \bar{C}_0\right)$ to the server. Here $\tilde{C}_0$ is the header and $\bar{C}_0$ is the body. After a specific epoch e, the serve will send back the header $\tilde{C}_e$. The client will generate a new key k_{e+1}, compute a token $\Delta_{e,\tilde{C}_e}$ and send it back to the server. The server will update the old ciphertext C_e to the new one C_{e+1} with the token $\Delta_{e,\tilde{C}_e}$. Formally, we have the following definition.

DEFINITION 5.2.1 (Updatable Encryption). The ciphertext dependent updatable encryption (CDUE) consists of the following six algorithms

$$\textbf{CDUE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{ReKeyGen}, \mathsf{Recrypt}).$$

- $\mathsf{Setup}(1^\lambda)$ is a randomized algorithm run by the client. It takes the security parameter λ as input and outputs the public parameter pp which will be shared with the server. Later all algorithms take pp as input implicitly.
- $\mathsf{KeyGen}(e)$ is a randomized algorithm run by the client. It takes the epoch index e as input and outputs a secret key k_e for the epoch e.
- $\mathsf{Encrypt}(k_e, m)$ is a randomized algorithm run by the client. It takes the secret key k_e and the message m as inputs, and outputs the ciphertext $C_e = (\tilde{C}_e, \bar{C}_e)$ which consists of two parts, i.e., the header $\tilde{C}_e$ and the body $\bar{C}_e$.
- $\mathsf{Decrypt}(k_e, C_e)$ is a deterministic algorithm run by the client. It takes the secret key k_e and the ciphertext C_e as inputs, and outputs the message m or the symbol $\perp$.
- $\mathsf{ReKeyGen}$ $(k_e, k_{e+1}, \tilde{C}_e)$ is a randomized algorithm run by the client. It takes the header $\tilde{C}_e$, the old secret key k_e of the last epoch and the new secret key k_{e+1} of the current epoch as inputs, and generates a re-encrypt token $\Delta_{e,\tilde{C}}$ or outputs the symbol $\perp$.
- $\mathsf{Recrypt}(\Delta_{e,\tilde{C}_e}, C_e)$ is a deterministic algorithm run by the server. It takes the re-encrypt token $\Delta_{e,\tilde{C}_e}$ and the ciphertext $C_e = (\tilde{C}_e, \bar{C}_e)$ as inputs, and outputs a new ciphertext $C_{e+1} = \left(\tilde{C}_{e+1}, \bar{C}_{e+1}\right)$ under the secret key k_{e+1} or the symbol $\perp$.

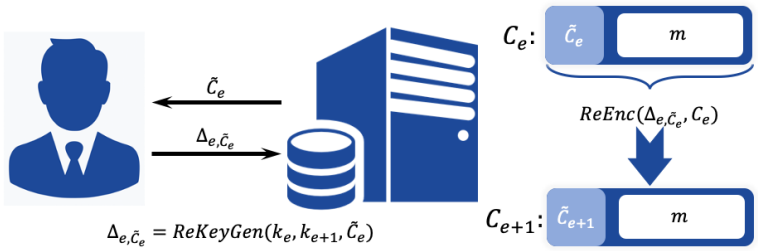


FIGURE 5.1. The data flow between client and cloud during the key update of the ciphertext $C_e = \left(\tilde{C}_e, \bar{C}_e\right)$ for the epoch e. The client receives a small ciphertext header $\tilde{C}_e$, and runs ReKeyGen to produce a compact update token $\Delta_{e,\tilde{C}_e}$. The server uses this token to re-encrypt the ciphertext C_e to C_{e+1}.

Note that the above formalization is tailored to our ciphertext integrity definition. Particularly, here we require the algorithm Recrypt to be deterministic. It is because, if the server is allowed to randomly re-encrypt the ciphertext given the token and the header, a malicious server may run this procedure more than one time, and get multiple (maybe exponentially large number of) versions of the updated ciphertext. Consequently, this makes the challenger to track the trivially obtained ciphertext in the CTXT game extremely difficult. Moreover, such a restriction of the syntax has little impact on the construction, since the algorithm Recrypt is deterministic for almost all existing CDUE schemes [25, 124].

Besides, the syntax of the CIUE scheme can be viewed as a special case of the ciphertext dependent scheme in Definition 5.2.1 when choosing a dummy header, although its security definition may be different. In this case, the server has no need to send the header back, and the update token is generated from the old and new keys directly.

Correctness. We define the correctness of CDUE if the ciphertext can be correctly decrypted after arbitrary times of key update. Specifically, we have the following formal definition.

DEFINITION 5.2.2 (Correctness). For an updatable encryption scheme **CDUE**, each epoch key k_i is generated by **CDUE**.KeyGen(i) for epoches from 0 to e. For a message m and any integer i such that $0 \leq i \leq e$, let $c_i \leftarrow$ **CDUE**.Encrypt(k_i, m) and recursively define for each

j from $j = i + 1$ to $j = e$,

$$\Delta_{j-1,\tilde{C}_{j-1}} \leftarrow \mathsf{ReKeyGen}\left(k_{j-1}, k_j, \tilde{C}_{j-1}\right),$$
$$C_j \leftarrow \mathsf{Recrypt}\left(\Delta_{j-1,\tilde{C}_{j-1}}, C_{j-1}\right).$$

Then **CDUE** is correct if $\Pr[\textbf{CDUE}.\mathsf{Decrypt}(k_e, C_e) = m] = 1$ for any message m, any integer e and any integer i such that $0 \leq i \leq e$.

Compactness. We say that a CDUE scheme is compact if the size of total communications between client and server during update is independent of the length of the plaintext. In practice, the compactness guarantees that the communication cost for the key update procedure is efficient.

5.3 Insufficiency of Existing Models

As we explained in the introduction, the previous model combination UP-IND + UP-INT [25, 124] needs to be strengthened in the following three aspects.

Malicious re-encryption attack. All previous CDUE definitions [25, 124] did not consider malicious re-encryption threats, particularly for integrity, i.e. the adversary may query *maliciously generated* ciphertexts to the re-encryption oracle. However, a real-world adversary who can temporarily compromise the server may inject arbitrary ciphertexts in data storage. These injected ciphertexts may be automatically updated by the server, even if they may not be decrypted successfully. Such possibilities can be leveraged by the adversary to attack the integrity or the confidentiality. In Sec 5.3.1, we show that an adversary of the **KSS** scheme [25] can fabricate a valid ciphertext by querying re-encryption oracle with an ill-formed ciphertext. The intuition of the attack is that the adversary may generate a valid ciphertext C_1 for epoch 1 by corrupting key k_1. But instead of querying the re-encryption oracle with C_1 directly, the adversary may query with a invalid ciphertext $C_1' = f(C_1)$ which is a modification of C_1' via certain operation f. After getting an updated ciphertext C_2' (which is still invalid), the adversary can recover a valid ciphertext C_2 from C_2' though an inverse operation f^{-1}. More importantly, since C_2 is not directly generated via querying the re-encryption oracle or the

encryption oracle, and the epoch key k_2 has not been corrupted, C_2 will be considered as a legitimate forgery in the CTXT game!

Post-compromise security. The security model in [25, 124], as discussed in [27], is hard to capture the post compromise security. More precisely, the UP-IND model is ambiguous that whether the adversary is allowed to view certain version of the challenge ciphertext updated from a key corrupt epoch. We gave exemplary explanations in the introduction. and we will give a concrete example in Sec 5.3.2 to show a scheme proved secure under UP-IND model, but can be attacked by a real world adversary. As pointed by Lehmann and Tackmann in [27], this ambiguity is caused by the missing of the epoch notion in UP-IND. The integrity model UP-INT has a similar problem. Of course, the definition is more involved as we also need to consider the leaked headers, and flexible generation of tokens.

Chosen ciphertext attack. The chosen ciphertext attack is a real threat to a UE system. One the one hand, a malicious server may choose an arbitrary ciphertext to answer the retrieve query of the client, and learn the information about the decryption result later on from side channels (e.g. the server may easily learn whether the decryption is successful according the response of the client.); on the other hand, temporary breaches of the client's device may happen occasionally. Although the secret key may not be easy to steal due to the limit of time, the adversary may use the compromised device as an decryption oracle. Nevertheless, the previous models for CDUE in [25, 124] have not considered the chosen ciphertext attack. One may hope that UP-IND plus UP-INT can imply a CCA style security analogous to the AE setting, but such a relation have never been proved for UE. We will show soon that it turns out to be false!

5.3.1 Malicious re-encryption threats.

In [126] Klooß, Lehmann and Rupp pointed out the model in [25] has an artificial restriction on the queries to the re-encryption oracle. Specifically, the model only allows the adversary to query honestly generated ciphertexts to the re-encryption oracle. In practice, the model

with such restriction could not capture some malicious re-encryption threats on existing UE schemes. We take the KSS scheme in [25] as an example.

KSS is a ciphertext-dependent updatable encryption scheme which is proved to satisfy the UP-IND and UP-INT security defined in [25]. The KSS uses two AE schemes[4] $\pi_{kem} = (\mathcal{K}_{kem}, \mathcal{E}_{kem}, \mathcal{D}_{kem})$ and $\pi_{dem} = (\mathcal{K}_{dem}, \mathcal{E}_{dem}, \mathcal{D}_{dem})$ and key sharing to build an updatable encryption scheme. The message is encrypted with $(C, \tau) \leftarrow\$ \mathcal{E}_{dem}(x, m)$ where the key $x = r \oplus y$ produces two shares r, y with secret sharing, one share r is clear in the ciphertext and another share y with the DEM tag τ is encrypted in KEM with the master key. Key rotating is to update the master key and re-randomize the two key shares leaving the DEM static.

- **KSS**.Setup(λ): Return $(k_1, \ldots, k_{t+\kappa}) \leftarrow\$ \mathcal{K}_{kem}$.
- **KSS**.Enc(k, m): Compute $x \leftarrow\$ \mathcal{K}_{dem}, r \leftarrow\$ \mathcal{K}_{dem}, (C, \tau) \leftarrow\$ \mathcal{E}_{dem}(x, m)$, $y \leftarrow x \oplus r$, $\tilde{C} \leftarrow\$ \mathcal{E}_{kem}(k, y\|\tau)$, and return $(\tilde{C}, (r, C))$
- **KSS**.ReKeyGen$(k_i, k_j, \tilde{C})$: compute $(y, \tau) \leftarrow \mathcal{D}_{kem}(k_i, \tilde{C})$, $r' \leftarrow\$ \mathcal{K}$, and $\tilde{C}' \leftarrow\$ \mathcal{E}_{kem}(k_j, y \oplus r'\|\tau)$, return $\Delta_{i,j,\tilde{C}} = (\tilde{C}', r')$
- **KSS**.ReEnc$(\Delta_{i,j,\tilde{C}}, (\tilde{C}, (r, C)))$: parse $(\tilde{C}', r') = \Delta_{i,j,\tilde{C}}$, return $(\tilde{C}', (r \oplus r', C))$
- **KSS**.Dec$(k, (\tilde{C}, (r, C))$: run $(y\|\tau) \leftarrow \mathcal{D}_{kem}(k, \tilde{C})$, return $m \leftarrow \mathcal{D}_{dem}(y \oplus r, C, \tau)$

Malicious re-encryption threat on the ciphertext integrity. The adversary corrupts the current secret key $k_{e^\star}$ and generates a ciphertext $(\tilde{C}, (r, C))$ of message m with $k_{e^\star}$. Then she intrudes the storage sever and injects an invalid ciphertext $(\tilde{C}, (\tilde{r}, C))$ with wrong key share $\tilde{r}$. After the intrusion, the server rotates from $k_{e^\star}$ to $k_{e^\star+1}$ where $k_{e^\star+1}$ is unknown to the adversary. The adversary then intrudes the sever again, and finds the update of the invalid ciphertext denoted by $(\tilde{C}', (\tilde{r}', C))$. The adversary extracts $\tilde{r}'$ and recover $r' \leftarrow \tilde{r}' \oplus \tilde{r} \oplus r$. So the adversary can produce a new valid ciphertext $(\tilde{C}', (r', C))$ of message m in current epoch $e^\star + 1$.

[4] As noted earlier, AE's notation is slightly abused to have two forms of ciphertext, a single string c or a pair of bit strings (c, τ) where τ denote an authentication tag made explicit in the paired form.

Malicious re-encryption threat on the confidentiality. In the UP-IND model, the adversary is disallowed to make re-encryption query of any malicious ciphertext having the same header with the challenge ciphertext to a corrupted key. However, in a strengthened confidentiality model allowing malicious update, the adversary has more capability, who can query the re-encryption to a corrupted key with any ciphertext except the exact challenge ciphertext. The capible adversary can utilize such ability to break the confidentiality of KSS in the following way. The adversary obtains the challenge ciphertext $(\tilde{C}, (r, C))$ of message m_b with $k_{e^\star}$. She corrupts the next key $k_{e^\star+1}$ and makes an re-encryption query to the next corrupted key $k_{e^\star+1}$ with the modified ciphertext $(\tilde{C}, (\hat{r}, C))$. The replied update $(\tilde{C}', (\hat{r}', C'))$ can be be used to recover the real update of challenge ciphertext under the corrupted key, i.e., $(\tilde{C}', (\hat{r}' \oplus \hat{r} \oplus r, C'))$, which can be decrypted to the selected challenge message m_b. Thus, the confidentiality is broken.

5.3.2 Post-compromise corruption threats.

We all know that in the real world our storage is vulnerable, which means that both the keys stored in the client and the ciphertexts stored in the cloud may suffer from the adversary's corruption. Updatable encryption is proposed to protect the security by rotating the ciphertext to under a new secret key, even if some previous components, such as the previous key, token and ciphertext, have been corrupted as long as the those corruptions do not occur at the same time in which message confidentiality is impossible. The post-compromise corruption aims to describe all those corruptions about some parts of the previously protected information. Therefore, essentially UE is supposed to guarantee security in face of the post-compromise corruption.

However, the existing models UP-IND [25] for the message confidentiality suffers from the post-compromise corruption. In their models, the ciphertext update can cross several epochs. For example, there are three epochs one to five where the corresponding keys are k_1 to k_3. A ciphertext encrypted with the key k_1 can be updated directly and jumping to a ciphertext under the key k_3 without passing through the update under the key k_2. When the intermediate key k_2 is corrupted, the ciphertext under the k_3 updated directly from the k_1 may avoid some

threats, but the ciphertext under the k_3 which is generate under two consecutive updates from k_1 to k_2 and then from k_2 to k_3 may suffer from those threats.

Now we construct a detail scheme which is UP-IND secure but sUP-IND-CPA insecure to show that the previous confidentiality model cannot capture the post-compromise corruption. Intuitively, we add an extra component to the UP-IND secure ciphertext. This extra component is an encryption of some secret information about the plaintext of the total ciphertext, and can be corrupted only when the ciphertext is updated from a key-corrupted epoch.

For simplicity, we take the **KSS** as the building block which is UP-IND secure to construct **KSS**' which is UP-IND secure but sUP-IND-CPA insecure.

- **KSS**'.$\mathsf{Setup}(\lambda)$: Run **KSS**..Setup.
- **KSS**'.$\mathsf{Enc}(k, m)$: Run **KSS**.$\mathsf{Enc}(k, m)$ to get $(\tilde{C}, (r, C))$. The detail steps are $x \leftarrow\$ \mathcal{K}_{dem}, r \leftarrow\$ \mathcal{K}_{dem}, (C, \tau) \leftarrow\$ \mathcal{E}_{dem}(x, m)$, $y \leftarrow x \oplus r$, $\tilde{C} \leftarrow\$ \mathcal{E}_{kem}(k, y\|\tau)$. And then return $(\tilde{C}, (r, C, \perp))$ as the ciphertext of **KSS**'.
- **KSS**'.$\mathsf{ReKeyGen}(k_i, k_j, \tilde{C})$: Run the **KSS**.$\mathsf{ReKeyGen}(k_i, k_j, \tilde{C})$ algorithm, that is, compute $(y, \tau) \leftarrow \mathcal{D}_{kem}(k_i, \tilde{C})$, $r' \leftarrow\$ \mathcal{K}$, while $\tilde{C}' \leftarrow\$ \mathcal{E}_{kem}(k_j, y \oplus r'\|\tau)$, and get $\Delta_{i,j,\tilde{C}} = (\tilde{C}', r')$. In addition, run $C'_{pre} \leftarrow\$ \mathcal{E}(k_i, y \oplus r')$, where $\mathcal{E}$ is a symmetric key encryption algorithm and get the token $\Delta'_{i,j,\tilde{C}} = (\Delta_{i,j,\tilde{C}}, C'_{pre})$ for **KSS**'.
- **KSS**'.$\mathsf{ReEnc}(\Delta'_{i,j,\tilde{C}}, (\tilde{C}, (r, C, \perp/C_{pre})))$: First parse $\Delta'_{i,j,\tilde{C}} = (\Delta_{i,j,\tilde{C}}, C_{pre})$ and run **KSS**.$\mathsf{ReEnc}(\Delta_{i,j,\tilde{C}}, (\tilde{C}, (r, C)))$ to get $(\tilde{C}', (r \oplus r', C))$. Then add C'_{pre} and return $(\tilde{C}', (r \oplus r', C, C'_{pre}))$ as the ciphertext of **KSS**'.
- **KSS**'.$\mathsf{Dec}(k, (\tilde{C}, (r, C, \perp/C_{pre})))$: Remove the ciphertext suffix $\perp/C_{pre}$ and run **KSS**.$\mathsf{Dec}(k, (\tilde{C}, (r, C))$ to get message m. Return m as the decrypted message.

The modification in **KSS**' does not increase the adversary's advantage, as the UP-IND model is not allowed to see a challenge ciphertext updated from a corrupted key. For all challenge ciphertexts, the plaintext $y \oplus r'$ in C_{pre} is unknown to the adversary as the previous key is unknown. However, in the sUP-IND-CPA model, the adversary is allowed to corrupt the key of the former epoch i and query the $\mathcal{O}_{\mathsf{ChallengeCT}}(i+1)$ if the key of the latter epoch $i+1$ is uncorrupted. Then the adversary can decrypt C_{pre} in the challenge ciphertext of epoch

$i+1$ with the prior epoch key k_i to get $y \oplus r'$, thus decrypting the challenge message, so the message confidentiality can be easily broken.

5.4 Strengthened Security Models

In this section, we systematically study the security definitions of the CDUE. We formally define our strengthened security models for CDUE: for confidentiality, we provide the sUP-IND-CCA model; for integrity, we provide the sUP-INT-CTXT model; for re-encryption indistinguishability, we provide the sUP-REENC-CCA model in Sec 5.4.4. Moreover, we also provide the sUP-IND-CPA model without the decryption oracle for completeness, and show a counter example where a CDUE scheme is sUP-IND-CPA and sUP-INT-CTXT but not sUP-IND-CCA secure. That inspires us that the corresponding model relation is different with the case for authenticated encryption.

5.4.1 Confidentiality

Now we start from the confidentiality, and describe models strengthened UP-IND-CPA and strengthened UP-IND-CCA (sUP-IND-CPA and sUP-IND-CCA for short) which mimic the standard CPA and CCA model of AE. In these models, the key is evolving with the epochs. Beside the challenge ciphertext and the encryption/decryption oracle, the adversary is additionally allowed to obtain keys of some epochs. This captures that the client's keys are leaked. Also the adversary has the ability to get some previous versions of the challenge ciphertexts and update tokens. This captures that previous storage in the server may not be securely erased in time. To exclude the trivial impossibility, we disallow the adversary to learn a version of the challenge ciphertext and corrupt the key *within the same epoch*. However, the adversary is always allowed to see the header of any updated version of the original challenge ciphertext, even getting its body is forbidden. This is because the adversary may pretend the client in front of the server and ask the header[5].

[5]In the real world, the communication between a client and a server is typically via TLS without the user authentication [67], since the client does not have a PKI certificate. Therefore pretending the client in front of the server is not difficult.

Note that our models sUP-IND-CPA and sUP-IND-CCA have fully considered that the cases that the adversary may compromise the server during some epoch and read its memory or tamper some ciphertexts. So we allow the adversary to query the re-encryption oracle with maliciously generated ciphertexts. However, the key update procedure should follow the instructions of the UE scheme, i.e., the server will recover at the end of the epoch and *honestly execute the key rotation instructions*. The assumption is inevitable for UE, since no UE scheme can achieve the basic security if a fully malicious server refuses to execute the update operation. In practice, a benign server can quickly detect the invasion by the intrusion detection systems (IDSs), recover from the breach in time before the next key rotation with a high probability.

Experiment structure. We first describe the structure of the confidentiality game in Fig.5.2, and explain in detail how the oracles are defined right after Definition 5.4.1. As mentioned above, we also introduce the epoch notion to denote the time sequence following [27]. We index every epoch in the experiments according to its order from 0, and record the index of the current epoch with variable e. Note that in our game the challenge ciphertexts are automatically updated when moving to the next epoch. This enables us to provide to the adversary some updated versions of the challenge ciphertext which are indeed updated from an epoch in which the key is corrupted, as well as the header of the version of the challenge ciphertext in the key corrupted epoch, thus our model easily captures the post-compromise security (which was ambiguous in existing models).

DEFINITION 5.4.1 (sUP-IND-CPA(CCA)). Define the sUP-IND-CPA(CCA) experiment as Fig.5.2 where ATK is CPA(CCA). An updatable encryption scheme is called sUP-IND-CPA(CCA) secure if for any P.P.T adversary $\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\text{sUP-IND-CPA(CCA)}}_{\mathcal{A}} := \left| \Pr[Exp^{\mathcal{A}}_{\text{Adaptive UE-CPA}}(\lambda) \Rightarrow 1] - \frac{1}{2} \right|$$

is negligible for the security parameter λ.

As explained before, our sUP-IND-CPA(CCA) strengthen previous confidentiality model in aspects of the malicious update resistance and the post-compromise security. Also. the

sUP-IND-ATK $Exp^{\mathcal{A}}_{\mathsf{sUP\text{-}IND\text{-}ATK}}(\lambda)$

1 : $pp \leftarrow\$ \mathsf{Setup}(\lambda)$, Initialize $e, \mathbb{K}, \mathbb{IC}, \mathbb{KC}, \mathbb{TO}, \mathbb{CE}$
2 : $k_0 \leftarrow \mathsf{KeyGen}(pp)$, $\mathbb{K}(0) \leftarrow k_0$
3 : $(m_0, m_1, state) \leftarrow\$ \mathcal{A}^{\mathcal{O}_1}$
4 : Procced only if $|m_0| = |m_1|$
5 : $b \leftarrow\$ \{0,1\}$, $C^* \leftarrow \mathsf{Encrypt}(k_e, m_b)$, Set $\mathbb{CE}(e) \leftarrow C^*$
6 : $b' \leftarrow\$ \mathcal{A}^{\mathcal{O}_2}(state)$
7 : **for** $i = 1$ *to* e
8 : **if** $\mathbb{KC}(i) = \mathsf{true} \wedge \mathbb{IC}(i) = \mathsf{true}$ **then return** $\perp$
9 : **return** $(b' == b)$

FIGURE 5.2. The sUP-IND-ATK experiment, where ATK could be CPA or CCA. When ATK is CPA, $\mathcal{O}_1$:=($\mathcal{O}_{\mathsf{Enc}}$, $\mathcal{O}_{\mathsf{Next}}$, $\mathcal{O}_{\mathsf{KeyCorrupt}}$, $\mathcal{O}_{\mathsf{ReEnc}}$, $\mathcal{O}_{\mathsf{Token}}$) and $\mathcal{O}_2$:= ($\mathcal{O}_{\mathsf{Enc}}$, $\mathcal{O}_{\mathsf{Next}}$, $\mathcal{O}_{\mathsf{KeyCorrupt}}$, $\mathcal{O}_{\mathsf{ReEnc}}$, $\mathcal{O}_{\mathsf{Token}}$, $\mathcal{O}_{\mathsf{Header}}$, $\mathcal{O}_{\mathsf{ChallengeCT}}$). When ATK is CCA, $\mathcal{O}_1$ additionally includes $\mathcal{O}_{\mathsf{Dec}}$ and $\mathcal{O}_2$ additionally includes $\mathcal{O}_{\mathsf{Dec}}$.

sUP-IND-CCA strengthens the security against chosen ciphertext attack. To more clearly elaborate this claim, next we will describe the behaviour of the challenger during the game in detail. Especially, we will show how the challenge to maintain his internal states and answer each queries of the adversary.

The internal state of the challenger. During the games, with respect to the adversary's behaviour and the key evolution, the challenger will maintain and update the following tables to keep track of the overall state, which will be used to rule out the trivial impossibility. The rows of each table are indexed by the epoch indices.

Special cares are needed for those tables related to challenge ciphertexts. To explain, we call the ciphertexts that are updated from the challenge ciphertext *challenge-equal* ciphertexts. There is at least one challenge-equal ciphertext for every epoch since the challenge epoch. And the adversary can choose to view the challenge-equal ciphertext in any key-uncorrupted epoch and the header of the challenge-equal ciphertext in any key-corrupted epoch (via concrete oracles defined below). Note that our model does not limit to repeat querying the $\mathcal{O}_{\mathsf{ReEnc}}$ oracle with the challenge ciphertext and the challenge-equal ciphertext. Since the ReKeyGen

algorithm (hence the ciphertext update procedure) could be randomize, the adversary can acquire multiple the challenge-equal ciphertexts of the same epoch.

As previous models [25, 124], we also consider static key corruption, which means that the adversary is required to commit whether he will corrupt the key of the current epoch in advance before the challenger generating this epoch key, computing the tokens and updating all the ciphertexts to this epoch.

- Table $\mathbb{K}$ is used to record the secret key of every epoch, each entry is the secret key k_i of epoch i. All entries of $\mathbb{K}$ are initialized as $\perp$.
- Table $\mathbb{KC}$ is used to keep track of the adversary's commitments about the key corruption. Each entry is one Boolean value $b \in \{\mathsf{true}, \mathsf{false}\}$. When an epoch i begins, the static adversary needs to set $\mathbb{KC}(i)$ as true or false, which denotes her commitment about whether the secret key of that epoch i can be corrupted in the game.
- Table $\mathbb{CE}$ is used to record all the challenge-equal ciphertexts during the experiment. Specifically, each entry $\mathbb{CE}(i)$ contains all the challenge-equal ciphertexts of the corresponding epoch. All the ciphertexts are updated to the current epoch automatically with key update. All entries will be initially set as $\perp$ during the experiment.
- Table $\mathbb{TO}$ is used to keep track of the event that a token related to challenge-equal ciphertext is corrupted. Specifically, the i-th entry is one Boolean value $b \in \{\mathsf{true}, \mathsf{false}\}$. Here $\mathbb{TO}(i+1) = \mathsf{true}$ denotes that the following event has happened during the game: a valid token updating *any* one challenge-equal ciphertext from epoch i to epoch $i+1$ has been queried by the adversary. All entries will be initially set as false during the experiment.
- Table $\mathbb{IC}$ is used to keep track of the event of the adversary's corruption of the challenge-equal ciphertexts. Specifically, each entry i contains one Boolean value $b \in \{\mathsf{true}, \mathsf{false}\}$. Here $\mathbb{IC}(i) = \mathsf{true}$ means the following event has happened during the game: there are certain challenge-equal ciphertext in the epoch i has been learned by adversary via different oracles (to be defined below) directly or indirectly. Note that there may be multiple challenge-equal ciphertexts for one epoch due the

randomized key update procedure. Here we make $\mathbb{IC}(i) = \mathsf{true}$ if *anyone* of the challenge-equal ciphertexts for epoch i is leaked to the adversary. All entries will be set false when the game starts.

Oracles of the adversary. We now formally define the queries that adversary is allowed to ask. Note that the epoch variable e will automatically increase during the game, and the key and the challenge-equal ciphertexts are automatically updated accordingly. This procedure is triggered by the oracle $\mathcal{O}_{\mathsf{Next}}$. Hence the challenge-equal ciphertexts will be updated to the key-corrupted epochs, and the adversary can see their headers but not bodies. This feature helps us to go beyond the restriction of the models in [25], and capture post compromise security. Also note that we allow the adversary to query $\mathcal{O}_{\mathsf{ReEnc}}$ with maliciously generated ciphertexts, and $\mathcal{O}_{\mathsf{ReEnc}}$ may return $\perp$ if the ReKeyGen and Recrypt algorithms include a invalid ciphertext detection mechanism. Similarly, $\mathcal{O}_{\mathsf{Token}}$ may reply $\perp$ when queried with an invalid header.

In a brief overview, the following oracles mainly enable the adversary to do the following corruptions. $\mathcal{O}_{\mathsf{Next}}$ enables $\mathcal{A}$ to make the key evolve by turning to the next epoch. $\mathcal{O}_{\mathsf{Enc}}$, $\mathcal{O}_{\mathsf{Dec}}$, and $\mathcal{O}_{\mathsf{ReEnc}}$ oracles allow $\mathcal{A}$ to query the encryption, decryption, and re-encryption on and to the latest epoch. $\mathcal{O}_{\mathsf{KeyCorrupt}}$ and $\mathcal{O}_{\mathsf{Token}}$ allows $\mathcal{A}$ to corrupt any of the existing epoch keys and any ciphertext update tokens except the challenge ciphertext. $\mathcal{O}_{\mathsf{Header}}$ and $\mathcal{O}_{\mathsf{ChallengeCT}}$ oracles are related to the challenge ciphertext and provide $\mathcal{A}$ the corresponding headers and ciphertext.

- *Turn to next epoch oracle* $\mathcal{O}_{\mathsf{Next}}(b)$: This oracle is to used to inform the challenger to evolve to the next epoch $e + 1$, and update all challenge-equal ciphertexts in table $\mathbb{CE}(e)$ to the epoch $e + 1$. Specifically, the input of the oracle $\mathcal{O}_{\mathsf{Next}}$ is a bit b which denotes whether the epoch key k_{e+1} will be corrupted later on, the challenger will record $\mathbb{KC}(e + 1) = b$ in the key corruption table. Moreover, the challenger runs $\mathsf{KeyGen}(pp)$ to produce a new key k_{e+1} for the new epoch $e + 1$ and sets $\mathbb{K}(e + 1) = k$ in the key record table. For each challenge-equal ciphertext $C_e = (\tilde{c}_e, \bar{c}_e) \in \mathbb{CE}(e)$ (if the challenge-equal ciphertext table $\mathbb{CE}(e)$ is not empty),

run the token generation algorithm $\Delta_{e,e+1,\tilde{c}} \leftarrow\$ \mathsf{ReKeyGen}(k_e, k_{e+1}, \tilde{c}_e)$ and the update algorithm $C' \leftarrow \mathsf{Recrypt}(\Delta_{e,e+1,\tilde{c}_e}, C_e)$ for each ciphertext and import all the updated ciphertexts to the row $\mathbb{CE}(e)$. Finally, the challenger updates the current epoch variable e by adding one as $e \leftarrow e + 1$.

- *Encrypt oracle* $\mathcal{O}_{\mathsf{Enc}}(m)$: This oracle is used to ask the challenger to encrypt a message m under the current epoch key. The challenger will run $C \leftarrow \mathsf{Encrypt}(k_e, m)$ and return the ciphertext C to the adversary.
- *Decrypt oracle* $\mathcal{O}_{\mathsf{Dec}}(C)$: This oracle is to ask the challenger to decrypt ciphertext C under the current epoch key. When queried with a ciphertext C, the challenger will check the table $\mathbb{CE}$ to identify whether C could be a challenge-equal ciphertext. If $C \notin \mathbb{CE}(i)$ for i from 0 to e, the challenger will run the algorithm $m \leftarrow \mathsf{Decrypt}(k_e, C)$ to decrypt C with current key k_e and return m to the adversary; otherwise, return $\perp$. This is to avoid the trivial attack that the adversary may query $\mathcal{O}_{\mathsf{Dec}}$ on a challenge-equal ciphertext.
- *Key corrupt oracle* $\mathcal{O}_{\mathsf{KeyCorrupt}}(i)$: This oracle is used to corrupt the keys for previous epochs. Note that in our static model the adversary is only allowed to corrupt the key that he has committed before. When queried the epoch index i, the challenger checks the key corruption commit table $\mathbb{KC}(i)$ at first. If $\mathbb{KC}(i) = \mathsf{true}$, the challenger returns the secret key k_i of the epoch i. Otherwise, he returns $\perp$.
- *Token corrupt oracle* $\mathcal{O}_{\mathsf{Token}}(i, \tilde{c})$: The adversary is allowed to query this oracle to obtain update tokens. When queried with an epoch index i and the corresponding ciphertext header $\tilde{c}$, the challenger will run the token generation algorithm $\Delta_{i,i+1,\tilde{c}} \leftarrow\$ \mathsf{ReKeyGen}(k_i, k_{i+1}, \tilde{c})$, and return the token $\Delta_{i,i+1,\tilde{c}}$ to the adversary. If $\Delta_{i,i+1,\tilde{c}} \neq \perp$ and the header $\tilde{c}$ has even appeared in $\mathbb{CE}(i)$, the challenger will update the token corruption table $\mathbb{TO}$, the challenge-equal ciphertext table $\mathbb{CE}$ and the challenge-equal ciphertext corruption table $\mathbb{IC}$ accordingly.
  - The challenger sets $\mathbb{TO}(i + 1)$ as true to mark the event that some update token of certain challenge-equal ciphertexts for epoch i has been leaked to the adversary.

  - The challenger automatically updates all the challenge-equal ciphertexts with header same to $\tilde{c}$ in $\mathbb{CE}(i)$ from epoch i to the current epoch e. Particularly, the challenger iteratively runs ReKeyGen and Recrypt algorithm to update these ciphertexts by epoch, while archiving all generated challenge-equal ciphertexts along the way to the corresponding rows of $\mathbb{CE}$.
  - Update the table $\mathbb{IC}$ to mark the epochs in which the adversary may see challenge-equal ciphertexts as follows: for each ℓ from i to e, if $\mathbb{IC}(\ell) \wedge \mathbb{TO}(\ell+1) = \mathsf{true}$, then set $\mathbb{IC}(\ell+1)$ set as true. Moreover, for most existing CDUE schemes [25, 124], given the updated ciphertext in the second epoch, the corresponding token from the first epoch to the second epoch, and the header of ciphertext in the first epoch, it is not difficult to recover the complete ciphertext in the second epoch. This property is called the *bi-directional update* by Everspauph et al., which also should be taken into consideration for the game winning condition. Hence for any ℓ decreasing from $i+1$ to 0, if $\mathbb{IC}(\ell) \wedge \mathbb{TO}(\ell) = \mathsf{true}$, we let the challenger set $\mathbb{IC}(\ell-1)$ as true.

- *Challenge-equal ciphertexts' header oracle* $\mathcal{O}_{\mathsf{Header}}(i)$: This oracle is used to acquire the header of the challenge-equal ciphertext in the key corrupted epoch i. When queried with the epoch index i, the challenger will return all the headers of the challenge-equal ciphertexts in $\mathbb{CE}$.
- *Challenge-equal ciphertexts oracle* $\mathcal{O}_{\mathsf{ChallengeCT}}(i)$: This oracle is used to acquire the existing challenge-equal ciphertexts in the epoch i. When queried with the epoch index i, the challenger will return all the challenge-equal ciphertexts in the row $\mathbb{CE}(i)$ and update the challenge-equal ciphertext corruption table $\mathbb{IC}$ to mark the leakage of challenge-equal ciphertexts as following:
  - Set $\mathbb{IC}(i)$ as true to mark the leakage of challenge-equal ciphertexts in epoch i.
  - For any ℓ from $i+1$ to e, if $\mathbb{IC}(\ell-1) = \mathsf{true} \wedge \mathbb{TO}(\ell) = \mathsf{true}$, then set $\mathbb{IC}(\ell)$ as true to mark the leakage of the challenge-equal ciphertexts that may be updated by the adversary herself via leaked tokens.

  - For any ℓ from i to 1, if $\mathbb{IC}(\ell) \wedge \mathbb{TO}(\ell) = \mathsf{true}$, then set $\mathbb{IC}(\ell-1)$ as true to mark the leakage of former challenge-equal ciphertexts that may be recovered by the adversary herself via leaked tokens and the bi-directional update property.[6]

- *Re-encryption oracle* $\mathcal{O}_{\mathsf{ReEnc}}(i, C)$: This oracle is used to update any ciphertexts of the epoch i to the current epoch. As considering the adversary may query the oracle $\mathcal{O}_{\mathsf{ReEnc}}$ with maliciously generated ciphertexts, the oracle $\mathcal{O}_{\mathsf{ReEnc}}$ is allowed to return $\perp$ according to the scheme specification, which is different with the previous works [25, 27, 126]. Specifically, when $\mathcal{O}_{\mathsf{ReEnc}}$ is queried with a ciphertext C and an epoch index i, the challenger defines $C_i = (\tilde{c}_i, \bar{c}_i)$ as $C = (\tilde{c}, \bar{c})$, and iteratively runs token generation algorithm $\Delta_{k_i,k_{i+1},\tilde{c}_l} \leftarrow\$ \mathsf{ReKeyGen}(k_l, k_{l+1}, \tilde{c}_l)$ and the re-encryption algorithm $C_{l+1} \leftarrow \mathsf{Recrypt}(\Delta_{k_i,k_{i+1},\tilde{c}_l}, C_l)$ for all integers $l \in [i, e)$. If all $\mathsf{Recrypt}$ procedures are carried out successfully, the challenger will return the generated C_e to the adversary. Moreover, if the queried ciphertext $C \in \mathbb{CE}$ (i.e., it is the challenge-equal ciphertext), the challenger will update the tables $\mathbb{IC}$ and $\mathbb{CE}$ accordingly:
  - For all $l \in [i, e)$, the challenger archives the newly generated challenge-equal ciphertext C_l in $\mathbb{CE}(l)$.
  - The challenger sets $\mathbb{IC}(e)$ as true to mark the leakage of the challenge-equal ciphertext in epoch e.
  - Additionally, the challenger may have to go backward and update the entry $\mathbb{IC}(l)$ for the epochs before e. This is because given the challenge-equal ciphertext of the epoch e, the adversary may recover the former challenge-equal ciphertext via the leaked tokens and the bi-directional update property. Specifically, for l start decreasing from e, the challenger sets $\mathbb{IC}(l-1) = \mathsf{true}$ until he finds $\mathbb{IC}(l) \wedge \mathbb{TO}(l) = \mathsf{false}$.

sUP-IND-CPA vs. UP-IND. Note that even our sUP-IND-CPA security is stronger than UP-IND [25] in following aspects. Firstly, sUP-IND-CPA can characterize the post-compromise

[6] For simplicity, we assume that if the adversary can acquire one of the challenge-equal ciphertext in the epoch e, she can automatically get all other challenge-equal ciphertexts in the same epoch.

security which is ignored in UP-IND. Although the constructions in [25, 124] is post-compromise secure, there do exist constructions (see inSec 5.3.2) which are UP-IND secure but without the post-compromise security. Secondly, unlike sUP-IND-CPA, UP-IND does not allow the adversary to query the re-encryption oracle with malformed ciphertexts with the same header as the challenge ciphertext. Therefore, the KSS scheme in [25] is proved secure under UP-IND, but can be attacked by maliciously re-encrypting a forged ciphertext with the same header of the challenge ciphertext to a key corrupted epoch. In this way, the adversary can somehow compute the challenge-equal ciphertext that he is not supposed to see in a key corrupted epoch. The detailed attack is shown in Sec 5.3.1.

Bi-directional update. Given the previous update token and the former ciphertext header, we assume that one can reversely downgrade a ciphertext to a previous epoch. This property is naturally satisfied by the two constructions **KSS** and **ReCrypt** in [25]. Therefore, for fully capturing the challenge-equal ciphertext corruption to avoid trivial win, the challenger needs to update the challenge-equal ciphertext corruption table $\mathbb{IC}$ forward and backward whenever a challenge-equal ciphertext or token is corrupted. This backward inference should have appeared in the model of [25], but due to the inherent limitation of their model, the challenge-equal ciphertext that the adversary can see is always directly updated from a key-uncorrupted epoch. So this negligence has not been fully reflected in their paper.

5.4.2 Integrity

Then we describe our model sUP-INT-CTXT for CDUE. Like our sUP-IND-CCA model, our integrity model strengthens the UP-INT model in [25] in the sense that allowing the adversary to query the ReEnc oracle with maliciously generated ciphertexts and introducing the epoch notion to capture the post-compromise security. Similar to our confidential models, the challenger needs to maintain table $\mathbb{K}$ to record generated secret keys, and table $\mathbb{KC}$ to keep track of the adversary's key corruption commitment. Besides, the challenger also needs to maintain the following trivially obtained ciphertexts table $\mathbb{T}$ especially for the sUP-INT-CTXT model.

- Table $\mathbb{T}$ is used to keep track of ciphertexts that the adversary can trivially obtain. These ciphertexts are acquired by adversary from three sources: 1) directly response from the $\mathcal{O}_{\mathsf{Enc}}$ oracle, 2) response from the $\mathcal{O}_{\mathsf{ReEnc}}$ oracle, and 3) derived by the adversary herself from querying ciphertexts and update tokens. Specifically, its rows are indexed by the epoch index and ciphertext header pairs $(i, \tilde{c})$, and entries are the header's associated ciphertext body $\bar{c}$. To make the definition more general, we allow $\mathbb{T}(i, \tilde{c})$ to include multiple ciphertext bodies $\bar{c}$ associated to the same header. All entries will be set $\perp$ when the game start.

Specifically, we define the sUP-INT-CTXT experiment as Fig. 5.3. Similar to [126], we only accept forgeries that the adversary makes in the current and final epoch e_{end}, but not in the past. This matches the concept of UE where the secret keys and update tokens of old epochs will (ideally) be deleted, and thus a forgery for an old key is meaningless anyway. The experiment requests the adversary, after engaging with the oracles $\mathcal{O}_{\mathsf{Enc'}}$, $\mathcal{O}_{\mathsf{Dec}}$, $\mathcal{O}_{\mathsf{Token'}}$, $\mathcal{O}_{\mathsf{Next}}$, $\mathcal{O}_{\mathsf{KeyCorrupt}}$ and $\mathcal{O}_{\mathsf{ReEnc'}}$, to generate a new legal ciphertext C^* for the current epoch. The adversary wins if the two requirements hold simultaneously. One is the new ciphertext C^* can be successfully decrypted by the current epoch key k_e. The other is that C^* is not a trivial win, i.e. the ciphertext C^* is not in the trivially obtained table ciphertext table $\mathbb{T}$ and the current epoch key k_e has not been corrupted.

During the sUP-INT-CTXT experiment, the challenger's behaviours to response the oracles $\mathcal{O}_{\mathsf{Dec}}$, $\mathcal{O}_{\mathsf{Next}}$ and $\mathcal{O}_{\mathsf{KeyCorrupt}}$ are similar to the sUP-IND-CCA experiment. However, there are three different oracles $\mathcal{O}_{\mathsf{Enc'}}$, $\mathcal{O}_{\mathsf{Token'}}$ and $\mathcal{O}_{\mathsf{ReEnc'}}$ in sUP-INT-CTXT that require the challenger to update the table $\mathbb{T}$ accordingly in order to track the ciphertexts that adversary can trivially obtained via oracle queries.

- *Encryption oracle* $\mathcal{O}_{\mathsf{Enc'}}(m)$: This oracle is used to query the encryption of the message m under the current epoch key k_e. Specifically, the challenger will return $\mathsf{Enc}(k_e, m)$ to the adversary. Also he will parse the ciphertext $\mathsf{Enc}(k_e, m) = (\tilde{c}, \bar{c})$ and update the table $\mathbb{T}$ as $\mathbb{T}(e, \tilde{c}) \leftarrow \bar{c}$.

$Exp^{\mathcal{A}}_{\mathsf{sUP\text{-}INT\text{-}CTXT}}(\lambda)$

1 : $pp \leftarrow\$ \mathsf{Setup}(\lambda)$
2 : Initialize $e, \mathbb{K}, \mathbb{T}, \mathbb{KC}$
3 : $k_0 \leftarrow \mathsf{KeyGen}(pp)$; $\mathbb{K}(0) \leftarrow k_0$
4 : $C^* = (\tilde{c}^*, \bar{c}^*) \leftarrow\$ \mathcal{A}^{\mathcal{O}_{\mathsf{Enc'}}, \mathcal{O}_{\mathsf{Next}}, \mathcal{O}_{\mathsf{KeyCorrupt}}, \mathcal{O}_{\mathsf{ReEnc'}}, \mathcal{O}_{\mathsf{Token'}}}$
5 : **if** $(\mathsf{Decrypt}(k_e, C^*) \neq \perp) \wedge (\bar{c}^* \notin \mathbb{T}(e, \tilde{c}^*)) \wedge (\mathbb{KC}(e) \neq \mathsf{true})$
6 : **return** 1
7 : **else return** 0

FIGURE 5.3. The sUP-INT-CTXT experiment.

- *Re-encryption oracle* $\mathcal{O}_{\mathsf{ReEnc'}}(i, C)$: This oracle is used to update any ciphertexts of the epoch i to the current epoch like $\mathcal{O}_{\mathsf{ReEnc}}$ in sUP-IND-CCA. When the oracle $\mathcal{O}_{\mathsf{ReEnc'}}$ is queried with an epoch index i and a ciphertext C, if the challenger can successfully update C to $C' = (\hat{c}', \bar{c}')$ of the current epoch e, he will return C' to the adversary. Additionally, $\bar{c}'$ will be added to $\mathbb{T}(e, \hat{c}')$.
- *Token corrupt oracle* $\mathcal{O}_{\mathsf{Token'}}(i, \bar{c})$: When the oracle $\mathcal{O}_{\mathsf{Token'}}$ is queried with an epoch index i and a ciphertext header $\tilde{c}$ during the sUP-INT-CTXT experiment, the challenger will return $\perp$ if $\mathbb{KC}(i) = \mathsf{true}$, otherwise the challenger will run the token generation algorithm $\Delta_{i,i+1,\tilde{c}} \leftarrow\$ \mathsf{ReKeyGen}(k_i, k_{i+1}, \tilde{c})$ and return the token $\Delta_{i,i+1,\tilde{c}}$ to adversary $\mathcal{A}$. If $\Delta_{i,i+1,\tilde{c}}$ is not $\perp$, the challenger will updates the trivially obtained ciphertext $\mathbb{T}$ accordingly: for all ciphertext bodies $\bar{c} \in \mathbb{T}(i, \tilde{c})$, the challenger will automatically generate the corresponding ciphertext $C' \leftarrow \mathsf{Recrypt}(\Delta_{k_i,k_{i+1},\tilde{c}}, (\tilde{c}, \bar{c}))$ for next epoch, parse $C' = (\tilde{c}', \bar{c}')$ and record them in the row $\mathbb{T}(i, \tilde{c}')$.

DEFINITION 5.4.2 (sUP-INT-CTXT). Define the sUP-INT-CTXT experiment as Fig. 5.3. An updatable encryption scheme is called sUP-INT-CTXT secure if for any P.P.T. adversary $\mathcal{A}$ the following advantage

$$\mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\mathcal{A}} := \Pr[Exp^{\mathcal{A}}_{\text{sUP-INT-CTXT}}(\lambda) \Rightarrow 1]$$

is negligible in the security parameter λ.

Note that any token corruption is disallowed from a key corrupted epoch to a key uncorrupted epoch in the sUP-INT-CTXT model, as well as in the existing models [25, 124] for ciphertext integrity. Since in a key corrupted epoch, the adversary can generate any ciphertext, and the challenger does not know which ciphertexts the header used to query the $\mathcal{O}_{\mathsf{Token}}$ oracle is corresponding to. Thus, such attack should be restricted in the ciphertext integrity game. We also know that in the message confidentiality models, sUP-IND-CPA and sUP-IND-CCA, the adversary is allowed to query any token except for the challenge-equal ciphertext from the key corrupted epoch to the key uncorrupted epoch in which the challenge-equal ciphertext is corrupted. Such a difference also cause that the combination of sUP-IND-CPA security and sUP-INT-CTXT security is not sufficient to imply the sUP-IND-CCA security, which we will discuss in the next subsection.

5.4.3 sUP-IND-CPA + sUP-INT-CTXT $\not\Rightarrow$ sUP-IND-CCA

It is widely known that for the authenticated encryption, the IND-CPA security plus the INT-CTXT security imply the IND-CCA security [129]. This implication still holds for CIUE[125]. However, the case for CDUE is different. More interestingly, we find this particularity is inherent for general CDUE, since even under weaker security models, this implication does not work either, including under a weaken version of our models without malicious update and under existing models in [25, 124] which do not capture post-compromise security or malicious update security. In the following, we will show a special CDUE scheme which is sUP-IND-CPA and sUP-INT-CTXT secure but not sUP-IND-CCA secure. Our counter example is inspired by our own construction **ReCrypt**$^+$, but we believe it can be generalized to a large class of CDUE schemes.

This counterintuitive gap comes from the fact that querying $\mathcal{O}_{\mathsf{Token}}$ from a key-corrupted epoch to a key-uncorrupted epoch is forbidden during the sUP-INT-CTXT game, but the adversary in the sUP-IND-CCA game has the ability to acquire that kind of tokens for non-challenge-equal ciphertexts. Such token queries in sUP-INT-CTXT are forbidden, since in a key corrupted epoch the header used to query the $\mathcal{O}_{\mathsf{Token}}$ oracle is unknown to the challenger.

Thus an sUP-IND-CCA adversary can leverage such tokens and the decryption oracle to launch attacks.

Intuitively, if an updating token contains secret information which can be leveraged by the adversary who knows the previous epoch key, the adversary may be able to modify the challenge-equal ciphertext and use the result to query the decryption oracle to get more information about the challenge ciphertext. More precisely, we add the ciphertext header of the new scheme with a redundant MAC, and make the encryption of the MAC key contained in the token. If the adversary corrupt the key of the former epoch and query a token for a non-challenge-equal ciphertext from that epoch, she can learn the MAC key and modify the MAC in the next epoch challenge-equal ciphertext. After that, she may query the modified challenge-equal ciphertext to the decryption oracle. Note that this attack even does not leverage the malicious re-encryption ability!

Suppose the **CDUE** is the CDUE scheme which is both sUP-IND-CPA and sUP-INT-CTXT secure. Moreover, CDUE has a special property: the update token $\Delta_{i,\tilde{c}_i}$ must explicitly contain the header $\tilde{c}_{i+1}$ of the new ciphertext in epoch $i+1$. Such a property is satisfied by most CDUE schemes, say **KSS** and **ReCrypt** in [25] and our **ReCrypt**$^+$in Section 5.5.

Let **SKE** $=$ $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CPA secure symmetric key encryption. Let **MAC** $=$ $(\mathsf{KeyGen}, \mathsf{Tag}, \mathsf{Verify})$ be a deterministic MAC scheme which is unforgerable under chosen message attack (e.g. hash-based MACs). Note that the deterministic property guarantees that there is only one valid MAC for each message under one secret key. Then we construct the scheme **CDUE**$'$ as follows:

- **CDUE**$'$.$\mathsf{Setup}(1^\lambda)$: Generate the public parameter pp via **CDUE**.Setup.
- **CDUE**$'$.$\mathsf{KeyGen}(pp)$: Use **CDUE**.KeyGen to generate an epoch key k_e of **CDUE** and use **MAC**.KeyGen to generate a MAC key mk_e. The new epoch key k'_e of **CDUE**$'$ is (k_e, mk_e).
- **CDUE**$'$.$\mathsf{Encrypt}(k'_e, m)$: Parse the secret key $k'_e = (k_e, mk_e)$. Given the plaintext m, firstly use **CDUE**.Enc to encrypt m under the secret key k_e and generate the ciphertext $C_e = (\tilde{c}_e, \bar{c}_e)$. Secondly, concatenate the header $\tilde{c}_e$ with one bit 1 and

compute a MAC $\tau_e = \textbf{MAC}.\mathsf{Tag}(mk_e, \tilde{c}_e \| 1)$. Finally, output the ciphertext $C'_e = (\tilde{c}'_e, \bar{c}_e)$ where the new header $\tilde{c}'_e = (\tilde{c}_e, \tau_e)$.

- $\textbf{CDUE}'.\mathsf{Decrypt}(k'_e, C'_e)$: Parse $C'_e = (\tilde{c}'_e, \bar{c}_e)$ where $\tilde{c}'_e = (\tilde{c}_e, \tau_e)$. Verify whether $\textbf{MAC}.\mathsf{Verify}(mk_e, \tau_e, \tilde{c}_e \| 1) = 1$ or $\textbf{MAC}.\mathsf{Verify}(mk_e, \tau_e, \tilde{c}_e \| 0) = 1$. If one of above two cases is true, use the $\textbf{CDUE}.\mathsf{Decrypt}$ to decrypt the ciphertext $C_e = (\tilde{c}_e, \bar{c}_e)$ and return the decryption result.
- $\textbf{CDUE}'.\mathsf{ReKeyGen}(k'_e, k'_{e+1}, \tilde{c}'_e)$: Parse $\tilde{c}'_e = (\tilde{c}_e, \tau_e)$, $k'_e = (k_e, mk_e)$ and $k'_{e+1} = (k_{e+1}, mk_{e+1})$. Firstly, verify whether $\textbf{MAC}.\mathsf{Verify}(\tau_e, \tilde{c}_e \| 1) = 1$. If it is true, invoke $\textbf{CDUE}.\mathsf{ReKeyGen}(k_e, k_{e+1}, \tilde{c}_e)$ to generate the token $\Delta_{e,\tilde{c}_e}$. Note that according to our assumption about **CDUE**, $\Delta_{e,\tilde{c}_e}$ has the form $(\tilde{c}_{e+1}, \delta_{e,\tilde{c}_e})$ where $\tilde{c}_{e+1}$ is the new header and $\delta_{e,\tilde{c}_e}$ denotes the other information. Secondly, compute the new MAC $\tau_{e+1} = \textbf{MAC}.\mathsf{Tag}(mk_{e+1}, \tilde{c}_{e+1} \| 1)$ and the new header $\tilde{c}'_{e+1} = (\tilde{c}_{e+1}, \tau_{e+1})$. Finally, encrypt mk_{e+1} under the key k_e as $\textbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1})$, and output the update token $\Delta'_{e,\tilde{c}'_e} = \big(\tilde{c}'_{e+1}, \delta_{e,\tilde{c}_e}, \textbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1})\big)$ for $\textbf{CDUE}'$.
- $\textbf{CDUE}'.\mathsf{ReEncrypt}(\Delta'_{e,\tilde{c}'_e}, C'_e)$: First parse the token $\Delta'_{e,\tilde{c}'_e} = (\tilde{c}'_{e+1}, \delta_{e,\tilde{c}_e}, \textbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1}))$ and the ciphertext $C'_e = (\tilde{c}'_e, \bar{c}_e) = ((\tilde{c}_e, \tau_e), \bar{c}_e)$. Then derive the **CDUE** token $\Delta_{e,\tilde{c}_e} = (\tilde{c}_{e+1}, \delta_{e,\tilde{c}_e})$ from $\Delta'_{e,\tilde{c}'_e}$, and $C_e = (\tilde{c}_e, \bar{c}_e)$ from C'_e. Invoke $\textbf{CDUE}.\mathsf{ReEncrypt}(\Delta_{e,\tilde{c}_e}, C_e)$ to get $C_{e+1} = (\tilde{c}_{e+1}, \bar{c}_{e+1})$. Finally output $C'_{e+1} = (\tilde{c}'_{e+1}, \bar{c}_{e+1})$ by replacing $\tilde{c}_{e+1}$ with the new header $\tilde{c}'_{e+1}$ in the token $\Delta'_{e,\tilde{c}'_e}$.

In the following two lemmas, we show that the above $\textbf{CDUE}'$ is sUP-IND-CPA and sUP-INT-CTXT secure when **MAC** is deterministic (like HMAC [130]). The sUP-IND-CPA is obvious since the augmented MAC will not leak any information about the plaintext. Since the CTXT model disallows the adversary to see the token from a key-corrupted epoch to a key-uncorrupted epoch, the MAC key will never be leaked. The sUP-INT-CTXT comes from the MAC's unforgerability.

LEMMA 1. *If **CDUE** is sUP-IND-CPA secure and **MAC** is deterministic (i.e. there is only one valid MAC for each message under one secret key), $\textbf{CDUE}'$ is sUP-IND-CPA secure.*

PROOF. This proof can be done by running a **CDUE**′ adversary $\mathcal{A}$ to construct a **CDUE** adversary $\mathcal{B}$ in the sUP-IND-CPA game. When $\mathcal{A}$ asks any oracle $\mathcal{O}_{\mathsf{Enc}}$, $\mathcal{O}_{\mathsf{Next}}$, $\mathcal{O}_{\mathsf{KeyCorrupt}}$, $\mathcal{O}_{\mathsf{ReEnc}}$, $\mathcal{O}_{\mathsf{Token}}$, $\mathcal{O}_{\mathsf{Header}}$ or $\mathcal{O}_{\mathsf{ChallengeCT}}$, $\mathcal{B}$ will query the same oracle to the **CDUE** challenger with corresponding inputs. Moreover, $\mathcal{B}$ generates a sequence of MAC key $mk_0, \ldots, mk_e$, and uses them to generate MAC tags in the header of the **CDUE**′ ciphertexts. Since there is only one valid MAC for each message under same secret key, there is a one-to-one correspondence between the **CDUE**′ header $\tilde{c}'$ queried by $\mathcal{A}$ and the **CDUE** header $\tilde{c}$ queried by $\mathcal{B}$ for the inputs of $\mathcal{O}_{\mathsf{Token}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$. This is because the MAC τ in $\tilde{c}' = (\tilde{c}, \tau)$ is exactly determined by its key mk and the value of $\tilde{c}$. That means for every oracle query made by $\mathcal{A}$ there is a correspondence query made by $\mathcal{B}$, so the view of $\mathcal{A}$ can be easily simulated by $\mathcal{B}$ by adding a simulated MAC. □

LEMMA 2. *If **CDUE** is sUP-INT-CTXT secure, **SKE** is IND-CPA secure and **MAC** is multi-user CMA unforgerable, then **CDUE**′ is sUP-INT-CTXT secure.*

PROOF. Here we show a brief proof idea with a sequence of games. Game G0 is exactly an sUP-INT-CTXT game between the adversary $\mathcal{A}$ and the challenger $\mathcal{C}$. In Game G1, challenger $\mathcal{C}$ changes its token generation process. Instead, for all tokens not between two key corrupted epochs, $\mathcal{C}$ replaces encrypting the MAC key with encrypting a random selected string (same length with the MAC key). To state $\mathcal{A}$ cannot distinguish G1 from G0, we need to consider two aspects. The one is in which case $\mathcal{A}$ can see this part and the other is whether the corruption of this part helps $\mathcal{A}$ distinguish G0 and G1. For the first aspect, we know that actually this part of token only exists on the token but never participates the re-encryption of the ciphertext. So $\mathcal{A}$ has exact one way to get this part, that is corrupting the token. However, $\mathcal{A}$ can only obtain the tokens from key uncorrupted epochs (this is the inherent restriction of sUP-INT-CTXT security), which means this part of token is indistinguishable as its encryotion key (a.k.a. the key of the starting epoch) is unknown. Thus, the indistingushability of G0 and G1 can be reduced to the real or random security (proved equivalent with IND-CPA security) of the underlying **SKE** scheme. In Game G2, challenger $\mathcal{C}$ finishes the mac generation via querying the multi-user MAC challenger with the same message. Here multi-user challenger has all the honest MAC keys for those key-uncorrupted epochs and answers MAC queries

with the specified key (indexed like epoch). Challenger also replaces generating the **CDUE** components by querying the **CDUE** challenger. G1 and G2 are essentially same, so they are indistinguashable. For the challenge query, $\mathcal{C}$ will parse the ciphertext to get a **CDUE** ciphertext and a message-tag pair. Then $\mathcal{C}$ submits them to the corresponding challenger, respectively. If there is at least one challenger returns true to $\mathcal{C}$, $\mathcal{C}$ will return true to the $\mathcal{A}$. In this way, we reduce the security of the multi-user CMA unforgeability of **MAC** and sUP-IND-CTXT of **CDUE**. □

The CCA attack. We provide a CCA attack as follows. The adversary commits to corrupt the key of the epoch e, but will not corrupt the key of the epoch $e+1$. Then the adversary queries a token of non-challenge ciphertext header $\tilde{c}_{e,0}$, and she will get a token $\Delta'_{e,\tilde{c}'_{e,0}} = \big(\tilde{c}'_{e+1,0}, \delta_{e,\tilde{c}_{e,0}}, \mathbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1})\big)$. Since the key $k'_e = (k_e, mk_e)$ has been corrupted by the adversary, she can recover mk_{e+1} for $\mathbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1})$ easily. Then the adversary acquires the challenge-equal ciphertext $C'_{e+1,1} = ((\tilde{c}_{e+1,1}, \tau_{e+1,1}), \bar{c}_{e+1,1})$ in the epoch $e+1$, where $\tau_{e+1,1} = \mathbf{MAC}_{mk_{e+1}}(\tilde{c}_{e+1,1}\|1)$. Since the adversary knows mk_{e+1}, she can modify $C'_{e+1,1}$ into a new ciphertext $C'_{e+1,2} = ((\tilde{c}_{e+1,1}, \tau'_{e+1}), \bar{c}_{e+1,,1})$ by shifting the attached bit in the MAC message and acquiring $\tau'_{e+1} = \mathbf{MAC}_{mk_{e+1}}(\tilde{c}_{e+1,1}\|0)$. According to the design of our decryption algorithm, τ'_{e+1} still can pass the verification even the attached bit is 0 but not 1. So $C'_{e+1,2}$ is still a valid ciphertext of the epoch $e+1$, and it will not be recognized as a challenge-equal ciphertext by the sUP-IND-CCA challenger. The adversary can query $\mathcal{O}_{\mathsf{Dec}}$ with $C'_{e+1,2}$ in the epoch $e+1$, and learn the challenge bit. Therefore, we have the following theorem.

THEOREM 5.4.1. For **CDUE**, the security combination of sUP-IND-CPA and sUP-INT-CTXT cannot imply sUP-IND-CCA security.

The gap is inherent. One may be curious about whether the counter-intuitive gap is caused by the malicious update resistance or the post-compromise security. However, we find that the gap between the CPA+CTXT and CCA is inherent for general CDUE. To note that, firstly we show the implication does not hold for a weaker collection of our models (we call UP-IND-CPA, UP-INT-CTXT and UP-IND-CCA that follow the former paradigm but have

a restriction to the re-encryption oracle), which only capture the post-compromise security but not malicious update security. Then we have the following Theorem 5.4.2. The intuition comes from that the CCA attack on our artificially designed **CDUE**$'$ scheme does not need to query malicious ciphertexts on the re-encryption oracle. Moreover, the security gap holds even for the weakest models[7] in [25, 124] without the post-compromise security or the malicious update resistance. Indeed, it is not hard to see that the above **CDUE**$'$ is also UP-IND and UP-INT secure, while the CCA attack can still apply.

THEOREM 5.4.2. For a ciphertext dependent UE, the security combination of UP-IND-CPA and UP-INT-CTXT do not imply UP-IND-CCA security.

5.4.4 Update unlinkability

We now present the model sUP-REENC-CCA for updatable encryption with update unlinkability, which improve upon the UP-IND model [25] by giving adversary access to the decryption oracle.

DEFINITION 5.4.3 (sUP-REENC-CCA). An updatable encryption scheme is called sUP-REENC-CPA secure if for any P.P.T adversary $\mathcal{A}$ the following advantage:

$$\mathsf{Adv}^{\text{sUP-REENC-CCA}}_{\mathcal{A}} := \Pr[Exp^{\mathcal{A}}_{\mathsf{sUP\text{-}REENC\text{-}CCA}}(\lambda) \Rightarrow 1] - \frac{1}{2}$$

is negligible in the security parameter λ.

Perfect re-encryption. To demonstrate that our **ReCrypt**$^+$ scheme is sUP-REENC-CCA secure, we firstly introduce a property called **perfect re-encryption** proposed in [126]. Perfect re-encryption assures that for any ciphertext of updatable encryption, decrypt-then-encrypt has the same distribution with re-encryption. We give a formal definition of perfect re-encryption for UE setting in the following.

DEFINITION 5.4.4 (Perfect re-encryption). **UE**=(Setup, KeyGen, Encrypt, Decrypt, ReKeyGen, Recrypt) is an updatable encryption where update process is probabilistic. We say

[7]The similar CCA model can be trivially obtained by adding an additional decryption oracle for ciphertexts decryption except for the challenge-equal ciphertexts.

$Exp^{\mathcal{A}}_{\mathsf{sUP\text{-}REENC\text{-}CCA}}(\lambda)$

1 : $pp \leftarrow\$\ \mathsf{Setup}(\lambda)$
2 : Initialize $e, \mathbb{K}, \mathbb{CE}, \mathbb{IC}, \mathbb{KC}, \mathbb{TO}$
3 : $k_1 \leftarrow \mathsf{KeyGen}(pp);\ \mathbb{K}(1) \leftarrow k_1$
4 : $(C_0, C_1, state) \leftarrow\$\ \mathcal{A}^{\mathcal{O}_{\mathsf{Enc}},\mathcal{O}_{\mathsf{Dec}},\mathcal{O}_{\mathsf{Next}},\mathcal{O}_{\mathsf{KeyCorrupt}},\mathcal{O}_{\mathsf{ReEnc}},\mathcal{O}_{\mathsf{Token}}}$
5 : Proceed only if $|C_0| = |C_1| \wedge m_0 \neq \perp \wedge m_1 \neq \perp$,
where $m_0 \leftarrow \mathsf{Dec}(e-1, C_0)$ and $m_1 \leftarrow \mathsf{Dec}(e-1, C_1)$
6 : $b \leftarrow\$\ \{0,1\};$ Parse $C_b = (\tilde{c}_b, \bar{c}_b)$
7 : $\Delta_{k_{e-1},k_e,\tilde{c}_b} \leftarrow\$\ \mathsf{ReKeyGen}(k_{e-1}, k_e, \tilde{c}_b)$
8 : $C^* \leftarrow \mathsf{Recrypt}(\Delta_{k_{e-1},k_e,\tilde{c}_b}, C_b)$
9 : Set $\mathbb{CE}(e) \leftarrow C^*$
10 : $b' \leftarrow\$\ \mathcal{A}(state)^{\mathcal{O}_{\mathsf{Enc}},\mathcal{O}_{\mathsf{Dec}},\mathcal{O}_{\mathsf{Next}},\mathcal{O}_{\mathsf{KeyCorrupt}},\mathcal{O}_{\mathsf{ReEnc}},\mathcal{O}_{\mathsf{Token}},\mathcal{O}_{\mathsf{Header}},\mathcal{O}_{\mathsf{ChallengeCT}}}$
11 : **for** $i = 1$ *to* e
12 : **if** $\mathbb{KC}(i) = \mathsf{true} \wedge \mathbb{IC}(i) = \mathsf{true}$ **then**
13 : **return** $\perp$
14 : **return** b'

FIGURE 5.4. The experiment for sUP-REENC-CCA

that the re-encryption (of **UE**) is perfect, if for all $pp \leftarrow\$\ \mathsf{Setup}(\lambda)$, all keys $k_i, k_{i+1} \leftarrow\$\ \mathsf{KeyGen}(pp)$, and all message m, we have

$$\mathsf{Encrypt}(k_{i+1}, m) \overset{dist}{\equiv} \mathsf{Recrypt}\left(\mathsf{ReKeyGen}(k_i, k_{i+1}, \tilde{c}), C\right),$$

where $C \leftarrow\$\ \mathsf{Encrypt}(k_i, m)$ and $C = (\tilde{c}, \bar{c})$.

LEMMA 3 (sUP-REENC-CCA security of UE). *Let **UE**=(Setup, KeyGen, Encrypt, Decrypt, ReKeyGen, Recrypt) be an updatable encryption with perfect re-encryption. Suppose that **UE** is sUP-IND-CCA secure. Then **UE** is sUP-REENC-CCA secure (via a tight reduction to sUP-IND-CCA).*

PROOF. Let $\mathcal{A}$ be the sUP-REENC-CCA adversary who plays with $\mathcal{B}$ and let $\mathcal{C}$ be the sUP-IND-CCA challenger(who palys with $\mathcal{B}$). $\mathcal{B}$ is constructed by simply forwarding (and recording) all oracle queries from $\mathcal{A}$ to $\mathcal{C}$ except for the challenge query from $\mathcal{A}$. In particular, they aways stay in the same epoch. When $\mathcal{A}$ submits a pair of ciphertexts C_0, C_1 of epoch $e-1$ as challenge to ask for either of two re-encryptions in epoch e, $\mathcal{B}$ is supposed to

embed his challenge to query $\mathcal{C}$ so that $\mathcal{A}$'s challenge answer can also be forwarded to $\mathcal{C}$ as $\mathcal{B}$'s challenge answer. $\mathcal{B}$ hopes to know the decryption of C_0 and C_1, and it has access to $\mathcal{O}_{\mathsf{Dec}}$ oracle. $\mathcal{B}$ first queries the $\mathcal{O}_{\mathsf{ReEnc}}$ oracle with the two ciphertexts C_0, C_1 to $\mathcal{C}$ and get the re-encryption C_0' and C_1' under the current key k_e. Then $\mathcal{B}$ queries the $\mathcal{O}_{\mathsf{Dec}}$ oracle with C_0', C_1' to get the decryption message m_0 and m_1. (Remember that $\mathcal{O}_{\mathsf{Dec}}$ oracle works always for the current epoch, the epoch e in this case.) Now $\mathcal{B}$ can issue m_0, m_1 as its own challenge message to $\mathcal{C}$ under epoch e. Thus $\mathcal{B}$ obtains $C_b^* \leftarrow\$ \mathsf{Encrypt}(k_e, m_b)$ where $\mathcal{C}$ picks $b \leftarrow\$ \{0,1\}$. Since **UE** has perfect re-encryption, the distribution of C_b^* is indistinguishable with $\mathsf{Recrypt}(\mathsf{ReKeyGen}(k_{e-1}, k_e, \tilde{c}_b), C_b)$. Consequently, $\mathcal{B}$ perfectly simulates the sUP-REENC-CCA game for $\mathcal{A}$. Finally, $\mathcal{B}$ simply forwards $\mathcal{A}$'s guess to $\mathcal{C}$as its guess. Due to perfect simulation, $\mathcal{B}$ wins the sUP-IND-CCA game if and only if $\mathcal{A}$ wins the sUP-REENC-CCA game.

$$\mathsf{Adv}_{\mathrm{UE}}^{\text{sUP-REENC-CCA}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathrm{UE}}^{\text{sUP-IND-CCA}}(\mathcal{B})$$

□

5.5 UE Construction with Strengthened Integrity

Next we describe our new CDUAE construction **ReCrypt**$^+$. Comparing with previous CDUAE constructions [25, 124], our scheme not only naturally inherits their advantage that the plaintext space could be a bit string with arbitrary length, but also has the strengthened security to resist the malicious re-encryption attack. During the security analysis, we prove our scheme secure under sUP-IND-CCA and sUP-INT-CTXT as above mentioned. So our scheme has a strengthened security in aspects of the post-compromise security, the malicious re-encryption resistance and the chosen ciphertexts attack resistance.

5.5.1 Construction framework

Our construction **ReCrypt**$^+$ follows the paradigm of the **ReCrypt** scheme proposed by Everspauph et al. The original **ReCrypt** in [25] not only follows the KEM + DEM with

the secret sharing structure, but also involves the key-homomophic PRF to achieve the re-encryption indistinguishability. However, as pointed by in the introduction, **ReCrypt** in [25] suffers the malicious re-encryption attack.

The key to resist the malicious re-encryption attack is to verify the validity of the ciphertext before re-encryption. Therefore our scheme not only involves the AEAD to enable the client to verify the header of the ciphertext, but also uses the collision-resistant homomorphic hash function and homomorphic commitment to help the server to check the consistency of the body with the header. These measures guarantee that the adversary always learns nothing when querying the ReEnc oracle with a forged ciphertexts. In the meantime, the homomorphic properties of the hash function and the commitment scheme make that the update operations to apply smoothly. The detailed construction is as follows, and also shown in Figure 5.5.

Let **HomHash**.Setup and **HomHash**.Eval be the algorithms of a homomorphic collision-resistant hash function with the following syntax.

DEFINITION 5.5.1. A homomorphic hash function H_{hom} is a linear function that maps vectors of starting group elements $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{G}_{HS}^n$ into one target group element $u \in \mathbb{G}_{HT}$ which is defined by the following two algorithms:

- **HomHash**.Setup(1^λ) : On input the security parameter λ, output an evaluation key hk;
- **HomHash**.Eval$(hk, \mathbf{v})$: On input the evaluation key hk and a vector of starting group elements $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{G}_{HS}^n$, output one target group element $u \in \mathbb{G}_{HT}$.

Hence fixed the evaluation key hk, we can write as

$$\mathsf{H}_{hom}(\mathbf{v}) = \textbf{HomHash}.\mathsf{Eval}(hk, \mathbf{v}) = u.$$

Specifically, it should satisfies the following properties:

- *Collision resistance:* the probability for any P.P.T adversary to generate the two vectors $\mathbf{v}$ and $\mathbf{v}'$ in $\mathbb{G}_{HS}^n$ which satisfy $\mathsf{H}_{hom}(\mathbf{v}) = \mathsf{H}_{hom}(\mathbf{v}')$ is negligible.
- *Homomorphism:* we have $\mathsf{H}_{hom}(\mathbf{v}) + \mathsf{H}_{hom}(\mathbf{v}') = \mathsf{H}_{hom}(\mathbf{v} + \mathbf{v}')$.

Setup(λ)

$hk \leftarrow\$ \textbf{HomHash}.\mathsf{Setup}(\lambda), \mathsf{hcom}.pp \leftarrow\$ \textbf{HCOM}.\mathsf{Init}(\lambda)$

return $(hk, \mathsf{hcom}.pp)$

KeyGen(λ)

$k \leftarrow\$ \textbf{AEAD}.\mathsf{KeyGen}(1^\lambda)$, **return** k

Encrypt(k, m)

Map $m \to (m_1, m_2, \ldots, m_n) \in \mathbb{G}_{PRF}^n, z \leftarrow\$ \mathbb{K}_{PRF}$

$d_i \leftarrow m_i + \mathsf{F}(z, i), d = (d_1, d_2, \ldots, d_n) \in \mathbb{G}_{PRF}^n, h \leftarrow \textbf{HomHash}.\mathsf{Eval}(hk, d)$

$y \leftarrow\$ \mathbb{K}_{PRF}, hcom \leftarrow \textbf{HCOM}.\mathsf{com}(y; hopen), x = z - y$

$ct \leftarrow\$ \textbf{AEAD}.\mathsf{Enc}(k, x, (h, hcom))$

$\tilde{c} = (ct, h, hcom),\ \bar{c} = (y, hopen, d)$ / ciphertext header and body

return $C = (\tilde{c}, \bar{c})$

Decrypt(k, C)

Parse $C = ((ct, h, hcom), (y, hopen, d))$

if $h == \textbf{HomHash}.\mathsf{Eval}(hk, d) \wedge \textbf{HCom}.\mathsf{Open}(hcom, y, hopen) == 1$ **then**

/ check the body is consistent with the header.

$x^\star \leftarrow \textbf{AEAD}.\mathsf{Dec}(k, \tilde{c}^1, (h, hcom))$

for $1 \le i \le n$ **do** $m_i^\star \leftarrow d_i - \mathsf{F}(x^\star - y, d_i)$ **return** $m^\star = m_1^\star, \ldots, m_n^\star$

return $\perp$

ReKeygen$(k, k', \tilde{c})$

Parse $\tilde{c} = (ct, h, hcom), m' \leftarrow \textbf{AEAD}.\mathsf{Dec}(k, ct, (h, hcom))$

if $m' \neq \perp$ **then** / check the returned header is valid.

$\Delta z \leftarrow\$ \mathbb{K}_{PRF},\ \Delta d_i \leftarrow \mathsf{F}(\Delta z, i),\ \Delta d = \Delta d_1, \Delta d_2, \ldots, \Delta d_n$

$h' \leftarrow h + \textbf{HomHash}.\mathsf{Eval}(hk, \Delta d),\ \Delta y \leftarrow\$ \{0,1\}^*$

$hcom' \leftarrow hcom + \textbf{HCom}.\mathsf{Com}(\Delta y, hopen_\Delta),\ x' = x + \Delta z - \Delta y,$

$ct' \leftarrow\$ \textbf{AEAD}.\mathsf{Enc}(k', x', (h', hcom')),\ \tilde{c}' = (ct', h', hcom')$

return $\Delta = (\tilde{c}', \Delta y, hopen_\Delta, \Delta z)$

else **return** $\perp$

ReEncrypt(C, Δ)

Parse $C = ((ct, h, hcom), (y, hopen, d)), \Delta = (\tilde{c}', (\Delta y, hopen_\Delta, \Delta z))$

if $\textbf{HCOM}.\mathsf{Open}(hcom, y, r) == 1 \wedge h == \textbf{HomHash}.\mathsf{Eval}(hk, d)$ **then**

/ check the body is consistent with the header.

$y' = y + \Delta y, \quad r' = r + \Delta r, \quad$ Parse $d = (d_1, d_2, \ldots, d_n)$

$d_i' \leftarrow d_i + \mathcal{F}(\Delta z, i), \quad d' = (d_1', d_2', \ldots, d_n')$

$hopen' = hopen + hopen_\Delta, \quad y' = y + \Delta y, \quad \bar{c}' = (y', hopen', d')$

return $C' = (\tilde{c}', \bar{c}')$

return $\perp$

FIGURE 5.5. The construction for ReCrypt$^+$.

Let $\mathsf{F} : \mathbb{K}_{PRF} \times \mathcal{M}_{PRF} \to \mathbb{G}_{PRF}$ be the key homomorphic PRF as described in Subsection 2.4, whose codomain is a cyclic group $\mathbb{G}_{PRF} \subseteq \mathbb{G}_{HS}$ and key space $\mathbb{K}_{PRF}$ is also an additive group. Let **HCOM**.Init, **HCOM**.Com and **HCOM**.Open be the algorithms for the homomorphic commitment scheme described in Subsection 2.3, whose message space, opening randomness space and commitment value are $\mathbb{M}_{COM}$, $\mathbb{O}_{COM}$ and $\mathbb{C}_{COM}$, respectively. Specifically, we require that the message space $\mathbb{M}_{COM}$ contains the PRF key space $\mathbb{K}_{PRF}$. Let the **AEAD**.KeyGen, **AEAD**.Enc and **AEAD**.Dec be the algorithms for AEAD as described in Subsection 2.2, whose key space, message space and ciphertext space are $\mathbb{K}_{AEAD}$, $\mathbb{M}_{AEAD}$ and $\mathbb{C}_{AEAD}$.

- **ReCrypt**$^+$.Setup(λ): Run the **HomHash**.Setup algorithm to generate the parameter hk for the homomorphic collision-resistant hash function. Also run the **HCOM**.Init to generate the parameter $\mathsf{hcom}.pp$ for the homomorphic commitment. The public parameter **ReCrypt**$^+$.pp=$(hk, \mathsf{hcom}.pp)$ will be taken as the implicit input of the following algorithm.
- **ReCrypt**$^+$.KeyGen(λ): Run the **AEAD**.KeyGen(λ) to generate the key of AEAD $k \in \mathbb{K}_{AEAD}$.
- **ReCrypt**$^+$.Encrypt(k, m): The algorithm proceeds as follows.
  (1) Map the message m into n group elements $m_1, m_2, \ldots, m_n \in \mathbb{G}_{PRF}^n$.
  (2) Use the key-homomorphic PRF to encrypt each block m_i. Specifically, sample a PRF key $z \in \mathbb{K}_{PRF}$ and then mask each message m_i as $d_i = m_i + \mathsf{F}(z, i) \in \mathbb{G}_{PRF}$.
  (3) Let $d = (d_1, d_2, \ldots, d_n) \in \mathbb{G}_{PRF}^n$. Since $d \in \mathbb{G}_{PRF}^n \subseteq \mathbb{G}_{HS}^n$, one can compute the homomorphic hash function on d and derive **HomHash**.Eval$(hk, d) = h \in \mathbb{G}$.
  (4) Randomly choose two shares $x, y \in \mathbb{K}_{PRF}$ of z such that $x + y = z$.
  (5) Use the homomorphic commitment scheme to commit the share y, and generate the commitment **HCom**.Com$(y, hopen) = hcom \in \mathbb{C}_{COM}$, where $hopen \in \mathbb{O}_{COM}$ is the corresponding opening randomness.

(6) Use the AEAD to encrypt the key share $x \in \mathbb{K}_{PRF} \subseteq \{0,1\}^{\lambda}$ with the auxiliary data the HCRH value $h \in \mathbb{G}_{HT} \subseteq \{0,1\}^{\lambda}$ and the homomorphic commitment $hcom \in \mathbb{C}_{COM} \subseteq \{0,1\}^{\lambda}$. Get the ciphertext $ct \in \mathbb{C}_{AEAD}$.

(7) The header of the UE ciphertext is $\tilde{c} = (ct, h, hcom) \in \mathbb{C}_{AEAD} \times \mathbb{G}_{HT} \times \mathbb{C}_{COM}$, and the body of the UE ciphertext $\bar{c} = (y, hopen, d) \in \mathbb{K}_{PRF} \times \mathbb{O}_{COM} \times \mathbb{G}_{PRF}^{n}$.

- **ReCrypt**$^{+}$.$\mathsf{Decrypt}(k, C)$: Given $k \in \mathbb{K}_{PRF}$ and the ciphertext $C = (\tilde{c}, \bar{c})$, the UE decryption algorithm first parses the ciphertext C as the header $\tilde{c} = (ct, h, hcom) \in \mathbb{C}_{AEAD} \times \mathbb{G}_{HT} \times \mathbb{C}_{COM}$ and the body $\bar{c} = (y, hopen, d) \in \mathbb{K}_{PRF} \times \mathbb{O}_{COM} \times \mathbb{G}_{PRF}^{n}$, and proceeds as follows:

  (1) Verify **HomHash**.$\mathsf{Eval}(hk, d) \stackrel{?}{=} h \in \mathbb{G}_{HT}$ for $d \in \mathbb{G}_{PRF}^{n} \subseteq \mathbb{G}_{HS}^{n}$,

  (2) Verify whether $hcom \in \mathbb{C}_{COM}$ is a valid commitment of $y \in \mathbb{K}_{PRF} \subseteq \mathbb{M}_{COM}$, so one invokes the homomorphic commitment opening algorithm **HCom**.$\mathsf{Open}(hcom, y, hopen)$ and check the results whether equals to 1.

  (3) Decrypt the AEAD ciphertext ct with the current epoch key k and the auxiliary data h and $hcom$.

  (4) If above verification passes and the AEAD decryption algorithm successfully outputs $x \in \mathbb{K}_{PRF}$, the UE decryption algorithm will recover all $m_i \in \mathbb{G}_{PRF}$ by computing $m_i = d_i - \mathsf{F}(x - y, i)$, otherwise it returns $\perp$.

- **ReCrypt**$^{+}$.$\mathsf{ReKeyGen}\ (k, \tilde{c})$: The algorithm first parses the header $\tilde{c} = (ct, h, hcom) \in \mathbb{C}_{AEAD} \times \mathbb{G}_{HT} \times \mathbb{C}_{COM}$, and proceeds as follows:

  (1) Use the currency secret key $k \in \mathbb{K}_{AEAD}$ to decrypt ct with the auxiliary data $(h, hcom)$. If the AEAD decryption successfully return $x \in \mathbb{K}_{PRF}$, execute following steps, otherwise return $\perp$.

  (2) Choose a random $\Delta z \in \mathbb{K}_{PRF}$, and compute $\Delta d_i = \mathsf{F}(\Delta z, i) \in \mathbb{G}_{PRF}$.

  (3) Let $\Delta d = (\Delta d_1, \ldots, \Delta d_n) \in \mathbb{G}_{PRF}^{n} \subseteq \mathbb{G}_{HS}^{n}$. Compute the new hash value $h' = h +$ **HomHash**.$\mathsf{Eval}(hk, \Delta d) \in \mathbb{G}_{HT}$.

  (4) Generate a new group element $\Delta y \in \mathbb{K}_{PRF}$ and its homomorphic commitment **HCom**.$\mathsf{com}(\Delta y, hopen_{\Delta}) = hcom_{\Delta} \in \mathbb{G}_{COM}$. So the new commitment is $hcom' = hcom + hcom_{\Delta}$.

(5) Compute $x' = x + \Delta z - \Delta y \in \mathbb{K}_{PRF}$. Encrypt x' with the new master key k' and auxiliary data $(h', hcom')$, and get the AEAD ciphertext $ct' = \textbf{AEAD}.\mathsf{Enc}(k', x', (h', hcom'))$.

(6) Let the new header $\tilde{c}' = (ct', h', hcom') \in \mathbb{C}_{AEAD} \times \mathbb{G}_{HT} \times \mathbb{C}_{COM}$. Return the update token $\Delta = (\tilde{c}', \Delta y, hopen_{\Delta}, \Delta z)$.

- **ReCrypt**$^{+}$.$\mathsf{ReEncrypt}(C, \Delta)$: The algorithm will first parse the ciphertext header $\tilde{c} = (ct, (h, hcom))$, the ciphertext body $\bar{c} = (y, hopen, d)$ and the update token $\Delta = (\tilde{c}', \Delta y, hopen_{\Delta}, \Delta z)$, then proceeds as follows.

  (1) Verify whether **HomHash**.$\mathsf{Eval}(hk, d) = h \in \mathbb{G}_{HT}$ for $d \in \mathbb{G}_{HS}^{n}$,

  (2) Verify whether $hcom \in \mathbb{G}_{COM}$ is a valid commitment of $y \in \mathbb{K}_{PRF}$, i.e., invoke the opening algorithm **HCom**.$\mathsf{Open}(hcom, y, hopen)$ and check the result whether equals to 1.

  (3) If above verification can be passed, compute $d' = (d'_1, d'_2, \ldots, d'_n) \in \mathbb{G}_{PRF} \subseteq R^n$ where $d'_i = d_i + \mathsf{F}(\Delta z, i) \in \mathbb{G}_{PRF}$.

  (4) Compute the new commitment opening $hopen' = hopen + hopen_{\Delta}$.

  (5) Compute $y' = y + \Delta y$.

  (6) Generate new ciphertext $C' = (\tilde{c}', \bar{c})'$ by taking $\tilde{c}'$ from the token Δ as the new header and setting $\bar{c}' = (y', hopen', d') \in \mathbb{K}_{PRF} \times \mathbb{O}_{COM} \times \mathbb{G}_{PRF}^{n}$.

5.5.2 Homomorphic hash functions from DDH groups

To make the following **ReCrypt**$^{+}$framework works, we should construct a homomorphic embedding from the range of the key homomorphic PRF into the domain of the collision-resistant hash function (i.e, $\mathbb{G}_{PRF} \to \mathbb{G}_{HS}$). Note that trivial dictionary maps do not work here, since we should make those homomorphic properties still hold. To handle this issue, we will involve a critical primitive named *the homomorphic hash function from DDH groups*. Previous homomorphic hash function schemes only allow the messages to be exponents [131, 132, 133] or short ring elements [134]. In contrast, we hope the message can be chosen from a group where the decisional Diffie-Hellman (DDH) problem is hard, since the domain of the hash function will be the range of the key-homomorphic PRF.

If there is not requirement for the message group $\mathbb{G}$, a homomorphic hash scheme is not hard to obtain. Chaum et al. have shown a homomorphic collision-resistant hash function can be constructed from an *exponential homomorphic hash* scheme [131, 132]. In their construction, $\mathbb{G}'$ is a finite cyclic group of order p. The public key hk contain $h_1, \ldots, h_n$ as generators of $\mathbb{G}'$. Let $\mathbb{G} = \mathbb{Z}_p$ be a group of exponents for $\mathbb{G}'$. For any positive integer n, $\mathsf{H}_{Hom} : \mathbb{G}^n \to \mathbb{G}'$ is defined as $\mathsf{H}_{hom}(v_1, \ldots, v_n) = \prod_{j=1}^{n} h_j^{v_j}$. The homomorphic property is easily verified, and collision resistance is implied by the discrete logarithm assumption in $\mathbb{G}'$.

However, in our construction **ReCrypt**$^+$, the DDH problem is required to be hard over $\mathbb{G}$, since $\mathbb{G}$ will be the range of the key-homomorphic pseudorandom function. The above exponential homomorphic hash construction does not trivially satisfy this requirement, since the operation over $\mathbb{G} = \mathbb{Z}_p$ is the *addition* but not the *multiplication*. To find the relation between a random element and a generator is easy in $\mathbb{G}$.

Our homomorphic hash function from DDH groups is based on a bilinear map over elliptic curves where the external Diffie-Hellman (XDH) assumption is hard. Specifically, the homomorphic function works on a bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ where p is a k-bit prime, $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are cyclic groups of order p and $e : \mathbb{G}_1 \times \mathbb{G}_2 \leftarrow \mathbb{G}_T$ is a non-degenerate bilinear map. The XDH assumption states that the Decisional Diffie Hellman (DDH) assumption is hard in the group $\mathbb{G}_1$ (not necessarily hard in $\mathbb{G}_2$). The XDH is believed to be true in asymmetric pairings generated using special MNT curves [135, 136].

So the message are chosen from the group $\mathbb{G}_1^n$, the algorithms of the homomorphic hash function are defined as follows.

- **HomHash**.$\mathsf{Setup}(\mathbb{G}, n)$: Pick at random $g \leftarrow \mathbb{G}_2 \backslash \{1\}$ and random elements $x_1, \ldots, x_n \leftarrow\!\!\$\ \mathbb{Z}_p$. Define $h_1 = g^{x_1}, \ldots, h_n = g^{x_n}$. Output $hk = (h_1, \ldots, h_n) \in \mathbb{G}_2^n$.
- **HomHash**.Eval $(hk, \mathbf{v})$: Given a key $hk = (h_1, \ldots, h_n) \in \mathbb{G}_2^n$ and a vector $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{G}_1^n$, output $\prod_{j=1}^{n} e(v_j, h_j) \in \mathbb{G}_T$.

For a fixed hk, $\mathsf{H}_{hom} : \mathbb{G}_1^n \to \mathbb{G}_T$ is defined as $\mathsf{H}_{hom}(\mathbf{v}) = \textbf{HomHash}.\mathsf{Eval}(hk, \mathbf{v})$.

The homomorphism can be easily verified. Suppose $\mathsf{H}_{hom}(\mathbf{v}) = \prod_{j=1}^{n} e(v_j, h_j)$ and $\mathsf{H}_{hom}(\mathbf{v}') = \prod_{j=1}^{n} e(v'_j, h_j)$, and we have

$$\mathsf{H}_{hom}(\mathbf{v}) \cdot \mathsf{H}_{hom}(\mathbf{v}') = \prod_{j=1}^{n} e(v_j, h_j) \cdot \prod_{j=1}^{n} e(v'_j, h_j) = \prod_{j=1}^{n} e(v_j v'_j, h_j).$$

The collision resistance is based on the double pairing assumption whose hardness is shown by Groth in [137]. The double pairing problem is given random elements $g_r, g_t \in \mathbb{G}_2$ to find a non-trivial couple $(r, t) \in \mathbb{G}_1^2$ such that $e(r, g_r)e(t, g_t) = 1$. Then we have the collision resistance lemma as follows:

LEMMA 4 (Collision resistance). *The double pairing assumption holds for the bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$. The homomorphic hash function H_{hom} defined as above is collision resistant.*

PROOF. We will show that if $\mathcal{A}$ has non-negligible probability of the collision resistance, then there is an algorithm $\mathcal{B}$ that breaks the double pairing assumption with at least non-negligible chance. Let (g_r, g_t) be a random double pairing challenge given to $\mathcal{B}$. If $g_r \neq 1$, $g_t \neq 1$ it selects $\rho_1, \tau_1, \ldots, \rho_n, \tau_n \leftarrow \mathbb{Z}_p$ and computes $h_1 = g_r^{\rho_1} g_t^{\tau_1}, \ldots, h_n = g_r^{\rho_n} g_t^{\tau_n}$. It runs $\mathcal{A}$ on $(h_1, \ldots, h_n) \in \mathbb{G}_2^n$ and with more than $\epsilon - 1/p$ probability it gets two different openings to the same commitment. If $\mathsf{H}_{hom}(\mathbf{v}) = \mathsf{H}_{hom}(\mathbf{v}')$, by the homomorphic property of the hash function we have $\prod_{j=1}^{n} e({v_j}^{-1} v'_j, h_j) = 1$. Define $\mu_i = {v_j}^{-1} v'_j$ for $j = 1, \ldots, n$, then we have at least one $\mu_j \neq 1$. This implies

$$\prod_{j=1}^{n} e\left(\mu_j, g_r^{\rho_j} g_t^{\tau_j}\right) = e\left(\prod_{j=1}^{n} \mu_j^{\rho_j}, g_r\right) e\left(\prod_{j=1}^{n} \mu_j^{\tau_j}, g_t\right).$$

So the element $r = \prod_{j=1}^{n} \mu_j^{\rho_j}$ and $t = \prod_{j=1}^{n} \mu_j^{\tau_j}$ is the answer of the double pairing problem.

□ □

5.5.3 Instantiation

To make the above framework works, we should construct a homomorphic embedding from the range of the key homomorphic PRF into the domain of the collision-resistant hash

function (i.e, $\mathbb{G}_{PRF} \rightarrow \mathbb{G}_{HS}$), as well as a homomorphism from the key space of $\mathbb{K}_{PRF}$ to the commitment message space $\mathbb{M}_{COM}$.

ReCrypt$^+$ can be instantiated over a bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ over elliptic curves where the external Diffie-Hellman (XDH) assumption and the double pairing assumption are hard. To handle the homomorphic embedding from $\mathbb{G}_{PRF}$ to $\mathbb{G}_{HS}$, we adopt the DDH based key-homomorphic PRF described in Subsection 2.4 over $\mathbb{G}_{PRF} = \mathbb{G}_1$ and $\mathbb{K}_{PRF} = \mathbb{Z}_p$, and the homomorphic hash function described in Subsection 5.5.2 over $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.

To handle the homomorphism from $\mathbb{K}_{PRF}$ to $\mathbb{M}_{COM}$, we adopt the Pedersen commitment over the group $\mathbb{G}_1$. The commitment scheme is specified with two random public group generators g and h in $\mathbb{G}_1$. The opening randomness $hopen$ is randomly chosen from $\mathbb{Z}_p$ and the commitment message m is also from $\mathbb{Z}_p$. The commitment is $\mathsf{Com}(m, hopen) = h^{hopen} g^m \in \mathbb{G}_1$. Since the PRF key space $\mathbb{K}_{PRF}$ and the commitment message $\mathbb{M}_{COM}$ are both $\mathbb{Z}_p^*$, the homomorphism is naturally inherent.

5.5.4 Security analysis

Now we show that our construction **ReCrypt**$^+$ is secure under the models sUP-IND-CCA, sUP-INT-CTXT and sUP-REENC-CCA with formal security proofs.

sUP-IND-CCA. We are now ready to state the sUP-IND-CCA security of our **ReCrypt**$^+$ scheme. Our security proof is similar to the **ReCrypt** except that 1) sUP-IND-CCA has $\mathcal{O}_{\mathsf{Dec}}$, 2) and allow to query malicious generated ciphertext to $\mathcal{O}_{\mathsf{ReEnc}}$ and malicious header to $\mathcal{O}_{\mathsf{Token}}$. Besides, 3) we put the commitment of the secret share of DEM key in the head. So the intuition of the security proof comes from: First of all, the authenticity of AEAD, the binding property of the commitment and the collision-resistance of the hash function guarantee that all ciphertexts that could be successful decrypted or reencrypted is honestly generated. Secondly, the authenticity of AEAD guarantee that all token is generated from honest generated ciphertext headers. Thirdly, the hiding property of the commitment can hide the secret share of DEM key y. Formally, we have the following theorem and proof.

THEOREM 5.5.1 (sUP-IND-CCA Security of ReCrypt$^+$). Let **ReCrypt**$^+$ be an updatable encryption scheme as defined in Section 5.5.1. **ReCrypt**$^+$is sUP-IND-CCA secure if **AEAD** is MU-RoR-AE secure (2.2), the homomorphic commitment **HCOM** is statistic hiding and computation binding, the homomorphic hash **HomHash** is collision resistant, and the key homomorphic PRF is pseudorandom.

PROOF. Our proof consists a sequence of games. Let G_0 be the sUP-IND-CCA game.

The behavior of the challenger in G_1 is similar to G_0, except the way to deal with the AEAD ciphertexts ct in the header: when queried with $\mathcal{O}_{\mathsf{Enc}}$,$\mathcal{O}_{\mathsf{Token}}$,$\mathcal{O}_{\mathsf{Next}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$ for all epoch that keys $k_e \notin \mathbb{KC}$ (i.e. the keys that will never be corrupted by the adversary), the challenger generates the CDUAE headers $\tilde{c} = (ct, h, hcom)$ by picking up a random AEAD ciphertext ct instead of invoking AEAD encryption. Additionally the challenger of G_1 will build a table $\mathbb{T}_{AEAD}$ to record all the tuples of the AEAD ciphertext, its plaintext and auxiliary data according to the epoch indexes. When querid with $\mathcal{O}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$ related to the epochs that keys $k_e \notin \mathbb{KC}$, instead of directly invoking the AEAD decryption algorithm, the challenger may recovering the AEAD plaintext though looking up the table $\mathbb{T}_{AEAD}$. The distinguishability of G_0 and G_1 can be reduced to the MU-ROR-AE security of the used AEAD scheme, since these AEAD secret keys will never be learned by the adversary.

In G_2, the challenger's behavior is similar to G_1, except that the way to deal with the commitment $hcom$ in the header: when queried with $\mathcal{O}_{\mathsf{Enc}}$,$\mathcal{O}_{\mathsf{Token}}$,$\mathcal{O}_{\mathsf{Next}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$ for all epoch that keys $k_e \notin \mathbb{KC}$ (i.e. the keys that will never be corrupted by the adversary), the challenger generates $hcom$ in CDUAE headers $\tilde{c} = (ct, h, hcom)$ by picking up a random commitment value $hcom$ instead of invoking the commitment algorithm. Additionally the challenger of G_1 will build a table $\mathbb{T}_{Hcom}$ to record all the tuples of the AEAD ciphertext, the commitment, its message and openning according to the epoch indexes. When querid with $\mathcal{O}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$ related to the epochs that keys $k_e \notin \mathbb{KC}$, instead of directly invoking the commitment verification algorithm, the challenger check the commitment by looking up the table $\mathbb{T}_{Hcom}$ and checking whether the tuple $(hcom, hopen, y) \in \mathbb{T}_{Hcom}$ and wether $hcom$

is included in the table $\mathbb{T}_{AEAD}$. The distinguishability of G_1 and G_2 can be reduced to the hiding property of the commitment.

In G_3, the challenger's behavior is similar to G_2, except that the way to deal with the decryption oracle $\mathcal{O}_{\mathsf{Dec}}$ and the oracle $\mathcal{O}_{\mathsf{ReEnc}}$: when queried with $\mathcal{O}_{\mathsf{Enc}}$,$\mathcal{O}_{\mathsf{Token}}$,$\mathcal{O}_{\mathsf{Next}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$ for all epoch that keys $k_e \notin \mathbb{KC}$ (i.e. the keys that will never be corrupted by the adversary), the challenger of G_3 will build a table $\mathbb{T}_{Dec}$ to record all the tuples of all CDUAE ciphertexts. When queried with $\mathcal{O}_{\mathsf{Dec}}$ and $\mathcal{O}_{\mathsf{ReEnc}}$ related to the epochs that keys $k_e \notin \mathbb{KC}$ via the ciphertext not in the table $\mathbb{T}_{Dec}$, the challenger return $\perp$ directly instead of running the decryption algorithm or the reencryption algorithm. The indistinguishability of G_2 and G_3 can be reduced to the authenticity of the AEAD the binding property of the commitment and the collision resistance hash function.

In the game G_4, the challenger encrypts a random string (has the same length) instead of the challenge message, the $\mathcal{A}$ cannot distinguish G_3 and G_4 due to the pusedorandomness of the key homomorphic PRF. Since the challenge message is submitted only once for each key, the security is also called OT-ROR(2.2). And in the game G_4, the probability of adversary outputing the correct challenge bit is equal to the probability of guessing randomly, which is exactly $\frac{1}{2}$.

Overall, we solve the problem by excluding some impossible decryption queries, playing the game with MU-RoR-AE challenger and embedding the challenge of sUP-IND-CCA into the MU-RoR-AE game, using the security of secret sharing, playing the OT-RoR (2.2) security game step by step, and eventually reducing to a random guessing event. □

sUP-INT-CTXT. We first provide the analysis result for sUP-INT-CTXT. Intuitively, we first assume that **ReCrypt**$^+$ is not sUP-INT-CTXT secure, and then construct contradictions with the existing conditions to prove the lemma. As a ciphertext contains a ciphertext header and a ciphertext body, a successful forgery can forge the ciphertext header or the ciphertext body. we make a reduction from the ciphertext header forgery to the break of ciphertext integrity of AEAD scheme, and make reductions from the ciphertext body forgery to the break of binding

of commitment scheme **HCom** or the break of collision resistance of homomorphic hash function **HomHash**. Formally, we have the following theorem and proof.

THEOREM 5.5.2 (sUP-INT-CTXT Security of ReCrypt$^+$). Let **ReCrypt**$^+$ be an updatable encryption scheme as defined in Section 5.5.1. **ReCrypt**$^+$is sUP-INT-CTXT secure, if **AEAD** scheme is CTXT scheme, **HCom** scheme has computational binding property, and **HomHash** scheme is collision resistant.

PROOF. We assume that **ReCrypt**$^+$ is not sUP-INT-CTXT secure and consider the winning query whose input is $C^* = (\tilde{c}^*, \bar{c}^*)$ and the corresponding epoch should be the current epoch e^*. There is no more queries after this winning query and the epoch stays in the current epoch. As the winning query $C^* = (\tilde{c}^*, \bar{c}^*)$ containing two parts ciphertext header and body, is newly generated by the adversary, we can divide into two winning cases: one is that the ciphertext header $\tilde{c}^*$ is newly generated, and the other case is that the ciphertext is new where the header $\tilde{c}^*$ is from querying oracle but the recorded ciphertexct body is different from the $\bar{c}^*$. The second case means that there exists another query that adds a record $\mathbb{T}(e^*, \tilde{c}^*) = \bar{c}$, where $\bar{c} \neq \bar{c}^*$ meaning that the ciphertext body is newly generated by the adversary. We will demonstrate that the first case suffices for winning the game $\mathsf{CTXT}_{\mathbf{AEAD}}$ and we an construct an adversary running the $\mathcal{A}_t$ of the second case to win the game $\mathsf{BIND}_{\mathbf{HCOM}}$ or $\mathsf{Col}_{\mathbf{Homhash}}$.

Firstly, we start from the first wining case and show that an adversary $\mathcal{A}_1$ winning the first case can be used to build a CTXT adversary $\mathcal{B}$ against the underlying encryption scheme **AEAD**. Here we gradualy reduct to it with a sequence of games.

Game G_0 is a standard strong UE-INT-CTXT game except that the winning query requires the ciphertext body has never been queried.

$$\mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\text{ReCrypt}^+}(\mathcal{A}) = \mathsf{Adv}^{\text{G0}}_{\text{ReCrypt}^+}(\mathcal{A}_1)$$

Game G_1 equals game G_0, except that we try to guess the exact epoch number of the challenge epoch e^*. We know that the challenge is adaptively submitted by adversary and the exact challenge epoch can never be known before it is. But in order to make a reduction to CTXT,

pre-deciding the challenge epoch is necessary but not easy. Note that such a number exists in any execution of Game G1 in which $\mathcal{A}_1$ tries to win the game. We follow the method used in [126] and guess this number by simply selecting an integer from $1, 2, \ldots, e_{max}$ uniformly at random. Thus, this guess is correct with probability at least $1/e_{max}$. (If the guess turns out to be wrong, we abort.) In this way, we reduce the advantage by a polynomial factor$1/e_{max}$. Thus,

$$\mathcal{A}^{\text{G1}}_{\text{ReCrypt}^+}(\mathcal{A}_1) = \frac{1}{e_{max}} \mathsf{Adv}^{\text{G0}}_{\text{ReCrypt}^+}(\mathcal{A}_1)$$

Game G_2 is the same as game G_1 but $\mathcal{B}$ embeds the challenge key k as the **AEAD** as key k_{e^*} in G_2 he simulates for adversary $\mathcal{A}$. $\mathcal{B}$ achieves this by replacing calling all **AEAD**.Enc and **AEAD**.Dec algorithm using k_{e^*} with querying the encryption and decryption oracle of CTXT security game. As the epoch e^* is the challenge epoch, the key k_{e^*} should not be corrupted by adversary $\mathcal{A}$, otherwise $\mathcal{A}$ loses due to trivial win. For the encryption and decryption query on the e^* epoch, it can be answered directly by querying the encryption and decryption oracle of CTXT game. For $\mathcal{A}$'s token query from $e^* - 1$ to e^* epoch, first decrypt with k_{e^*-1} and then submit the updated decryption message to the encryption oracle of CTXT. $\mathcal{A}$'s re-encryption query can be addressed in a similar way. As a result, Game G_2 can be perfectly simulated by $\mathcal{B}$ and is perfectly indistinguishable with game G_1. Finallly, the ciphertext header of the forgery $\mathcal{A}$ outputs is submitted by $\mathcal{B}$ to the CTXT game as a forgery. $\mathcal{A}$'s successful forgery is also $\mathcal{B}$'s successful forgery.

$$\mathsf{Adv}^{\text{G1}}_{\text{ReCrypt}^+}(\mathcal{A}_1) = \mathsf{Adv}^{\text{G2}}_{\text{ReCrypt}^+}(\mathcal{A}_1) \leq \mathsf{Adv}^{CTXT}_{AEAD}(\mathcal{B})$$

Combining all the above in G_0, G_1, G_2, we have the following:

$$\mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\text{ReCrypt}^+}(\mathcal{A}) \leq e_{max} \cdot \mathsf{Adv}^{CTXT}_{AEAD}(\mathcal{B})$$

We turn to the second winning case. Towards this, note that it must be that the winning query accesses the header $\tilde{c}^* = (\tilde{c}^{*1}, \tilde{c}^{*2}, \tilde{c}^{*3})$ through querying $\mathcal{O}_{\mathsf{Enc}}$, $\mathcal{O}_{\mathsf{Token}}$ or $\mathcal{O}_{\mathsf{ReEnc}}$ oracle.Let $x^*, h^*, \tilde{c}^{*2}, \tilde{c}^{*3}, y^*, r^*, z^*, d^*, \tilde{c}^*, \bar{c}^*, e^*$ be varibles associated to the winning query submitted by the adversary $\mathcal{A}_2$. We have that $\tilde{c}^{*2} = \mathsf{HomHash}(d^*)$ and $\mathbf{HCOM}.\mathsf{Open}(pp_{pcom}, \tilde{c}^{*3}, y^*; r^*) =$

y^*. As there exists a previous query that sets $\mathbb{T}(e^*, \tilde{c}^*) = \bar{c} \neq \bar{c}^*$, we will analyze all the possible queries.

Consider the first possibility that the previous query was $\mathcal{O}_{\mathsf{Enc}}(e^*, m)$. Necessarily this generated a valid ciphertext $C = (\tilde{c}^*, \bar{c})$ such that $\mathbb{T}(e^*, \tilde{c}^*) = \bar{c}$. Because $\mathbb{T}(e^*, \tilde{c}^*) \neq \bar{c}^*$ for a winning query, it must be that $(y^*, r^*, d^*) \neq (y, r, d)$. Thus there are two situations: either $(y^*, r^*) \neq (y, r)$ or $d^* \neq d$. We therefore let our $\mathsf{BIND}_{\Pi_{COM}}$ winning values $\tilde{c}^{*3}, (y^*, r^*), (y, r)$, or let our $\mathsf{Col}_{\mathcal{H}}$ winning values h^*, d^*, d.

Consider the second possibility that the previous query was $\mathcal{O}_{\mathsf{ReEnc}}(i, e^*, C_i)$. Necessarily, C_i is a valid ciphertext of epoch i and this generated a valid ciphertext $C = (\tilde{c}^*, \bar{c})$ such that $\mathbb{T}(e^*, \tilde{c}^*) = \bar{c}$. Because $\mathbb{T}(e^*, \tilde{c}^*) \neq \bar{c}^*$ for a winning query, it must be that $(y^*, r^*, d^*) \neq (y, r, d)$. Thus the result is the same as the former.

Consider the third possibility that the previous query was $\mathcal{O}_{\mathsf{Token}}(e^* - 1, \tilde{c}_{e^*-1})$. Necessarily $\mathbb{T}(e^* - 1, \tilde{c}_{e^*-1}) \neq \perp$ and $\mathbb{KC}(e^* - 1) \neq ture$, since if $\mathbb{KC}(e^* - 1) = ture$ then $\mathcal{A}_2$ lose due to key k_{e^*} corrupted and if $\mathbb{T}(e^* - 1, \tilde{c}_{e^*-1}) = \perp$ then there is no recording $\mathbb{T}(e^*, \tilde{c}^*)$ which is contradict to the conditions. After the query $\mathcal{O}_{\mathsf{Token}}(e^* - 1, \tilde{c}_{e^*-1})$, there is a record. $\mathbb{T}(e^*, \tilde{c}^*) = \bar{c}$. Then the result is similar to the former two and we omit it. All the three possible queries show that

$$\mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\text{ReCrypt}^+}(\mathcal{A}_2) = \mathsf{Adv}^{\text{bind}}_{\text{Hcom}}(\mathcal{C}) + \mathsf{Adv}^{\text{col}}_{\text{HomHash}}(\mathcal{D})$$

Combining the two cases, we get the following:

$$\begin{aligned}\mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\text{ReCrypt}^+}(\mathcal{A}) &= \mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\text{ReCrypt}^+}(\mathcal{A}_1) + \mathsf{Adv}^{\text{sUP-INT-CTXT}}_{\text{ReCrypt}^+}(\mathcal{A}_2) \\ &\leq e_{max} \cdot \mathsf{Adv}^{CTXT}_{AEAD,q}(\mathcal{B}) + \mathsf{Adv}^{\text{bind}}_{\text{Hcom},t}(\mathcal{C}) + \mathsf{Adv}^{\text{col}}_{\text{HomHash}}(\mathcal{D})\end{aligned}$$

□

sUP-REENC-CCA. To demonstrate that our **ReCrypt**$^+$ scheme is sUP-REENC-CCA secure, we introduce a property called *perfect re-encryption* proposed in [126]. Perfect re-encryption assures that for any ciphertext of updatable encryption, decrypt-then-encrypt has the same

distribution with re-encryption. We give a formal definition of perfect re-encryption for UE setting defined in Sec 5.4.4. We notice that **ReCrypt**$^+$naturally satisfy the perfect re-encryption property. As pointed by [126], the perfect re-encryption property plus the sUP-IND-CCA security imply the sUP-REENC-CCA security. So we have the following theorem whose the formal proof is in 5.4.4.

THEOREM 5.5.3 (sUP-REENC-CCA Security of ReCrypt$^+$). Let **ReCrypt**$^+$ be an updatable encryption scheme as defined in section 5.5.1, with the perfect re-encryption property. Then for any P.P.T adversary $\mathcal{A}_r$, we have:

$$\mathsf{Adv}^{\text{sUP-REENC-CCA}}_{\text{ReCrypt}^+}(\mathcal{A}_r) \leq neg(\lambda).$$

PROOF. Theorem 5.5.1 shows that **ReCrypt**$^+$ is a sUP-IND-CCA secure updatable encryption scheme. We know that **ReCrypt**$^+$ has perfect re-encryption property. From lemma 3, we have

$$\mathsf{Adv}^{\text{sUP-REENC-CCA}}_{\text{ReCrypt}^+}(\mathcal{A}_r) \leq \mathsf{Adv}^{\text{sUP-IND-CCA}}_{\text{ReCrypt}^+}(\mathcal{A}_c) \leq neg(\lambda).$$

□

5.6 Future Works

UE is a cryptographic primitive derived from the real-world need for key rotation to mitigate the impact of key compromise. Some schemes currently in use are not sufficiently secure. While most secure UE schemes demonstrate theoretical feasibility, they remain far from practical deployment.

Towards practical efficiency. It is promising to bring UE back to addressing the key compromise challenge effectively. One possible direction is to improve the efficiency of UE constructions. Additionally, recent research [138] indicates that CIUE with forward secrecy and post-compromise security implies public-key encryption, which strongly suggests that encrypting data with such UE may not be as efficient as using symmetric encryption. Another interesting avenue is to revise UE for application setting to include necessary security properties along with corresponding practical constructions.

CHAPTER 6

Efficient Secure Storage with Post-Compromise Security

6.1 Introduction

An increasing number of companies, government bodies, and personal users are choosing to store their data on the cloud instead of local devices. However, as a public infrastructure, frequent data breaches from the cloud have been reported. One potential mitigation strategy is to allow users to upload encrypted data and keep the decryption key locally. However, even with encryption mechanisms in place, there is still a risk that users' decryption keys may become compromised over time.

To address this issue, it is widely acknowledged and implemented in the industry to periodically refresh the secret key used to protect the data and to update the corresponding ciphertext in the cloud. For example, the Payment Card Industry Data Security Standard (PCI DSS)[122, 123] requires credit card data to be stored in encrypted form and mandates key rotation, whereby encrypted data is regularly refreshed from an old to a newly generated key. This strategy has also been adopted by many cloud storage providers, such as Google and Amazon [25]. By regularly refreshing encryption keys, the risk of data compromise can be significantly reduced. This approach ensures that even if a decryption key is compromised, it will only affect a limited amount of data that was encrypted with that key. Furthermore, this strategy is relatively easy to implement and can be automated, making it an effective way to improve cloud security.

While standardized encryption tools are available, facilitating key rotation requires careful consideration. A naive solution is to have the client download all encrypted data, decrypt

it, choose a new key, encrypt the data, and upload the new ciphertext to the cloud server. However, this approach is inefficient, especially for large amounts of data. To address this issue, Boneh et al. [21] proposed a new primitive called updatable encryption (UE) for efficiently updating ciphertexts with a new key. Everspaugh et al. [25] gave a systematic study of Updatable Authenticated Encryption (UAE), especially on the key rotation on *authenticated encryption*, which is the standard practice for encryption. Standard UAE constructions can guarantee the confidentiality and integrity of the plaintext. With UAE, a client only needs to retrieve at most a short piece of information (known as the header) and generate a short update token that enables the server to re-encrypt the data from the existing ciphertext while preserving encryption security.

UAE constructions [27, 126, 125, 89, 124] are particularly appealing due to their ability to provide post-compromise security. This ensures that outsourced storage can regain its security, even in the event of a temporary client hack, as long as the system executes the update process by updating both the secret key and ciphertext. Notably, after re-encryption, adversaries cannot determine whether the data has been modified, even if they have seen both the old key and the previous version of the ciphertext.

Integrity vs. frequent data update. In numerous real-world applications, such as E-commerce websites, social media platforms, financial institutions, and logistics companies, users require dynamic databases that can accommodate real-time changes in data. For instance, E-commerce websites need to manage inventory in real-time as products are added, sold, or restocked. Social media platforms must store and update user-generated content, such as posts, comments, and likes, in real-time. Financial institutions require the processing and storage of large volumes of transactional data in real-time, such as stock trades or credit card transactions. Logistics companies need to track and manage shipments in real-time as they move through the supply chain. These databases must be capable of handling continuous updates and modifications. To handle large volumes of data with varying attributes, such databases must be designed to facilitate fast data retrieval and frequent data updates.

However, despite the existence of several proposed constructions for Updatable Encryption (UE) in the literature [25, 126, 125, 89, 124], these schemes mainly focus on ensuring the

confidentiality and integrity of static databases and cannot be applied directly to dynamic databases. If the client encrypts each file using traditional UE before uploading it to the server, this approach fails to ensure data integrity if the client performs data updates on the database. The main issue with this approach is that the client needs a mechanism to revoke the previous UE ciphertexts associated with outdated data stored by the untrusted server. Otherwise, the server may provide the client with an obsolete version of the file instead of the latest one. Although the client could keep track of every change locally, this contradicts the primary objective of utilizing fewer resources compared to storing the entire database locally.

A promising approach for tracking changes of all files in a database is to use vector commitment (VC), a powerful primitive proposed by Catalano and Fiore [139]. VCs enable the commitment of an ordered sequence of n values $(m_1, \ldots, m_n)$ into a concise commitment while allowing for later opening of the commitment at specific positions with a membership proof to prove that m_i is the i-th committed message. To ensure security, VCs must satisfy the position-binding property, which requires that an adversary cannot open a commitment to two different values at the same position. The size of the commitment and each opening must be independent of the vector degree. To guarantee the integrity of a dynamic database, each file could be treated as an element of the vector.

The vector commitment property, especially its support for element updates, plays a crucial role in ensuring the integrity of a dynamic database. VC has two algorithms to update the commitment and corresponding openings. The first algorithm updates a commitment Com by changing the i-th message from m_i to m'_i, and results in a modified Com' containing the updated message. The second algorithm updates an opening for a message at position j with respect to Com to a new opening with respect to the new Com'. Indeed, Catalano and Fiore [139] have shown that the verifiable database with efficient updates (VDB) [140] can be constructed from the VC scheme.

Key rotation for a verifiable database. Although VC can address the integrity problem of dynamic databases, its compatibility with encryption schemes featuring key rotation is not straightforward. Specifically, applying VC to commit UE ciphertexts as vector elements may result in linear communication costs with the entire storage during each key update. This is

because updating the VC content requires linear communication with the updated ciphertext, which constitutes the entire content of the user's encrypted storage.

An alternative approach to reducing communication costs is to apply VC and UE directly to the plaintext, similar to the Enc-and-Mac combination of AE. In this approach, users store the UE ciphertext and VC membership proof of each file on the server while keeping the commitment locally as metadata for integrity checks. However, this construction fails to satisfy the confidentiality requirement if using a general VC without position hiding property, as the vector commitment and membership proofs could potentially leak information about the plaintexts. Although this information leakage can be avoided by committing each file first and running VC on those commitments, it is not sufficient to achieve post-compromise security. Since the membership proofs are not updated during key rotation, an adversary may be able to learn the updated pattern of the files in each vector element. This information leakage can further reveal whether each file has been updated after the epoch has evolved, which is a critical concern for post-compromise security. Therefore, it is important to hide the information of the membership proofs, as well as the plaintext, when considering post-compromise security.

To address the information leakage of VC proofs, one possible suggestion is to encrypt the VC proofs using UE schemes as well. However, this means that updating one file would require updating all other VC proofs in the UE construction. One straightforward solution would be to retrieve all ciphertexts, decrypt them, update their contents, re-encrypt them, and then upload them. However, this approach incurs linear communication costs for each file update in relation to the entire storage. To mitigate the information leakage of the vector commitment, a desirable solution would be to have re-randomizable vector commitment that supports periodic re-randomization.

To tackle these challenges, new encryption with key rotation and vector commitment techniques are needed that can adapt to the evolving needs of dynamic databases. In a nutshell, the following question arises:

Is there an efficient method that enables the highest levels of confidentiality and integrity in a frequently updated database, while minimizing communication overhead?

6.1.1 Our results

This chapter introduces a novel primitive called updatable secure storage (short for USS), which provides a secure solution for dynamic databases with version control. The USS scheme ensures both data confidentiality and integrity, even in the event of key compromises. By using efficient key rotation and file update procedures, the communication costs of these operations are independent of the size of the database. This makes the USS scheme particularly well-suited for managing large and frequently updated databases in a secure and efficient manner. The USS scheme is built on the KEM + DEM paradigm, where the DEM part can be any UE ciphertext with IND-CPA security. This allows the USS scheme to benefit from the efficiency of existing UE schemes while also providing strong security guarantees against attacks on data confidentiality and integrity. Overall, the USS scheme provides an effective solution for secure database management in dynamic environments, where frequent data updates are necessary.

Confidentiality in the event of key leakage. The USS scheme is designed to ensure basic content confidentiality even in the event of temporary key leakage or storage breach, as long as the server conducts the key rotation process in an honest manner. This process of key rotation is an essential aspect of the storage system, serving to limit the amount of data that could be compromised in the event of a key breach. More precisely, USS can guarantee the confidentiality of the data unless the attacker can learn the key and the ciphertexts in the same epoch. In more precise terms, the USS scheme can guarantee data confidentiality, unless the attacker can learn both the key and the ciphertexts within the same epoch. Hence USS scheme can effectively mitigate the impact of key breaches, as it reduces the window of opportunity for attackers to gain access to both pieces of information simultaneously.

Integrity for dynamic databases. The USS scheme provides strong integrity guarantees in dynamic databases, where a malicious server may attempt to deceive the client by providing

an outdated version of data. More precisely, the strong integrity allows the server to be fully malicious and may not follow the protocol to behave most of the time. In contrast, existing UE schemes, such as those proposed in [25, 27, 126, 125, 89, 124], assume that the server will honestly proceed with the key rotation procedure. However, this assumption is unrealistic in many scenarios, and it limits the ability of UE schemes to provide comprehensive protection against attacks on data integrity. Furthermore, while UE schemes exclude the case where an adversary forges ciphertexts after learning the current secret key, the USS scheme can guarantee data integrity even if the secret key is leaked. This is a significant advantage of the USS scheme, which offers stronger protection against a wider range of attacks.

Post-compromise security. The USS scheme offers post-compromise security for confidentiality. Specifically, if an adversary compromises both the secret key and storage in some epoch, they cannot gain any advantage in decrypting ciphertexts obtained in epochs after the compromisation. To capture the notion of post-compromise security, we introduce a security game called key update unlinkability. This game requires that attackers cannot distinguish whether an updated ciphertext is key updated from a previously corrupted ciphertext. Additionally, the USS scheme provides file update unlinkability, which guarantees that attackers who corrupt storage before and after a file update operation learn nothing about the update itself, such as whether the file content has changed and what the current content is. These security notions are essential for protecting sensitive data in scenarios where confidentiality is of utmost importance, such as in healthcare, finance, and government applications.

6.1.2 Technique overview

Intuitively, USS can be regarded as a secure version of Github that provides secure outsourced storage and version control services even when the server is not fully trusted. USS enables users to create remote repositories on the server while keeping a secret key and a public stub on the client side. Its primary goal is to provide the best possible security under the key compromise. To this end, USS employs a periodic key rotation mechanism similar to UE schemes, which prevents an adversary from learning stored data even with a leaked key.

Unlike UE schemes, the integrity of USS relies on the public stub of the repository rather than the secret key. As long as the stub is correctly kept by the user, the server cannot deceive the user. Keeping a public stub is much easier than secretly keeping a key on the client side. Furthermore, the stub changes if any file in the repository gets updated, which makes it convenient for users to track the versions of the entire repository. Even if the server is malicious, it cannot force the client to accept an old version of a file instead of the latest one.

One possible solution is to use a vector commitment to commit to all files in the repository. The commitment value, which is the stub stored on the client side, can be updated efficiently using the vector commitment whenever there are changes to the files [139]. However, this approach raises the concern that the commitment value itself may reveal information about the repository. Another approach is to encrypt the files during updates and use a vector commitment to commit to the resulting ciphertexts. However, this approach faces the challenge that the ciphertexts may change during key updates, causing the commitment value to update accordingly. Since the new ciphertexts are computed on the server side, the client cannot update the stub locally.

An alternative solution is to use a classic commitment to commit to each file and then use a vector commitment to commit to each classic commitment value. The content of each file can then be encrypted using the updated encryption. However, this approach may raise concerns about post-compromise security. Specifically, an attacker who gains access to the server can track the membership proofs and the vector commitment value stored on the server. These values will not change if the files remain unchanged, allowing the attacker to easily determine whether any files have been modified.

Homomorphic vector commitment. To address this dilemma, we introduce the concept of homomorphic vector commitment (HVC), which extends the classical additive homomorphic commitment (e.g., Pedersen commitment [26]). Besides the position-binding property, HVC offers a significant advantage over existing VC constructions [139, 141, 142, 143] by satisfying both the position hiding and homomorphic properties simultaneously. The *position hiding* property states that one cannot distinguish whether a commitment was created to a vector $(m_1, \ldots, m_n)$ or $(m'_1, \ldots, m'_n)$, even after seeing the openings of some same elements.

Although many existing vector commitment constructions already satisfy the homomorphic property, we observe that augmenting them with position hiding using the hybrid methodology would destroy the homomorphic property. By contrast, HVC provides a more elegant solution that preserves both properties. Specifically, HVC allows a user to commit to a vector of values and later reveal the value at a certain position of the committed vector without revealing any information about other vector elements. Moreover, HVC supports efficient homomorphic operations on both the commitment and the openings. The detailed construction of HVC is in Sec 6.3.

In our proposed USS construction, each file in the repository is represented as an entry in a vector. The commitment to this vector serves as a concise stub representing the entire repository. The binding property of HVC ensures that any changes to the files will be detected by the client, even if the stub is public and the key is leaked. The homomorphic property of HVC allows the client to efficiently update the stub by homomorphically adding a new commitment to the vector of changes whenever any file in the repository changes. Finally, the hiding property of HVC ensures that an adversary cannot learn any information about individual files from the public stub and other files' membership proofs. When entering a new epoch, the client can re-randomize the stub by homomorphically adding a commitment to the zero vector. This approach is effective because an attacker cannot distinguish between the commitment to the zero vector and to the difference vector of the new and old files, and thus cannot track whether files have been changed or not.

Homomorphic updatable encryption. However, the above approach alone is insufficient to guarantee post-compromise security. If an attacker gains access to the membership proof of the vector commitment, they could determine whether a file has changed over two epochs. One possible way is to wrap the vector commitment membership proofs with UE, but this approach may present additional challenges for proof updates, particularly when updating a single file. In VC, changes to one element require updates to membership proofs of all elements[144]. Therefore, all UE ciphertexts of the membership proofs must be updated with the plaintext. To ensure efficient database management, the updatable encryption scheme

must enable the server to compute the membership proof update in a homomorphic manner without requiring the retrieval of the ciphertexts of the membership proofs.

Thus, homomorphic updatable encryption is a critical feature for efficient database management, as it enables updates to be performed on the server side on the ciphertexts of the membership proofs without decryption. This reduces communication costs and enhances the scalability of the system. To the best of our knowledge, only one existing updatable encryption scheme, RISE [27], has homomorphic properties for plaintexts. RISE only supports the homomorphic operation by multiplying a new element. Fortunately, we discovered that the opening update of the bilinear pairing-based VC [139] also involves the multiplication of group elements. As a result, VC membership proofs could be encrypted via RISE and we can leverage RISE's homomorphic property to update the encryption of VC membership proofs.

6.2 Updatable Secure Storage

As previously introduced, an updatable secure storage (USS) system can be considered a secure version of GitHub that offers secure outsourced storage and version control services, even in scenarios where the trustworthiness of the server is not fully assured. USS provides users with the capability to create and update remote encrypted repositories on the server while maintaining a secret key and a public stub on the client side. The stored data and its updated version remain confidential and can only be accessed by authorized parties with the secret key. Additionally, USS ensures that a malicious server is unable to manipulate the client into accepting a tampered database or an outdated file, even if it gains access to the client's secret key and violates the protocol during each interactive procedure including file updates, key updates, and data retrieval.

Moreover, the USS system supports *key rotation*, a feature that is similar to updatable encryption schemes [25]. Key rotation is a critical security mechanism that ensures the confidentiality of the database, even if either the key or the storage is compromised, but not both simultaneously. By periodically rotating the secret key, the USS system can prevent an attacker who has gained access to an old key from knowing the current plain version

of the database. This is particularly important in scenarios where the key may have been compromised, as it ensures that any data stored on the server remains secure.

Furthermore, we consider the possibility of external attackers gaining access to the repository stored on the server temporarily and occasionally, mirroring frequently reported data breaches. Despite the repository being encrypted, monitoring the alterations in the encrypted repository could unveil its update history, which has the potential to expose users' activities and preferences, thus compromising the privacy of the individual. For instance, if the user updates a file related to their medical records, an attacker who gains access to the update history can infer that the user has medical issues, even if they cannot access the actual contents of the file. To mitigate this issue, we propose a *file update unlinkability* mechanism in USS during key rotation. This mechanism utilizes a re-randomization algorithm to conceal the update history of the database if the attacker has only intermittent access to the stored encrypted repository.

To ensure system efficiency, USS uses efficient data update techniques and secret key refresh/rotation mechanisms. These mechanisms guarantee that communication costs and client workloads remain independent of the number and size of files stored in the system.

6.2.1 Syntax of USS

To ensure the security of remotely stored data, the USS creates a unique secret key for each encrypted repository. The user will possess the secret key and a unique stub associated with the respective repository on the client side. Each repository can store a predetermined number of encrypted files. When a specific file is required, the user can retrieve it from the remote repository, decrypt its content using the secret key, and verify its integrity. If a file needs to be updated, the client will interactively communicate with the server to modify the file within the repository. Additionally, the client will periodically generate new keys and update the encrypted files in the repository to maintain security.

Accordingly, the syntax of USS should be as follows.

- $\mathsf{USS.ParGen}(1^\lambda, \mathcal{M}, n) \rightarrow pp$: Given the security parameter λ, the description of message space $\mathcal{M}$, and the size of vector degree n, the **parameter generation** algorithm generates the public parameter pp.
- $\mathsf{USS.KeyGen}(1^\lambda, pp) \rightarrow sk$: Given the security parameter λ and the public parameter pp, the **key generation** algorithm generates the secret key sk.
- $\mathsf{USS.Store}(db, sk, pp) \rightarrow (rep, sb)$: Given a database db that contains n independent files $m_1, \ldots, m_n$, and the public parameter pp, the client executes the **data storing** algorithm. The output of this algorithm is a repository $rep = (c_1, \ldots, c_n)$, which will be stored on the server side, along with a stub sb that will be accessible to both the server and the client. Each c_i is the ciphertext corresponding to the file m_i.
- $\mathsf{USS.Rev}_{client}(i, sk, sb, pp) \leftrightarrows \mathsf{USS.Rev}_{server}(rep, sb, pp) \rightarrow \langle m_i/\perp; \cdot \rangle$: The **data retrieval** algorithm is an interactive procedure that enables the client to retrieve file i from the server. The client provides the index i, secret key sk, stub sb, and public parameter pp. The server holds the repository rep and public parameter pp. If the data retrieval procedure succeeds, the client will output m_i; otherwise, it will output $\perp$.
- $\mathsf{USS.FileUp}_{client}(i, m_i', sk, sb, pp) \leftrightarrows \mathsf{USS.FileUp}_{server}(rep, sb, pp) \rightarrow \langle sb'; sb', rep' \rangle$: The **file update** is an interactive procedure that allows the client to update the i-th file to m_i'. Specifically, the client holds the index i, the new i-th file m_i', the secret key sk, the stub sb and the public parameter pp, and the server has the storing repository rep together with the public parameter pp. After the interaction, the client will have a new stub sb', and the server will store a new repository rep'.
- $\mathsf{USS.KeyUp}_{client}(sk, sk', sb, pp) \leftrightarrows \mathsf{USS.KeyUp}_{server}(sb, rep, pp) \rightarrow \langle sb'; rep' \rangle$: The **key update** is an interactive procedure that makes the server to update the storing ciphertexts to encryptions under a new key sk'. After the interaction, the client will have a new stub sb', and the server will store a new repository rep'.

Basically, the USS scheme should satisfy the following properties for correctness and efficiency.

Correctness.: The correctness guarantees that when the client invokes the **data retrieval** procedure to fetch the i-th file if the server is honest, the client always successfully gets m_i no matter how many times the key has been updated and m_i is the latest updated version of the i-th file deposited by the client.

Client storage efficiency.: The client storage efficiency requires that the size of the information stored on the client side, including the secret key sk and the stub sb, should be independent of the size of database db and even the number of the files n.[1]

Retrieve efficiency.: To ensure efficient retrieval, the communication cost for the interactive procedure **data retrieval** should be independent of the size of other plaintext files m_j for $j \neq i$ and the number of files n in the entire repository when retrieving the i-th file. However, it may depend on the size of the retrieved file m_i.

File update efficiency.: For efficient file updates, the communication cost for the **file update** procedure should be independent of the size of other plaintext files m_j for $j \neq i$ and the number of files n in the entire repository when updating the i-th file. However, it may depend on the size of the updating file m_i.

Key update efficiency.: In order to ensure efficient key updates, it is desirable that the communication cost associated with the key update procedure remains independent of the size of all files m_i for $i = 1, \ldots, n$. If the communication cost is also independent of the number of files n in the repository, we classify these schemes as ciphertext-independent. Conversely, if the communication cost depends on the number of files, we refer to them as ciphertext-dependent schemes.[2]

6.2.2 Security models

The security threats associated with USS arise from three main sources. Firstly, the system must safeguard the confidentiality of the stored database against the honest but curious server and external attackers who have temporary and intermittent access to server storage and user secrets with no time overlap. Secondly, the system must retain the integrity of the stored

[1] The size of public parameter stored on both client and server should be independent of the size of database db, although it may be related to the number of files n.

[2] This notion is directly borrowed from the updatable encryption framework, which distinguishes between ciphertext-dependent and ciphertext-independent versions.

database against the malicious server. Thirdly, given the possibility of the encrypted repository being compromised by external attackers multiple times, and potentially exposing the update history of the repository, the system must offer security guarantees that prevent update history leakage, even if the user's secret is ever revealed once simultaneously.

We present three models aimed at addressing confidentiality security challenges: IND-DD-CPA (indistinguishability of dynamic databases under chosen plaintext attacks), IND-REENC-CPA (indistinguishability of re-encryption ciphertext under chosen plaintext attacks), and ciphertext indistinguishability for file update (IND-FileUp-CPA for short). The IND-DD-CPA and IND-FileUp-CPA models ensure the fundamental confidentiality of the original database and updated files respectively, while the IND-REENC-CPA model provides post-compromise security and the ability to conceal the update history of the repository. Intuitively, the IND-DD-CPA model stipulates that an adversary cannot distinguish between two vectors of messages once they are encrypted. This holds even if the adversary has the ability to corrupt keys, trigger file updates, or initiate key rotations. The IND-FileUp-CPA model ensures that an adversary cannot distinguish between two files used to replace the current file in the repository, even if the attacker knows the previously stored data. On the other hand, the IND-REENC-CPA model ensures that an adversary cannot distinguish whether the ciphertexts have been updated after a key rotation. This is also true even if the adversary has the ability to corrupt keys, trigger file updates, or initiate key rotations. Our proposed models are similar to the IND-ENC and IND-UPD models presented in [27], respectively.

To address the security challenges related to message integrity, we propose a model called ordered full plaintext integrity (OF-PTXT for short). Intuitively, OF-PTXT imposes stricter requirements than the INT-PTXT game for the updatable encryption proposed in [126], as it guarantees message integrity even in the presence of a leaked secret key and non-compliant servers that do not rotate keys as required by the protocol.

Confidentiality. Our confidentiality-related models consider three critical security properties: message confidentiality (IND-DD-CPA), file update unlinkability (IND-FileUp-CPA), and re-encryption indistinguishability (IND-REENC-CPA). In these models, the adversary may attempt to compromise the confidentiality of any files in a target repository. The encryption

of the target file is referred to as challenge ciphertext. Furthermore, the repository that contains the target files is called the challenge repository. However, since the key is rotated when the epoch evolves, the key-updated version of the challenge ciphertext is referred to as challenge-equal ciphertext. For simplicity, we also use the term “challenge-equal ciphertext" to represent both original and key-updated versions of the challenge ciphertext. Consequently, the repository containing the challenge-equal ciphertexts is called the challenge-equal repository.

More precisely, the challenger maintains the following internal states.

e**:** The current epoch number. It is initialized as 1.

e^***:** The epoch number from which the challenge begins. It is initialized as $\perp$.

sb^***:** The current stub of a repository including challenge-equal ciphertexts, which is initialized as $\perp$.

$\mathcal{K}$**:** The set of epochs in which the adversary has corrupted the epoch key by querying the key corruption oracle.

$\mathcal{C}$**:** The set of epochs in which the adversary corrupts a challenge-equal ciphertext by querying the challenge-equal ciphertext corruption oracle.

$\mathcal{I}$**:** The set of indices of challenge-equal ciphertexts in the repository. Before the adversary submits the challenge, the set is empty $\mathcal{I} = \emptyset$.

$\mathcal{L}$**:** The collection consists of tuples (t, sb, rep, db) which will be used to track all the repositories on the server, where t represents the epoch number, sb represents the corresponding stub, and rep and db represent the corresponding encrypted repository and plaintext database, respectively.

$\mathcal{S}$**:** The collection contains tuples (t, Trans), where t represents the epoch number and Trans represents all the corresponding key update transcripts from epoch $t-1$ to epoch t.

$\mathcal{F}$**:** The collection of challenge-equal ciphertexts' file update transcripts contains tuples (sb, t, i, ft_i), where ft_i represents the file update transcript generated in the file update query $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ at epoch t, and the index i is the index of a challenge-equal ciphertext.

$\mathcal{T}$**:** The set of epochs in which the adversary queries the key update transcript oracle. If the transcript of the key update procedure from sk_t to sk_{t+1} is corrupted, it is represented

as $t \in \mathcal{T}$. The set $\mathcal{T}$ is initially empty, indicating that the adversary has not yet queried the key update transcript oracle.

The adversary is given the following oracles to store files as users via $\mathcal{O}.\mathsf{Store}$, enable the key evolving via $\mathcal{O}.\mathsf{Next}$, corrupt key via $\mathcal{O}.\mathsf{KeyCorr}$, corrupt key update transcript via $\mathcal{O}.\mathsf{KeyUpTrans}$, and corrupt the file update ciphertext via $\mathcal{O}.\mathsf{FileUp}$. In terms of challenge related corruption, adversary can query $\mathcal{O}.\mathsf{ChaCTCorr}$ and $\mathcal{O}.\mathsf{ChaFTCorr}$ to get the challenge ciphertext and the file update transcripts of those challenge-equal ciphertexts.

$\mathcal{O}.\mathsf{Store}(db)$**:** The purpose of this oracle is to enable the adversary to deposit a database on the server. To this end, the challenge invokes the storing algorithm $\mathsf{Store}(sk, db, pp)$ to generate the stored file rep and the stub sb, which are then given to the adversary. Furthermore, the challenger adds the tuple (e, sb, rep, db) to the set $\mathcal{L}$.

$\mathcal{O}.\mathsf{Next}$**:** The adversary uses this oracle to initiate the update of all repositories on the server. The adversary will automatically gain access to their updated versions, except for the challenge-equal ciphertexts.

Specifically, the challenger retrieves all entries (e, sb, rep, db) from the database $\mathcal{L}$ having the current epoch number e. Subsequently, the challenger generates a new epoch key sk' and executes the key update procedure to produce new stubs sb' and new repository rep' for each retrieved entry. The transcripts of the generated key updates are denoted as Trans. The current epoch number is then incremented to $e + 1$, and new entries (e, sb', rep', db) are appended to the database $\mathcal{L}$. Additionally, a new entry (e, Trans) is added to the list $\mathcal{S}$. Furthermore, if there exists an entry $(e, sb, *, *) \in \mathcal{L}$ with $sb = sb^*$, then sb^* is also updated accordingly. The challenger provides the adversary with the stub and ciphertext elements having non-challenge indices $\{j\}_{j \notin \mathcal{I}}$ for the current epoch. For all other entries $(e, sb, *, *) \in \mathcal{L}$ such that $sb \neq sb^*$, the adversary is furnished with the updated versions of (sb, rep).

$\mathcal{O}.\mathsf{KeyCorr}(t)$**:** The purpose of this oracle is to facilitate the adversary in retrieving the secret key. If the epoch number t is not greater than the current epoch number e, the oracle will provide the adversary with the secret key sk_t corresponding to epoch t. Additionally, epoch t will be included in the set of key corruptions $\mathcal{K}$.

$\mathcal{O}.\mathsf{KeyUpTrans}(t)$**:** The purpose of this oracle is to facilitate the adversary in retrieving the key update transcript. If epoch number t is no more than the current epoch number e, retrieve the entry (t, Trans), and return the key update transcript Trans of all ciphertexts from epoch $t-1$ to epoch t to the adversary. Add epoch t to the set of key corruptions $\mathcal{T}$.

$\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$**:** This oracle enables the adversary to modify the i-th file of the current epoch to be the encryption of m'_i. The input includes the stub sb, the new file m'_i, and its index i, where $i \in [1, n]$. The challenger first retrieves the entry (t, sb, rep, db) in $\mathcal{L}$ with the current epoch number $t = e$ and the same stub sb. If the entry is empty, the oracle outputs $\perp$. Otherwise, the challenger executes $\mathsf{FileUp}_{client}(i, m'_i, sk_e, sb, pp) \leftrightarrows \mathsf{FileUp}_{server}(rep, sb, pp)$ and updates the entry with (e, sb', rep', db).

If $sb \neq sb^*$, then the challenger returns (sb', rep') and the file update transcript ft_i to the adversary. If $sb = sb^*$, which means that the queried stub sb is the stub of a challenge-equal ciphertext in the current epoch, then the challenger updates $sb^* \leftarrow sb'$ and checks whether index i belongs to a challenge-equal ciphertext. If $i \in \mathcal{I}$, remove i from $\mathcal{I}$, add a tuple (sb, e, i, ft_i) to collection $\mathcal{F}$. The challenger returns the updated stub sb' and each updated ciphertext f'_j with index $j \notin \mathcal{I}$ to the adversary. If $i \notin \mathcal{I}$, the challenger returns the updated stub sb', the file update transcript ft_i, and each updated ciphertext f'_j with index $j \notin \mathcal{I}$ to the adversary.

$\mathcal{O}.\mathsf{ChaCTCorr}(j)$**:** This oracle helps the adversary to learn the jth ciphertext of the challenge-equal ciphertext vector in the current epoch. If $j \in \mathcal{I}$, the jth element is a challenge-equal ciphertext for the current epoch. Then the challenger finds the entry (t, sb, rep, db) of $\mathcal{L}$ with the current epoch number $t = e$ and the stub sb is equal to $sb^* \neq \perp$, and add the current epoch e to the challenge-equal ciphertext corruption set $\mathcal{C}$ and give the adversary the jth ciphertext f_j where $rep = (f_1, \ldots, f_n)$. If $j \notin \mathcal{I}$, return $\perp$.

$\mathcal{O}.\mathsf{ChaFTCorr}(sb, t, i)$**:** This oracle helps the adversary to learn the file update transcripts of challenge-equal ciphertexts. The challenger finds the entry (sb, t, i, ft_i) of F with the same stub sb, epoch t, and index i, and returns the transcript ft_i to the adversary.

Trivial win condition. Adversaries could trivially win the confidentiality game if they corrupt both the epoch key and the challenge ciphertext or the updated version at that epoch. Since adversaries are given access to multiple oracles, where key update transcripts could help to update ciphertexts to the new key due to USS's function and even downgrade ciphertexts to the previous key if USS's update function is bi-directional. To exclude the trivial win conditions, we define an extended ciphertext corruption set $\tilde{\mathcal{C}}$ to record the epochs at which adversaries corrupt the challenge ciphertext via directly querying the challenge ciphertext oracle $\mathcal{O}.\mathsf{ChaCTCorr}$ or indirectly referring to the challenge ciphertext based on queries of $\mathcal{O}.\mathsf{KeyUpTrans}$ and $\mathcal{O}.\mathsf{ChaCTCorr}$. Here we assume the key update transcripts could update/downgrade ciphertexts in bi-direction since the scheme we use to construct USS supports it. Then we have $i \in \tilde{\mathcal{C}}$ if $i \in \mathcal{C}$, or $i-1 \in \mathcal{C}$ & $i \in \mathcal{T}$, or $i+1 \in \mathcal{C}$ and $i+1 \in \mathcal{T}$. The trivial win condition is $\mathcal{K} \cap \tilde{\mathcal{C}} \neq \emptyset$.

Message Confidentiality. Here we defined IND-DD-CPA security which aims to capture CPA style message confidentiality in the key updatable and file updatable setting. Concretely, the adversary can query $\mathcal{O}.\mathsf{Store}$ oracle for repository encryption in the storage. The adversary is allowed to engage the key rotation and get the update of non-challenge-equal files via the $\mathcal{O}.\mathsf{Next}$ oracle. The adversary can also corrupt some epoch key and challenge-equal file via the $\mathcal{O}.\mathsf{KeyCorr}, \mathcal{O}.\mathsf{ChaCTCorr}$ oracles. Furthermore, the adversary is allowed to query file update of each repository via $\mathcal{O}.\mathsf{FileUp}$ oracle. To exclude the trivial win of the security game, the adversary is not allowed to see the key and the challenge-equal file encryption simultaneously. Such requirements are similar to the restrictions in the security models of the updatable encryptions [126]. Formally, we have the IND-DD-CPA game as Figure 6.1.

DEFINITION 6.2.1 (IND-DD-CPA). An updatable secure storage scheme USS is called IND-DD-CPA secure if for any PPT adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\mathsf{ind\text{-}dd\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) :=$$
$$\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right|$$

$\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$

1 : $pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize $e = 1, e^* = \perp, sb^* = \perp$, Set $\mathcal{K}, \mathcal{C}, \mathcal{I}, \mathcal{L}, \mathcal{S}, \mathcal{T}, \mathcal{F}$ as $\emptyset$
2 : $sk_1 \leftarrow \mathsf{KeyGen}(pp)$
3 : $(db_0, db_1, state) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans}}(pp)$
4 : Parse $db_0 = (m_{0,1}, \ldots, m_{0,n}), db_1 = (m_{1,1}, \ldots, m_{1,n})$
5 : $(sb^*, rep^*) \leftarrow \mathsf{Store}(db_b, sk_e, pp)$
6 : $e^* = e, \quad \mathcal{L} = \mathcal{L} \cup (e, sb^*, rep^*, db_b)$
7 : **for** $i = 0$ *to* n
8 : **if** $m_{0,i} \neq m_{1,i}$, **then** $\mathcal{I} = \mathcal{I} \cup \{i\}$
9 : $b' \leftarrow \mathcal{A}_2^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans},\mathcal{O}.\mathsf{ChaFTCorr},\mathcal{O}.\mathsf{ChaCTCorr}}(state, rep^*)$
10 : **return** b' **if** $\mathcal{K} \cap \tilde{\mathcal{C}} = \emptyset$

FIGURE 6.1. The game of IND-DD-CPA.

File update unlinkability. To capture the security that the file update operation does not leak the confidentiality of the updated file, we define the file update unlinkability via the following experiment with the adversary. Intuitively, it ensures that attackers corrupting the storage before and after a file update operation learn nothing about file updates, such as whether the file content has changed, and what the current file content is.

We describe the file update security experiment $\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}$ for the key updatable dynamic secure storage scheme USS and adversary $\mathcal{A}$. In $\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}$ experiment, $\mathcal{A}$ submits two possible file $(m_{0,i}, m_{1,i})$ for challenge, where $i \in \{1, \ldots, n\}$. The challenger updates the i-th file of the stored storage with one of the two submissions selected randomly and gives the updated ciphertext to the adversary as the challenge ciphertext. $\mathcal{A}$'s goal is to give a correct guess on which file is chosen to update. The trivial win situation is that the adversary corrupts both the epoch key and the challenge-equal ciphertext at the same epoch, i.e., $\mathcal{K} \cap \mathcal{C} \neq \emptyset$.

DEFINITION 6.2.2 (IND-FileUp-CPA). An updatable secure storage scheme USS is called IND-FileUp-CPA secure if for any PPT adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) :=$$
$$\left| \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] \right|$$

$\mathsf{Exp}^{\mathsf{ind\text{-}fileup\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$

1 : $pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, $\mathsf{Initialize}$ $e = 1, e^* = \perp, sb^* = \perp$, Set $\mathcal{K}, \mathcal{C}, \mathcal{I}, \mathcal{L}, \mathcal{S}, \mathcal{T}, \mathcal{F}$ as $\emptyset$

2 : $sk_1 \leftarrow \mathsf{KeyGen}(pp)$

3 : $(sb, i, m_{0,i}, m_{1,i}, state_1) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans}}(pp)$

4 : **if** $(e, sb, *, *) \notin \mathcal{L}$, or $i \notin \{1, \ldots, n\}$, or $|m_{0,i}| \neq |m_{1,i}|$, **then return** $\perp$

5 : $\mathsf{Retrieve}$ $(e, sb, rep, db = (m_1, \ldots, m_n))$, Set $e^* = e$, $\mathcal{I} = \mathcal{I} \cup \{i\}$, $\mathcal{C} = \mathcal{C} \cup \{e\}$

6 : Run $\mathsf{FileUp}_{token}(sk_e, sb, m_{b,i}, i, pp) \leftrightarrows \mathsf{FileUp}_{server}(sb, rep, pp) \rightarrow \langle sb^*; rep^* \rangle$

7 : $\mathcal{L} = \mathcal{L} \cup (e, sb^*, rep^*, db_b = (m_1, \ldots, m_{b,i}, \ldots, m_n))$

8 : Record the file update transcript as fpt

9 : $b' \leftarrow \mathcal{A}_2^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans},\mathcal{O}.\mathsf{ChaFTCorr},\mathcal{O}.\mathsf{ChaCTCorr}}(state_1, sb^*, rep^*, fpt)$

10 : **return** b' **if** $\mathcal{K} \cap \tilde{\mathcal{C}} = \emptyset$

FIGURE 6.2. The game of IND-FileUp-CPA

Key update unlinkablity. Intuitively, key update unlinkability is aimed to capture the security for key updates after both corruptions. More concretely, attackers may corrupt both the client and the server at the same epoch. After the key rotation, attackers corrupt the server and obtain the updated ciphertext. Key update unlinkability ensures that attackers cannot detect whether the updated ciphertext contains the same plaintext as the previous corrupted ciphertext. The security is similar to the IND-UPD security of UE [27] since we provide the adversary with all the oracles IND-UPD security provides. In addition, our key update unlinkability allows the adversary to have additional access to the file update oracle.

We define the following security experiment $\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}$ for updatable secure cloud storage scheme USS and adversary $\mathcal{A}$, who has access to the oracle tuple ($\mathcal{O}.\mathsf{Store}$, $\mathcal{O}.\mathsf{Next}$, $\mathcal{O}.\mathsf{KeyCorr}$, $\mathcal{O}.\mathsf{FileUp}$, $\mathcal{O}.\mathsf{KeyUpTrans}$, $\mathcal{O}.\mathsf{ChaCTCorr}$) like in the above confidentiality games. So, the trivial win condition is triggered in the same case.

DEFINITION 6.2.3 (IND-REENC-CPA). An updatable secure storage scheme USS is called IND-REENC-CPA secure if for any PPT adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) :=$$
$$\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right|$$

$\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$

1 : $pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize e, e^*, sb^*, Set $\mathcal{K}, \mathcal{C}, \mathcal{I}, \mathcal{L}, \mathcal{S}, \mathcal{T}, \mathcal{F}$ as $\emptyset$
2 : $sk_1 \leftarrow \mathsf{KeyGen}(pp)$
3 : $(sb_0, sb_1, state_1) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans}}(pp)$
4 : Retrieve $(e, sb_0, rep_0 = (c_{0,1} \ldots, c_{0,n}), db_0), (e, sb_1, rep_1 = (c_{1,1}, \ldots, c_{1,n}), db_1)$ from $\mathcal{L}$
5 : Set $\mathcal{I} = \{i\}_{i \in \{1,\ldots,n\}, c_{0,i} \neq c_{1,i}}$, $e = e + 1$, $e^* = e$, $\mathcal{C} = \mathcal{C} \cup \{e\}$
6 : $sk_e \leftarrow \mathsf{KeyGen}(pp)$
7 : Run $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb_b, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep_b, pp)$ to output $\langle sb^*; rep^* \rangle$
8 : **for each** $(e-1, sb, rep, db) \in \mathcal{L}$, where $sb \notin \{sb_0, sb_1\}$
9 : Run $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb'; rep' \rangle$
10 : Set $\mathcal{L} = \mathcal{L} \cup (e, sb', rep', db)$
11 : $b' \leftarrow \mathcal{A}_2^{\mathcal{O}.\mathsf{Store},\mathcal{O}.\mathsf{Next},\mathcal{O}.\mathsf{FileUp},\mathcal{O}.\mathsf{KeyCorr},\mathcal{O}.\mathsf{KeyUpTrans},\mathcal{O}.\mathsf{ChaCTCorr}}(state_1, sb^*, rep^*, \text{ all } (sb', rep'))$
12 : **return** b' **if** $\mathcal{K} \cap \tilde{\mathcal{C}} = \emptyset$

FIGURE 6.3. The game of IND-REENC-CPA

Integrity. We define a kind of strong plaintext integrity notion called *ordered full plaintext integrity* (OF-PTXT for short). For the classic authenticated encryption schemes, plaintext integrity ensures that attackers cannot make any forgery for new plaintext except the queried ones, which work for static storage with appending function. But for dynamic storage, where some storage may be changed or even deleted, the old or deleted messages could be leveraged by dishonest storage providers to cheat users, which is not covered by classic integrity. Here OF-PTXT provides stronger integrity in the dynamic storage setting, where data could be updated dynamically. OF-PTXT ensures attackers cannot forge for a plaintext that does not belong to the current storage. To be formal, we show a security experiment between the adversary who acts as the malicious storage server, and the challenger who acts as the honest user. More precisely, the challenger will maintain the following list to record the latest version of databases.

$\mathcal{R}$: The list recording the latest stub and the message vector pair (sb, db) generated during the integrity game. $\mathcal{R}$ is initialized as empty and will be updated for each file update and key update. $\mathcal{R}$ has only entries for the latest epoch.

We use the stub to track the target stored database. For a certain pair (sb, db) of $\mathcal{R}$ and a certain index i, the adversary aims to make the client accept m_i' as the i-th element of the database but $m_i' \neq db[i]$.

Moreover, we allow the adversary to launch active attacks in the integrity game. The server which may be corrupted by the adversary may manipulate the storage and even to not follow the protocol during the file update or key update procedures. To capture adversary's above capability, three special oracles, including the database storing oracle $\mathcal{O}.\mathsf{StoreINT}$, the next oracle $\mathcal{O}.\mathsf{NextINT}$ and the file update token oracle $\mathcal{O}.\mathsf{FileUpINT}$ are provided for the adversary in integrity game. Besides, the adversary in the integrity game can also learn the security key via the key corruption oracle $\mathcal{O}.\mathsf{KeyCorr}$, which means USS can guarantee the integrity even when the key is leaked.

We will elaborate the special oracles for the integrity game in the following. For brevity, please refer to our previous descriptions about the similar oracles in the confidentiality Sec 6.2.2.

$\mathcal{O}.\mathsf{StoreINT}(db)$**:** This oracle is to let the adversary learn the stored file generated by the storing algorithm. The challenge will invoke the storing algorithm $\mathsf{Store}(sk_e, db, pp)$ to generate the stored file rep and the stub sb, and give rep and sb to the adversary. And add the pair (sb, db) to $\mathcal{R}$.

$\mathcal{O}.\mathsf{NextINT}$**:** This oracle is to let the adversary to invoke the client to launch the key update procedure. The challenger updates the epoch number $e = e + 1$, runs the KeyGen algorithm to generate the new epoch key $sk_e = sk'$, and runs the key update client-side algorithm to update all stubs sbs into the corresponding sb's and to generate client-side key update transcripts Trans_c for the server. Then the challenger returns all the new stubs sb's and transcripts Trans_c to the adversary, and updates each entry (sb, db) in $\mathcal{R}$ with the corresponding (sb', db).

$\mathcal{O}.\mathsf{FileUpINT}(m_i', i, sb)$**:** The adversary uses this oracle to invoke the client to launch the file update procedure and replace the i-th element of the database to m_i'. More precisely, the input of this oracle contains the new file m_i', its index i, and the corresponding stub sb. The challenger will first check whether the stub is contained in the list $\mathcal{R}$. During this interaction, the client will communicate with the corrupted server

according to the specification of the designed scheme, while the adversary could respond to the client with an arbitrary message and violate the protocol design. If the client finally accepts the update results, the challenger will update the entry (sb, db) of $\mathcal{R}$ with (sb', db').

We describe the integrity experiment $\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}$ for key updatable dynamic secure storage scheme USS and adversary $\mathcal{A}$, who has access to the oracle tuple $(\mathcal{O}.\mathsf{StoreINT}, \mathcal{O}.\mathsf{NextINT}, \mathcal{O}.\mathsf{FileUpINT}, \mathcal{O}.\mathsf{KeyCorr})$.

$\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$

$pp \leftarrow \mathsf{ParGen}(1^\lambda, \mathcal{M}, n)$, Initialize $\mathcal{R}, e$
$sk_e \leftarrow \mathsf{KeyGen}(pp)$
$(sb, i, state_1) \leftarrow \mathcal{A}_1^{\mathcal{O}.\mathsf{StoreINT},\mathcal{O}.\mathsf{NextINT},\mathcal{O}.\mathsf{FileUpINT},\mathcal{O}.\mathsf{KeyCorr}}(pp)$
if $(sb, *) \notin \mathcal{R}$ or $i \notin [1, n]$: **return** 0
else Run $\mathsf{Rev}_{client}(i, sk_e, sb, pp) \leftrightarrows \mathcal{A}$ to output $\langle m_i^*; \cdot \rangle$
  for $\forall \mathbf{m}$ s.t. $(sb, \mathbf{m}) \in \mathcal{R}$
    if $m_i^* \neq \mathbf{m}[i]$ **return** 1
  endfor
return 0

FIGURE 6.4. The game of OF-PTXT

DEFINITION 6.2.4 (OF-PTXT). An updatable secure storage scheme USS is called OF-PTXT secure if for any PPT adversary $\mathcal{A}$ the following advantage is negligible in the security parameter λ:

$$\mathsf{Adv}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) := \Pr[\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n) = 1]$$

6.3 Homomorphic Vector Commitment

This section presents an introduction to Homomorphic Vector Commitment (HVC) and explores the difficulties involved in constructing an HVC that can simultaneously satisfy both the position hiding and homomorphic properties.

6.3.1 Syntax and notions

HVC is defined with a tuple of algorithms $\mathsf{HVC} = (\mathsf{HVC.Setup}, \mathsf{HVC.Com}, \mathsf{HVC.Open}, \mathsf{HVC.Ver}, \mathsf{HVC.ComHom}, \mathsf{HVC.OpenHom})$ that works as following:

$\mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n) \rightarrow crs_n$**:** Given the security parameter λ, the description of committed message space $\mathcal{M}$, and the size of committed vector n, the probabilistic setup algorithm outputs a common reference string crs_n.

$\mathsf{HVC.Com}_{crs_n}(\mathbf{m}) \rightarrow (C, aux)$**:** On input an ordered sequence of n messages $\mathbf{m} = (m_1, \ldots, m_n)$ and the common reference string crs_n, the commitment algorithm outputs a commitment string C and the auxiliary information aux. We denote the commitment space as $\mathcal{C}$. The auxiliary information aux is succinct, say independent of the vector degree n.

$\mathsf{HVC.Open}_{crs_n}(i, \mathbf{m}, aux) \rightarrow \Lambda_i$**:** This algorithm is run by the committer to produce a proof (also known as opening) Λ_i that the i-th element $\mathbf{m}[i]$ is the committed message. We denote the proof space as $\mathcal{P}$.

$\mathsf{HVC.Ver}_{crs_n}(C, m, i, \Lambda_i) \rightarrow 1/0$**:** The verification algorithm outputs 1 only if Λ_i is a valid proof that m is the i-th committed message to the C.

$\mathsf{HVC.ComHom}_{crs_n}(C, C' \in \mathcal{C}) \rightarrow C''$ **:** This algorithm can be run by any user who holds two commitment belonging to the commitment space $\mathcal{C}$, and it allows the user to compute another commitment $C'' = C \oplus C' \in \mathcal{C}$, where $\oplus$ denotes the homomorphic operation for the commitment.

$\mathsf{HVC.OpenHom}_{crs_n}(\Lambda_j, \Lambda'_j \in \mathcal{P}) \rightarrow \Lambda''_j$**:** This algorithm can be run by any user who holds two membership proofs Λ_j and Λ'_j for some message on position j w.r.t. to some C and C'' (which contains m and m' as the message at position j), and it allows the user to compute another proof $\Lambda''_j = \Lambda_j \otimes \Lambda'_j \in \mathcal{P}$ (w.r.t. some C'' which contains m'' as the new message at position j), where $\otimes$ denotes the homomorphic operation for the proof.

Basically, a HVC scheme should satisfy correctness, conciseness and homomorphic property.

Correctness. A vector commitment is correct if for all honestly generated $crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$, $\forall i \in [n]$, if C is a commitment on a vector $(m_1, \cdots, m_n) \in \mathcal{M}^n$, Λ_i is a proof for position i generated by $\mathsf{HVC.Open}_{crs_n}$, then $\mathsf{HVC.Ver}_{crs_n}(C, m_i, i, \Lambda_i)$ outputs 1 with overwhelming probability.

Conciseness. A vector commitment is concise if the size of the commitment C and the outputs of $\mathsf{HVC.Open}$ are both independent of the size n of the vector.

Homomorphic property. Formally, $\forall i \in [n]$, for all honestly generated $crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$, for all honestly generated

$$(C, aux) \leftarrow \mathsf{HVC.ComHom}_{crs_n}(\mathbf{m}),\ (C', aux') \leftarrow \mathsf{HVC.ComHom}_{crs_n}(\mathbf{m}'),$$

$$\Lambda_i \leftarrow \mathsf{HVC.Open}_{crs_n}(i, \mathbf{m}, aux),\ \Lambda_i' \leftarrow \mathsf{HVC.Open}_{crs_n}(i, \mathbf{m}', aux'),$$

where $\mathbf{m} = (m_1, \ldots, m_n), \mathbf{m}' = (m_1', \ldots, m_n')$, if

$$C'' \leftarrow \mathsf{HVC.ComHom}_{crs_n}(C, C'),\ \Lambda_i'' \leftarrow \mathsf{HVC.OpenHom}_{crs_n}(\Lambda_i, \Lambda_i')$$

then we have $\mathsf{HVC.Ver}_{crs_n}(C'', m_i + m_i', i, \Lambda_i'') = 1$.

6.3.2 Security models

In this section, we formally define the security models for binding and hiding on the situation that the corresponding membership proofs are leaked.

Position-Binding: It requires that for any well-formed commitment, the *PPT* adversary cannot find two different messages on the same position that the verification algorithm accepts both. Formally, we have the HVC Position-Binding game as Figure 6.5.

The advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\text{position-binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n) = \Pr[\mathsf{Exp}^{\text{position-binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n) = 1].$$

DEFINITION 6.3.1. A HVC scheme satisfies HVC Position-Binding if for every PPT adversary $\mathcal{A}$ the advantage function $\mathsf{Adv}^{\text{position-binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$ is negligible in λ.

$\mathsf{Exp}^{\text{position-binding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$

1 : $crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$
2 : $(C, i, m_i, m_i', \Lambda_i, \Lambda_i') \leftarrow \mathcal{A}_1(crs_n)$
3 : **if** $m_i \neq m_i' \wedge \mathsf{HVC.Ver}_{crs_n}(C, m, i, \Lambda_i) = 1 \wedge \mathsf{HVC.Ver}_{crs_n}(C, m', i, \Lambda_i') = 1$
4 : **return** 1,
5 : **else return** 0

FIGURE 6.5. The game of HVC Position-Binding

$\mathsf{Exp}^{\text{position-hiding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n, b)$

1 : $crs_n \leftarrow \mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n)$
2 : $(i, m_1, \cdots, m_{i-1}, m_{i,0}, m_{i,1}, m_{i+1}, \cdots m_n, state) \leftarrow \mathcal{A}_1(crs_n)$
3 : $\mathbf{m}_b = (m_1, \ldots, m_{i-1}, m_{i,b}, m_{i+1}, \ldots, m_n)$
4 : $(C, aux) \leftarrow \mathsf{HVC.com}(\mathbf{m}_b)$
5 : $\Lambda_j \leftarrow \mathsf{HVC.open}(j, \mathbf{m}_b, aux), \forall j \in [n]/\{i\}$
6 : $\forall j \in [n]/\{i\}, \Lambda_i \leftarrow VC.open(i, m_{i,b}, aux)$
7 : $b' \leftarrow \mathcal{A}_2(crs_n, C, \{\Lambda_j\}_{j\in[n]\setminus\{i\}}, state)$,
8 : **return** b'

FIGURE 6.6. The game of HVC Position-Hiding

Position-Hiding: The position hiding property not only requires that the adversary cannot distinguish whether a commitment is for a vector $(m_1, \ldots, m_n)$ or $(m_1', \ldots, m_n')$, but also guarantees that the adversary cannot learn any information about m_i from the opening of m_j where $i \neq j$. Formally, we have the HVC Position-Hiding game as Figure 6.6.

The advantage of $\mathcal{A}$ is defined as

$$\mathsf{Adv}^{\text{position-hiding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n) =$$
$$\left|\Pr[\mathsf{Exp}^{\text{position-hiding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\text{position-hiding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right|$$

DEFINITION 6.3.2. A HVC scheme satisfies HVC Position-Hiding if for every PPT adversary $\mathcal{A}$ the advantage function $\mathsf{Adv}^{\text{position-hiding}}_{\mathsf{HVC},\mathcal{A}}(1^\lambda, \mathcal{M}, n)$ is negligible in λ.

6.3.3 Construction

Although there are several vector commitment (VC) constructions [139, 141, 142, 143] that satisfy the homomorphic property, none of them can directly satisfy both the position hiding and homomorphic properties simultaneously. Furthermore, they cannot be made position hiding through the composition approach. For example, if one first commits to each message using a standard commitment scheme and then applies a VC to the resulting sequence of commitments, the resulting hybrid scheme will not satisfy the homomorphic property.

Even if one first commits to each message separately using a homomorphic commitment scheme (such as Pedersen commitment) and then applies a VC construction to the obtained sequence of commitments, the compatibility of the algebraic structures of these two underlying primitives is still unclear.[3] Some existing VC schemes are based on bilinear maps [139, 141], or RSA groups [139, 141, 142, 143], or lattice assumptions [147]. However, for pairing-based or RSA-based VC constructions, the messages (commitment values themselves in the above composition construction) are encoded into the exponents of group elements, which restricts the operation on messages to addition. This means that we require a homomorphic commitment scheme where the committed values lie in an additive group. Unfortunately, the existence of such a commitment scheme is elusive as most of the well-known computational assumptions do not hold, making it unclear how to construct such a scheme. For example, the homomorphic operation for Peterson commitment is multiplication, making it incompatible with pairing-based or RSA-based VC constructions for obtaining an HVC. Similarly, the message space of lattice-based VC consists of short vectors, while the standard lattice commitment consists of pseudorandom ring elements. It is also challenging to directly combine a lattice-based VC with a lattice-based commitment scheme to obtain an HVC.

Our proposed HVC construction is based on the pairing-based VC scheme introduced by Catalano et al. [139], which already possesses homomorphic properties and position binding. To achieve position hiding without compromising the homomorphic property, we add a

[3] Some recently proposed functional commitment schemes [145, 146] may also satisfy similar security requirements of HVC. However, it is unclear how to make these schemes compatible with UE schemes and thus integrate them into our proposed USS construction.

dummy position at the end of the vector, which is used to store a random value. The random value is then used to mask the information about the membership proof, thus achieving position hiding. Since the dummy position is not used for any actual file, it does not affect the integrity of the VC scheme. With this approach, we can achieve both homomorphic properties and position hiding in our HVC scheme, which is essential for our proposed construction in a USS system.

Let $\mathbb{G}$, $\mathbb{G}_T$ be bilinear groups of prime order p equipped with a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. Let $g \in \mathbb{G}$ be random generators.

$\mathsf{HVC.Setup}(1^\lambda, \mathcal{M}, n) \to crs_n$**:** Randomly choose $z_1, \ldots, z_n, z_{n+1} \leftarrow\!\!\$\ \mathbb{Z}_p$. For all $i = 1, \ldots, n+1$, set $h_i = g^{z_i}$, For all $i, j = 1, \ldots, n+1$, $i \neq j$ set $h_{i,j} = g^{z_i z_j}$. Output $crs_n = \big(g, \{h_i\}_{i\in[n+1]}, \{h_{i,j}\}_{i,j\in[n+1], i\neq j}\big)$.

$\mathsf{HVC.Com}_{crs_n}(\mathbf{m} = (m_1, \ldots, m_n)) \to (C, aux)$**:** Randomly select $r \leftarrow\!\!\$\ \mathbb{Z}_p$, Compute $C = h_1^{m_1} h_2^{m_2} \cdots h_n^{m_n} \cdot h_{n+1}^r$ and output C and the auxiliary information $aux = r$

$\mathsf{HVC.Open}_{crs_n}(\mathbf{m}, i, aux) \to \Lambda_i$ **:** Compute

$$\Lambda_i = \prod_{j=1, j\neq i}^{n} h_{i,j}^{m_j} \cdot h_{i,n+1}^r = \left(\prod_{j=1, j\neq i}^{n} h_j^{m_j} \cdot h_{n+1}^r \right)^{z_i}$$

$\mathsf{HVC.Ver}_{crs_n}(C, m, i, \Lambda_i) \to 1/0$**:** Check $e(C/h_i^m, h_i) = e(\Lambda_i, g)$.

$\mathsf{HVC.ComHom}_{crs_n}(C, C' \in \mathcal{C}) \to C''$**:** Compute $C'' = C \cdot C'$.

$\mathsf{HVC.OpenHom}_{crs_n}(\Lambda_j, \Lambda'_j \in \mathcal{P}) \to \Lambda''_j$**:** Compute $\Lambda''_j = \Lambda_j \cdot \Lambda'_j$.

The correctness and homomorpihc property of the scheme can be easily verified by inspection. We prove its security via the following theorem.

THEOREM 6.3.1. If the Square-CDH Assumption holds, then the scheme defined above satisfies the Position-Binding property.

PROOF. We prove the theorem by showing that the scheme satisfies the Position-Binding property. For sake of contradiction assume that there exists an efficient adversary $\mathcal{A}$ who produces two valid openings to two different messages at the same position, then we show how to build an efficient algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the Square-CDH Assumption.

To break the Square-CDH Assumption, $\mathcal{B}$ takes $g, g^a \in \mathbb{G}$ as its input and its goal is to compute g^{a^2}.

First, $\mathcal{B}$ selects a random $i \leftarrow\$ [n]$ as a guess for the index i on which $\mathcal{A}$ will break the position binding. And set $h_i = g^a$ Next, $\mathcal{B}$ chooses $z_j \leftarrow\$ \mathbb{Z}_p$, $\forall j \in [n+1]\backslash\{i\}$ and it computes:

$$\forall j \in [n+1]\backslash\{i\} : h_j = g^{z_j}, h_{i,j} = h_i^{z_j} = g^{az_j}$$

$$\forall k, j \in [n+1]\backslash\{i\}, k \neq j : h_{k,j} = g^{z_k z_j}$$

and outputs $crs_n = (g, \{h_j\}_{j\in[n+1]}, \{h_{j,k}\}_{j,k\in[n+1],j\neq k})$. Notice that the public parameters are perfectly distributed as the real ones. The adversary is supposed to output a tuple $(C, m, m', \Lambda, \Lambda')$ such that: $m \neq m'$ and both Λ and Λ' correctly verify at position i. If the position is not i, then $\mathcal{B}$ aborts the simulation. Otherwise, it computes $g^{a^2} = (\Lambda/\Lambda')^{(m'-m)^{-1}}$.

To see that the output is correct, observe that since the two openings verify correctly, then it holds: $e(C, h_i) = e(h_i^{m'}, h_i)e(\Lambda', g) = e(h_i^m, h_i)e(\Lambda, g)$. Notice that if $\mathcal{A}$ succeeds with probability ϵ, then $\mathcal{B}$ has probability ϵ/n of breaking the Square-CDH assumption. □

THEOREM 6.3.2. The scheme defined above is perfectly position hiding.

PROOF. We prove the theorem by showing that for two given vectors of messages $\mathbf{m}_0 = \{m_1, \ldots, m_{i-1}, m_{i,0}, m_{i+1}, \ldots, m_n\}$ and $\mathbf{m}_1 = \{m_1, \ldots, m_{i-1}, m_{i,1}, m_{i+1}, \ldots, m_n\}$, we can find two random values r_0 and r_1 such that $(\mathbf{m}_0, r_0)$ and $(\mathbf{m}_1, r_1)$ map to the same commitment value C and same proofs $\{\Lambda_j\}_{j\in[n]\backslash\{i\}}$ except Λ_{ib}. Since r_0, r_1 are chosen with equal probabilities according to the commitment algorithm $\mathsf{HVC.Com}$, any adversary $\mathcal{A}$ has a success to win the Position-Hiding game with a probability of exactly $1/2$.

Concretely, the challenger $\mathcal{C}$ sets the public parameters as the real environment: randomly choose $z_1, \ldots, z_n, z_{n+1} \leftarrow\$ \mathbb{Z}_p$. For all $i = 1, \ldots, n+1$, set $h_i = g^{z_i}$. For all $i, j = 1, \ldots, n+1$, $i \neq j$ set $h_{i,j} = g^{z_i z_j}$.

Upon receiving $\{m_1, \ldots, m_{i-1}, m_{i,0}, m_{i,1}, m_{i+1}, \ldots, m_n\}$ from $\mathcal{A}$, $\mathcal{C}$ randomly chooses r_0, computes commitment $C = h_1^{m_1} \cdot h_{i-1}^{m_{i-1}} \cdot h_i^{m_{i,0}} \cdot h_{i+1}^{m_{i+1}} \cdots h_n^{m_n} \cdots h_{n+1}^{r_0}$, and proofs $\Lambda_j = (C/{h_j}^{m_j})^{z_j}, j \in [n]\backslash\{i\}$. Obviously, $(C, \{\Lambda_j\}_{j\in[n]\backslash\{i\}})$ is the corresponding commitment

and openings to $(\mathbf{m}_0, r_0)$. $\mathcal{C}$ outputs $(C, \{\Lambda_j\}_{j\in[n]\setminus\{i\}})$. Note that if we set $r_1 = r_0 + (m_{i,0} - m_{i,1})z_i/z_{n+1}$, then the tuple $(C, \{\Lambda_j\}_{j\in[n]\setminus\{i\}})$ mentioned above can also serve as a commitment and openings for $(\mathbf{m}_1, r_1)$.

Since r_0 is randomly chosen, both r_0 and r_1 occur with equal probability. Therefore, the probability for $\mathcal{A}$ to win the Hiding game is exactly $1/2$. □

6.4 Construction of USS

In this section, we present our construction for achieving both confidentiality and integrity in a USS system. Our approach combines UE for confidentiality and VC for integrity. Specifically, we follow the VC-then-UE paradigm, where each file is treated as an element of VC, and its membership proof is appended at the end of the file. The file and its membership proof are then encrypted using UE schemes.

The main challenge in this approach is that updating one file changes all other membership proofs, which are encrypted together with the files using UE. However, general UE does not support file updates, which means that all storage must be retrieved, decrypted, and re-encrypted after each update. To reduce communication and user computation costs, we require a UE with homomorphic properties. To our knowledge, only the scheme RISE [27] satisfies this requirement and is compatible with the update operation of the membership proofs of our HVC scheme in Sec 6.3.

More precisely, let UE be any IND-ENC and IND-UPD secure updatable encryption scheme. RISE [27] is an IND-ENC and IND-UPD secure updatable encryption with homomorphic property described in Sec 2.6. Let COM be a standard commitment scheme with the hiding and binding property, and HVC be a homomorphic vector commitment with the position hiding and position binding property.

$\mathsf{USS.ParGen}(1^\lambda, \mathcal{M}, n)$**:** λ is the security parameter. $\mathcal{M}$ denotes the message space. n specifies the vector degree and the total number of stored files.

- Run the setup of UE and RISE to generate public parameter $ue.pp, rise.pp$.

- Let $hvc.crs_n$ be the public parameter of HVC.
- Let $com.pp$ be the public parameter of COM.

Then the public parameter $pp = (hvc.crs_n, ue.pp, rise.pp, com.pp)$ will be taken as the implicit input of the following algorithm.

$\mathsf{USS.KeyGen}(pp)$**:** take the public parameter pp as input, run the key generation algorithm of UE and RISE to generate the secret key $ue.sk, rise.sk$, and output the secret key $sk = (ue.sk, rise.sk)$.

$\mathsf{USS.Store}(\mathbf{m}, sk, pp)$**:** $\mathbf{m} = \{m_1, \ldots, m_n\}$, where each m_i denotes one file. The algorithm proceeds as follows:

(1) For each $i \in [n]$, randomly sample $r_i \leftarrow\!\!\$\ \{0,1\}^\lambda$, run $\mathsf{COM}(m_i; r_i) \to h_i$.

(2) For each $i \in [n]$, run $\mathsf{UE.Enc}(ue.sk, i\|m_i\|h_i\|r_i) \to \bar{f}_i$, where $\|$ denotes concatenation. Run $\mathsf{HVC.Com}(\mathbf{h}) \to (C, aux)$ to get the vector commitment C and the auxiliary input aux, where the message vector is $\mathbf{h} = (h_1, \ldots, h_n)$.

(3) For each $i \in [n]$, compute the proof Λ_i via running $\mathsf{HVC.Open}(i, \mathbf{h}, aux)$.

(4) For each $i \in [n]$, run $\mathsf{RISE.Enc}(rise.sk, \Lambda_i) \to \hat{f}_i$.

(5) Let $f_i = (\bar{f}_i, \hat{f}_i)$. Upload the total ciphertexts $\mathbf{f} = (f_1, \ldots, f_n)$ to the cloud storage service. Client stores the stub $sb = C$ and the secret key sk for the current epoch in the local storage.

$\mathsf{USS.Rev}_{client}(i, sk, sb, pp) \leftrightarrows \mathsf{USS.Rev}_{server}(\mathbf{f}, sb, pp)$**:** The client interacts with the server to retrieve the i-th file through the following procedure.

- $\mathsf{Rev}_{request}(i, sk, sb, pp) \to (q_{rev}, st_{rev})$: The client sends $q_{rev} = i$ to the server, where $i \in [n]$ and keeps a state $st_{rev} = (\text{“}retrieve\text{”}, i)$.
- The server holds the public parameter pp, and the storing file $\mathbf{f}$. When given the retrieve request q_{rev}, the server returns the q_{rev}-th file $f_{q_{rev}}$ as the response r_{rev}.
- $\mathsf{Rev}_{decrypt}(sk, sb, pp, st_{rev}, r_{rev}) \to m_i/\perp$: When given the server's response r_{rev}, the client parses the $sk = (ue.sk, rise.sk)$ and $f_i = (\bar{f}_i, \hat{f}_i)$. Run UEdecryption algorithm $\mathsf{UE.Dec}\left(ue.sk, \bar{f}_i\right) \to i\|m_i\|h_i\|r_i$. Run RISE decryption algorithm $\mathsf{RISE.Dec}\left(rise.sk, \hat{f}_i\right) \to \Lambda_i$.

- If the commitment verification $\mathsf{Com.Open}(com_i, m_i, r_i) \rightarrow 1$ and the homomorphic vector commitment verification $\mathsf{HVC.Ver}(C, h_i, i, \Lambda_i) \rightarrow 1$, then the client will output m_i, otherwise output $\perp$.

$\mathsf{USS.FileUp}_{client}(m_i', i, sk, sb, pp) \leftrightarrows \mathsf{USS.FileUp}_{server}(\mathbf{f}, pp)$**:** The file update procedure allows the client to update the i-th file m_i to m_i' with the collaboration of the server. More precisely, the interaction procedure is as follows.

(1) $\mathsf{FileUp}_{request}(m_i', i, sk, sb, pp) \rightarrow q_{fup}, st_{fup}$: The client sends the file update request $q_{fup} = (\text{“}FileUpdate\text{”}, i) = st_{fup}$ to the server to request the i-th encrypted file and keep a state st_{fup}.

(2) $\mathsf{FileUp}_{response}(\mathbf{f}, pp, q_{fup}) \rightarrow (f_i, sr_{fup})$: The server returns the i-th encrypted file f_i to the client as the response r_{fup} and keep the internal state $sr_{fup} = (\text{“}FileUpdate\text{”}, i)$.

(3) $\mathsf{FileUp}_{token}(sk, sb, pp, m_i', f_i, st_{fup}) \rightarrow (sb', tk_{fup})$: The client first parses $sk = (ue.sk, rise.sk)$ and $f_i = (\bar{f}_i, \hat{f}_i)$. Then run UE decryption algorithm $\mathsf{UE.Dec}(ue.sk, \bar{f}_i) \rightarrow i \| m_i \| h_i \| r_i / \perp$, and RISE decryption algorithm $\mathsf{RISE.Dec}\left(rise.sk, \hat{f}_i\right) = \Lambda_i / \perp$. If both decryptions are not $\perp$, then run the following verification algorithms. If the commitment opens to the different file $\mathsf{COM.open}(h_i; r_i) \neq m_i$, or the homomorphic vector commitment verification $\mathsf{HVC.Ver}(sb, h_i, i, \Lambda_i) \neq 1$, then output $\perp$, otherwise continues: The client replaces the plaintext file with m_i', samples a randomness $r_i' \leftarrow\!\!\$\ \{0,1\}^\lambda$, and commits it by running $\mathsf{COM}(m_i'; r_i') \rightarrow h_i'$. Encrypt the new file with $\mathsf{UE.Enc}(ue.sk, i \| m_i' \| h_i' \| r_i') \rightarrow \bar{f}_i'$. Set the change of message vector $\mathbf{m}_\delta = (0, \ldots, \delta_i = h_i' - h_i, \ldots, 0)$, and get homomorphic vector commitment $C_{\mathbf{m}_\delta}$ via running $\mathsf{HVC.Com}(\mathbf{m}_\delta) = (C_{\mathbf{m}_\delta}, aux)$. Then update the stub sb by running the vector commitment homomorphic algorithm $\mathsf{HVC.ComHom}(sb, C_{\mathbf{m}_\delta}) = sb'$. The client keeps the new stub sb' and sends the file update token $tk_{fup} = (\bar{f}_i', \mathbf{m}_\delta, aux, y = g^{rise.sk})$ to the server. Please note that the change of message vector $\mathbf{m}_\delta$ could be compressed to a constant size independent of the vector degree since it contains redundant 0 with $n-1$ degrees.

(4) $\mathsf{FileUp}_{update}(\mathbf{f}, pp, sr_{fup}, tk_{fup}) \to \mathbf{f}'$: On receiving $tk_{fup} = (\bar{f}_i', \mathbf{m}_\delta, aux, y)$ from the client, parse $\mathbf{f} = \Big((\bar{f}_1, \hat{f}_1), \ldots, (\bar{f}_n, \hat{f}_n)\Big)$. For all $j \in [n]$, the server will run $\mathsf{HVC.Open}(j, \mathbf{m}_\delta, aux) = \Lambda_{\delta_j}$, get RISE ciphertext $rise.C_{\delta_j} = (y^r, g^r \cdot \Lambda_{\delta_j})$, and get updated proof encryption

$$\hat{f}_j' = \hat{f}_j \cdot rise.C_{\delta_j} = \mathsf{RISE.Enc}(rise.sk, \Lambda_j) \cdot rise.C_{\delta_j} = \mathsf{RISE.Enc}(rise.sk, \Lambda_j \cdot \Lambda_{\delta_j})$$

(The last equation is a result of the homomorphic property of RISE). Then the updated ciphertexts are $\mathbf{f}' = (f_1', \ldots, f_n')$, where $f_j' = (\bar{f}_j, \hat{f}_j')$ for $j \in [n] \setminus i$ and $f_i' = (\bar{f}_i', \hat{f}_i')$.

$\mathsf{USS.KeyUp}_{client}(sk, sk', sb, pp) \leftrightarrows \mathsf{USS.KeyUp}_{server}(\mathbf{f}, pp)$**:** The interactive procedure between the client and the server updates the stored ciphertexts to the encryption under a new key sk'. The details are as follows.

(1) $\mathsf{KeyUp}_{token}(sk, sk', sb, pp) \to (tk, sb')$: The client first parses $sk = (ue.sk, rise.sk)$, $sk' = (ue.sk', rise.sk')$. Then get a homomorphic commitment on $\mathbf{0}$ via $\mathsf{HVC.Com}(\mathbf{0}) = (C_{\mathbf{0}}, aux)$, and re-randomize the stub via running $\mathsf{HVC.ComHom}(sb, C_{\mathbf{0}}) \to sb'$. Run UE's token generation algorithm $\mathsf{UE.Next}$ $(ue.sk, ue.sk') \to \Delta$ to generate the UE key update token Δ. Run RISE token generation algorithm $\mathsf{RISE.Next}(rise.sk, rise.sk') \to rise.\Delta$ to generate the RISE key update token $rise.\Delta = (rise.\Delta_1, y)$. Send $tk = (\Delta, rise.\Delta, \mathbf{0}, aux)$ as the key update token for the repository. The stub is updated as sb'. Please note that $\mathbf{0}$ could be compressed into constant size, so tk is still succinct.

(2) $\mathsf{KeyUp}_{update}(\mathbf{f}, pp, tk) \to (\mathbf{f}')$: Server parses $tk = (\Delta, rise.\Delta, \mathbf{0}, aux)$, $rise.\Delta = (rise.\Delta_1, y)$, and $\mathbf{f} = (f_1, \ldots, f_n)$, where $f_i = (\bar{f}_i, \hat{f}_i)$ for $i \in [n]$. For each $i \in [n]$, run UE re-encryption algorithm $\mathsf{UE.Upd}(\Delta, \bar{f}_i) \to \bar{f}_i'$ to update the data part. For each $i \in [n]$, run RISE re-encryption algorithm $\mathsf{RISE.Upd}(rise.\Delta_1, \hat{f}_i) \to \hat{f}_i'$ to re-encrypt the proof part. Then for $i \in [n]$, the server runs $\mathsf{HVC.Open}(i, \mathbf{0}, aux) = \Lambda_{\mathbf{0}_i}$, get RISE its encryption $rise.C_i = (y^r, g^r \cdot \Lambda_{\mathbf{0}_i})$ and to re-randomize the updated proof encryption

$$\hat{f}_i'' = \hat{f}_i' \cdot rise.C_i = \mathsf{RISE.Enc}(rise.sk', \Lambda_i \cdot \Lambda_{\mathbf{0}_i}).$$

(The last equation is because of the homomorphic property of RISE)
Finally, the updated ciphertexts are $\mathbf{f}' = (f_1'', \ldots, f_{n-1}'')$, where $f_i'' = \left(\bar{f}_i', \hat{f}_i''\right)$ for $i \in [n]$.

Instantiation. In the USS construction, the HVC is instantiated in section 6.3, the COM scheme could be instantiated with any secure commitment scheme with hiding and binding property, and the UE could be instantiated with any IND-ENC and IND-UPD secure UE schemes [27]. We know that in USS, the membership proof of HVC is encrypted by the RISE encryption algorithm. We require that the homomorphism of HVC and RISE is compatible.

6.4.1 Security analysis

We formally state the security properties of the construction USS in the following theorems and prove our theorems.

THEOREM 6.4.1. If UE is an IND-ENC secure updatable encryption scheme, COM is a secure commitment scheme with hiding property, then our USS is IND-DD-CPA secure.

PROOF. As IND-DD-CPA security is defined in Definition 6.2.1, we need to prove that

$$\left|\Pr[\mathsf{Exp}^{\text{ind-dd-up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\text{ind-dd-up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right| = negl(\lambda).$$

We prove our claim via a game sequence. In a high level, we consider the game $\mathsf{Exp}^{\text{ind-dd-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$ as the starting point. In the first game, we replace the updatable encryption ciphertexts $\bar{f}_i$ with ciphertexts of random strings. The IND-ENC security of UE ensures indistinguishability. In the second game, we replace the commitment of message $m_{0,i}$ with the commitment values of message $m_{1,i}$. In the third game, we replace the updatable encryption ciphertexts $\bar{f}_i$ with ciphertexts of $i\|m_{1,i}, \|h_{1,i}\|r_{1,i}$, which is the experiment $\mathsf{Exp}^{\text{ind-dd-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$. The detailed games are as follows:

Game$_0$**:** This game is same as the experiment $\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. We define G_0 to be the event that $\mathcal{A}$ outputs 1 in Game$_0$. So,

$$\Pr[G_0] = \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1].$$

Game$_1$**:** In this game, the challenger $\mathcal{C}$ modifies the behavior of $\mathsf{Store}(db_0, sk_e, pp)$ in response to the challenge, compared to **Game**$_0$. Specifically, for each $i \in \{1, \ldots, n\} \wedge m_{0,i} \neq m_{1,i}$, $\mathcal{C}$ randomly selects $(m_{r,i}, r_{r,i}, h_{r,i})$ from the corresponding message, randomness, and commitment spaces and replaces the invocation of $\mathsf{UE.Enc}(i\|m_{0,i}\|h_{0,i}\|r_{0,i}) \rightarrow \bar{f}_i$ with $\mathsf{UE.Enc}(i\|m_{r,i}\|h_{r,i}\|r_{r,i}) \rightarrow \bar{f}_{r,i}$. Correspondingly, in rep^*, $\bar{f}_i$ is replaced with $\bar{f}_{r,i}$. Additionally, $\mathcal{C}$ records $h_{0,i}$ for file update queries. All other operations, including answering the challenge and post-processing, remain the same as **Game** $_0$.

Specifically, $\mathcal{C}$ still uses the same $h_{0,i}$ to run the HVC algorithm to generate the stub and openings when answering the challenge. If the file update query $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ is related to file $m_{0,i}$, i.e., $sb = sb^* \wedge i \in \mathcal{I}$, $\mathcal{C}$ skips the UE decryption and COM verification steps. Instead, $\mathcal{C}$ retrieves the record $h_{0,i}$ as the decrypted commitment to perform the HVC verification and calculates the change of commitment $\delta_i = h'_i - h_{0,i}$ as a part of the file update transcript ft_i, which will be added to set $\mathcal{I}$. All other operations are identical to those in **Game**$_0$.

Let G_1 be the event that $\mathcal{A}$ outputs 1 in Game$_1$. We claim that

$$|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc},$$

where $\epsilon_{ind-enc}$ is the IND-ENC advantage of the UE scheme (which is negligible if UE is IND-ENC secure). This claim can be proven by observing that in Game $_0$, $\bar{f}_i$ is a UE encryption of message $i\|m_{0,i}\|h_{0,i}\|r_{0,i}$, while in Game $_1$, it is a UE encryption of message $i\|m_{r,i}\|h_{r,i}\|r_{r,i}$. Since UE is IND-ENC secure, the adversary $\mathcal{A}$ cannot distinguish between the two games.

Game$_2$**:** In this game, the challenger $\mathcal{C}$ introduces a small modification when running Store to answer the challenge. Specifically, for each $i \in \{1, \ldots, n\} \wedge m_{0,i} \neq m_{1,i}$, instead of using $\mathsf{COM}(m_{0,i}; r_{0,i})$ to generate $h_{0,i}$, $\mathcal{C}$ replaces $h_{0,i}$ with a commitment

$h_{1,i} \leftarrow \mathsf{COM}(m_{1,i}; r_{1,i})$ to message $m_{1,i}$. Then, $\mathcal{C}$ uses $h_{1,i}$ to run the HVC algorithm to obtain sb^*, rep^*, and update the set $\mathcal{L}$. Finally, replace the record $h_{0,i}$ with $h_{1,i}$.

Let G_2 be the event that $\mathcal{A}$ outputs 1 in Game $_2$. We claim that $|\Pr[G_2] - \Pr[G_1]| \leq \epsilon_{hiding}$, where ϵ_{hiding} is the hiding-advantage of the COM scheme (which is negligible if COM is a secure commitment scheme with hiding property). This claim can be proven by observing that in Game $_1$, $\mathcal{A}$ obtains the commitment $h_{0,i}$ to message $m_{0,i}$, while in Game $_2$, $\mathcal{A}$ obtains the commitment $h_{1,i}$ to message $m_{1,i}$. However, due to the hiding property of COM, $\mathcal{A}$ cannot distinguish the two games.

Game$_3$: In this game, challenger $\mathcal{C}$ replaces the UE encryption on the random message to the encryption on a tuple of $i \| m_{1,i}, \| h_{1,i} \| r_{1,i}$ where $h_{1,i}$ is generated from $\mathsf{COM}(m_{1,i}; r_{1,i})$ as in Game$_2$. Then Game$_3$ is the same as experiment $\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$.

We define G_3 to be the event that $\mathcal{A}$ outputs 1 in Game$_3$. Then we claim

$$|\Pr[G_3] - \Pr[G_2]| = \left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \Pr[G_2]\right| \leq \epsilon_{ind-enc},$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of UE scheme (which is negligible if UE is IND-ENC secure).

The proof of this claim is identical to the claim $|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc}$. Thus, noticing the difference between Game$_3$ and Game$_2$ is in negligible probability due to the IND-ENC security of UE.

Then we get

$$\left|\Pr[\mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \mathsf{Exp}^{\mathsf{ind\text{-}dd\text{-}up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1]\right| \leq 2\epsilon_{ind-enc} + \epsilon_{hiding}$$

□

THEOREM 6.4.2. If UE is an IND-ENC secure updatable encryption scheme, COM is a secure commitment scheme with hiding property, then our USS is IND-FileUp-CPA secure.

PROOF. As IND-FileUp-CPA security is defined in Definition 6.2.2, we need to prove that $\left|\Pr[\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]\right| = neg(\lambda)$. We prove the IND-FileUp-CPA security via a sequence of games.

Specifically, we start from the original game $\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. In the first game, we replace the updatable encryption ciphertexts $\bar{f}_i$ with ciphertexts generated during the execution of $\mathsf{FileUp}_{token}(sk_e, sb, m_{0,i}, i, pp) \leftrightarrows \mathsf{FileUp}_{server}(sb, rep, pp) \rightarrow \langle sb^*; rep^* \rangle$. The IND-ENC security of UE ensures indistinguishability. In the second game, we replace the commitment of message $m_{0,i}$ with the commitment values of message $m_{1,i}$. In the third game, we replace the updatable encryption ciphertexts $\bar{f}_i$ with ciphertexts of $i \| m_{1,i}, \| h_{1,i} \| r_{1,i}$ generated during the execution of

$$\mathsf{FileUp}_{token}(sk_e, sb, m_{0,i}, i, pp) \leftrightarrows \mathsf{FileUp}_{server}(sb, rep, pp) \rightarrow \langle sb^*; rep^* \rangle,$$

which is the experiment $\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$.

Game$_0$: This game is same as the experiment $\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. We define G_0 to be the event that $\mathcal{A}$ outputs 1 in Game$_0$. So,

$$\Pr[G_0] = \Pr[\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1].$$

Game$_1$: In this game, challenger $\mathcal{C}$ changes a bit from **Game**$_0$ in running the $\mathsf{FileUp}_{token}(sk_e, sb, m_{0,i}, i, pp) \leftrightarrows \mathsf{FileUp}_{server}(sb, rep, pp) \rightarrow \langle sb^*; rep^* \rangle$ to answer the challenge. Concretely, for challenge index $i \in \mathcal{I}$, $\mathcal{C}$ randomly chooses $(m_{r,i}, r_{r,i}, h_{r,i})$ from the corresponding message, randomness, and commitment spaces and replaces running $\mathsf{UE.Enc}(i \| m_{0,i} \| h_{0,i} \| r_{0,i}) \rightarrow \bar{f}_i^*$ with running $\mathsf{UE.Enc}(i \| m_{r,i} \| h_{r,i} \| r_{r,i}) \rightarrow \bar{f}_{r,i}^*$. Correspondingly, in rep^*, replace the value of $\bar{f}_i^*$ with $\bar{f}_{r,i}^*$. $\mathcal{C}$ still use $h_{0,i}$ to continue the FileUp procedure and the i-th element of $\mathbf{m}_\delta$ is still $h_{0,i} - h_i$ in the file update transcript fpt. Besides, $\mathcal{C}$ record $h_{0,i}$ for file update query. Other operations including answering the challenge and the post-processing remain the same as **Game**$_0$. Specifically, $\mathcal{C}$ still uses the same $h_{0,i}$ to run the HVC algorithm to generate the stub and openings in answering the challenge. If the file update query $\mathcal{O}.\mathsf{FileUp}(sb, m_i', i)$ is related to file $m_{0,i}$, i.e., $sb = sb^* \wedge i \in \mathcal{I}$, $\mathcal{C}$ retrieves

e, sb, rep, db from $\mathcal{L}$, changes as follows in running FileUp procedure. $\mathcal{C}$ gets rid of UE decryption and verification check on $\bar{f}_i$, retrieves the record $h_{0,i}$ to calculate the change of commitment $\delta_i = h'_i - h_{0,i}$ as a part of file update transcript ft_i which will be added to set $\mathcal{I}$. Then follow the same operations to continue as in Game$_0$.

We define G_1 to be the event that $\mathcal{A}$ outputs 1 in Game$_1$. We claim that

$$|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of UE scheme (which is negligible if UE is IND-ENC secure).

The proof of this claim is essentially the observation that in Game$_0$ $\bar{f}_i^*$ is a UE encryption on message $i\|m_{0,i}\|h_{0,i}\|r_{0,i}$, while in Game$_1$, it is a UE encryption on the message $i\|m_{r,i}\|h_{r,i}\|r_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since UE is IND-ENC secure.

Game$_2$: In this game, challenger $\mathcal{C}$ makes one small change to the above game. Specifically, during running FileUp procedure to answer the challenge. For the challenge index $i \in \mathcal{I}$, instead of running COM$(m_{0,i}; r_{0,i})$ to generate $h_{0,i}$, $\mathcal{C}$ replaces $h_{0,i}$ with a commitment $h_{1,i} \leftarrow$ COM$(m_{1,i}; r_{1,i})$ to message $m_{1,i}$ to run the HVC algorithm to get sb^*, rep^* and update the set $\mathcal{L}$. Then the record on $h_{0,i}$ is changed to $h_{1,i}$.

We define G_2 to be the event that $\mathcal{A}$ outputs 1 in Game$_2$. We claim that

$$|\Pr[G_2] - \Pr[G_1]| = \epsilon_{hiding}$$

where ϵ_{hiding} is the hiding-advantage of COM scheme (which is negligible if COM is a secure commitment scheme with hiding property).

The proof of this claim is essentially the observation that in Game$_1$, $\mathcal{A}$ could get the commitment $h_{0,i}$ to message $m_{0,i}$, while in Game$_2$, $\mathcal{A}$ could get the commitment $h_{1,i}$ to message $m_{1,i}$. So $\mathcal{A}$ cannot distinguish them due to the hiding property of COM.

Game$_3$: In this game, challenger $\mathcal{C}$ replaces the UE encryption on the random message to the encryption on a tuple of $i\|m_{1,i}, \|h_{1,i}\|r_{1,i}$ where $h_{1,i}$ is generated from

$\mathsf{COM}(m_{1,i}; r_{1,i})$ as in Game$_2$. Then Game$_3$ is the same as experiment $\mathsf{Exp}^{\text{ind-dd-up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$.

We define G_3 to be the event that $\mathcal{A}$ outputs 1 in Game$_3$. Then we claim

$$|\Pr[G_3] - \Pr[G_2]| = \left|\Pr[\mathsf{Exp}^{\text{ind-dd-up}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \Pr[G_2]\right| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of UE scheme (which is negligible if UE is IND-ENC secure).

The proof of this claim is identical to the claim $|\Pr[G_1] - \Pr[G_0]| = \epsilon_{ind-enc}$. Thus, noticing the difference between Game$_3$ and Game$_2$ is in negligible probability due to the IND-ENC security of UE.

Then we get

$$\left|\Pr[\mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1] - \mathsf{Exp}^{\text{ind-fileup-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1]\right| \leq 2\epsilon_{ind-enc} + \epsilon_{hiding}$$

□

THEOREM 6.4.3. If UE is an IND-UPD secure updatable encryption scheme, RISE is an IND-ENC and IND-UPD secure updatable encryption scheme with homomorphic property, and HVC is a secure vector commitment with homomorphic property and position hiding, then our USS is IND-REENC-CPA secure.

PROOF. We prove the IND-REENC-CPA security via a sequence of games. As IND-REENC-CPA security is defined in Definition 6.2.3, we need to prove that $|\Pr[\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1] - \Pr[\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]| = negl(\lambda)$. In the high level, we start from $\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$ as the original game, and make one small change in each of a sequence of games, and end up at $\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$ as the final game. We reduce the advantage of distinguishing between every two adjacent games to the advantage of breaking the security of one of the building blocks including IND-UPD security of UE, IND-UPD security of RISE, IND-ENC security of RISE, and the hiding property of HVC. Since the building blocks satisfy the security requirements, the advantage of distinguishing every two adjacent games is negligible. There are a constant number of games, and the

summed advantage is still negligible, so the first game and last game are indistinguishable, which means the probability difference that the adversary outputs the same is negligible.

Game$_0$.: This game is same as $\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0)$. We define G_0 to be the event that $\mathcal{A}$ outputs 1. Then we get

$$\Pr[G_0] = \Pr[\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 0) = 1].$$

Game$_1$.: In this game, challenger $\mathcal{C}$ changes a bit from **Game**$_0$ in running $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^* \rangle$. Concretely, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ randomly chooses $m_{r,i}, h_{r,i}, r_{r,i}$ from the message space, commitment space and randomness space, respectively, where $|m_{r,i}| = |m_{0,i}|$. Then run

$\mathsf{UE.Enc}(sk_{e-1}, i \| m_{r,i} \| h_{r,i} \| r_{r,i}) \rightarrow \bar{f}_{r,i}$ and replaces $c_{0,i} = (\bar{f}_{0,i}, \hat{f}_{0,i})$ in rep_0 with $c_{r,i} = (\bar{f}_{r,i}, \hat{f}_{0,i})$ as input to run the KeyUp procedure. Since the key update version of $\bar{f}_{r,i}$ will be used in answering the $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ query when $sb = sb^* \wedge i \in \mathcal{I}$, challenger $\mathcal{C}$ will change the behavior as follows to reply such query. In the FileUp procedure, $\mathcal{C}$ gets rid of the decryption and integrity check of retrieved ciphertext, directly commits and encrypts on the new message m'_i and uses $h_{0,i}$, calculates the change of i-th commitment $\delta_i = h'_i - h_{0,i}$ where $h'_i \rightarrow \mathsf{COM}(m'_i; r'_i)$ to get the file update token and continue. Since the stub sb is generated using $h_{0,i}$ and binds with its vector commitment proof in the second part of i-th ciphertext, the same $h_{0,i}$ is used to update file to ensure the file updated version is a valid ciphertext, consistent with Game$_0$. $\mathcal{A}$ cannot notice the difference via corrupting both the epoch key and ciphertext of the file's updated version.

We define G_1 to be the event that $\mathcal{A}$ outputs 1 in Game$_1$. We claim that

$$|\Pr[G_1] - \Pr[G_0]| \leq \epsilon_{ind-upd}$$

where $\epsilon_{ind-upd}$ is the IND-UPD-advantage of UE scheme (which is negligible if UE is IND-UPD secure).

The claim can be proven by observing that in Game $_0$, $rep^* = \{c^*_{0,1}, \ldots, c^*_{0,n}\}$, where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{0,i})$, includes a UE re-encryption of ciphertext $\bar{f}_{0,i}$ that encrypts

message $i\|m_{0,i}\|h_{0,i}\|r_{0,i}$, while in Game $_1$, $c^*_{0,i}$ includes a UE re-encryption of ciphertext $\bar{f}_{r,i}$ that encrypts message $i\|m_{r,i}\|h_{r,i}\|r_{r,i}$. Since UE is IND-UPD secure, the adversary $\mathcal{A}$ should not be able to distinguish between the two games.

Game$_2$: This game, makes one small change from Game$_1$ to answer the challenge. Concretely, in $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^*\rangle$, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ randomly chooses $\Lambda_{r,i}$ from the vector commitment proof space to replace $\Lambda_{0,i} \leftarrow \mathsf{RISE.Dec}(sk_{e-1}, \hat{f}_{0,i})$, encrypts it $\mathsf{RISE.Enc}(sk_{e-1}, \lambda_{r,i}) \rightarrow \hat{f}_{r,i}$, replaces $\hat{f}_{0,i}$ with $\hat{f}_{r,i}$ as input to run the KeyUp procedure. Since the key update version of $\hat{f}_{r,i}$ will be used in answering the $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ query when $sb = sb^* \wedge i \in \mathcal{I}$, challenger $\mathcal{C}$ will change the behavior as follows to reply such query. In the FileUp procedure, when updating $\hat{f}_{r,i}$, $\mathcal{C}$ addtionally run $\mathsf{RISE.Enc}(sk_e, \Lambda_{0,i}/\Lambda_{r,i}) \leftarrow rise.C_\Delta$, and replace $\hat{f}^*_{r,i}$ with $\hat{f'}^*_{r,i} = \hat{f}^*_{r,i} \cdot rise.C_\Delta$ to answer the query. Since the stub sb is generated using $h_{0,i}$ and binds with its vector commitment proof $\Lambda_{0,i}$, while the second part of i-th ciphertext in set $\mathcal{L}$ is related to $\Lambda_{r,i}$ due to this game's change, $\mathcal{C}$ recover the i-th ciphertext to base on $\Lambda_{0,i}$ by running $\hat{f'}^*_{r,i} = \hat{f}^*_{r,i} \cdot rise.C_\Delta$, which is consistent with Game$_1$. $\mathcal{A}$ cannot notice the difference via corrupting both the epoch key and ciphertext of the file's updated version.

We define G_2 to be the event that $\mathcal{A}$ outputs 1 in Game$_2$. We claim that

$$|\Pr[G_2] - \Pr[G_1]| \leq \epsilon'_{ind-upd}$$

where $\epsilon'_{ind-upd}$ is the IND-UPD-advantage of RISE scheme (which is negligible if RISE is IND-UPD secure).

The proof of this claim is essentially the observation that in Game$_1$ $rep^* = \{c^*_{0,1}, \ldots, c^*_{0,n}\}$ where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{0,i})$ includes a RISE re-encryption of ciphertext $\hat{f}_{0,i}$ which encrypts proof $\Lambda_{0,i}$, while in Game$_2$, $c^*_{0,1}$ include a RISE re-encryption of ciphertext $\hat{f}_{r,i}$ which encrypts the proof $\Lambda_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-UPD secure.

Game$_3$.: This game, makes one small change from Game$_2$ to answer the challenge. Specifically, in $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^*\rangle$,

for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ randomly chooses $\Lambda_{\delta_r,i}$ from the vector commitment proof space to replace $\Lambda_{\mathbf{0}_i}$ to continue the KeyUp procedure. Since the challenge ciphertext with one change in this game can be updated to a non-challenge ciphertext if $\mathcal{A}$ queries $\mathcal{O}.\mathsf{FileUp}(sb, m'_i, i)$ where $sb = sb^* \wedge i \in \mathcal{I}$. The non-challenge ciphertext could be decrypted if $\mathcal{A}$ corrupts the epoch key, which is allowed. To make $\mathcal{A}$'s view the same as in Game$_2$, challenger $\mathcal{C}$ will change the behavior as follows to reply to such query. In the FileUp procedure, when updating $\hat{f}_{r,i}$, $\mathcal{C}$ addtionally run $\mathsf{RISE.Enc}(sk_e, \Lambda_{\mathbf{0}_i}/\Lambda_{\delta_r,i}) \leftarrow rise.C'_{\Delta}$, and replace $\hat{f'}^*_{r,i}$ with $\hat{f''}^*_{r,i} = \hat{f'}^*_{r,i} \cdot rise.C'_{\Delta}$ to answer the query. Since the stub sb binds with its vector commitment proof $\Lambda_{\mathbf{0}_i}$, while the second part of i-th ciphertext in set $\mathcal{L}$ is related to $\Lambda_{\delta_r,i}$ due to this game's change, $\mathcal{C}$ recover the i-th ciphertext to base on $\Lambda_{\mathbf{0}_i}$ by running $\hat{f''}^*_{r,i} = \hat{f'}^*_{r,i} \cdot rise.C'_{\Delta}$, which is consistent with Game$_2$. $\mathcal{A}$ cannot notice the difference via corrupting both the epoch key and ciphertext of the file's updated version.

We define G_3 to be the event that $\mathcal{A}$ outputs 1 in Game$_3$. We claim that

$$|\Pr[G_3] - \Pr[G_2]| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of RISE scheme (which is negligible if RISE is IND-ENC secure).

The proof of this claim is essentially the observation that in Game$_2$ $rep^* = \{c^*_{0,1}, \ldots, c^*_{0,n}\}$ where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{0,i})$ includes a RISE encryption of proof $\Lambda_{\mathbf{0}_i}$, while in Game$_3$, $c^*_{0,1}$ include a RISE encryption of the proof $\Lambda_{\delta_r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-ENC secure.

Game$_4$.: This game makes one small change of the stub of challenge ciphertext from Game$_3$ to answer the challenge. Specifically, in running KeyUp procedure to output $\langle sb^*; rep^* \rangle$, for each $i \in \{1, \ldots, n\}$, challenger $\mathcal{C}$ runs $\mathsf{UE.Dec}(sk_{e-1}, \bar{f}_{0,i}) \rightarrow (i \| m_{0,i} \| h_{0,i} \| r_{0,i})$ and $\mathsf{UE.Dec}(sk_{e-1}, \bar{f}_{1,i}) \rightarrow (i \| m_{1,i} \| h_{1,i} \| r_{1,i})$ and gets $\mathbf{m}_{\delta_{0-1}} = (h_{1,1} - h_{0,1}, \ldots, h_{1,n} - h_{0,n})$. Replace $\mathbf{0}$ with $\mathbf{m}_{\delta_{0-1}}$ to run $\mathsf{HVC.Com}(\mathbf{m}_{\delta 0-1}) \rightarrow (C_{\mathbf{m}_{\delta 0-1}}, aux_{\delta})$, updates stub $sb'^* = sb^* \cdot C_{\mathbf{m}_{\delta 0-1}}$, and generates key update token $tk^* = (\Delta^*, rise.\Delta^*, \mathbf{m}_{\delta 0-1}, aux_{\delta})$ to continue the KeyUp procedure. In this game

each value of $h_{0,i}, \Lambda_{0,i}$ are changed to $h_{1,i}, \Lambda_{1,i}$. If $i \notin \mathcal{I}$, which means $c_{0,i} = c_{1,i}$, also means $h_{0,i} = h_{1,i}$, the i-th ciphertext is a valid ciphertext that could be decrypted correctly with the corrupted epoch key and does not leak information about $h_{1,j}$ with other index j.

We define G_4 to be the event that $\mathcal{A}$ outputs 1 in Game$_4$. We claim that

$$|\Pr[G_4] - \Pr[G_3]| = \epsilon_{position-hiding}$$

where $\epsilon_{position-hiding}$ is the position-hiding-advantage of HVC scheme (which is negligible if HVC is a secure vector commitment scheme with position hiding and homomorphic property).

The proof of this claim is essentially the observation that in Game$_3$ sb^* is vector commitment corresponding to $rep_0 = \{c_{0,1}, \ldots, c_{0,n}\}$, while in Game$_4$, sb^* is a vector commitment corresponding to $rep_1 = \{c_{1,1}, \ldots, c_{1,n}\}$. For those indices i satisfying $c_{0,i} \neq c_{1,i}$, the vector commitment and other indices' message do not leak information about the i-th element. So the adversary $\mathcal{A}$ should not notice the difference since HVC is position hiding.

Game$_5$: This game makes one small change on Game$_4$, which is a reverse change of Game$_3$ or getting rid of the change in Game$_3$. Specifically, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ changes $\Lambda_{\delta_r,i}$ back to the proof generated from HVC.Open to continue the KeyUp procedure. We omit the details for simplicity. Please refer to Game$_3$ for details.

We define G_5 to be the event that $\mathcal{A}$ outputs 1 in Game$_5$. We claim that

$$|\Pr[G_5] - \Pr[G_4]| \leq \epsilon_{ind-enc}$$

where $\epsilon_{ind-enc}$ is the IND-ENC-advantage of RISE scheme (which is negligible if RISE is IND-ENC secure).

The proof of this claim is identical to the observation of Game$_3$ that in Game$_5$ $\hat{f}^*_{1,i}$) includes a RISE encryption of proof $\Lambda_{\mathbf{1}_i}$, while in Game$_4$, $\hat{f}^*_{r,i}$) include a RISE encryption of the proof $\Lambda_{\delta_r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-ENC secure.

Game$_6$: This game makes one small change on Game$_5$, which is a reverse change of Game$_2$ or getting rid of the change in Game$_2$. Specifically, in running $\mathsf{KeyUp}_{client}(sk_{e-1}, sk_e, sb, pp) \leftrightarrows \mathsf{KeyUp}_{server}(rep, pp)$ to output $\langle sb^*; rep^* \rangle$, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ replace $\Lambda_{r,i}$ with $\Lambda_{1,i} \leftarrow \mathsf{RISE.Dec}(sk_{e-1}, \hat{f}_{1,i})$, encrypts it $\mathsf{RISE.Enc}(sk_{e-1}, \lambda_{1,i}) \rightarrow \hat{f}_{1,i}$, replaces $\hat{f}_{r,i}$ with $\hat{f}_{1,i}$ as input to run the KeyUp procedure. We omit the details for simplicity. Please refer to Game$_2$ for details.

We define G_6 to be the event that $\mathcal{A}$ outputs 1 in Game$_6$. We claim that

$$|\Pr[G_6] - \Pr[G_5]| \leq \epsilon'_{ind-upd}$$

where $\epsilon'_{ind-upd}$ is the IND-UPD-advantage of RISE scheme (which is negligible if RISE is IND-UPD secure).

The proof of this claim is identical to the observation of Game$_2$. That is in Game$_6$ $rep^* = \{c^*_{0,1}, \ldots, c^*_{0,n}\}$ where $c^*_{0,i} = (\bar{f}^*_{0,i}, \hat{f}^*_{1,i})$ includes a RISE re-encryption of ciphertext $\hat{f}_{1,i}$ which encrypts proof $\Lambda_{1,i}$, while in Game$_5$, $c^*_{0,1}$ include a RISE re-encryption of ciphertext $\hat{f}_{r,i}$ which encrypts the proof $\Lambda_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since RISE is IND-UPD secure.

Game$_7$: This game makes one small change on Game$_6$, which is a reverse change of Game$_1$ or getting rid of the change in Game$_1$. Concretely, for each $i \in \mathcal{I}$, which means $c_{0,i} \neq c_{1,i}$, challenger $\mathcal{C}$ replaces $\bar{f}_{r,i}$ with $\bar{f}_{1,i}$ where $c_{1,i} = (\bar{f}_{1,i}, \hat{f}_{1,i})$ in rep_1, as input to run the Keyup procedure. We omit the details for simplicity. Please refer to Game$_1$ for details.

We define G_7 to be the event that $\mathcal{A}$ outputs 1 in Game$_7$. Game$_7$ is the same as $\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1)$. So $\Pr[G_7] = \Pr[\mathsf{Exp}^{\mathsf{ind\text{-}reenc\text{-}cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda, \mathcal{M}, n, 1) = 1]$. We claim that

$$|\Pr[G_7] - \Pr[G_6]| \leq \epsilon_{ind-upd}$$

where $\epsilon_{ind-upd}$ is the IND-UPD-advantage of UE scheme (which is negligible if UE is IND-UPD secure).

The proof of this claim is identical to the observation of Game$_1$. That is, in Game$_7$ $rep^* = \{c^*_{1,1}, \ldots, c^*_{1,n}\}$ where $c^*_{1,i} = (\bar{f}^*_{1,i}, \hat{f}^*_{1,i})$ includes a UE re-encryption of ciphertext $\bar{f}_{1,i}$ which encrypts message $i \| m_{1,i} \| h_{1,i} \| r_{1,i}$, while in Game$_6$, $c^*_{0,1}$ include

a UE re-encryption of ciphertext $\bar{f}_{r,i}$ which encryptps the message $i\|m_{r,i}\|h_{r,i}\|r_{r,i}$. So the adversary $\mathcal{A}$ should not notice the difference since UE is IND-UPD secure.

Then we get

$$\begin{aligned}&\left|\Pr[\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda,\mathcal{M},n,1)=1]-\mathsf{Exp}^{\text{ind-reenc-cpa}}_{\mathsf{USS},\mathcal{A}}(1^\lambda,\mathcal{M},n,0)=1]\right|\\ \leq&2\epsilon_{ind-upd}+2\epsilon'_{ind-upd}+2\epsilon_{ind-enc}+\epsilon_{position-hiding}\end{aligned}$$

□

THEOREM 6.4.4. Let USS denote an updatable secure storage scheme. COM is secure commitment with binding property, and HVC is a secure vector commitment with position binding, then USS is OF-PTXT secure.

PROOF. Intuitively, we first assume that USS is not OF-PTXT secure, and then construct contradictions with the existing properties of building blocks to prove the theorem. In the $\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda,\mathcal{M},n)$ experiment, given the stub sb_e and the epoch key sk_e, to win the game, $\mathcal{A}$ needs to provide a ciphertext f_i and interact with challenger running USS.Rev procedure and enable the challenger to retrieve a file m'_i which is different from the i-th element m_i of the latest message vector $\mathbf{m}$ corresponding to the stub sb_e.

Since we assume USS is not OF-PTXT secure, then there exists $\mathcal{A}$ winning the experiment $\mathsf{Exp}^{\text{of-ptxt}}_{\mathsf{USS},\mathcal{A}}(1^\lambda,\mathcal{M},n)$ by enabling the file $m'_i \neq m_i$ retrieval. That means the commitment of m'_i passes the verification algorithm of HVC.Ver. So $\mathcal{A}$ could win in two cases: one case is that $\mathcal{A}$ finds a collision for the commitment h_i, i.e., $\mathsf{COM}(m'_i;r'_i)=\mathsf{COM}(m_i;r_i)=h_i$, which is contradictory with the binding property of COM; the other case is that $\mathcal{A}$ finds the collision for the HVC, i.e., two different elements $h_i \neq h'_i$ pass the i-th vector commitment verification for the same stub sb_e, which is contradictory with HVC's position-binding property. So we can reduce USS's OF-PTXT property to COM's binding property and HVC's position binding property. □

6.5 Future Works

Although Updatable Encryption (UE) is a promising approach for secure storage of data, it cannot be directly applied to frequently updated databases due to the risk of a malicious server inducing the client to accept an outdated version of a file instead of the latest one. To address this issue, we propose a scheme called Updatable Secure Storage (USS) that provides a secure and key-rotatable solution for dynamic databases. However, we acknowledge that the USS scheme presented in this chapter is still far from practical due to its high computational and communication overheads. Therefore, we suggest several directions for future work to improve the efficiency and usability of the USS scheme.

General construction for general UE. In this paper, we present a construction based on ciphertext-independent UE (CIUE) schemes, which we describe using the UE syntax introduced in this work. Our construction can be extended to more general ciphertext-dependent UE (CDUE) schemes. From a construction perspective, this extension is straightforward. However, ensuring security in the presence of both CIUE and CDUE requires a careful consideration of the security model. CDUE models offer more fine-grained control over token corruption, which requires a more nuanced modeling and analysis approach. We show that existing CDUE schemes are secure under some applicable security models and can be used as black-box components in the proof of USS security. This result has not been formalized or proven before.

Enhancing performance for files of significant size. Our USS scheme has the potential to offer practical efficiency, even when encrypting large files, thanks to its adherence to the KEM + DEM paradigm. Specifically, the DEM component of our scheme can be any UE ciphertext with IND-CPA security, allowing for the construction of an efficient DEM component directly from existing UE schemes. A black-box approach, such as the one proposed in [124], can be used to achieve this goal, thereby leveraging the efficiency of UE schemes. Meanwhile, the KEM component operates on a constant amount of data, ensuring strong security guarantees against attacks on data confidentiality and integrity without sacrificing efficiency. The ability

to efficiently encrypt large files is particularly important in scenarios where substantial volumes of data are outsourced.

Towards full-fledged version control with post-compromise security. In this paper, we focus on achieving the first step of version control, which involves ensuring the integrity of the latest version of a database. However, we do not currently support a fall-back function that would allow users to revert to any prior version of the database. Enabling this feature would require the database to store all historical versions or changes. One approach to implementing a limited fall-back function involves pre-configuring a set of versions for a file with empty or meaningless data and updating the corresponding version when the user makes changes. The stub would contain information on all versions to prevent malicious servers from manipulating any version to deceive users. During each key update, all versions of ciphertext undergo key rotation to hide the update history from external attackers.

CHAPTER 7

Conclusion and Future Works

In this dissertation, we initiate the study of end-to-end secure storage and focus our research on three key dimensions: end-to-end secure storage services for other Apps, with version control functionality and full-fledged end-to-end security, and with post-compromise security. The specific open problems related to each dimension are discussed in detail in the respective technical chapters. Here, we take a broader perspective to provide an overview of secure storage as a service, highlighting broader and more intriguing research directions.

End-to-end secure storage is both an important and fascinating area of study. The secure storage industry predates its academic exploration, but both remain far from achieving the maturity seen in fields like end-to-end secure messaging. Numerous interesting challenges persist, including understanding the security of practical systems mentioned in Sec 4.6, supporting more advanced security such as post-compromise security in practical design mentioned in Sec3.8 and Sec 6.5, enabling more advanced data operations supported in plain cloud storage, such as search mentioned in Sec 3.8, code review of GitHub mentioned in Sec 4.6, and more.

Looking beyond its standalone significance, end-to-end secure storage can also be envisioned as a foundational service that facilitates and enhances the security and functionality of other services mentioned a bit in Sec 3.8, such as end-to-end secure messaging, online collaboration, and more. One of our ongoing projects explores how end-to-end secure storage can be leveraged to build a more robust end-to-end secure messaging service. Additionally, services like online collaboration and even web service could benefit significantly from end-to-end secure storage, not only in terms of enhanced security but also by addressing further challenges such as accessibility, service robustness, and data processing efficiency.

Bibliography

[1] Maya Kamath. *The Internet Era before Facebook : AltaVista, GeoCities, Lycos, Netscape and Other Giants Of Web 1.0!* https://www.techworm.net/2015/03/internet-era-before-facebook-altavista-geocities-lycos-netscape-other-giants-web-1-0.html. 2015.

[2] Amazon Web Services. *AWS Simple Cloud Storage*. https://aws.amazon.com/pm/serv-s3/. Accessed: 2024-12-26. 2024.

[3] Microsoft Azure. *Introduction to Azure Storage*. https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction. Accessed: 2024-12-26. 2024.

[4] Google Cloud. *Object storage for companies of all sizes*. https://cloud.google.com/storage. Accessed: 2024-12-26. 2024.

[5] Apple Newsroom. *Report: 2.6 billion personal records compromised by data breaches in past two years — underscoring need for end-to-end encryption*. https://www.apple.com/newsroom/2023/12/report-2-point-6-billion-records-compromised-by-data-breaches-in-past-two-years/. 2023.

[6] EDPB. *1.2 billion euro fine for Facebook as a result of EDPB binding decision*. https://www.edpb.europa.eu/news/news/2023/12-billion-euro-fine-facebook-result-edpb-binding-decision_en. 2023.

[7] BDO. *TikTok Receives Significant GDPR Fine for Mishandling Children's Data*. https://www.bdo.co.uk/en-gb/insights/advisory/risk-and-advisory-services/tiktok-receives-significant-gdpr-fine-for-mishandling-childrens-data. 2023.

[8] Vulnerable U. *U.S. Officials Tell Americans to Use Encrypted Apps as Scope of Cyberattack Grows*. https://www.vulnu.com/p/u-s-officials-tell-americans-to-use-encrypted-apps-as-scope-of-cyberattack-grows. 2024.

[9] Martin R. Albrecht et al. 'Caveat Implementor! Key Recovery Attacks on MEGA'. In: *EUROCRYPT (5)*. Vol. 14008. Lecture Notes in Computer Science. Springer, 2023, pp. 190–218.

[10] Matilda Backendal, Miro Haller and Kenneth G. Paterson. 'MEGA: Malleable Encryption Goes Awry'. In: *SP*. IEEE, 2023, pp. 146–163.

[11] Jonas Hofmann and Kien Tuong Truong. 'End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem'. In: *CCS*. ACM, 2024, pp. 3988–4001.

[12] Internet Engineering Task Force (IETF). *The Transport Layer Security (TLS) Protocol Version 1.3*. https://datatracker.ietf.org/doc/html/rfc8446. 2018.

[13] Signal Messenger. *Speak Freely*. https://signal.org/. Accessed: 2024-12-26.

[14] WhatsApp. *WhatsApp | Secure and Reliable Free Private Messaging and Calling*. https://www.whatsapp.com/. Accessed: 2024-12-26.

[15] Craig Gentry. 'Fully homomorphic encryption using ideal lattices'. In: *STOC*. ACM, 2009, pp. 169–178.

[16] MEGA Limited. *MEGA*. https://mega.io/about. Apr. 2024.

[17] pCloud. *pCloud - The Most Secure Cloud Storage*. https://www.pcloud.com/. Accessed: 2024-12-26. 2024.

[18] Proton. *Protocol Drive*. https://proton.me/drive.

[19] Matilda Backendal et al. 'A Formal Treatment of End-to-End Encrypted Cloud Storage'. In: *CRYPTO (2)*. Vol. 14921. Lecture Notes in Computer Science. Springer, 2024, pp. 40–74.

[20] Long Chen et al. 'End-to-Same-End Encryption: Modularly Augmenting an App with an Efficient, Portable, and Blind Cloud Storage'. In: *USENIX Security Symposium*. USENIX Association, 2022, pp. 2353–2370.

[21] Dan Boneh et al. 'Key Homomorphic PRFs and Their Applications'. In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. 2013, pp. 410–428.

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

[23] Dick Hardt. *The OAuth 2.0 authorization framework*. 2012.

[24] Sampath Srinivas et al. 'Universal 2nd factor (U2F) overview'. In: *FIDO Alliance Proposed Standard* . (2015).

[25] Adam Everspaugh et al. 'Key Rotation for Authenticated Encryption'. In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*. 2017, pp. 98–129.

[26] Torben Pryds Pedersen. 'Non-interactive and information-theoretic secure verifiable secret sharing'. In: *Annual international cryptology conference*. Springer. 1991, pp. 129–140.

[27] Anja Lehmann and Björn Tackmann. 'Updatable Encryption with Post-Compromise Security'. In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*. 2018, pp. 685–716.

[28] 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. Online; accessed 6 January 2020. 2016.

[29] Ali Bagherzandi et al. 'Password-protected secret sharing'. In: *CCS*. New York: ACM, 2011, pp. 433–444.

[30] Jan Camenisch, Anna Lysyanskaya and Gregory Neven. 'Practical yet universally composable two-server password-authenticated secret sharing'. In: *CCS*. New York: ACM, 2012, pp. 525–536.

[31] Jan Camenisch et al. ‘Memento: How to reconstruct your secrets from a single password in a hostile environment’. In: *CRYPTO*. Berlin: Springer, 2014, pp. 256–275.

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

[33] Stanislaw Jarecki, Aggelos Kiayias and Hugo Krawczyk. ‘Round-optimal password-protected secret sharing and T-PAKE in the password-only model’. In: *ASIACRYPT*. Berlin: Springer, 2014, pp. 233–253.

[34] Stanislaw Jarecki et al. ‘Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online)’. In: *EuroS&P*. New York: IEEE, 2016, pp. 276–291.

[35] Stanisław Jarecki et al. ‘TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF’. In: *ACNS*. Berlin: Springer, 2017, pp. 39–58.

[36] Lin Zhang, Zhenfeng Zhang and Xuexian Hu. ‘UC-secure two-server password-based authentication protocol and its applications’. In: *AsiaCCS*. New York: ACM, 2016, pp. 153–164.

[37] Russell WF Lai et al. ‘Simple password-hardened encryption services’. In: *USENIX Security*. Berkeley: USENIX Association, 2018, pp. 1405–1421.

[38] Stanislaw Jarecki, Hugo Krawczyk and Jason Resch. ‘Updatable oblivious key management for storage systems’. In: *CCS*. New York: ACM, 2019, pp. 379–393.

[39] Kathleen Moriarty, Burt Kaliski and Andreas Rusch. *PKCS# 5: password-based cryptography specification version 2.1*. Tech. rep. IETF, 2017.

[40] Xavier Boyen. ‘Hidden credential retrieval from a reusable password’. In: *ASIACCS*. New York: ACM, 2009, pp. 228–238.

[41] Adam Everspaugh et al. ‘The Pythia PRF Service’. In: *USENIX Security*. Berkeley: USENIX Association, 2015, pp. 547–562.

[42] Jonas Schneider et al. ‘Efficient cryptographic password hardening services from partially oblivious commitments’. In: *CCS*. New York: ACM, 2016, pp. 1192–1203.

[43] Russell WF Lai et al. 'Phoenix: Rebirth of a cryptographic password-hardening service'. In: *USENIX Security*. Berkeley: USENIX Association, 2017, pp. 899–916.

[44] Philip MacKenzie, Thomas Shrimpton and Markus Jakobsson. 'Threshold password-authenticated key exchange'. In: *CRYPTO*. Berlin: Springer, 2002, pp. 385–400.

[45] Maliheh Shirvanian et al. 'Sphinx: A password store that perfectly hides passwords from itself'. In: *ICDCS*. New York: IEEE, 2017, pp. 1094–1104.

[46] Sarah Pearman et al. 'Let's go in for a closer look: Observing passwords in their natural habitat'. In: *CCS*. New York: ACM, 2017, pp. 295–310.

[47] Jad S Boutros et al. *Device independent encrypted content access system*. US Patent 10,341,304. July 2019.

[48] Brandon Jones. *Do Snapchat Memories Take Up Space on Your Phone?* https://www.psafe.com/en/blog/snapchat-memories-take-space-phone/. Online; accessed 6 January 2020. 2017.

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

[50] WhatsApp Security. *How WhatsApp is enabling end-to-end encrypted backups*. Website. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf. 2021.

[51] Jo Van Bulck, Frank Piessens and Raoul Strackx. 'SGX-Step: A practical attack framework for precise enclave execution control'. In: *SysTEXSOSP*. New York: ACM, 2017, pp. 1–6.

[52] Ferdinand Brasser et al. 'Software grand exposure: SGX cache attacks are practical'. In: *WOOT*. Berkeley: USENIX Association, 2017.

[53] Oleksii Oleksenko et al. 'Varys: Protecting SGX enclaves from practical side-channel attacks'. In: *USENIX ATC*. Berkeley: USENIX Association, 2018, pp. 227–240.

[54] Andrea Biondo et al. 'The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX'. In: *USENIX Security*. Berkeley: USENIX Association, 2018, pp. 1213–1227.

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

[56] Brendan McMahan and Daniel Ramage. 'Federated learning: Collaborative machine learning without centralized training data'. In: *Google Research Blog* 3 (2017), p. .

[57] IBM Security. *IBM HSM Products*. https://www.ibm.com/security/cryptocards. May 2018.

[58] Michel Abdalla et al. 'Robust password-protected secret sharing'. In: *ESORICS*. Berlin: Springer, 2016, pp. 61–79.

[59] 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)*. Berkeley: USENIX Association, 2020, pp. 1121–1138.

[60] Moni Naor and Omer Reingold. 'Number-theoretic constructions of efficient pseudo-random functions'. In: *FOCS*. New York: IEEE, 1997, pp. 458–467.

[61] Michael J Freedman et al. 'Keyword search and oblivious pseudorandom functions'. In: *TCC*. Berlin: Springer, 2005, pp. 303–324.

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

[63] Apple Inc. *Set up iCloud Keychain*. Website. SetupiCloudKeychain. 2021.

[64] Intricately. *EVERYTHING YOU WANT TO KNOW ABOUT TikTok*. https://my.intricately.com/companies/tiktok. Online; accessed 6 January 2020. 2017.

[65] Cade Metz. *How Facebook Moved 20 Billion Instagram Photos Without You Noticing*. https://www.wired.com/2014/06/facebook-instagram/. Online; accessed 6 January 2020. 2014.

[66] Tibor Jager et al. 'On the security of TLS-DHE in the standard model'. In: *CRYPTO*. Berlin: Springer, 2012, pp. 273–293.

[67] Hugo Krawczyk, Kenneth G Paterson and Hoeteck Wee. 'On the security of the TLS protocol: A systematic analysis'. In: *Annual Cryptology Conference*. Springer. 2013, pp. 429–448.

[68] 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*. Berlin: Springer, 2015, pp. 456–477.

[69] Ding Wang et al. 'Targeted online password guessing: An underestimated threat'. In: *CCS*. New York: ACM, 2016, pp. 1242–1254.

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

[71] Google. *Cloud Storage Client Libraries*. https://cloud.google.com/storage/docs/reference/libraries. Accessed July 30, 2020. 2020.

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

[73] Dropbox. *Dropbox for Java Developers*. https://www.dropbox.com/developers/documentation/java. Accessed July 30, 2020. 2020.

[74] AWS. *Amazon EC2 On-Demand Pricing*. https://aws.amazon.com/ec2/pricing/on-demand. Accessed July 30, 2020. 2020.

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

[76] Colin Percival and Simon Josefsson. *The scrypt password-based key derivation function*. Tech. rep. 2016.

[77] Dan Boneh, Henry Corrigan-Gibbs and Stuart Schechter. 'Balloon hashing: A memory-hard function providing provable protection against sequential attacks'. In: *ASIACRYPT*. Berlin: Springer, 2016, pp. 220–248.

[78] Alex Biryukov, Daniel Dinu and Dmitry Khovratovich. 'Argon2: new generation of memory-hard functions for password hashing and other applications'. In: *EuroS&P*. New York: IEEE, 2016, pp. 292–302.

[79] D Moody. *Post-quantum cryptography: NIST's plans for the future. Presentation at PKC 2016*. 2016.

[80] Martin R Albrecht et al. 'Round-optimal verifiable oblivious pseudorandom functions from ideal lattices'. In: *IACR International Conference on Public-Key Cryptography*. Springer. 2021, pp. 261–289.

[81] Dan Boneh, Dmitry Kogan and Katharine Woo. 'Oblivious pseudorandom functions from isogenies'. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2020, pp. 520–550.

[82] Peter Schwabe, Douglas Stebila and Thom Wiggers. 'Post-quantum TLS without handshake signatures'. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020.

[83] Christian Paquin, Douglas Stebila and Goutam Tamvada. 'Benchmarking post-quantum cryptography in TLS'. In: *International Conference on Post-Quantum Cryptography*. Springer. 2020, pp. 72–91.

[84] Rick Mugridge. 'Test driven development and the scientific method'. In: *ADC*. New York: IEEE, 2003, pp. 47–52.

[85] Tim Menzies. 'Occam's Razor and Simple Software Project Management'. In: *Software Project Management in a Changing World*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 447–472. ISBN: 978-3-642-55035-5. DOI: 10.1007/978-3-642-55035-5_18. URL: https://doi.org/10.1007/978-3-642-55035-5_18.

[86] 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. 2020.

[87] 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. 2020.

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

[89] Long Chen, Yanan Li and Qiang Tang. 'CCA Updatable Encryption Against Malicious Re-encryption Attacks'. In: *ASIACRYPT (3)*. Vol. 12493. Lecture Notes in Computer Science. Springer, 2020, pp. 590–620.

[90] Github Docs. *About commit signature verification*. https://docs.github.com/en/enterprise-cloud@latest/authentication/managing-commit-signature-verification/about-commit-signature-verification. Accessed: 2025-01-09.

[91] GitLab.com. *Signed commits*. https://docs.gitlab.com/ee/user/project/repository/signed_commits/. Accessed: 2025-01-09.

[92] Gitea Docs. *GPG Commit Signatures, Version: 1.22.6*. https://docs.gitea.com/administration/signing. Accessed: 2025-01-09. [n.d.]

[93] Bitbucket support. *Verify commit signatures*. https://confluence.atlassian.com/bitbucketserver/verify-commit-signatures-1279066267.html. Accessed: 2025-01-09.

[94] Andrew Ayer. *git-crypt - transparent file encryption in git*. https://github.com/AGWA/git-crypt. Apr. 2024.

[95] Keybase Book. *Security on Keybase*. https://book.keybase.io/security. Apr. 2024.

[96] bluss, Joey Hess and Sean Whitton. *git-remote-gcrypt: a gitremote helper to push and pull from repositories encrypted with GnuPG*. https://spwhitton.name/tech/code/git-remote-gcrypt. Apr. 2024.

[97] Nik Unger et al. 'SoK: Secure Messaging'. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 232–249.

[98] Kenneth G. Paterson, Matteo Scarlata and Kien T. Truong. 'Three Lessons From Threema: Analysis of a Secure Messenger'. In: *USENIX Security Symposium*. USENIX Association, 2023, pp. 1289–1306.

[99] Anrin Chakraborti, Darius Suciu and Radu Sion. 'Wink: Deniable Secure Messaging'. In: *USENIX Security Symposium*. USENIX Association, 2023, pp. 1271–1288.

[100] Cas Cremers, Charlie Jacomme and Aurora Naska. 'Formal Analysis of Session-Handling in Secure Messaging: Lifting Security from Sessions to Conversations'. In: *USENIX Security Symposium*. USENIX Association, 2023, pp. 1235–1252.

[101] Gareth T. Davies et al. 'Security Analysis of the WhatsApp End-to-End Encrypted Backup Protocol'. In: *CRYPTO (4)*. Vol. 14084. Lecture Notes in Computer Science. Springer, 2023, pp. 330–361.

[102] Joseph Jaeger, Akshaya Kumar and Igors Stepanovs. 'Symmetric Signcryption and E2EE Group Messaging in Keybase'. In: *EUROCRYPT (3)*. Vol. 14653. Lecture Notes in Computer Science. Springer, 2024, pp. 283–312.

[103] Martin R. Albrecht, Benjamin Dowling and Daniel Jones. 'Formal Analysis of Multi-device Group Messaging in WhatsApp'. In: *EUROCRYPT (8)*. Vol. 15608. Lecture Notes in Computer Science. Springer, 2025, pp. 242–271.

[104] Joseph Jaeger and Akshaya Kumar. 'Analyzing Group Chat Encryption in MLS, Session, Signal, and Matrix'. In: *EUROCRYPT (8)*. Vol. 15608. Lecture Notes in Computer Science. Springer, 2025, pp. 272–301.

[105] Matilda Backendal, Miro Haller and Kenneth G. Paterson. 'MEGA: Malleable Encryption Goes Awry'. In: *SP*. IEEE, 2023, pp. 146–163.

[106] Jonas Hofmann and Kien Tuong Truong. 'End-to-End Encrypted Cloud Storage in the Wild: A Broken Ecosystem'. In: *CCS*. ACM, 2024, pp. 3988–4001.

[107] Wenhan Xu et al. 'Gringotts: An Encrypted Version Control System With Less Trust on Servers'. In: *IEEE Trans. Dependable Secur. Comput.* 21.2 (2024), pp. 668–684.

[108] Andrés Fábrega et al. 'Injection Attacks Against End-to-End Encrypted Applications'. In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2023, pp. 82–82.

[109] Xin Xu et al. 'Enforcing Access Control in Distributed Version Control Systems'. In: *IEEE International Conference on Multimedia and Expo, ICME 2019, Shanghai, China, July 8-12, 2019*. IEEE, 2019, pp. 772–777.

[110] Nikita Sobolev. *git-secret: A bash-tool to store your private data inside a git repository*. https://github.com/sobolevn/git-secret. Apr. 2024.

[111] Bo Chen and Reza Curtmola. 'Auditable Version Control Systems'. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[112] David Balbás, Daniel Collins and Serge Vaudenay. 'Cryptographic Administration for Secure Group Messaging'. In: *USENIX Security Symposium*. USENIX Association, 2023, pp. 1253–1270.

[113] Weikeng Chen and Raluca Ada Popa. 'Metal: A Metadata-Hiding File-Sharing System'. In: *NDSS*. The Internet Society, 2020.

[114] Enrico Buonanno, Jonathan Katz and Moti Yung. 'Incremental Unforgeable Encryption'. In: *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*. Ed. by Mitsuru Matsui. Vol. 2355. Lecture Notes in Computer Science. Springer, 2001, pp. 109–124.

[115] IPFS Docs. *Merkle Directed Acyclic Graphs (DAGs)*. https://docs.ipfs.tech/concepts/merkle-dag/. Accessed: 2025-06-26. [n.d.]

[116] diff-match-patch. https://github.com/google/diff-match-patch.

[117] awesome. https://github.com/sindresorhus/awesome.

[118] free-programming-books. https://github.com/EbookFoundation/free-programming-books.

[119] bootstrap. https://github.com/twbs/bootstrap.

[120] react. https://github.com/facebook/react.

[121] freeCodeCamp. https://github.com/freeCodeCamp/freeCodeCamp.

[122] Martin Bradley and Alexander Dent. 'Payment Card Industry Data Security Standard'. In: (2010). URL: https://media.techtarget.com/searchSecurityUK/downloads/RHUL_Bradley_2010.pdf.

[123] Payment Card Industry. *Data Security Standard. Requirements and Security Assessment Procedures. Version 3.2 PCI Security Standards Council (2016)*. 2016. URL: https://listings.pcisecuritystandards.org/documents/PA-DSS-v3_2-Program-Guide.pdf.

[124] Dan Boneh et al. ‘Improving Speed and Security in Updatable Encryption Schemes’. In: *ASIACRYPT (3)*. Vol. 12493. Lecture Notes in Computer Science. Springer, 2020, pp. 559–589.

[125] Colin Boyd et al. ‘Fast and Secure Updatable Encryption’. In: *CRYPTO (1)*. Vol. 12170. Lecture Notes in Computer Science. Springer, 2020, pp. 464–493.

[126] Michael Klooß, Anja Lehmann and Andy Rupp. ‘(R)CCA Secure Updatable Encryption with Integrity Protection’. In: *EUROCRYPT (1)*. Vol. 11476. Lecture Notes in Computer Science. Springer, 2019, pp. 68–99.

[127] Ran Canetti, Hugo Krawczyk and Jesper B Nielsen. ‘Relaxing chosen-ciphertext security’. In: *Annual International Cryptology Conference*. Springer. 2003, pp. 565–582.

[128] Ueli Maurer, Andreas Rüedlinger and Björn Tackmann. ‘Confidentiality and integrity: A constructive perspective’. In: *Theory of Cryptography Conference*. Springer. 2012, pp. 209–229.

[129] Mihir Bellare and Chanathip Namprempre. ‘Authenticated encryption: Relations among notions and analysis of the generic composition paradigm’. In: *Journal of Cryptology* 21.4 (2008), pp. 469–491.

[130] Hugo Krawczyk, Mihir Bellare and Ran Canetti. *HMAC: Keyed-hashing for message authentication*. 1997.

[131] David Chaum, Eugène van Heijst and Birgit Pfitzmann. ‘Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer’. In: *Advances in Cryptology — CRYPTO ’91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 470–484. ISBN: 978-3-540-46766-3.

[132] Maxwell N Krohn, Michael J Freedman and David Mazieres. ‘On-the-fly verification of rateless erasure codes for efficient content distribution’. In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE. 2004, pp. 226–240.

[133] Rosario Gennaro et al. ‘Secure network coding over the integers’. In: *International Workshop on Public Key Cryptography*. Springer. 2010, pp. 142–160.

[134] Vadim Lyubashevsky et al. ‘SWIFFT: A modest proposal for FFT hashing’. In: *International Workshop on Fast Software Encryption*. Springer. 2008, pp. 54–72.

[135] Dan Boneh, Xavier Boyen and Hovav Shacham. 'Short group signatures'. In: *Annual international cryptology conference*. Springer. 2004, pp. 41–55.

[136] Atsuko Miyaji, Masaki Nakabayashi and Shunzo Takano. 'Characterization of elliptic curve traces under FR-reduction'. In: *International Conference on Information Security and Cryptology*. Springer. 2000, pp. 90–108.

[137] Jens Groth. 'Homomorphic Trapdoor Commitments to Group Elements.' In: *IACR Cryptology ePrint Archive* 2009 (2009), p. 7.

[138] Navid Alamati, Hart Montgomery and Sikhar Patranabis. 'Symmetric Primitives with Structured Secrets'. In: *CRYPTO (1)*. Vol. 11692. Lecture Notes in Computer Science. Springer, 2019, pp. 650–679.

[139] Dario Catalano and Dario Fiore. 'Vector commitments and their applications'. In: *International Workshop on Public Key Cryptography*. Springer. 2013, pp. 55–72.

[140] Siavosh Benabbas, Rosario Gennaro and Yevgeniy Vahlis. 'Verifiable delegation of computation over large datasets'. In: *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*. Springer. 2011, pp. 111–131.

[141] Russell WF Lai and Giulio Malavolta. 'Subvector commitments with application to succinct arguments'. In: *Annual International Cryptology Conference*. Springer. 2019, pp. 530–560.

[142] Dan Boneh, Benedikt Bünz and Ben Fisch. 'Batching techniques for accumulators with applications to IOPs and stateless blockchains'. In: *Annual International Cryptology Conference*. Springer. 2019, pp. 561–586.

[143] Matteo Campanelli et al. 'Vector Commitment Techniques and Applications to Verifiable Decentralized Storage.' In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 149.

[144] Miranda Christ and Joseph Bonneau. 'Limits on revocable proof systems, with applications to stateless blockchains'. In: *IACR Cryptol. ePrint Arch.* (2022), p. 1478.

[145] Hoeteck Wee and David J Wu. 'Succinct vector, polynomial, and functional commitments from lattices'. In: *Advances in Cryptology–EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*. Springer. 2023, pp. 385–416.

[146] Leo de Castro and Chris Peikert. ‘Functional Commitments for All Functions, with Transparent Setup and from SIS’. In: *Advances in Cryptology–EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*. Springer. 2023, pp. 287–320.

[147] Chris Peikert, Zachary Pepin and Chad Sharp. ‘Vector and functional commitments from lattices’. In: *Theory of Cryptography Conference*. Springer. 2021, pp. 480–511.