CCA Updatable Encryption Against Malicious Re-Encryption Attacks

Long Chen, Yanan Li, and Qiang Tang

New Jersey Institute of Technology, Newark NJ 07102, USA
{longchen,ly252,qiang}@njit.edu

Abstract. Updatable encryption (UE) is an attractive primitive, which allows the secret key of the outsourced encrypted data to be updated to a fresh one periodically. Several elegant works exist 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. Those malformed ciphertext could be updated and leveraged by the adversary and cause serious security issues.
In this paper, we fill the gap and strengthen the security definitions in multiple aspects: most importantly our integrity and CCA security models remove the restriction 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 interests. We also study the relations among security notions; and a bit surprisingly, the folklore result in authenticated encryption that IND-CPA plus ciphertext integrity imply IND-CCA security does *not* hold for ciphertext dependent updatable encryption.

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) [6,13] 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 storage providers, such as Google and Amazon [10].

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. [4] 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 [10] 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 [3] proposed strengthening on confidentiality and improved the efficiency of [10].

Security of updatable encryption in a nutshell. The security models of updatable encryption mimic those of authenticated encryption (AE) to capture both the confidentiality and integrity of the massage. 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 [10], 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 [10,3], 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 [5] 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 [3], which we will discuss in detail in the section of related works. In this paper, we focus on the general ciphertext dependent UE, as [4,10,3].

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 [14], 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 [10] "*are only guaranteed against passive adversaries*".

Indeed, existing constructions of updatable encryption will become insecure if we allow the *malicious re-encryption queries*. In the full version [9], we provide a concrete example to show an active "attack" on the integrity of the KSS scheme proposed by Everspauph et al. [10]. 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 [3].

Having noticed the problem, some partial progresses have been made in the ciphertext independent setting [14].[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 [7]). It is widely believed that PTXT does not provide a strong enough integrity guarantee for secure storage [20], as the adversary may still be able to generate a ciphertext that was mauled from a target ciphertext. While RCCA has another restriction that a ciphertext generated by re-randomizing a challenge ciphertext is not allowed to query decryption oracle, thus clearly not CCA.

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 [18] that the security model of [10] 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 [10] 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

[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.

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 [10], 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 [3]).

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 [18] 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.

1.1 Our Contributions

In this paper, 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. 1.

Scheme	**Update Manner**	**Conf.**	**Integrity**	**ReEnc IND**
BLMR [4]	CD	CPA	No	CPA
KSS [10]	CD	CPA	CTXT^-	$\perp$
ReCrypt [10]	CD	CPA	CTXT^-	CPA
Nested UAE[3]	CD	CPA	CTXT^-	CPA
KH-PRF UAE [3]	CD	CPA	CTXT^-	CPA
RISE [18]	CI	CPA	No	CPA
E&M [14]	CI	CCA^-	CTXT^-	CPA
NYUE [14]	CI	RCCA	PTXT	CPA
SHINE [5]	CI	CCA^-	CTXT^-	CCA^-
ReCrypt$^+$	CD	CCA	CTXT	CCA

Table 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 [10,3], 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 [10] when facing malicious re-encryption in the full version [9].
– We use the notion of epoch from [18] 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 the full version [9], we provide a variation of KSS scheme from [10] 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 to the full version [9].

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 [10], 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** follows the standard Key Encapsulation Mechanism (KEM) + Data Encapsulation Mechanism (DEM) 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 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) [4]. 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.

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 [4,10,3], (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 [18,14,5] (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 [3], 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 also refer to the full version [9] for more detailed comparisons.

Other related works. The first updatable encryption scheme (**BLMR**) is proposed by Boneh et al. [4]. However, only the confidentiality is considered in this work, and the other security notions have not been formalized. Later, Everspauph et al.[10] 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 paper. Very recently, Boneh et.al [3] revisit the results of Everspauph et al. about CDUE. Their security notion is similar to [10], 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 [18] point out the models UP-IND and UP-REENC in [10] 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. [14] add the integrity considerations to [18], and provide two constructions (**E&M** without malicious update resistance and **NYUE** with only plaintext integrity and the weaker RCCA security).

Boy et.al [5] 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.

2 Preliminary

Here we describe several primitives that will be used in our construction.

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.
- $\mathsf{Dec}(k, c, ad)$: 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.

For the detailed security definition, we refer to the full version [9].

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 [22], 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 $\mathsf{Com}(m_1, open_1) \otimes \mathsf{Com}(m_2, open_2) = \mathsf{Com}(m_1 \oplus m_2, open_1 \odot open_2)$.

Key-homomorphic pusedorandom function The notion of key-homomorphic PRFs was proposed by Boneh et al. [4], and used in the UE constructions [10,18]. 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.

3 Formalization

In this section, we formalize the syntax of the ciphertext dependent updatable encryption scheme following [10].

Intuitively, the data flow of the outsource storage from CDUE can be seen in Fig. 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 1 (Updatable Encryption). *The ciphertext dependent updatable encryption (CDUE) consists of the following six algorithms*

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

- $\mathsf{Setup}(1^\lambda)$ *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 $\perp$.*

Note that the above formalization is tailored to our ciphertext integrity definition. Particularly, here we require the algorithm $\mathsf{Recrypt}$ 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 $\mathsf{Recrypt}$ is deterministic for almost all existing CDUE schemes [10,3].

Besides, the syntax of the CIUE scheme can be viewed as a special case of the ciphertext dependent scheme in Definition 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.

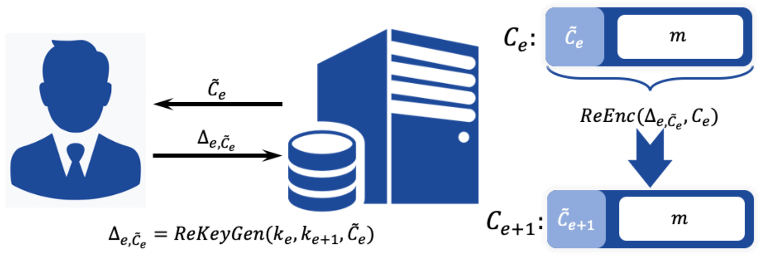
$\tilde{C}_e$
$\Delta_{e,\tilde{C}_e}$
$\Delta_{e,\tilde{C}_e} = ReKeyGen(k_e, k_{e+1}, \tilde{C}_e)$
C_e:
$\tilde{C}_e$
m
$ReEnc(\Delta_{e,\tilde{C}_e}, C_e)$
C_{e+1}:
$\tilde{C}_{e+1}$
m

Fig. 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}.

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

Definition 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 $i < j \leq e$,*

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

Then ***CDUE*** *is correct if* $\Pr[\textbf{\textit{CDUE}}.\mathsf{Decrypt}(k_e, C_e) = m] = 1$ *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.

4 Strengthened Security Models

In this section, we systematically study the security definitions of the CDUE. As we explained in the introduction, the previous model combination UP-IND + UP-INT [10,3] needs to be strengthened in multiple aspects.

Malicious re-encryption attack. All previous CDUE definitions [10,3] 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 the full version [9], we show that an adversary of the **KSS** scheme [10] 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 [10,3], as discussed in [18], 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 [9] 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 [18], 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 [10,3] 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!

In the following, 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 the full version [9]. 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.

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[3].

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.2, and explain in detail how the oracles are defined right after Definition 3. As mentioned above, we also introduce the epoch notion to denote the time sequence following [18]. 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

[3] In the real world, the communication between a client and a server is typically via TLS without the user authentication [16], since the client does not have a PKI certificate. Therefore pretending the client in front of the server is not difficult.

challenge ciphertext in the key corrupted epoch, thus our model easily captures the post-compromise security (which was ambiguous in existing models).

```
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 $(b' == b)$
```

Fig. 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 $\mathcal{O}_{\mathsf{Dec}}$.

Definition 3 (sUP-IND-CPA(CCA)). *Define the sUP-IND-CPA(CCA) experiment as Fig.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*

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

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-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 $\mathsf{ReKeyGen}$ 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 [10,3], 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 [10], 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 $\mathsf{ReKeyGen}$ and $\mathsf{Recrypt}$ algorithms include a invalid ciphertext detection mechanism. Similarly, $\mathcal{O}_{\mathsf{Token}}$ may reply $\perp$ when queried with an invalid header.

– *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 $\mathsf{ReKeyGen}$ and $\mathsf{Recrypt}$ 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 [10,3], 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.[4]

– *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

[4] 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.

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 [10,18,14]. 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 v.s. UP-IND. Note that even our sUP-IND-CPA security is stronger than UP-IND [10] in following aspects. Firstly, sUP-IND-CPA can characterize the post-compromise security which is ignored in UP-IND. Although the constructions in [10,3] is post-compromise secure, there do exist constructions (see in the full version [9]) 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 [10] 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 the full version [9].

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 [10]. 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 [10], 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.

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 [10] 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. 3. Similar to [14], 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.

– *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 $\mathbb{T}(e, \tilde{c}) \leftarrow \bar{c}$.
– *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 $C' = (\tilde{c}', \bar{c}')$ 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 4 (sUP-INT-CTXT). *Define the sUP-INT-CTXT experiment as Fig. 3. An updatable encryption scheme is called sUP-INT-CTXT secure if for any P.P.T. adversary $\mathcal{A}$ the following advantage*

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

is negligible in the security parameter λ.

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

Fig. 3. The sUP-INT-CTXT experiment.

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 [10,3] 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.

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 [1]. This implication still holds for CIUE[5]. 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 [10,3] 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 [10] and our **ReCrypt**$^{+}$in Section 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.

- $\mathbf{CDUE}'.\mathsf{KeyGen}(pp)$: Use $\mathbf{CDUE}.\mathsf{KeyGen}$ to generate an epoch key k_e of $\mathbf{CDUE}$ and use $\mathbf{MAC}.\mathsf{KeyGen}$ to generate a MAC key mk_e. The new epoch key k'_e of $\mathbf{CDUE}'$ is (k_e, mk_e).
- $\mathbf{CDUE}'.\mathsf{Encrypt}(k'_e, m)$: Parse the secret key $k'_e = (k_e, mk_e)$. Given the plaintext m, firstly use $\mathbf{CDUE}.\mathsf{Enc}$ 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 $\tau_e = \mathbf{MAC}.\mathsf{Tag}(mk_e, \tilde{c}_e \| 1)$. Finally, output the ciphertext $C'_e = (\tilde{c}'_e, \bar{c}_e)$ where the new header $\tilde{c}'_e = (\tilde{c}_e, \tau_e)$.
- $\mathbf{CDUE}'.\mathsf{Decrypt}(k'_e, C'_e)$: Parse $C'_e = (\tilde{c}'_e, \bar{c}_e)$ where $\tilde{c}'_e = (\tilde{c}_e, \tau_e)$. Verify whether $\mathbf{MAC}.\mathsf{Verify}(mk_e, \tau_e, \tilde{c}_e \| 1) = 1$ or $\mathbf{MAC}.\mathsf{Verify}(mk_e, \tau_e, \tilde{c}_e \| 0) = 1$. If one of above two cases is true, use the $\mathbf{CDUE}.\mathsf{Decrypt}$ to decrypt the ciphertext $C_e = (\tilde{c}_e, \bar{c}_e)$ and return the decryption result.
- $\mathbf{CDUE}'.\mathsf{ReKeyGen}(k'_e, k'_{e+1}, \tilde{c}'_e)$: 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 $\mathbf{MAC}.\mathsf{Verify}(\tau_e, \tilde{c}_e \| 1) = 1$. If it is true, invoke $\mathbf{CDUE}.\mathsf{ReKeyGen}(k_e, k_{e+1}, \tilde{c}_e)$ to generate the token $\Delta_{e,\tilde{c}_e}$. Note that according to our assumption about $\mathbf{CDUE}$, $\Delta_{e,\tilde{c}_e}$ 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 $\tau_{e+1} = \mathbf{MAC}.\mathsf{Tag}(mk_{e+1}, \tilde{c}_{e+1} \| 1)$ 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 $\mathbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1})$, and output the update token $\Delta'_{e,\tilde{c}'_e} = \big(\tilde{c}'_{e+1}, \delta_{e,\tilde{c}_e}, \mathbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1})\big)$ for $\mathbf{CDUE}'$.
- $\mathbf{CDUE}'.\mathsf{ReEncrypt}(\Delta'_{e,\tilde{c}'_e}, C'_e)$: First parse the token $\Delta'_{e,\tilde{c}'_e} = (\tilde{c}'_{e+1}, \delta_{e,\tilde{c}_e}, \mathbf{SKE}.\mathsf{Enc}_{k_e}(mk_{e+1}))$ and the ciphertext $C'_e = (\tilde{c}'_e, \bar{c}_e) = ((\tilde{c}_e, \tau_e), \bar{c}_e)$. Then derive the $\mathbf{CDUE}$ 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 $\mathbf{CDUE}.\mathsf{ReEncrypt}(\Delta_{e,\tilde{c}_e}, C_e)$ 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 $\mathbf{CDUE}'$ is sUP-IND-CPA and sUP-INT-CTXT secure when $\mathbf{MAC}$ is deterministic (like HMAC [15]). 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. We put the formal proof in the full version [9].

Lemma 1. *If* $\boldsymbol{CDUE}$ *is sUP-IND-CPA secure and* $\boldsymbol{MAC}$ *is deterministic (i.e. there is only one valid MAC for each message under one secret key),* $\boldsymbol{CDUE}'$ *is sUP-IND-CPA secure.*

Lemma 2. *If* $\boldsymbol{CDUE}$ *is sUP-INT-CTXT secure,* $\boldsymbol{SKE}$ *is IND-CPA secure and* $\boldsymbol{MAC}$ *is multi-user CMA unforgerable, then* $\boldsymbol{CDUE}'$ *is sUP-INT-CTXT secure.*

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 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 define UP-IND-CPA, UP-INT-CTXT and UP-IND-CCA in Appendix XXX following the former paradigm but adding 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 2. The intuition comes from that the CCA attack on our artificially designed $\mathbf{CDUE}'$ scheme does not need to query malicious ciphertexts on the re-encryption oracle. Moreover, the security gap holds even for the weakest models[5] in [10,3] without the post-compromise security or the malicious update resistance. Indeed, it is not hard to see that the above $\mathbf{CDUE}'$ is also UP-IND and UP-INT secure, while the CCA attack can still apply.

Theorem 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 UE Construction with Strengthened Integrity

Next we describe our new CDUAE construction $\mathbf{ReCrypt}^+$. Comparing with previous CDUAE constructions [10,3], 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] The similar CCA model can be trivially obtained by adding an additional decryption oracle for ciphertexts decryption except for the challenge-equal ciphertexts.

5.1 Construction framework

Our construction **ReCrypt**$^+$ follows the paradigm of the **ReCrypt** scheme proposed by Everspauph et al. The original **ReCrypt** in [10] 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 [10] 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 4.

Let **HomHash**.Setup and **HomHash**.Eval be the algorithms of a homomorphic collision-resistant hash function with the following syntax.

Definition 5. *A homomorphic hash function H_{hom} is a linear function that maps vectors of starting group elements $\mathbf{v} = (v_1, \dots, v_n) \in \mathbb{G}_{HS}^n$ 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$(hk, \boldsymbol{v})$: On input the evaluation key hk and a vector of starting group elements $\mathbf{v} = (v_1, \dots, v_n) \in \mathbb{G}_{HS}^n$, output one target group element $u \in \mathbb{G}_{HT}$.*

*Fixed the evaluation key hk, we can write as $\mathsf{H}_{hom}(\boldsymbol{v}) =$ **HomHash**.Eval$(hk, \boldsymbol{v}) = u$. Specifically, it should satisfies the following properties:*

- Collision resistance: *the probability for any P.P.T adversary to generate the two vectors $\boldsymbol{v}$ and $\boldsymbol{v}'$ in $\mathbb{G}_{HS}^n$ which satisfy $\mathsf{H}_{hom}(\boldsymbol{v}) = \mathsf{H}_{hom}(\boldsymbol{v}')$ is negligible.*
- Homomorphism: *we have $\mathsf{H}_{hom}(\boldsymbol{v}) + \mathsf{H}_{hom}(\boldsymbol{v}') = \mathsf{H}_{hom}(\boldsymbol{v} + \boldsymbol{v}')$.*

Let $\mathsf{F} : \mathbb{K}_{PRF} \times \mathcal{M}_{PRF} \to \mathbb{G}_{PRF}$ be the key homomorphic PRF as described in Subsection 2, 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, 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, whose key space, message space and ciphertext space are $\mathbb{K}_{AEAD}$, $\mathbb{M}_{AEAD}$ and $\mathbb{C}_{AEAD}$.

```
Setup(λ)
1 : $hk \leftarrow_\$ \mathbf{HomHash}.\mathsf{Setup}(\lambda), \mathsf{hcom}.pp \leftarrow_\$ \mathbf{HCOM}.\mathsf{Init}(\lambda)$
2 : $\mathbf{return}\ (hk, \mathsf{hcom}.pp)$

KeyGen(λ)
1 : $k \leftarrow_\$ \mathbf{AEAD}.\mathsf{KeyGen}(1^\lambda), \quad \mathbf{return}\ k$

Encrypt(k, m)
1 : Map $m \rightarrow (m_1, m_2, \ldots, m_n) \in \mathbb{G}^n_{PRF}, z \leftarrow_\$ \mathbb{K}_{PRF}$
2 : $d_i \leftarrow m_i + \mathsf{F}(z, i), d = (d_1, d_2, \ldots, d_n) \in \mathbb{G}^n_{PRF}, h \leftarrow \mathbf{HomHash}.\mathsf{Eval}(hk, d)$
3 : $y \leftarrow_\$ \mathbb{K}_{PRF}, hcom \leftarrow \mathbf{HCOM}.\mathsf{com}(y; hopen), x = z - y$
4 : $ct \leftarrow_\$ \mathbf{AEAD}.\mathsf{Enc}(k, x, (h, hcom))$
5 : $\tilde{c} = (ct, h, hcom)$// Ciphertext header
6 : $\bar{c} = (y, hopen, d)$// Ciphertext body
7 : $\mathbf{return}\ C = (\tilde{c}, \bar{c})$

Decrypt(k, C)
1 : Parse $C = ((ct, h, hcom), (y, hopen, d))$
2 : $\mathbf{if}\ h == \mathbf{HomHash}.\mathsf{Eval}(hk, d) \wedge \mathbf{HCom}.\mathsf{Open}(hcom, y, hopen) == 1\ \mathbf{then}$
// Check the body is consistent with the header.
3 :     $x^\star \leftarrow \mathbf{AEAD}.\mathsf{Dec}(k, \tilde{c}^1, (h, hcom))$
4 :     $\mathbf{for}\ 1 \leq i \leq n\ \ \mathbf{do}\ m_i^\star \leftarrow d_i - \mathsf{F}(x^\star - y, d_i) \quad \mathbf{return}\ m^\star = m_1^\star, \ldots, m_n^\star$
5 : $\mathbf{return}\ \perp$

ReKeygen(k, k', c̃)
1 : Parse $\tilde{c} = (ct, h, hcom), m' \leftarrow \mathbf{AEAD}.\mathsf{Dec}(k, ct, (h, hcom))$
2 : $\mathbf{if}\ m' \neq \perp\ \mathbf{then}$// Check the returned header is valid.
3 :     $\Delta z \leftarrow_\$ \mathbb{K}_{PRF}, \ \ \Delta d_i \leftarrow \mathsf{F}(\Delta z, i), \ \ \Delta d = \Delta d_1, \Delta d_2, \ldots, \Delta d_n$
4 :     $h' \leftarrow h + \mathbf{HomHash}.\mathsf{Eval}(hk, \Delta d), \ \ \Delta y \leftarrow_\$ \{0,1\}^*$
5 :     $hcom' \leftarrow hcom + \mathbf{HCom}.\mathsf{Com}(\Delta y, hopen_\Delta), \ \ x' = x + \Delta z - \Delta y,$
6 :     $ct' \leftarrow_\$ \mathbf{AEAD}.\mathsf{Enc}(k', x', (h', hcom')), \ \ \tilde{c}' = (ct', h', hcom')$
7 :     $\mathbf{return}\ \Delta = (\tilde{c}', \Delta y, hopen_\Delta, \Delta z)$
8 : $\mathbf{else} \quad \mathbf{return}\ \perp$

ReEncrypt(C, Δ)
1 : Parse $C = ((ct, h, hcom), (y, hopen, d)), \Delta = (\tilde{c}', (\Delta y, hopen_\Delta, \Delta z))$
2 : $\mathbf{if}\ \mathbf{HCOM}.\mathsf{Open}(hcom, y, r) == 1 \wedge h == \mathbf{HomHash}.\mathsf{Eval}(hk, d)\ \mathbf{then}$
// Check the body is consistent with the header.
3 :     $y' = y + \Delta y, \quad r' = r + \Delta r, \quad$ Parse $d = (d_1, d_2, \ldots, d_n)$
4 :     $d_i' \leftarrow d_i + \mathcal{F}(\Delta z, i), \quad d' = (d_1', d_2', \ldots, d_n')$
5 :     $hopen' = hopen + hopen_\Delta, \quad y' = y + \Delta y, \quad \bar{c}' = (y', hopen', d')$
6 :     $\mathbf{return}\ C' = (\tilde{c}', \bar{c}')$
7 : $\mathbf{return}\ \perp$
```

Fig. 4. Construction for ReCrypt$^+$

- **ReCrypt**$^+$.$\mathsf{Setup}(\lambda)$: 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**$^+$.$\mathsf{KeyGen}(\lambda)$: Run the **AEAD**.$\mathsf{KeyGen}(\lambda)$ to generate the key of AEAD $k \in \mathbb{K}_{AEAD}$.
- **ReCrypt**$^+$.$\mathsf{Encrypt}(k, m)$: The algorithm proceeds as follows.
  1. Map the message m into n group elements $m_1, m_2, \ldots, m_n \in \mathbb{G}^n_{PRF}$.
  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 $d = (d_1, d_2, \ldots, d_n) \in \mathbb{G}^n_{PRF}$. Since $d \in \mathbb{G}^n_{PRF} \subseteq \mathbb{G}^n_{HS}$, one can compute the homomorphic hash function on d and derive **HomHash**.$\mathsf{Eval}(hk, d)$ $= h \in \mathbb{G}$.
  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**.$\mathsf{Com}(y, hopen) = hcom \in \mathbb{C}_{COM}$, 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 $\bar{c} = (y, hopen, d) \in \mathbb{K}_{PRF} \times \mathbb{O}_{COM} \times \mathbb{G}^n_{PRF}$.
- **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 $\bar{c} = (y, hopen, d) \in \mathbb{K}_{PRF} \times \mathbb{O}_{COM} \times \mathbb{G}^n_{PRF}$, and proceeds as follows:
  1. Verify **HomHash**.$\mathsf{Eval}(hk, d) \overset{?}{=} h \in \mathbb{G}_{HT}$ for $d \in \mathbb{G}^n_{PRF} \subseteq \mathbb{G}^n_{HS}$,
  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 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 $h' = h + \mathbf{HomHash}.\mathsf{Eval}(hk, \Delta d) \in \mathbb{G}_{HT}$.
4. Generate a new group element $\Delta y \in \mathbb{K}_{PRF}$ and its homomorphic commitment $\mathbf{HCom}.\mathsf{com}(\Delta y, hopen_\Delta) = hcom_\Delta \in \mathbb{G}_{COM}$. 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 $ct' = \mathbf{AEAD}.\mathsf{Enc}(k', x', (h', hcom'))$.
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 $\Delta = (\tilde{c}', \Delta y, hopen_\Delta, \Delta z)$.

– $\mathbf{ReCrypt}^+.\mathsf{ReEncrypt}(C, \Delta)$: 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 $\mathbf{HomHash}.\mathsf{Eval}(hk, d) = h \in \mathbb{G}_{HT}$ 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 $\mathbf{HCom}.\mathsf{Open}(hcom, y, hopen)$ 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 $\bar{c}' = (y', hopen', d') \in \mathbb{K}_{PRF} \times \mathbb{O}_{COM} \times \mathbb{G}_{PRF}^n$.

5.2 Homomorphic hash functions from DDH groups

To make the following $\mathbf{ReCrypt}^+$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 [8,17,11] or short ring elements [19]. 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 [8,17]. 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 $\mathbf{ReCrypt}^+$, 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 [2,21].

So the message are chosen from the group $\mathbb{G}_1^n$, the algorithms of the homomorphic hash function are defined as follows.

- **HomHash**.$\mathsf{Setup}(\mathbb{G}, n)$: Randomly pick $g \leftarrow_{\$} \mathbb{G}_2 \backslash \{1\}$ and 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 $\mathsf{H}_{hom}(\mathbf{v}) = \mathbf{HomHash}.\mathsf{Eval}(hk, \mathbf{v})$.

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 [12]. 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 $e(r, g_r)e(t, g_t) = 1$. The proof could be found in the full version [9].

Lemma 3 (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.*

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, $\mathbb{G}_{PRF} \to \mathbb{G}_{HS}$), 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 over $\mathbb{G}_{PRF} = \mathbb{G}_1$ and $\mathbb{K}_{PRF} = \mathbb{Z}_p$, and the homomorphic hash function described in Subsection 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.

5.4 Security Analysis

Now we show that our construction **ReCrypt**$^+$ is secure under the models sUP-IND-CCA, sUP-INT-CTXT and sUP-REENC-CCA. Due to page limitation, we will provide detailed proofs in the full version [9].

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 give the formal proof in [9].

Theorem 3 (sUP-IND-CCA Security of ReCrypt$^+$). *Let* ***ReCrypt***$^+$ *be an updatable encryption scheme as defined in Section 5.1.* ***ReCrypt***$^+$ *is sUP-IND-CCA secure if* ***AEAD*** *is MU-RoR-AE secure (Section 2), the homomorphic commitment* ***HCOM*** *is statistic hiding and computation binding, and the key homomorphic PRF is pseudorandom.*

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 give the formal proof in [9].

Theorem 4 (sUP-INT-CTXT Security of ReCrypt$^+$). *Let* ***ReCrypt***$^+$ *be an updatable encryption scheme as defined in Section 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.*

sUP-REENC-CCA. To demonstrate that our **ReCrypt**$^+$ scheme is sUP-REENC-CCA secure, we introduce a property called *perfect re-encryption* proposed in [14]. 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 the full version [9]. We notice that **ReCrypt**$^+$naturally satisfy the perfect re-encryption property. As pointed by [14], the perfect re-encryption property plus the sUP-IND-CCA security imply the sUP-REENC-CCA security. So we have the following theorem whose formal proof is in [9].

Theorem 5 (sUP-REENC-CCA Security). *Since* ***ReCrypt***$^+$ *as defined in Section 5.1 has the perfect re-encryption property and satisfy the sUP-IND-CCA security,* ***ReCrypt***$^+$ *is sUP-REENC-CCA secure.*

Acknowledgement

We thank anonymous reviewers from ASIACRYPT 20 for valuable comments. Qiang and Ya-Nan are supported in part by NSF grant CNS #1801492. Qiang is also supported in part by a Google Faculty Award.

References

1. Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21(4):469–491, 2008.
2. Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *Annual international cryptology conference*, pages 41–55. Springer, 2004.
3. Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. Cryptology ePrint Archive, Report 2020/222, 2020. https://eprint.iacr.org/2020/222.
4. Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. 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*, pages 410–428, 2013.
5. Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. In *CRYPTO (1)*, volume 12170 of *Lecture Notes in Computer Science*, pages 464–493. Springer, 2020.
6. Martin Bradley and Alexander Dent. Payment card industry data security standard.
7. Ran Canetti, Hugo Krawczyk, and Jesper B Nielsen. Relaxing chosen-ciphertext security. In *Annual International Cryptology Conference*, pages 565–582. Springer, 2003.

8. David Chaum, Eugène van Heijst, and Birgit Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 470–484, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
9. Long Chen, Ya-Nan Li, and Qiang Tang. CCA updatable encryption against malicious re-encryption attacks (full version). Cryptology ePrint Archive, Report 2020/XXX, 2020.
10. Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. 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*, pages 98–129, 2017.
11. Rosario Gennaro, Jonathan Katz, Hugo Krawczyk, and Tal Rabin. Secure network coding over the integers. In *International Workshop on Public Key Cryptography*, pages 142–160. Springer, 2010.
12. Jens Groth. Homomorphic trapdoor commitments to group elements. *IACR Cryptology ePrint Archive*, 2009:7, 2009.
13. Payment Card Industry. Data Security Standard. Requirements and Security Assessment Procedures. Version 3.2 PCI Security Standards Council (2016).
14. Michael Klooß, Anja Lehmann, and Andy Rupp. (R)CCA secure updatable encryption with integrity protection. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, pages 68–99, 2019.
15. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication, 1997.
16. Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. On the security of the tls protocol: A systematic analysis. In *Annual Cryptology Conference*, pages 429–448. Springer, 2013.
17. 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*, pages 226–240. IEEE, 2004.
18. 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*, pages 685–716, 2018.
19. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In *International Workshop on Fast Software Encryption*, pages 54–72. Springer, 2008.
20. Ueli Maurer, Andreas Rüedlinger, and Björn Tackmann. Confidentiality and integrity: A constructive perspective. In *Theory of Cryptography Conference*, pages 209–229. Springer, 2012.
21. Atsuko Miyaji, Masaki Nakabayashi, and Shunzo Takano. Characterization of elliptic curve traces under fr-reduction. In *International Conference on Information Security and Cryptology*, pages 90–108. Springer, 2000.
22. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.