EPiC Series in Computing

Volume 106, 2025, Pages 16–30

Proceedings of the 20th Asia Joint Conference on Information Security

Key-Value Commitments with Unlinkability

Towa Tsujimura[1] and Atsuko Miyaji[2]

[1] Graduate School of Engineering, The University of Osaka, Suita, Osaka, Japan
towa.tsujimura@cy2sec.comm.eng.osaka-u.ac.jp
[2] Graduate School of Engineering, The University of Osaka, Suita, Osaka, Japan
miyaji@comm.eng.osaka-u.ac.jp

Abstract

Blockchain technology enables secure transactions without requiring a centralized administrator used in various applications. Therefore, efficient verification in terms of computation and memory is desired. To reduce computational and memory costs, Agrawal et al. proposed Key-Value Commitments (KVC) scheme supporting both new pair insertions and value updates with efficient data verification. However, in their scheme, each transaction reveals the User's key when a new key-value pair is inserted or an existing value is updated. As a result, it becomes possible to determine whether multiple transactions belong to the same User. Furthermore, KVC has two other issues. One is that the proof consists of three group elements, which yields the computational cost of updating proofs. The other is that the sign of the value change is leaked during value updates. This research defines the unlinkability in KVC as a condition where each transaction is not linked to any other transaction and constructs a scheme that satisfies unlinkability by integrating Oblivious Accumulators into the KVC. The proposed method resolves three issues. Unlinkability is achieved by outputting a different value each time instead of outputting the User's key for each operation. By processing updates as insertions, User performing the update discards the previous proof and obtains a new one, which simplifies the structure of the proof and reduces the computational cost of proof updates, and the sign of the value change is always positive by unifying with the insert operation. The key-binding of security for the scheme in our proposal is reduced to the GRSA assumption under the random oracle model and the SRSA assumption without random oracle. If the collision resistance, preimage resistance, and second preimage resistance of the hash function hold, KVC in our proposal satisfies unlinkability.

1 Introduction

In recent years, blockchain has gained attention as a decentralized ledger that enables reliable data management without the need for a centralized administrator. Since the advent of Bitcoin [1], cryptocurrencies have rapidly developed, with major platforms like Ethereum [2] and Ripple [3] representing their states as key-value map, where keys are User's public addresses and values are their associated attributes (e.g., account balances). However, a major challenge is that as the blockchain grows in size, the computational and memory costs required for transaction verification also increase. As a solution, Key-Value Commitments (KVC) scheme was

proposed in [4] based on the ideas of the RSA-based accumulator construction of [6] and [9]. The scheme manages key-value pair as a commitment to a compressed list of the key-value map and enables efficient verification of whether a key-value pair is included by using a membership proof, where key-binding is required for the security. It can be applied to a wide range of areas, such as group signatures, anonymous authentication, and verifiable databases. KVC supports both new key-value pair insertions and value updates. In [4], Agrawal et al. proposed two types of the scheme, KVC-Ins and KVaC. KVC-Ins allows only the insertion of pairs, and KVaC supports both insertion and value updates. A related technique to KVC is the vector commitment (VC), which is a commitment scheme for an ordered set of values that allows position-wise proof of values. In VC [7], value updates are supported, but the insertion of new elements is not possible.

Both KVC-Ins and KVaC enable a compact representation of key-value map, facilitating efficient updates and verification of proofs. For each insertion or value update, a new commitment value C, a proof Λ_k, and update information upd (key and value changes) are generated. User updates own proof from upd and verifies that it belongs to the User. Both key-binding of KVC-Ins and KVaC in [4] are proven under the GRSA assumption with the random oracle model, and the SRSA assumption without random oracle. However, there are three drawbacks. The first drawback is that the same key is output for every transaction, which reveals whether transactions are performed by the same User. The second drawback is the size of each proof, which consists of three group elements. The first is the commitment value before inserting new pairs or updating values, the second is used for updating the proof when other User updates the value, and the third represents the number of value updates. Consequently, each time a proof is updated, all three group elements must be recomputed, resulting in high computational cost for the User. The third drawback is that update operations leak the sign of the value, i.e., whether the value increases or decreases. In [8], KVC based on pairing is proposed for the first time, but similarly it leaks whether each transaction is of the same User or not.

In this research, we define the unlinkability in KVC as a condition where each transaction is not linked to any other transaction. We construct a new scheme that satisfies unlinkability and solve the three issues mentioned above by incorporating the Oblivious Accumulators (OblvAcc) [5] proposed by Baldimtsi et al. into KVaC [4]. Let us explain how we resolve the first drawback. Unlinkability is achieved by outputting a different value each time instead of outputting a key for each operation. Our proposal can achieve both insert and update, whose key-binding is proven under the GRSA assumption in the random oracle model, and the SRSA assumption without random oracle in the same way as [4]. If the collision resistance, preimage resistance, and second preimage resistance of the hash function hold, KVC in our proposal satisfies unlinkability. Let us explain how we resolve the second and third drawbacks. In our proposal, updates are treated as insertion, and User updating value discards the previous proof and obtains a new one. As a result, the number of value updates is not necessary for proof update, and since the proof consists of only two elements (the commitment value before the new pair insertion), the computational cost of proof updates for User is reduced. The third drawback is easily resolved since the sign of value is always positive by unifying with the insert operation.

Table 1 compares our proposal and KVaC from the size of data in possession, number of proof elements, and number of interactions. Both our proposal and KVaC use the same RSA group $\mathbb{G}$, but our proposal stores auxiliary information of size $||b| + 1|$, where b is the upper bound on the order of the group $\mathbb{G}$. With the introduction of OblvAcc, in addition to the proof size Λ_k for the key-value pair, it is necessary to store the sizes of two auxiliary information aux_1 and aux_2, and the membership proof Λ_a for OblvAcc. In existing research, the number of interactions for insertion and value updates is each one, while in our proposal, insertion

occurs once, the first value update requires three, and subsequent value updates require four. Table 2 compares them from the computational cost for User, H represents hash computation, E represents exponentiation, M represents multiplication, and A represents addition. Table 3 compares them from the security, feature of inserts or updates, and unlinkability properties.

Table 1: Comparison of data and transmission volumes

Research	**Size of data in possession**	$\#\Lambda_k$	# interactions (Insert, Update)
KVaC [4]	$5\|\mathbb{G}\|$	Three	(1,1)
This research	$3\|\mathbb{G}\| + 2\|b\| + 1\|$	Two	(1,3 or 4)

Table 2: Comparison of computational cost for User

Research	**Insert**	**Other than Insert** User	**Update**	**Other than Update** User
KVaC [4]	0	$1H + 4E + 6M$	$1H + 1E + 1A$	$1H + 4E + 6M$
This research	0	$3E + 1M$	0	$5E(4E, 3E) + 1M$

Table 3: Comparison between our proposal and related works (RO denotes random oracle)

Research	**Security**	**Insert**	**Update**	**Unlinkability**
KVC-Ins [4]	GRSA with RO SRSA without RO	✓	None	-
KVaC [4]	GRSA with RO SRSA without RO	✓	✓	None
[7]	RSA, CDH	None	✓	-
[8]	Pairings	✓	✓	None
This research	GRSA with RO SRSA without RO	✓	✓	✓

This paper is organized as follows. Section 2 introduces the background necessary for this research. Section 3 describes related works and the challenges of existing research. Section 4 explains the proposed method and security proof. Section 5 describes the comparison between existing research and our proposal. Section 6 concludes this paper.

2 Preliminary

This section describes the necessary background for this research and security assumptions in Section 2.1 and 2.2.

2.1 Symbols and Definitions

The symbols used in this paper are shown in Table 4.

Table 4: The description of symbols

Symbol	Description
$\mathbb{Z}$	The set of all integers
$\perp$	The output when an algorithm fails
$x \xleftarrow{\$} S$	Uniformly chosen element x from group S
$x \xleftarrow{\$} A(\cdot)$	Input $\cdot$ into algorithm A, and get output x
λ	Security parameter
$\mathsf{Primes}(\lambda)$	Set of prime numbers less than 2^λ
$[n]$	Set $\{1, 2, \ldots, n\}$ for a natural number n
C	Commitment value in the scheme
k	Key information held by a user
Λ_k	Membership proof for the pair (k, v)
upd	Update information
aux	Auxiliary info known only to the user
z	Value output instead of k each time
s	Element necessary for calculating aux
C_a	Commitment value in $\mathsf{OblvAcc}$
Λ_a	Membership proof for aux in $\mathsf{OblvAcc}$
$\mathsf{upd_a}$	Update info in $\mathsf{OblvAcc}$
z_A	$\mathsf{upd_a}$ value when an element is added
z_D	$\mathsf{upd_a}$ value when an element is deleted
$\mathsf{User_i}$	The ith user
$\Lambda_{k_{i,j}}$	Proof for (k, v) after j updates by $\mathsf{User_i}$
$v_{i,j}$	Value after the jth update by $\mathsf{User_i}$
$aux_{i,j}$	Auxiliary value for $(j+1)$th update computed at jth update
$z_{i,j}$	z value output at jth update by $\mathsf{User_i}$
$s_{i,j}$	s value needed for $(j+1)$th update output at jth update
$\Lambda_{a_{i,j}}$	Membership proof for aux at jth update by $\mathsf{User_i}$
$z_{A_{i,j}}$	Contents of $\mathsf{upd_a}$ when $\mathsf{User_i}$ adds the jth aux
$z_{D_{i,j}}$	Contents of $\mathsf{upd_a}$ when $\mathsf{User_i}$ deletes the jth aux

The definitions used in this paper are represented as follows.

Definition 2.1 (Cryptographic hash function)**.** *A cryptographic hash function is a hash function $H : \{0,1\}^* \to \{0,1\}^*$ that satisfies the following properties.*

- *Collision resistance : It is computationally infeasible to find two distinct inputs $x \neq x'$ such that $H(x) = H(x')$.*
- *Preimage resistance : Given a hash value $H(x)$, it is computationally infeasible to find any input x' such that $H(x') = H(x)$.*
- *Second preimage resistance : Given an input x, it is computationally infeasible to find a different input $x' \neq x$ such that $H(x') = H(x)$.*

In the following, cryptographic hash function is simply described as hash function.

Definition 2.2 (Key-Value Commitments [4])**.** Key-Value Commitments *(*KVC*) is a protocol for proving affiliation to* key-value map *using a commitment scheme.* key-value map $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$ *is*

a set of key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$. *Each* key-value pair *is assigned a proof* Λ *belonging to the set.* KVC *consists of the following five algorithms, each of which is outlined below.*

- *Key generation* $(\mathsf{pp}, C) \xleftarrow{\$} KeyGen(1^\lambda)$
  Input: security parameter λ
  Output: public parameter pp *(defined key space* $\mathcal{K}$ *and value space* $\mathcal{V}$*) and initial commitment* C *which corresponds to an empty* key-value map.
- *Pair insertion* $(C, \Lambda_k, \mathsf{upd}) \leftarrow Insert(\mathsf{C}, (\mathsf{k}, \mathsf{v}))$
  Input: commitment C, key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$ *Output: new commitment* C, *membership proof* Λ_k, *update information* upd
- *Value update* $(C, \mathsf{upd}) \leftarrow Update(C, (k, \delta))$
  Input: commitment C, *key* $k \in \mathcal{K}$, *difference of value* δ
  Output: new commitment C *and update information upd*
- *Proof update* $\Lambda_k \leftarrow Proofupdate(k, \Lambda_k, \mathsf{upd})$
  Input: key $k \in \mathcal{K}$, *membership proof* Λ_k *for key* k, *and update information* upd
  Output: updated membership proof Λ_k
- *Verify proof* $1/0 \leftarrow Ver(C, (k, v), \Lambda_k)$
  Input: commitment C, key-value pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, *membership proof* Λ_k
  Output: 1 (accept) or 0 (reject)

For correctness, we define the correctness game. In this game, adversary $\mathcal{A}$ *is introduced to capture the order in which inserts and updates are applied to commitment.*

Definition 2.3 (correctness game [4])**.** *We define the random variable* $g^{correct}_{\mathsf{KVC},\lambda,\mathcal{A}}$ *through the following game between a challenger* CH *and an adversary* $\mathcal{A}$*:*

1. CH *generates* $(\mathsf{pp}, \mathsf{C}) \xleftarrow{\$} KeyGen(1^\lambda)$ *and sends* $(\mathsf{pp}, \mathsf{C})$ *to* $\mathcal{A}$. CH *maintains the initial commitment* C, *the initial* key-value map $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$, *and a map* $\mathcal{P}$ *that associates each key with its corresponding proof.*
2. $\mathcal{A}$ *executes one of the following queries:*
   - *(Insert,*(k, v)*):* CH *checks whether the key* k *already exists in* $\mathcal{M}$. *If it does, output* $\perp$. *Otherwise,* CH *updates* $\mathcal{M}$ *by adding* (k, v), *executes* $Insert(k, v)$ *to obtain a new commitment* C, *a proof* Λ_k *corresponding to* k, *and update information* upd. CH *finally outputs* $\mathcal{P} \cup \{(k, \Lambda_k)\}$.
   - *(Update,*(k, δ)*):* CH *checks whether the key* k *exists in* $\mathcal{M}$. *If it does not, output* $\perp$. *Otherwise,* CH *updates* $\mathcal{M}$ *to* $(\mathcal{M} \cup \{(k, v + \delta)\}) \setminus \{(k, v)\}$. CH *executes* $Update(k, \delta)$ *to obtain a new commitment* C *and update information* upd.

   Whenever $\mathcal{A}$ *executes a query,* CH *processes the query in one of the above ways, then performs the following updates and checks.*

   - *For each* key-proof pair $(k, \Lambda_k) \in \mathcal{P}$, CH *executes* $Proofupdate(k, \Lambda_k, \mathsf{upd})$ *using the obtained update information* upd.
   - *For each pair* $(k, v) \in \mathcal{M}$ *and the corresponding* $(k, \Lambda_k) \in \mathcal{P}$, CH *computes* $b_k \leftarrow Ver(C, (k, v), \Lambda_k)$. *If there exists any* k *such that* $b_k = 0$, CH *outputs failure and terminates.*

3. CH *outputs success.*

Definition 2.4 (correctness [4])**.** KVC *is correct if the following probability is identically zero for every PPT adversary* $\mathcal{A}$*,*

$$Adv^{correct}_{\mathsf{KVC},\mathcal{A}}(\lambda) = \Pr\left[failure \xleftarrow{\$} g^{correct}_{\mathsf{KVC},\lambda,\mathcal{A}}\right]. \tag{1}$$

The security requirement for KVC is key-binding. To define key-binding, we define the key-binding game.

Definition 2.5 (key-binding game [4])**.** *We define the random variable* $g^{bind}_{\mathsf{KVC},\lambda,\mathcal{A}}$ *through the following game between a challenger* CH *and an adversary* $\mathcal{A}$*:*

1. CH *generates* $(\mathsf{pp}, \mathsf{C}) \xleftarrow{\$} KeyGen(1^\lambda)$ *and sends* $(\mathsf{pp}, \mathsf{C})$ *to* $\mathcal{A}$*.* CH *maintains the initial commitment* C*, the initial key-value map* $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{V}$*, and a map* $\mathcal{P}$ *that associates each key with its corresponding proof.*
2. $\mathcal{A}$ *executes one of the following queries:*
   - *(Insert,*(k, v)*):* CH *checks whether the key* k *already exists in* $\mathcal{M}$*. If it does, output* $\perp$*. Otherwise,* CH *updates* $\mathcal{M}$ *by adding* (k, v)*, executes* $Insert(k, v)$ *to obtain a new commitment* C*, a proof* Λ_k *corresponding to* k*, and update information* upd*.* CH *finally outputs* $\mathcal{P} \cup \{(k, \Lambda_k)\}$*.*
   - *(Update,*(k, δ)*):* CH *checks whether the key* k *exists in* $\mathcal{M}$*. If it does not, output* $\perp$*. Otherwise,* CH *updates* $\mathcal{M}$ *to* $(\mathcal{M} \cup \{(k, v+\delta)\}) \setminus \{(k, v)\}$*.* CH *executes* $Update(k, \delta)$ *to obtain a new commitment* C *and update information* upd*.*
3. *Eventually,* $\mathcal{A}$ *sends a final output to* CH *in one of the following forms:*
   - *Type 1: A key* k*, value* v*, and proof* Λ_k *where* k *does not exist in* $\mathcal{M}$*.*
   - *Type 2: A key* k *that exists in* $\mathcal{M}$ *together with two distinct values* v *and* v' $(v \neq v')$*, and corresponding proofs* Λ_k *and* Λ'_k*.*
4. *Upon receiving* $\mathcal{A}$*'s output,* CH *proceeds as follows:*
   - *Type 1: If* $Ver(C, (k, v), \Lambda_k) = 1$*, then* CH *outputs failure; otherwise, outputs success.*
   - *Type 2: If both* $Ver(C, (k, v), \Lambda_k) = 1$ *and* $Ver(C, (k, v'), \Lambda'_k) = 1$*, then* CH *outputs failure; otherwise, outputs success.*

Definition 2.6 (key-binding [4])**.** KVC *is key-binding if the following probability is negligible for every PPT adversary* $\mathcal{A}$*,*

$$Adv^{bind}_{\mathsf{KVC},\mathcal{A}}(\lambda) = \Pr\left[failure \xleftarrow{\$} g^{bind}_{\mathsf{KVC},\lambda,\mathcal{A}}\right]. \tag{2}$$

2.2 Security assumptions

In this section, we define RSA assumption, GRSA assumption, and SRSA assumption as the security assumptions used in this research. In all definitions, we use a polynomial-time algorithm $\mathsf{GGen}(\lambda)$ that, given a security parameter λ, outputs two integers a, b and a group $\mathbb{G}$ whose order lies within the range $[a, b]$, but whose exact order remains unknown.

Definition 2.7 (RSA assumption [4]). *RSA assumption holds if for any PPT adversary $\mathcal{A}$, the following probability is negligible in λ,*

$$Adv_{\mathcal{A}}^{RSA}(\lambda) = \Pr\left[u^{\ell} = w \ : \ \begin{array}{l}(a,b,\mathbb{G}) \xleftarrow{\$} \mathsf{GGen}(\lambda) \\ w \xleftarrow{\$} \mathbb{G} \\ \ell \xleftarrow{\$} \mathsf{Primes}(\lambda) \\ u \xleftarrow{\$} \mathcal{A}(a,b,\mathbb{G},w,\ell)\end{array}\right]. \tag{3}$$

Definition 2.8 (Generalized RSA assumption [4]). *GRSA assumption holds if for any PPT adversary $\mathcal{A}$, the following probability is negligible in λ,*

$$Adv_{\mathcal{A}}^{GRSA}(\lambda) = \Pr\left[u^{\ell} = w \ : \ \begin{array}{l}(a,b,\mathbb{G}) \xleftarrow{\$} \mathsf{GGen}(\lambda),\ |b| = \zeta \\ w \xleftarrow{\$} \mathbb{G} \\ \ell \xleftarrow{\$} \mathsf{Primes}(\zeta+1) \setminus [b] \\ u \xleftarrow{\$} \mathcal{A}(a,b,\mathbb{G},w,\ell)\end{array}\right]. \tag{4}$$

Definition 2.9 (Strong RSA assumption [4]). *SRSA assumption holds if for any PPT adversary $\mathcal{A}$, the following probability is negligible in λ,*

$$Adv_{\mathcal{A}}^{SRSA}(\lambda) = \Pr\left[\begin{array}{l}u^{\ell} = w \\ \ell \in \mathsf{Primes} \setminus \{2\}\end{array} \ : \ \begin{array}{l}(a,b,\mathbb{G}) \xleftarrow{\$} \mathsf{GGen}(\lambda) \\ w \xleftarrow{\$} \mathbb{G} \\ u \xleftarrow{\$} \mathcal{A}(a,b,\mathbb{G},w)\end{array}\right]. \tag{5}$$

3 Related Works

This Section describes Key-Value Commitments [4], the challenges of the existing research [4], and Oblivious Accumulators [5] in Sections 3.1, 3.2, and 3.3, respectively.

3.1 Key-Value Commitments [4]

In [4], two types of Key-Value Commitments were proposed: Key-Value Commitments with key-value pair insertion capability (KVC-Ins), and Key-Value Commitments with both insertion and value update capabilities (KVaC). In this section, we focus on KVaC, which allows both insertion and value updates, and omit the description of KVC-Ins which only supports insertions. In KVaC, a variable u_i is introduced to represent the number of times the value corresponding to key k_i has been updated. KVaC consists of the following algorithms. Algorithms 1 is executed by the Center, while Algorithms 2, 3, and 5 involve interactions between the Center and User. Algorithms 4 is performed individually by User.

Algorithm 1 Keygen

Input: security parameter λ
Output: $(\mathsf{pp}, \mathsf{C}) = ((\mathsf{a}, \mathsf{b}, \mathbb{G}, \mathsf{g}, \mathsf{H}), (1, \mathsf{g}))$
Center executes :
1: $(a, b, \mathbb{G}) \xleftarrow{\$} \mathsf{GGen}(\lambda), g \xleftarrow{\$} \mathbb{G}$
2: $\mathcal{V} = [0, a)$ and $\mathcal{K} = \{0,1\}^*, \zeta = |b|$
3: $H : \{0,1\}^* \to \mathsf{Primes}(\zeta + 1) \backslash [b]$
4: $\mathsf{pp} = (a, b, \mathbb{G}, g, H)$
5: $C = (1, g)$
Send pp to all User.

Algorithm 2 Insert

Input: $C = (C_1, C_2) \in (\mathbb{G} \times \mathbb{G}), (k, v) \in (\mathcal{K} \times \mathcal{V})$
Output: $(C, \Lambda_k, \mathsf{upd})$
User executes : Send (k, v) to Center.
Center executes :
1: $z = H(k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
2: $C = (C_1^z \cdot C_2^v, C_2^z) \in (\mathbb{G} \times \mathbb{G})$
3: $\Lambda_k = ((C_1, C_2), (g, 1, 1), 0) \in (\mathbb{G} \times \mathbb{G}, \mathbb{G} \times \mathbb{G} \times \mathbb{G}, \mathbb{Z}_{\geq 0})$
4: $\mathsf{upd} = (\mathsf{insert}, (\mathsf{k}, \mathsf{v}))$
Send Λ_k to User secretly, and upd to all User.

Algorithm 3 Update

Input: $C = (C_1, C_2) \in (\mathbb{G} \times \mathbb{G}), (k, \delta) \in (\mathcal{K} \times \mathbb{Z})$
Output: (C, upd)
User_i executes : Send (k, δ) to Center.
Center executes :
1: $z = H(k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
2: $C = (C_1^z \cdot C_2^\delta, C_2^z) \in (\mathbb{G} \times \mathbb{G})$
3: $\mathsf{upd} = (\mathsf{update}, (\mathsf{k}, \delta))$
Send upd to all User.

Algorithm 4 ProofUpdate

Input: $\Lambda_k = ((\Lambda_{k,1}, \Lambda_{k,2}), (\Lambda_{k,3}, \Lambda_{k,4}, \Lambda_{k,5}), u_k)$
$\in (\mathbb{G} \times \mathbb{G}, \mathbb{G} \times \mathbb{G} \times \mathbb{G}, \mathbb{Z}_{\geq 0})$
$\mathsf{upd} = (\mathsf{upd}_1, (\mathsf{upd}_2, \mathsf{upd}_3)), \mathsf{k} \in \mathcal{K}$
Output: Λ_k
All User executes :
1: $z = H(k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
2: **if** $\mathsf{upd}_2 = k$ **then**
3: $\Lambda_k = ((\Lambda_{k,1}, (\Lambda_{k,2})^z), (\Lambda_{k,3}, \Lambda_{k,4}, \Lambda_{k,5}), u_k + 1)$
$\in (\mathbb{G} \times \mathbb{G}, \mathbb{G} \times \mathbb{G} \times \mathbb{G}, \mathbb{Z}_{\geq 0})$
4: **else**
5: $\hat{z} = H(\mathsf{upd}_2) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
6: $z \neq \hat{z}$
7: $\alpha \cdot z + \beta \cdot \hat{z} = 1$
8: $\gamma = \beta \cdot \Lambda_{k,5} \mod z$
9: $\gamma \cdot \hat{z} + \eta \cdot z = \Lambda_{k,5}$
10: $\Lambda_k = (((\Lambda_{k,1})^{\hat{z}}, (\Lambda_{k,2})^{\mathsf{upd}_3}), ((\Lambda_{k,3})^{\hat{z}}, \Lambda_{k,4} \cdot \Lambda_{k,3}^{\eta}, \gamma), u_k)$
$\in (\mathbb{G} \times \mathbb{G}, \mathbb{G} \times \mathbb{G} \times \mathbb{G}, \mathbb{Z}_{\geq 0})$
11: **end if**

Algorithm 5 Ver

Input: $\Lambda_k = ((\Lambda_{k,1}, \Lambda_{k,2}), (\Lambda_{k,3}, \Lambda_{k,4}, \Lambda_{k,5}), u_k)$,
$\in (\mathbb{G} \times \mathbb{G}, \mathbb{G} \times \mathbb{G} \times \mathbb{G}, \mathbb{Z}_{\geq 0})$
$C = (C_1, C_2) \in (\mathbb{G} \times \mathbb{G}), (k, v) \in (\mathcal{K} \times \mathcal{V})$
Output: 1 or 0
Verifier who has $((k, v), \Lambda_k)$ executes :
Send $((k, v), \Lambda_k)$ to Center.
Center executes :
1: $z = H(k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
2: **if** the following 3-7 holds **then**
3: $v \in \mathcal{V}$ and $k \in \mathcal{K}$ and $u_k \in \mathbb{Z}_{\geq 0}$
4: $(\Lambda_{k,2})^z = C_2 \in \mathbb{G}$
5: $(\Lambda_{k,1})^{z^{u_k+1}} \cdot (\Lambda_{k,2})^v = C_1 \in \mathbb{G}$
6: $(\Lambda_{k,3})^{z^{u_k+1}} = C_2 \in \mathbb{G}$
7: $(\Lambda_{k,4})^z \cdot (\Lambda_{k,3})^{\Lambda_{k,5}} = g \in \mathbb{G}$
8: **return** 1
9: **else**
10: **return** 0
11: **end if**
Send the result to the verifier.

3.2 The challenges of the existing research

- The linkability

  Since Algorithms 2 and 3 output $(insert, (k, v))$ and $(update, (k, \delta))$, respectively, these transactions are linked from the output information of the same key k.

- The size of each proof

  The proof consists of three components, denoted as $\Lambda_k \in (\mathbb{G} \times \mathbb{G}, \mathbb{G} \times \mathbb{G} \times \mathbb{G}, \mathbb{Z}_{\geq 0})$.These five elements in $\mathbb{G}$ must be updated via exponentiations or multiplications in $\mathbb{G}$ when the value is updated by other Users. Since the number of group elements in the proof directly affects the computational cost, it is desirable to keep the proof as simple as possible.

- The sign of the value

  In Algorithms 2, the sign of the value is always positive, whereas in Algorithms 3, it can be either positive or negative. As a result, the update operations leak the sign of the value, i.e., whether the value increases or decreases.

3.3 Oblivious Accumulators [5]

In [5], the concept of Oblivious Accumulators (OblvAcc) was proposed. The goal of OblvAcc is to hide the details of the basic operations executed on the accumulator. OblvAcc provides only membership proofs and allows elements to be added and deleted. The existence of OblvAcc with non-membership proofs is an open problem. A set $\mathcal{S} \subseteq \mathcal{D}$ is a collection of elements $x \in \mathcal{D}$ where $\mathcal{D}$ is the Acc domain. We define OblvAcc by the following algorithm.

- Setup $(\mathsf{pp}, C) \xleftarrow{\$} \mathrm{Setup}(1^\lambda)$:

  Input: security parameter λ
  Output: public parameter pp (defined Acc domain $\mathcal{D}$) and initial commitment C which corresponds to an empty $\mathcal{S}$.

- Element Addition $(C, w_x, \mathsf{upd}, \mathsf{aux}) \leftarrow \mathrm{Add}(\mathsf{C}, \mathsf{x}, \mathsf{U})$:

  Input: commitment C, element $x \in \mathcal{S}$, the digest of all update information U
  Output: new commitment C, membership proof $w_x (x \in \mathcal{S})$, update information upd, auxiliary information aux (only known to the user that holds x)

- Element Delete $(C, \mathsf{upd}) \leftarrow \mathrm{Del}(C, x, U, aux)$:

  Input: commitment C, element $x \in \mathcal{S}$, the digest of all update information U, auxiliary information aux
  Output: new commitment C, update information upd

- Membership ProofUpdate

  $w_x \leftarrow \mathrm{MemProofUpdate}(w_x, upd)$:

  Input: membership proof w_x, update information upd
  Output: updated membership proof w_x

- Verify Membership proof

  $0/1 \leftarrow \mathrm{MemVer}(C, x, w_x, aux)$:

  Input: commitment C, element $x \in \mathcal{S}$, membership proof w_x, auxiliary information aux
  Output: 1 (accept) or 0 (reject)

For correctness, $\mathrm{MemVer}(C, x, w_x, aux)$ must always return 1 for any $x \in \mathcal{S}$ with correctly generated C, w_x, and aux. The security requirement for OblvAcc is weak soundness. To satisfy weak soundness, it must be computationally infeasible for a polynomially bounded adversary (with knowledge of pp) to come up with a valid proof for an element not added to an honestly generated accumulator. In addition, OblvAcc has the following properties.

Definition 3.1 (Element hiding [5]). *For any PPT adversary $\mathcal{A}$, if the following probability can be bounded by a negligible advantage over $\frac{1}{2}$, then* OblvAcc *satisfies Element hiding.*

$$\Pr\left[b' = b \;\middle|\; \begin{array}{l} (\mathsf{pp}, \mathsf{C}_0) \xleftarrow{\$} Setup(1^\lambda) \\ x_0, x_1 \xleftarrow{\$} \mathcal{A}(\mathsf{pp}, \mathsf{C}_0) \\ b \xleftarrow{\$} \{0,1\} \\ (C_1, w_{x_b}, u_1, aux) \xleftarrow{\$} Add(C_0, x_b, \varnothing) \\ (C_2, u_2) \xleftarrow{\$} Del(C_1, x_b, \{u_1\}, aux) \\ b' \xleftarrow{\$} \mathcal{A}(C_1, C_2, u_1, u_2) \end{array} \right] \tag{6}$$

The above property is meant to provide the guarantee that an adversary who observes the publicly available information does not learn about the elements in the accumulated set $\mathcal{S}$.

Definition 3.2 (Add-Delete indistinguishability [5]). *For any PPT adversary $\mathcal{A}$, if the following probability can be bounded by a negligible advantage over $\frac{1}{2}$, then* OblvAcc *satisfies Add-Delete indistinguishability.*

$$\Pr\left[b' = b \;\middle|\; \begin{array}{l} (\mathsf{pp}, C_0) \xleftarrow{\$} Setup(1^\lambda) \\ x_0, x_1 \xleftarrow{\$} \mathcal{A}(\mathsf{pp}, C_0) \\ (C_1, w_{x_0}, u_1, aux_0) \xleftarrow{\$} Add(C_0, x_0, \varnothing) \\ b \xleftarrow{\$} \{0,1\}, x_1 \xleftarrow{\$} D \\ \textbf{\textit{if}} \quad b = 0 : (C_2, w_{x_1}, u_2, aux_1) \xleftarrow{\$} Add(C_1, x_1, \{u_1\}) \\ \textbf{\textit{if}} \quad b = 1 : (C_2, u_2) \xleftarrow{\$} Del(C_1, x_0, \{u_1\}, aux_0) \\ b' \xleftarrow{\$} \mathcal{A}(C_1, C_2, u_1, u_2) \end{array} \right] \tag{7}$$

The above property is meant to provide the guarantee that an adversary who observes the publicly available information does not learn whether an operation is an Add or a Delete.

4 Proposed method

This Section describes the goal of this research, the detailed algorithms, and the security proof of unlinkability and key-binding in Sections 4.1, 4.2, and 4.3, respectively.

4.1 The goal of this research

The goal of this research is to achieve unlinkability of transactions corresponding to the same User from the key k output at each operation. In addition, we aim to reduce the burden of proof updating and to prevent leakage of the sign of the value change when updating values. These objectives are achieved by applying OblvAcc to KVaC.

Definition 4.1 (Unlinkability). *Our scheme is said to satisfy* unlinkability *if both insertion and update operations meet the following conditions:*

- *Given two update information values* $\mathsf{upd}_i = (z_i, v_i)$ *and* $\mathsf{upd}_j = (z_j, v_j)$ *output by insertion operations of* $\mathsf{User_i}$ *and* $\mathsf{User_j}$*, it is infeasible to determine whether they were generated by the same user or by different users.*
- *Given the update information* $\mathsf{upd}_i = (z_i, v_i)$ *output by the insertion operation of* $\mathsf{User_i}$*, and the update information* $\mathsf{upd}_j = (z_j, v_j)$ *output by the update operation of* $\mathsf{User_j}$*, it is infeasible to determine whether they were generated by the same user or by different users.*
- *Given two update information values* $\mathsf{upd}_i = (z_i, v_i)$ *and* $\mathsf{upd}_j = (z_j, v_j)$ *output by update operations of* $\mathsf{User_i}$ *and* $\mathsf{User_j}$*, it is infeasible to determine whether they were generated by the same user or by different users.*

4.2 Proposed method

We describe the algorithms of the proposed method, OblvAcc-KVC, which integrates OblvAcc into KVaC. OblvAcc-KVC consists of the following algorithms: Key generation 6 composed of 14, 16 executed by the Center; Pair insertion 7 composed of 15, Value update 8, 9 composed of 13, 15, 17, 18, and Verification proof 12, 13, which involved interactions between the Center and User; and Proof update 10, 11 performed individually by User.

We consider User_i who holds the key k_i and value $v_{i,0}$. When $j = 0$, it is the output of the initial pair insertion operation for User_i by Algorithm 7. During the value update, by executing

Algorithm 18 within Algorithm 9, the maximum number of elements in the aux stored in OblvAcc can be limited to the number of User.

While User_i executes Algorithm 7, other Users update their proofs from $\mathsf{upd} = (z_{i,0}, v_{i,0})$ by using Algorithm 10. The proof verifications of all User are performed by using Algorithm 12.

While User_i executes Algorithm 8, User storing aux in OblvAcc updates their Λ_a for aux from $\mathsf{upd} = (z_{i,0}, v_{i,0})$ by using Algorithm 11, and other Users update their proofs from $\mathsf{upd} = (z_{i,1}, v_{i,0}+\delta_1)$ by using Algorithm 10. The proof verifications of all User are performed by using Algorithm 12. User_i discards the previous proof $\Lambda_{k_{i,0}}$ as it is no longer needed. Also, $v_{i,0}+\delta_1$ is always positive.

While User_i executes Algorithm 9, User storing aux in OblvAcc updates their Λ_a for aux from $\mathsf{upd}_a = z_{D_{i,j}}, z_{A_{i,j+1}}$ by using Algorithm 11, and other Users update their proofs from $\mathsf{upd} = (z_{i,j+1}, v_{i,j}+\delta_{j+1})$ by using Algorithm 10. The proof verifications of all User are performed by using Algorithm 12. User_i discards $\Lambda_{a_{i,j}}$ associated with the deleted $aux_{i,j-1}$ and the previous proof $\Lambda_{k_{i,j}}$ as it is no longer needed. Also, $v_{i,j}+\delta_{j+1}$ is always positive.

Algorithm 6 OblvAcc-KVC.KeyGen

Input: λ
Output: $\mathsf{pp}, C, \mathsf{pp}_a, C_a$
1: $\mathsf{pp}, C \leftarrow$Algorithm 14 (λ)
2: $\mathsf{pp}_a, C_a \leftarrow$Algorithm 16 (λ)

Algorithm 7 OblvAcc-KVC.Insert
(Pair $(k_i, v_{i,0})$ insertion by User_i)

Input: $C, (k_i, v_{i,0})$
Output: $aux_{i,0}, C, \Lambda_{k_{i,0}}, \mathsf{upd}$
$\leftarrow$Algorithm 15 $(C, (k_i, v_{i,0}))$

Algorithm 8 OblvAcc-KVC.Update
(1st value update by User_i $v_{i,0} \to v_{i,0}+\delta_1$)

Input: $C_a, k_i, aux_{i,0}, \Lambda_{a_{i,1}}, s_{i,0}, C, (k_i, v_{i,0}+\delta_1)$
Output: $C_a, \Lambda_{a_{i,1}}, \mathsf{upd}_a$, 0 or 1,
$aux_{i,1}, C, \Lambda_{k_{i,1}}, \mathsf{upd}$
1: $C_a, \Lambda_{a_{i,1}}, \mathsf{upd}_a \leftarrow$Algorithm 17 $(C_a, k_i, aux_{i,0})$
2: 0 or 1
$\leftarrow$Algorithm 13 $(C_a, \Lambda_{a_{i,1}}, k_i, aux_{i,0}, s_{i,0})$
3: **if** Output 1 **then**
4: Delete past pair $(k_i, v_{i,0})$ from key-value map
5: $aux_{i,1}, C, \Lambda_{k_{i,1}}, \mathsf{upd}$
$\leftarrow$Algorithm 15 $(C, (k_i, v_{i,0}+\delta_1))$
6: **else**
7: **return** 0
8: **end if**

Algorithm 9 OblvAcc-KVC.Update2
(j+1th ($j \geq 1$) value update by User_i
$v_{i,j} \to v_{i,j}+\delta_{j+1}$)

Input: $C_a, k_i, aux_{i,j-1}, aux_{i,j}, \Lambda_{a_{i,j+1}}$,
$C, s_{i,j}, (k_i, v_{i,j}+\delta_{j+1})$
Output: $C_a, \mathsf{upd}_a = z_{D_{i,j}}, z_{A_{i,j+1}}, \Lambda_{a_{i,j+1}}$,,
0 or 1, $aux_{i,j+1}, C, \Lambda_{k_{i,j+1}}, \mathsf{upd}$
1: $C_a, \mathsf{upd}_a \leftarrow$Algorithm 18 $(C_a, k_i, aux_{i,j-1})$
2: $C_a, \Lambda_{a_{i,j+1}}, \mathsf{upd}_a$
$\leftarrow$Algorithm 17 $(C_a, k_i, aux_{i,j})$
3: 0 or 1
$\leftarrow$Algorithm 13 $(C_a, \Lambda_{a_{i,j+1}}, k_i, aux_{i,j}, s_{i,j})$
4: **if** Output 1 **then**
5: Delete past pair $(k_i, v_{i,j})$ from key-value map
6: $aux_{i,j+1}, C, \Lambda_{k_{i,j+1}}, \mathsf{upd}$
$\leftarrow$Algorithm 15 $(C, (k_i, v_{i,j}+\delta_{j+1}))$
7: **else**
8: **return** 0
9: **end if**

Algorithm 10 OblvAcc-KVC.ProofUpdate

Input: $\Lambda_k = (\Lambda_{k,1}\Lambda_{k,2}) \in (\mathbb{G} \times \mathbb{G})$
$\mathsf{upd} = (\mathsf{upd}_1, \mathsf{upd}_2)$
Output: Λ_k
All User execute :
1: $\Lambda_k = ((\Lambda_{k,1})^{\mathsf{upd}_1} \cdot (\Lambda_{k,2})^{\mathsf{upd}_2}, (\Lambda_{k,2})^{\mathsf{upd}_1}) \in (\mathbb{G} \times \mathbb{G})$

Algorithm 11 OblvAcc-KVC.MemProofUpdate

Input: $\Lambda_a \in \mathbb{G}, \mathsf{upd}_a$
Output: Λ_a
User storing aux in OblvAcc execute :
1: $\Lambda_a = \Lambda_a^{\mathsf{upd}_a} \in \mathbb{G}$

Algorithm 12 OblvAcc-KVC.Verification

Input: $\Lambda_k = (\Lambda_{k,1}, \Lambda_{k,2}) \in (\mathbb{G} \times \mathbb{G})$
$C = (C_1, C_2) \in (\mathbb{G} \times \mathbb{G}), (k, v) \in (\mathcal{K} \times \mathcal{V}), aux$
Output: 1 or 0
Verifier who has $((k, v), \Lambda_k, aux)$ executes :
Send $((k, v), \Lambda_k, aux)$ to Center.
Center executes :
$z = H(aux) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
if the following 3-5 holds **then**
$v \in \mathcal{V}$ and $k \in \mathcal{K}$
$(\Lambda_{k,2})^z = C_2 \in \mathbb{G}$
$(\Lambda_{k,1})^z \cdot (\Lambda_{k,2})^v = C_1 \in \mathbb{G}$
return 1
else
return 0
end if
Send the result to the verifier.

Algorithm 13 OblvAcc-KVC.MemVerification

Input: $C_a \in \mathbb{G}, \Lambda_a \in \mathbb{G}, k, aux, s$
Output: 1 or 0
Verifier who has (Λ_a, k, aux) executes :
Send (Λ_a, k, aux) to Center.
Center executes :
$z_A = H(aux||k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
if the following 3-4 holds **then**
$aux = H(s||k)$
$(\Lambda_a)^{z_A} = C_a \in \mathbb{G}$
return 1
else
return 0
end if
Send the result to the verifier.

Algorithm 14 KVC.Keygen

Input: security parameter λ
Output: $(\mathsf{pp}, C) = ((a, b, \mathbb{G}, g_1, H), (1, g_1))$
Center executes :
$(a, b, \mathbb{G}) \xleftarrow{\$} \mathsf{GGen}(\lambda), g_1 \xleftarrow{\$} \mathbb{G}$
$\mathcal{V} = [0, a)$ and $\mathcal{K} = \{0, 1\}^*, \zeta = |b|$
$H : \{0, 1\}^* \to \mathsf{Primes}(\zeta + 1) \backslash [b]$
$\mathsf{pp} = (a, b, \mathbb{G}, g_1, H)$
$C = (1, g_1)$
Send pp to all User.

Algorithm 15 KVC.Insert

Input: $C = (C_1, C_2) \in (\mathbb{G} \times \mathbb{G}), (k, v) \in (\mathcal{K} \times \mathcal{V})$
Output: $(aux, C, \Lambda_k, \mathsf{upd})$
User executes : Send (k, v) to Center.
Center executes :
$s \xleftarrow{\$} \{0, 1\}^\lambda$
$aux = H(s||k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
$z = H(aux) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
$C = (C_1^z \cdot C_2^v, C_2^z) \in (\mathbb{G} \times \mathbb{G})$
$\Lambda_k = C \in (\mathbb{G} \times \mathbb{G})$
$\mathsf{upd} = (z, v)$
Keep s corresponding to aux secretly
Send Λ_k, aux to User secretly, and upd to all User.

Algorithm 16 OblvAcc.Setup

Input: security parameter λ
Output: $(\mathsf{pp}_a, C_a) = ((a, b, \mathbb{G}, g_2, H), g_2)$
Center executes :
$(a, b, \mathbb{G}) \xleftarrow{\$} \mathsf{GGen}(\lambda), g_2 \xleftarrow{\$} \mathbb{G}, \zeta = |b|$
$H : \{0, 1\}^* \to \mathsf{Primes}(\zeta + 1) \backslash [b]$
$\mathsf{pp}_a = (a, b, \mathbb{G}, g_2, H)$
$C_a = g_2$
Send pp_a to all User.

Algorithm 17 OblvAcc.Add

Input: $C_a \in \mathbb{G}, k, aux$
Output: $(C_a, \Lambda_a, \mathsf{upd}_a)$
User executes : Send (k, aux) to Center.
Center executes :
$z_A = H(aux||k) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
$C_a = C_a^{z_A} \in \mathbb{G}$
$\Lambda_a = C_a \in \mathbb{G}$
$\mathsf{upd}_a = z_A$
Send Λ_a to User secretly, and upd_a to all User.

Algorithm 18 OblvAcc.Del

Input: $C_a \in \mathbb{G}, k, aux$
Output: (C_a, upd_a)
User executes : Send (k, aux) to Center.
Center executes :
$z_D = H(k||aux) \in \mathsf{Primes}(\zeta + 1) \backslash [b]$
$C_a = C_a^{z_D} \in \mathbb{G}$
$\mathsf{upd}_a = z_D$
Send upd_a to all User.

By utilizing OblvAcc for KVaC, we solve three challenges 3.2. Let us explain how we resolve the linkability. Instead of outputting key information k at each operation, unlinkability is achieved by outputting different values z each time, as shown in Algorithm 15. When User requests an insertion, Center randomly selects s each time and computes aux using the hash function H with the requested k. The calculation of z is performed by inputting aux into H. Since s is randomly selected each time during pair insertions, each time someone with the same key k updates v, different information z is obtained. aux is the information known only to User who holds the pair (k, v). Since s is necessary for verification, Center stores it as the value corresponding to aux. To perform a value update, User must add aux to OblvAcc. Let us explain how we resolve the size of each proof and the sign of the value. By processing updates as insertion, User updating value discards the previous proof and obtains a new one. As a result, the number of value updates is not necessary for proof update, and since the proof consists of only two elements (C before the new pair insertion), the computational cost of proof updates for User is reduced. Also, the sign of the value is always positive by unifying with the insert operation. Furthermore, by utilizing OblvAcc, the leakage of aux is prevented due to the Element hiding (Definition 3.1), which ensures that aux cannot be determined, and Add-Delete indistinguishability (Definition 3.2), which ensures that additions and deletions of aux are indistinguishable.

4.3 Security proof

Theorem 4.1. *If the properties of the hash function hold,* KVC *in our proposal satisfies unlinkability.*

Proof. If an adversary can find two distinct inputs $aux \neq aux'$ such that $H(aux) = H(aux')$, it breaks the Collision resistance. If the adversary is given a hash value $H(aux)$ and can find the corresponding input aux' such that $H(aux') = H(aux)$, it breaks the Preimage resistance. If the adversary is given an input aux and can find a different input $aux' \neq aux$ such that $H(aux') = H(aux)$, it breaks the Second preimage resistance. ∎

The soundness of OblvAcc is the same as the [5], we describe the key-binding in our proposal.

Theorem 4.2. *If GRSA assumption holds,* OblvAcc-KVC *satisfies key-binding under the random oracle assumption for the hash function H.*

Proof. We consider the following Lemma 4.1

Lemma 4.1. *In the random oracle model, assume the existence of a PPT adversary $\mathcal{A}$ satisfying the following equation 8*

$$Adv^{bind}_{\mathsf{OblvAcc\text{-}KVC},\mathcal{A}}(\lambda) = \epsilon \tag{8}$$

Then, there exists a PPT adversary $\mathcal{B}$ satisfying the following equation 9

$$Adv^{GRSA}_{\mathcal{B}}(\lambda) \geq \frac{\epsilon}{T_\lambda^2} - negl(\lambda) \tag{9}$$

where T_λ denotes the running time of $\mathcal{A}$ defined by λ.

The proof of Lemma 4.1 can be constructed based on [4]. If Lemma 4.1 holds, then the existence of an adversary $\mathcal{A}$ that breaks the key-binding of OblvAcc-KVC with non-negligible probability ϵ implies the existence of an adversary $\mathcal{B}$ that breaks the GRSA assumption with at least probability $\frac{\epsilon}{T_\lambda^2}$. From the above, Theorem 4.2 is proven. ∎

Moreover, based on properties derived from the random oracle model and the GRSA assumption, we can modify the properties to be based on the SRSA assumption.

Theorem 4.3. *If SRSA assumption holds,* OblvAcc-KVC *satisfies key-binding without random oracle assumption.*

Proof. We consider the following Lemma 4.2

Lemma 4.2. *Assume the existence of a PPT adversary $\mathcal{A}$ satisfying the following equation 10*

$$Adv^{bind}_{\mathsf{OblvAcc\text{-}KVC},\mathcal{A}}(\lambda) = \epsilon \tag{10}$$

Then, there exists a PPT adversary $\mathcal{B}$ satisfying the following equation 11

$$Adv^{SRSA}_{\mathcal{B}}(\lambda) \geq \epsilon - negl(\lambda) \tag{11}$$

The proof of Lemma 4.2 can be constructed based on [4]. If Lemma 4.2 holds, then the existence of an adversary $\mathcal{A}$ that breaks the key-binding of OblvAcc-KVC with non-negligible probability ϵ implies the existence of an adversary $\mathcal{B}$ that breaks the SRSA assumption with at least probability ϵ. From the above, Theorem 4.3 is proven. ∎

5 Comparison

This Section compares our proposal with KVaC [4] from the point of view of update information, computational cost, and data size.

We compare the upd. Table 5 shows the comparison on upd output during insert and update operations by User_i, as described in Section 4.2.

Table 5: Comparison of upd

Research	Insertion $(k_i, v_{i,0})$	Update $(k_i, v_{i,j}) \rightarrow (k_i, v_{i,j} + \delta_{j+1})$
KVaC	$(insert, (k_i, v_{i,0}))$	$(update, (k_i, \delta_{j+1}))$
This research	$(z_{i,0}, v_{i,0})$	$(z_{i,j}, v_{i,j} + \delta_{j+1})$

As shown in Table 5, KVaC reveals the key k and the sign of the value change δ_{j+1}, which can be positive or negative in updates. In contrast, our scheme outputs a different value z each time instead of k. By treating updates as insertion, the sign of $v_{i,j} + \delta_{j+1}$ is always positive, thus preventing the leakage of the sign.

We compare the computational cost. Compared to KVaC, in our proposal, the computational cost required for proof update is reduced as shown in Table 2. If a User_i inserts a pair, for User other than User_i, the cost is reduced by one hash computation, one exponentiation, and five multiplications. For the updating User_i, the computational cost becomes zero. If a User_i updates value, for User other than User_i, the cost is reduced by one hash computation and five multiplications. However, if User stores aux in OblvAcc, the number of exponentiations for the updating User_i in the first update (Algorithm 8) remains the same as in [4]. If the updating User_i performs a second or later update (Algorithm 9), the number of exponentiations increases by one. If User doesn't store aux in OblvAcc, the number of exponentiations decreases by one.

We compare the data size. In KVaC, User holds a proof Λ_k of size $|5\mathbb{G}|$. In this research, with the introduction of OblvAcc, User needs to hold Λ_k of size $|2\mathbb{G}|$, and two aux_1 and aux_2 of size $||b|+1|$, and a membership proof Λ_a of size $|\mathbb{G}|$ for the OblvAcc, as shown in Table 3. The total

data size is $|3\mathbb{G}| + 2||b| + 1|$. Also, in the existing research, the number of interactions between User and Center is one for both the insertion and the update. In our proposal, the number of interactions is one for insertion (Algorithm 7), three for the first value update (Algorithm 8), and four for subsequent value updates (Algorithm 9).

6 conclusion

In [4], KVC (KVaC) that supports new pair insertions and value updates was proposed, but it has the following three issues. First, the same key is output for every transaction, which reveals whether transactions are performed by the same User. Second, the proof consists of three group elements, leading to a high computational cost for proof update. Third, the sign of the value change is leaked during updates. In this research, we define the unlinkability in KVC as a condition where each transaction is not linked to any other transaction, construct a new KVC that satisfies unlinkability and solve the three issues by incorporating the OblvAcc into KVaC. Instead of outputting key k for each operation, we output a different value z each time, thereby achieving unlinkability. By processing updates as insertion, User performing the update discards the previous proof and obtains a new one, enabling proofs to consist of only two elements, which reduces the computational cost of proof updates. Also, the sign of the value change is always positive by unifying with the insert operation.

References

[1] Bitcoin. https://bitcoin.org/.

[2] Ethereum. https://www.ethereum.org/.

[3] Ripple - one frictionless experience to send money globally. https://www. ripple.com.

[4] Shashank Agrawal and Srinivasan Raghuraman. Kvac: Key-value commitments for blockchains and beyond. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*, pages 839–869. Springer, 2020.

[5] Foteini Baldimtsi, Ioanna Karantaidou, and Srinivasan Raghuraman. Oblivious accumulators. In *IACR International Conference on Public-Key Cryptography*, pages 99–131. Springer, 2024.

[6] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39*, pages 561–586. Springer, 2019.

[7] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*, pages 55–72. Springer, 2013.

[8] Dario Fiore, Dimitris Kolonelos, and Paola de Perthuis. Cuckoo commitments: registration-based encryption and key-value map commitments for large spaces. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 166–200. Springer, 2023.

[9] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *International Conference on Applied Cryptography and Network Security*, pages 253–269. Springer, 2007.