New Unbounded Verifiable Data Streaming for Batch Query with Almost Optimal Overhead

Jiaojiao Wu[1], Jianfeng Wang[1,2], Xinwei Yong[1],
Xinyi Huang[3], and Xiaofeng Chen[1]

[1] School of Cyber Engineering, Xidian University,
Xi'an, China
{jiaojiaowujj,xwyong}@stu.xidian.edu.cn, {jfwang,xfchen}@xidian.edu.cn
[2] Zhengzhou Xinda Institute of Advanced Technology,
Zhengzhou, China
[3] Fujian Provincial Key Laboratory of Network Security and Cryptology, College of Computer and Cyber Security, Fujian Normal University,
Fuzhou, China
xyhuang81@gmail.com

Abstract. Verifiable Data Streaming (VDS) enables a resource-limited client to continuously outsource data to an untrusted server in a sequential manner while supporting public integrity verification and efficient update. However, most existing VDS schemes require the client to generate all proofs in advance and store them at the server, which leads to a heavy computation burden on the client. In addition, all the previous VDS schemes can perform batch query (i.e., retrieving multiple data entries at once), but are subject to linear communication cost l, where l is the number of queried data. In this paper, we first introduce a new cryptographic primitive named Double-trapdoor Chameleon Vector Commitment (DCVC), and then present an unbounded VDS scheme VDS_1 with optimal communication cost in the random oracle model from aggregatable cross-commitment variant of DCVC. Furthermore, we propose, to our best knowledge, the first unbounded VDS scheme VDS_2 with optimal communication and storage overhead in the standard model by integrating Double-trapdoor Chameleon Hash Function (DCH) and Key-Value Commitment (KVC). Both of our schemes enjoy constant-size public key. Finally, we demonstrate the efficiency of our two VDS schemes with a comprehensive performance evaluation.

Keywords: Verifiable data streaming · Data integrity · Batch query · Optimal overhead.

1 Introduction

With the rapid development of IoT, 5G and cloud computing, a growing number of devices collect continuous and never-ending data streams and tend to outsource massive data streams to cloud servers. While it brings in inherent advantages such as ease of maintenance, convenient access, and lower costs, data

outsourcing also results in data integrity concerns due to the untrusted cloud servers. Traditional solutions, such as Merkle Hash Tree (MHT) and Verifiable Database (VDB), can guarantee the integrity of the outsourced database and support data updates. However, these approaches either require frequent updates to the public verification key as data is appended or have an upper bound on the size of the database.

Verifiable Data Streaming (VDS), initiated by Schröder and Schröder [14], enables a resource-limited client to outsource a (potentially unbounded) data stream $D = (d_1, d_2, \dots)$ to an untrusted server while supporting public integrity verification and efficient update. In particular, the public verification key remains unchanged as data entries are continuously appended to the database. However, their VDS scheme sets a prior upper bound on the database size.

Recently, a line of research works [11,14,15,19] got rid of the upper bound and these schemes can be categorized into two different types: tree-based unbounded VDS scheme [11,14,15] and signature-based unbounded VDS scheme [11,19]. However, most practical scenarios require databases to support batch query that retrieves multiple data entries at once. The first type of unbounded VDS schemes [11,14,15], constructed by tree-based authentication data structures, can perform batch query, but are subject to linear communication cost l, where l is the number of queried data. The second type of unbounded VDS scheme [19] significantly reduces the query communication cost by using BLS signature and RSA accumulator. Concretely, this scheme can aggregate signatures to a single value relying on homomorphic properties of BLS signature and generate a constant-size non-membership witness for these signatures using a batching technique of RSA accumulator [3]. Nevertheless, the size of the auxiliary proof information (i.e., signature identifiers) is linear in the size of a batch query, which cause high communication costs. In addition, most existing VDS schemes [11,14,15,19] require that the client generates all proofs in advance and stores them at the server for subsequent integrity verification, which leads to a heavy computation burden on the client and huge storage overhead on the server, respectively. To this end, a naive solution is to transfer the proof generation from the client to the server. However, this approach may suffer from a key exposure problem or even fail to support the integrity verification. Therefore, it is still challenging to design a secure and efficient unbounded VDS scheme supporting batch queries.

Our Contributions. In this paper, we put forward two new unbounded VDS schemes VDS_1 and VDS_2 for batch query with almost optimal overhead in random oracle model and standard model, respectively. Both of our schemes enjoy constant-size public key. A comprehensive comparison of our schemes with previous works is shown in Table 1. In detail, our main contributions are summarized as follows.

- We introduce a new cryptographic primitive, Double-trapdoor Chameleon Vector Commitment (DCVC), which allows us to transfer the computation of proof generation from the client to the server without key exposure to reduce client computation and server storage. Then we present an unbounded VDS

Table 1. Comparison with existing VDS schemes

Scheme		[14]	[15]	CVC [11]	ACC [11]	VADS [19]	VDS_1	VDS_2
Unbounded		✗	✓	✓	✓	✓	✓	✓
Standard Model		✓	✗	✓	✓	✗	✗	✓
Size of Public Key		$O(1)$	$O(1)$	$O(q^2)$	$O(u)$	$O(1)$	$O(1)$	$O(1)$
Server Storage		$O(m)$	$O(n)$	$O(qn)$	$O(n+u)$	$O(n+u)$	$O(n)$	$O(1)$
Communication	$\lvert\pi\rvert$	$O(\log_2 m)$	$O(\log_2 n)$	$O(\log_q n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
	$\lvert\pi_b\rvert$	$O(l\cdot\log_2 m)$	$O(l\cdot\log_2 n)$	$O(l\cdot\log_q n)$	$O(l)$	$O(l)$	$O(1)$	$O(1)$
Computation	Append	$O(\log_2 m)$	$O(\log_2 n)$	$O(1)^{\dagger}$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
	Query	$O(\log_2 m)$	$O(\log_2 n)$	$O(\log_q n)$	$O(u)$	$O(1)^{\ddagger}$	$O(\log_q n)$	$O(n)$
	Verify	$O(\log_2 m)$	$O(\log_2 n)$	$O(\log_q n)$	$O(1)$	$O(1)$	$O(\log_q n)$	$O(1)$
	Update	$O(\log_2 m)$	$O(\log_2 n)$	$O(q\cdot\log_q n)$	$O(1)$	$O(1)$	$O(\log_q n)$	$O(1)$

Note: m denotes the maximum database size. n is the current database size. u is the number of updates. l is the number of queried data in a batch query. q denotes the number of branches of a q-ary tree. $|\pi|$ is the proof size of a single query. $|\pi_b|$ denotes the proof size of a batch query. †: In the CVC-based scheme [11], the server is required to perform an additional proof update with $O(1)$ computational complexity after the client appends a data entry. ‡: In the recently proposed scheme [19], the query process contains an extended Euclidean algorithm with logarithmic time complexity.

scheme VDS_1 with optimal communication cost in the random oracle model from a variant of DCVC with aggregatable cross-commitment.

- We explore a new approach to construct an efficient unbounded VDS scheme VDS_2 by leveraging Double-trapdoor Chameleon Hash Function (DCH) and Key-Value Commitment (KVC). To the best of our knowledge, our scheme VDS_2 is the first unbounded VDS scheme with optimal communication cost and server storage in the standard model .
- We implement our schemes VDS_1 and VDS_2 and perform a comprehensive evaluation and comparison. The results show that VDS_1 and VDS_2 are efficient in terms of communication cost, storage cost, and computation cost.

1.1 Related Work

Schröder and Schröder [14] introduced Verifiable Data Streaming (VDS) and presented the first VDS scheme based on Chameleon Authentication Tree (CAT). Their proposed scheme sets a fixed upper bound m on the database size, and the query communication and computation are logarithmic in this upper bound m. After that, Schröder and Simkin [15] put forward the first unbounded VDS scheme in the random oracle model to break the upper bound. Subsequently, Krupp et al. [11] proposed two unbounded VDS schemes in the standard model. The first scheme is constructed on Chameleon Vector Commitment (CVC) and a tree structure with logarithmic query communication. The second scheme is based on bilinear-map accumulator and signature scheme and has constant-size query communication, but the query computation is linear in the number of updates. However, all the previous VDS schemes are evaluated in a single query, and for batch query, the query communication of these scheme is increasing linearly with the number of queried data. Very recently, Wei et al. [19] presented an unbounded VDS scheme with data integrity auditing (VADS) in the random

Table 2. Summary of notations

Notation	Meaning
λ	The security parameter
$\mathsf{negl}(\lambda)$	A negligible function
q	The branching number of the tree
pp	The public parameter
td_1, td_2	The double trapdoors of DCVC
cnt	The data append counter
f	A pseudorandom function
C	The commitment value
m_i	The i-th message of the vector $(m_1, \ldots, m_n)$
$\vec{m}_I$	The subvector of the vector $(m_1, \ldots, m_n)$
d_i	The i-th data entry of data streaming
$\vec{d}_I$	The queried data entries in a batch query
I	The position set of queried data entries in a batch query
$\lvert I \rvert$	The number of queried data entries in a batch query
Ch	The double-trapdoor chameleon hash value

oracle model by using BLS signature and RSA accumulator. This scheme significantly reduces the batch query communication, but it is still not optimal for batch query.

In addition, other works to extend VDS, explore many practical applications, such as integrity preservation and range query. Xu et al. [21] and Sun et al. [17] considered privacy-preserving data integrity verification in VDS, while Chen et al. [6] and Miao et al. [13] proposed efficient integrity preservation schemes for data streaming. Tsai et al. [18] and Xu et al. [20] developed verifiable range query in data streaming. Most VDS schemes and their extensions require the client to generate all proofs in advance and store them at the server, which leads to heavy computation and storage burden on the client and server, respectively.

Therefore, it is interesting to explore new approaches to designing unbounded VDS scheme for batch query with optimal query communication and server storage in the standard model.

2 Preliminaries

In this section, we first introduce the notations used in the following (as shown in Table 2) and briefly review the hardness assumptions and cryptography tools used in this work.

2.1 Hardness Assumption

Definition 1 (Strong RSA Problem). *Given an RSA modulus $N = pq$ and a random value $g \in \mathbb{Z}_N^*$, where p and q are two distinct prime number. The strong RSA assumption holds, if for any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ and a security parameter λ the probability of outputting a tuple (y, e) s.t. $y^e = g \mod N$ is negligible, namely,*

$$\Pr[\mathcal{A}(N, g) \to (y, e) : y^e = g \bmod N] \leq \mathsf{negl}(\lambda).$$

2.2 Shamir's Trick

Shamir's trick [16] is a method to compute the xy-root of $g \in \mathbb{Z}_N^*$ without knowing the group order $\phi(N)$, where $x, y \in \mathbb{Z}$ with $\gcd(x, y) = 1$. Concretely, given an x-root and a y-root of $g \in \mathbb{Z}_N^*$, i.e., $g^{1/x} \mod N$ and $g^{1/y} \mod N$, one can compute u, v s.t. $ux + vy = 1$ using the extended Euclidean algorithm and then compute the xy-root of g as $g^{1/xy} = g^{\frac{ux+vy}{xy}} = g^{\frac{u}{y}+\frac{v}{x}} = (g^{1/y})^u (g^{1/x})^v \mod N$. We often denote performing a Shamir's trick as $g^{1/xy} \leftarrow \mathsf{ShamirTrick}(g^{1/x}, g^{1/y}, x, y)$.

2.3 Double-trapdoor Chameleon Hash Function

A Double-trapdoor Chameleon Hash Function (DCH) [8,9] is a probabilistic hash function with collision resistance, which allows one holding hash trapdoors to find collisions. In particular, a DCH scheme has double hash trapdoors including the long-term and one-time trapdoors, and the long-term trapdoor is never exposed. The DCH scheme in [8] consists of the following algorithms $\mathsf{DCH} = (\mathsf{DCHKGen}, \mathsf{DCHTGen}, \mathsf{DCH}, \mathsf{DCHCol})$:

- $\mathsf{DCHKGen}(1^\lambda)$: It takes a security parameter λ as input, chooses at random a λ-bit safe prime p s.t. $q \stackrel{\text{def}}{=} \frac{p-1}{2}$ is also a prime, picks $x \in_R \mathbb{Z}_q^*$ and $g \in \mathbb{Z}_p^*$ with prime order q, and computes $Y = g^x \mod p$. Finally, it outputs the public parameters $\mathsf{pp} = (p, q, g)$ and a long-term hash/trapdoor key pair (Y, x). The message space is $\mathcal{M} = \mathbb{Z}_q$.
- $\mathsf{DCHTGen}(\mathsf{pp})$: It chooses at random $k \in_R \mathbb{Z}_q^*$, computes $K = g^k \mod p$, and outputs a one-time hash/trapdoor key pair (K, k).
- $\mathsf{DCH}_{\mathsf{pp}}(Y, K, m, r)$: It takes a message m, a random element $r \in \mathbb{Z}_q$ and the long-term/one-time public hash keys (Y, K), and outputs a hash value $\mathsf{DCh}(m, r) = (KY)^m g^r \mod p$.
- $\mathsf{DCHCol}_{\mathsf{pp}}(x, k, m, r, m')$: It takes a message m, a random element r, another message m' and the long-term/one-time trapdoors (x, k), and finally outputs a collision $r' = r + (k + x)(m - m') \mod q$, s.t. $\mathsf{DCh}(m, r) = \mathsf{DCh}(m', r')$.

2.4 Key-Value Commitment

A Key-Value Commitment (KVC) [1] is a cryptographic primitive, which allows one to commit key-value tuples $\{(k_1, v_2), (k_2, v_2), \dots\}$ and to later open the commitment at any key, and supports adding new key-value tuples and updating the old value to a new one at an existing key. The commitment size and the proof size are independent of the number of the tuples [1]. The KVC scheme in [1] consists of the following algorithms $\mathsf{KVC} = (\mathsf{KGen}, \mathsf{KAppend}, \mathsf{KUpdate}, \mathsf{KOpen}, \mathsf{KVer})$:

- $\mathsf{KGen}(1^\lambda, l)$: It takes a security parameter λ and an integer $l \in \mathbb{N}$ as input, and chooses two $\lambda/2$-bit primes p_1 and p_2 at random, sets $N = p_1 p_2$,

[1] In this work, we simply consider that the keys are integers $\{1, 2, \dots\}$.

picks $g \in \mathbb{Z}_N^*$, determines a deterministic collision-resistant function $\mathsf{PrimeGen}$ that maps integers to $l+1$-bit primes, initials the commitment $C \leftarrow (1, g)$ and the auxiliary information $\mathsf{aux} \leftarrow \emptyset$. Finally, it outputs $(\mathsf{pp}, C, \mathsf{aux}) \leftarrow ((N, g, \mathsf{PrimeGen}), (1, g), \emptyset)$. The message space is $\mathcal{M} = \{0,1\}^l$.

- $\mathsf{KAppend}_{\mathsf{pp}}(C, i, m_i, \mathsf{aux})$: It takes $C = (C_1, C_2)$, a new message m_i, its position i and the auxiliary information aux, updates $C \leftarrow ({C_1}^{e_i} \cdot {C_2}^{m_i} \bmod N, {C_2}^{e_i} \bmod N)$ where $e_i \leftarrow \mathsf{PrimeGen}(i)$, and appends (i, m_i) into aux, i.e., $\mathsf{aux} \leftarrow \mathsf{aux} \cup \{(i, m_i)\}$. Finally, it outputs the updated C and aux.
- $\mathsf{KUpdate}_{\mathsf{pp}}(C, i, m_i, m_i', \mathsf{aux})$: It takes $C = (C_1, C_2)$, the old message m_i, a new message m_i', the position i and the auxiliary information aux, updates $C \leftarrow (C_1 \cdot \sqrt[e_i]{{C_2}^{m_i' - m_i}} \bmod N, C_2)$ where $e_i \leftarrow \mathsf{PrimeGen}(i)$, and replaces the i-th message m_i with m_i' in aux. Finally, it outputs the updated C and aux.
- $\mathsf{KOpen}_{\mathsf{pp}}(i, m_i, \mathsf{aux})$: It takes the position i and $\mathsf{aux} = (m_1, \ldots, m_n)$, computes $S_i \leftarrow g^{\prod_{j=1, j\neq i}^{n} e_j} \bmod N$ and $\Lambda_i \leftarrow \sqrt[e_i]{\prod_{j=1, j\neq i}^{n} S_j^{m_j}} \bmod N$, and finally outputs a proof $\pi_i \leftarrow (S_i, \Lambda_i)$ that m_i is the i-th committed message.
- $\mathsf{KVer}_{\mathsf{pp}}(C, i, m_i, \pi_i)$: It takes $C = (C_1, C_2)$, the message m_i, its proof $\pi_i = (S_i, \Lambda_i)$ and its position i, and checks if

$$ {S_i}^{e_i} = C_2 \mod N \quad \wedge \quad C_1 = {S_i}^{m_i} \cdot \Lambda_i^{e_i} \mod N $$

  where $e_i \leftarrow \mathsf{PrimeGen}(i)$. If true, it outputs 1, else outputs 0.

Batch Opening: Next, we show that the above KVC supports batch openings (also called subvector openings [3,4,12]).

- $\mathsf{KBatchOpen}_{\mathsf{pp}}(I, \vec{m}_I, \mathsf{aux})$: It takes an ordered position set $I = \{i_1, \ldots, i_{|I|}\} \subset [n]$ of the message vector $\vec{m}_I = (m_{i_1}, \ldots, m_{i_{|I|}})$ and the auxiliary information $\mathsf{aux} = (m_1, m_2, \ldots, m_n)$, computes $S_I \leftarrow g^{\prod_{j=1, j\notin I}^{n} e_j}$ and $\Lambda_I \leftarrow \sqrt[e_I]{\prod_{j=1, j\notin I}^{n} S_j^{m_j}} \bmod N$ where $e_I \leftarrow \prod_{i\in I} e_i$, and finally outputs a proof $\pi_I := (S_I, \Lambda_I)$ that $\vec{m}_I$ is the I-subvector of the committed message.
- $\mathsf{KBatchVer}_{\mathsf{pp}}(C, I, \vec{m}_I, \pi_I)$: It takes $C = (C_1, C_2)$, the message subvector $\vec{m}_I$, its proof $\pi_I = (S_I, \Lambda_I)$ and its position set I, and checks if

$$ {S_I}^{e_I} = C_2 \mod N \ \wedge \ C_1 = \prod_{i\in I} {S_i}^{m_i} \cdot \Lambda_I^{e_I} \mod N $$

  where $e_I \leftarrow \prod_{i\in I} e_i$ and $S_i \leftarrow {S_I}^{e_I \setminus \{i\}}$ for every $i \in I$. If true, it outputs 1, else outputs 0.

2.5 Verifiable Data Streaming

VDS [11,14] is a protocol between a client and a server, which consists of the following algorithms $\mathsf{VDS} = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Update})$.

- $\mathsf{Setup}(1^\lambda)$: It takes a security parameter λ as input and generates a key pair (pk, sk). It outputs the public verification key pk to the server and the secret key sk to the client.

- $\mathsf{Append}(sk, d)$: It takes the secret key sk and a data entry d as inputs. Then the client sends an append request to the server and the server stores this new data entry d in the database DB. Finally, it may output an updated secret key sk' to the client, but the public verification key does not change.
- $\mathsf{Query}(pk, \mathrm{DB}, i)$: It takes the public verification key pk, the database DB and a queried index i. Finally, it outputs the i-th data entry (i, d) along with a proof π_i to the client.
- $\mathsf{Verify}(pk, i, d, \pi_i)$: It takes the public verification key pk and the query response (i, d, π_i) as inputs. If d is the i-th data entry in DB according to π_i, it outputs d, otherwise it outputs $\perp$.
- $\mathsf{Update}(pk, sk, \mathrm{DB}, i, d')$: It runs between the server and the client. Finally, the server updates the i-th data entry d with a new data entry d' and the client updates the public verification key to pk' as well as the secret key to sk'.

Security. Informally, the security of VDS schemes ensures that an attacker should not be able to modify stored data entries, append further data entries to the database, and pass the verification with an old data. We describe the security of VDS by the following experiment $\mathsf{VDSsec}^{\mathsf{VDS}}_{\mathcal{A}}(\lambda)$.

Setup: The challenger runs $(sk, pk) \leftarrow \mathsf{Setup}(1^\lambda)$, sets up an empty database DB, and sends the public verification key pk to the adversary $\mathcal{A}$.

Challenge: When the adversary $\mathcal{A}$ appends a new data entry d, the challenger runs $(sk', i, \pi_i) \leftarrow \mathsf{Append}(sk, d)$ to append d to its database, and then returns (i, π_i) to the adversary. When the adversary $\mathcal{A}$ updates the i-th data entry giving a new data entry d', the challenger runs $\mathsf{Update}(pk, sk, \mathrm{DB}, i, d')$ with the adversary $\mathcal{A}$ and then returns (i, π_i) to the adversary. The challenger will always keep the latest public key pk^* and an ordered sequence of the database $Q = \{(1, d_1), \ldots, (q(\lambda), d_{q(\lambda)})\}$.

Guess: The adversary $\mathcal{A}$ outputs a guess (i^*, d^*, π^*), and the experiment outputs 1 if $d^* \leftarrow \mathsf{Verify}(pk^*, i^*, d^*, \pi^*)$, $d^* \neq \perp$ and $(i^*, d^*) \notin Q$.

Definition 2 (VDS Security). *A VDS scheme is secure if for all $\lambda \in \mathbb{N}$ and any PPT adversary $\mathcal{A}$, its advantage $\Pr[\mathsf{VDSsec}^{\mathsf{VDS}}_{\mathcal{A}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$ is negligible.*

3 Double-trapdoor Chameleon Vector Commitment

In this section, we first introduce a new cryptographic primitive, Double-trapdoor Chameleon Vector Commitment (DCVC). Then we present a DCVC construction based on RSA and a variant of it with cross-commitment aggregation.

3.1 Definition of DCVC

DCVC is an enhancement of Chameleon Vector Commitment (CVC) [11]. Both of them allow one to commit a vector $(m_1, \ldots, m_q)$ and to open the commitment at any position, and one holding trapdoors can find a collision without changing the commitment. In particular, CVC provides a single trapdoor and may suffer

from key exposure [2,7], while DCVC enjoys double trapdoors, master trapdoor and specific trapdoor, which may be key-exposure free. A DCVC scheme consists of the following algorithms:

- $\mathsf{DCGen}(1^\lambda, q)$: It takes a security parameter λ and the size of a vector q, then outputs a public parameter pp, a master trapdoor td_1 and a specific trapdoor td_2.
- $\mathsf{DCCom}_{\mathsf{pp}}(m_1, \ldots, m_q)$: It takes q ordered message vector $(m_1, \ldots, m_q)$, and outputs a commitment C and the auxiliary information aux.
- $\mathsf{DCOpen}_{\mathsf{pp}}(i, m, \mathsf{aux})$: It takes the index i, the corresponding message m, and aux, outputs a proof π that m is the i-th message in the committed vector.
- $\mathsf{DCVer}_{\mathsf{pp}}(C, i, m, \pi)$: It takes the commitment C, the i-th message m and the corresponding proof π, and outputs 1 iff π is a valid proof that C was generated for $(m_1, \ldots, m_q)$ s.t. $m_i = m$.
- $\mathsf{DCCol}_{\mathsf{pp}}(i, m, m', \mathsf{td}_1, \mathsf{td}_2, \mathsf{aux})$: It takes the trapdoors td_1 and td_2, the index i, a message m, another message m', and aux, then outputs an updated aux' after finding a collision s.t. (C, aux') is indistinguishable from the output of $\mathsf{CCom}_{\mathsf{pp}}(m_1, \ldots, m', \ldots, m_q)$.
- $\mathsf{DCUpdate}_{\mathsf{pp}}(C, i, m, m')$: It takes the old commitment C, the old message m, a new message m', and the corresponding index i, then outputs a new commitment C' and an update information U.
- $\mathsf{DCProofUpdate}_{\mathsf{pp}}(C, \pi_j, j, U)$: It takes the commitment C, the old proof π_j at the position j, and the update information U, then outputs an updated proof π'_j that is valid with regard to the new commitment C'.

Definition 3 (Concise). *A DCVC scheme is concise if the commitment size and the proof size are independent of the vector size q.*

Definition 4 (Correctness). *A DCVC scheme is correct if for all $\lambda \in \mathbb{N}$, any vector size q, a vector $(m_1, \ldots, m_q)$ and any index $i \in \{1, \ldots, q\}$, we have*

$$\Pr\left[\mathsf{DCVer}_{\mathsf{pp}}(C, i, m, \pi) = 1 : \begin{array}{r}(\mathsf{pp}, \mathsf{td}_1, \mathsf{td}_2) \leftarrow \mathsf{DCGen}(1^\lambda, q) \\ (C, \mathsf{aux}) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(m_1, \ldots, m_q) \\ \pi \leftarrow \mathsf{DCOpen}_{\mathsf{pp}}(i, m, \mathsf{aux})\end{array}\right] = 1.$$

Definition 5 (Position Binding). *A DCVC scheme is position-binding if for any PPT adversary $\mathcal{A}$, the probability generating two valid proofs for different messages (m, m') at the same position i is negligible. Formally, for all $\lambda \in \mathbb{N}$ and any PPT adversary $\mathcal{A}$, the advantage of $\mathcal{A}$ winning the below experiment $\Pr[\mathsf{PosBdg}_{\mathcal{A}}^{\mathsf{DCVC}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$ is negligible.*

Definition 6 (Indistinguishable Collisions). *A DCVC scheme has indistinguishable collisions if for all $\lambda \in \mathbb{N}$ and any stateful PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, its advantage of winning the below experiment $\Pr[\mathsf{Collnd}_{\mathcal{A}}^{\mathsf{DCVC}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$ is negligible.*

Experiment $\mathsf{PosBdg}^{\mathsf{DCVC}}_{\mathcal{A}}(\lambda)$
$(\mathsf{pp}, \mathsf{td}_1, \mathsf{td}_2) \leftarrow \mathsf{DCGen}(1^\lambda, q)$
$(C, i, m, m', \pi, \pi') \leftarrow \mathcal{A}^{\mathsf{DCCol}}(\mathsf{pp})$
store (C, i) queried to DCCol in Q
if $m \neq m' \wedge (C, i) \notin Q$
$\wedge\ \mathsf{DCVer}_{\mathsf{pp}}(C, i, m, \pi)$
$\wedge\ \mathsf{DCVer}_{\mathsf{pp}}(C, i, m', \pi')$
output 1
else output 0

Experiment $\mathsf{CollInd}^{\mathsf{DCVC}}_{\mathcal{A}}(\lambda)$
$(\mathsf{pp}, \mathsf{td}_1, \mathsf{td}_2) \leftarrow \mathsf{DCGen}(1^\lambda, q)$
$b \leftarrow \{0, 1\}$
$((m_1, \ldots, m_q), (i, m'_i)) \leftarrow \mathcal{A}_0(\mathsf{pp}, \mathsf{td}_1, \mathsf{td}_2)$
$(C_0, \mathsf{aux}^*) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(m_1, \ldots, m_i, \ldots, m_q)$
$\mathsf{aux}_0 \leftarrow \mathsf{DCCol}_{\mathsf{pp}}(C_0, i, m_i, m'_i, \mathsf{td}_1, \mathsf{td}_2, \mathsf{aux}^*)$
$(C_1, \mathsf{aux}_1) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(m_1, \ldots, m'_i, \ldots, m_q)$
$b' \leftarrow \mathcal{A}_1(C_b, \mathsf{aux}_b)$
if $b = b'$ output 1 else output 0

3.2 DCVC based on RSA

We present a DCVC scheme based on RSA, which exquisitely combines RSA-based vector commitment [5] with chameleon hash without key exposure [2,10]. Furthermore, we develop a variant with cross-commitment aggregation. The details of our scheme DCVC is described as follows:

- $\mathsf{DCGen}(1^\lambda, l, q)$: It takes a security parameter λ and two integer $l, q \in \mathbb{N}$ as inputs, chooses two $\lambda/2$-bit primes p_1 and p_2 at random, sets $N = p_1 p_2$, picks $g \in \mathbb{Z}_N^*$ randomly, determines a deterministic collision-resistant function $\mathsf{PrimeGen}$ that maps integers to primes with length $l+1$ bits. Then it computes q primes $e_1, \ldots, e_q$ that are relatively prime to $\phi(N) = (p_1 - 1)(p_2 - 1)$, where $e_i \leftarrow \mathsf{PrimeGen}(i)$ for $i = 1, \ldots, q$. Finally, it outputs the public parameter $\mathsf{pp} \leftarrow (N, g, \mathsf{PrimeGen})$, the master trapdoor $\mathsf{td}_1 \leftarrow \{p_1, p_2\}$, and the specific trapdoor $\mathsf{td}_2 \leftarrow \{d_i\}_{i=1,\ldots,n}$, where d_i is computed s.t. $e_i d_i = 1 \bmod \phi(N)$. The message space is $\mathcal{M} = \{0, 1\}^l$.
- $\mathsf{DCCom}_{\mathsf{pp}}(m_1, \ldots, m_q)$: It takes a message vector $(m_1, \ldots, m_q)$ as input, chooses $r \leftarrow \mathbb{Z}_N^*$ randomly, and computes $S_i \leftarrow g^{\prod_{j=1, j\neq i}^{q} e_j}$ for $i = 1, \ldots, q$. Finally, it outputs $C \leftarrow S_1^{m_1} \cdots S_q^{m_q} r^{\prod_{i=1}^{q} e_i} \bmod N$ and $\mathsf{aux} \leftarrow (m_1, \ldots, m_q; r)$.
- $\mathsf{DCOpen}_{\mathsf{pp}}(i, m, \mathsf{aux})$: It computes $S_j^{1/e_i} \leftarrow g^{e_{[q]\setminus\{i,j\}}}$ for each $j \in [q] \setminus \{i\}$, and outputs $\pi \leftarrow \sqrt[e_i]{\prod_{j=1, j\neq i}^{q} S_j^{m_j}} \cdot r^{\prod_{j=1, j\neq i}^{q} e_j} \bmod N$.
- $\mathsf{DCVer}_{\mathsf{pp}}(C, i, m, \pi)$: If $C = S_i^m \cdot \pi^{e_i} \bmod N$ output 1, else output 0.
- $\mathsf{DCCol}_{\mathsf{pp}}(C, i, m, m', \mathsf{td}_2, \mathsf{aux})$: It computes $r' \leftarrow r \cdot (g^{d_i})^{m - m'}$ and outputs $\mathsf{aux}' \leftarrow (m_1, \ldots, m', \ldots, m_q; r')$.
- $\mathsf{DCUpdate}_{\mathsf{pp}}(C, i, m, m')$: It computes $C' \leftarrow C \cdot S_i^{m' - m} \bmod N$, then outputs C' and $U = (i, m, m')$.
- $\mathsf{DCProofUpdate}_{\mathsf{pp}}(C, \pi_j, j, U)$: If $j \neq i$, it computes $\pi'_j \leftarrow \pi_j \cdot (S_i^{m' - m})^{1/e_j} \bmod N$, else $\pi'_j \leftarrow \pi_j$.

Cross-Commitment Aggregation. Now we show that our scheme DCVC is aggregatable across multiple commitments, which means that different openings from different commitments (e.g., π_{i,k_i} and π_{j,k_j} at the position k_i and k_j of the commitments C_i and C_j, respectively) can be merged into a single concise opening π. Moreover, this aggregated proof can be further aggregated, namely cross-commitment incremental aggregation. Assume that $\hat{\pi}$ is already an aggregated

proof of $l-1$ commitments $\{C_j, k_j, m_{j,k_j}, \pi_{j,k_j}\}_{j\in[l-1]}$. The cross-commitment aggregation and verification algorithms are shown as follows:

- $\mathsf{DCAggCross}_{\mathsf{pp}}(\{k_j\}_{j\in[l-1]}, \hat{\pi}, (C_l, k_l, m_{l,k_l}, \pi_{l,k_l}))$:
  Case 1: If $k_l \notin \{k_j\}_{j\in[l-1]}$, compute

  $$\rho_K \leftarrow \hat{\pi} \cdot (\pi_{l,k_l})^{t_l e_{k_l}/e_K} \bmod N \quad \text{and} \quad \rho_l \leftarrow \hat{\pi}^{e_K/e_{k_l}} \cdot (\pi_{l,k_l})^{t_l} \bmod N,$$

  and then generate an aggregated proof $\pi \leftarrow \mathsf{ShamirTrick}(\rho_K, \rho_l, e_K, e_{k_l})$ [16].
  Case 2: If $k_l \in \{k_j\}_{j\in[l-1]}$, compute $\pi \leftarrow \hat{\pi} \cdot {\pi_{l,k_l}}^{t_l e_{k_l}/e_K}$. Note that $t_l \leftarrow H(l, C_l, k_l, m_{l,k_l})$ and $e_K \leftarrow \prod_{j\in[l-1]} e_{k_j}$.
- $\mathsf{DCAggVer}_{\mathsf{pp}}(\{C_j, k_j, m_{j,k_j}\}_{j\in[l]}, \pi)$: If the following equation holds output 1, else output 0. Note that $t_j \leftarrow H(j, C_j, k_j, m_{j,k_j})$.

$$\prod_{j\in[l]} {C_j}^{t_j} = \prod_{j\in[l]} S_{k_j}^{t_j m_{j,k_j}} \pi^{\frac{\prod_{j\in[l]} e_{k_j}}{\gcd(e_{k_1},\ldots,e_{k_l})}} \bmod N.$$

Concise. It is obvious that the commitment size and the proof size are independent of the vector size q.

Correctness. The correctness of DCVC is straightforward and the correctness of cross-commitment aggregation comes from the correctness of DCVC and Shamir's trick [16]. More details are shown in Appendix A.

Security. Our scheme DCVC is secure. The proofs of position binding, indistinguishable collisions and key exposure freeness are detailed in Appendix B.

4 Verifiable Data Streaming from DCVC

In this section, we propose our first scheme VDS_1 with optimal query communication from our scheme DCVC, and show that VDS_1 is secure in the random model.

4.1 High-level Description

Our first scheme VDS_1 follows the same framework as Krupp et al. CVC-based VDS scheme [11] which combines a q-ary tree with the cryptographic primitive CVC. However, CVC-based VDS scheme [11] requires the client to generate all proofs in advance and store them at the server, which leads to a heavy computational burden on the client and a large storage overhead on the server respectively. To reduce the client computation and server storage, our intuition is to transfer the proof generation from the client to the server. Unfortunately, this approach from our intuition may suffer from an inherent key exposure problem due to the CVC construction [2]. In addition, the query communication for a single query and a batch query is $2\log_q n - 1$ and $l \cdot (2\log_q n - 1)$ respectively, where q is the maximum number of q-ary tree children nodes, n is the size of the database, and l is the number of queried data in a batch query. To optimize

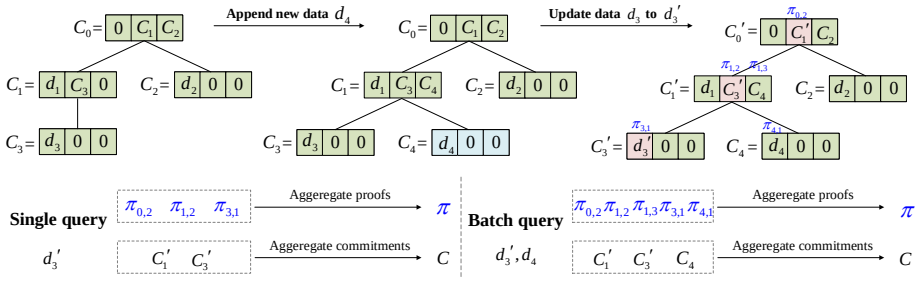


Fig. 1. Overview of VDS_1

query communication, a perfect solution is aggregating all CVC proofs on the authentication path to a constant-size value. However, this does not work because CVC does not support cross-commitment aggregation. Thus, our main task is to design a secure VDS scheme with optimal query communication and better client computation and server storage efficiency.

To this end, we combine a q-ary tree and an aggregatable cross-commitment variant of our proposed DCVC without key exposure. Concretely, we first build a q-ary tree, as shown in Fig. 1, where every node is a DCVC of a $q+1$-size vector. The first element of each vector is the data entry, and the q remaining elements are the node's q children (or 0 when the children do not exist). The root node of the tree is initialized to a DCVC of a zero vector and a new node is inserted into the tree by finding a collision in its parent node. Note that the nodes are appended to the tree from left to right and the tree grows from top to bottom. When appending a data entry, the client appends a node corresponding to this data entry into the tree. When querying a data entry, the server generates proofs of data along the authentication path without requiring the client to generate proofs in advance. Furthermore, we aggregate proofs on the authentication path into a constant-size value by using the feature of cross-commitment aggregation of DCVC. Thus we get an unbounded VDS scheme in the random model with optimal query communication $O(1)$ and better server storage $O(n)$.

4.2 Our Construction

Our scheme VDS_1 consists of five algorithms $\mathsf{VDS}_1 = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Update})$, which is based on our scheme DCVC and a q-ary tree. For sake of readability, we briefly describe the construction of the q-ary tree. In a q-ary tree, every node is a DCVC of a $q+1$-size vector. The first element of each vector is the data entry, and the q remaining elements are q children nodes (or 0 when children nodes do not exist). Particularly, the root node of the tree is initialized to a DCVC of a zero vector and a new node is appended into the tree by finding a collision in its parent node. Note that the nodes are appended to the tree from left to right and the tree grows from top to bottom. According to the structure of q-ary tree, we have three functions as following:

Algorithm 1 VDS from DCVC (VDS_1)

$\underline{\mathsf{Setup}(1^\lambda, l, q)}$ ▷ *Client*

$(\mathsf{pp}, \mathsf{td}_1, \mathsf{td}_2) \leftarrow \mathsf{DCGen}(1^\lambda, l, q+1)$
$cnt \leftarrow 0,\ k \leftarrow \{0,1\}^\lambda$
$r_0 \leftarrow f(k, 0)$
$(C_0, \mathsf{aux}_0) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(0, \ldots, 0; r_0)$
▷ $\mathsf{aux}_0 = (0, 0, \ldots, 0; r_0)$
$sk \leftarrow (\mathsf{td}_1, \mathsf{td}_2, cnt, k)$
$pk \leftarrow (\mathsf{pp}, C_0)$
return (sk, pk)

$\underline{\mathsf{Append}(pk, sk, d)}$ ▷ *Client*

$i \leftarrow cnt + 1,\ p \leftarrow \mathsf{parent}(i),\ j \leftarrow \#\mathsf{child}(i)$
$cnt \leftarrow cnt + 1$
$r_i \leftarrow f(k, i)$
$(C_i, \mathsf{aux}_i) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(0, \ldots, 0; r_i)$
▷ $\mathsf{aux}_i = (0, 0, \ldots, 0; r_i)$
$\mathsf{aux}'_i \leftarrow \mathsf{DCCol}_{\mathsf{pp}}(C_i, 1, 0, d, \mathsf{td}_1, \mathsf{td}_2, \mathsf{aux}_i)$
▷ $\mathsf{aux}'_i = (d, 0, \ldots, 0; r'_i)$
$r_p \leftarrow f(k, p)$
$(C_p, \mathsf{aux}_p) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(0, \ldots, 0; r_p)$
▷ $\mathsf{aux}_p = (0, \ldots, 0, \ldots, 0; r_p)$
$\mathsf{aux}'_p \leftarrow \mathsf{DCCol}_{\mathsf{pp}}(C_p, j, 0, C_i, \mathsf{td}_1, \mathsf{td}_2, \mathsf{aux}_p)$
▷ $\mathsf{aux}'_p = (0, \ldots, C_i, \ldots, 0; r'_p)$
return (i, d, C_i, r'_i, r'_p)

$\underline{\mathsf{Query}(pk, \mathrm{DB}, i)}$ ▷ *Server*

$\mathsf{Pos} \leftarrow \emptyset, C \leftarrow 1$
$L \leftarrow \mathsf{level}(i)$
$\pi_{i,1} \leftarrow \mathsf{DCOpen}_{\mathsf{pp}}(1, d, \mathsf{aux}_i)$
$\pi \leftarrow \pi_{i,1}$
$\mathsf{Pos} \leftarrow \mathsf{Pos} \cup \{1\}$
$a \leftarrow i$
$b \leftarrow \mathsf{parent}(i)$
for $h \in [L-1, 0]$ **do**
  $c \leftarrow \#\mathsf{child}(a)$
  $\pi_{b,c} \leftarrow \mathsf{DCOpen}_{\mathsf{pp}}(c, C_a, \mathsf{aux}_b)$
  $\pi \leftarrow \mathsf{DCAggCross}_{\mathsf{pp}}(\mathsf{Pos}, \pi, (C_b, c, C_a, \pi_{b,c}))$
  $C \leftarrow C \cdot {C_a}^{t_a}$ ▷ $t_a \leftarrow H(a)$
  $\mathsf{Pos} \leftarrow \mathsf{Pos} \cup \{c\}$
  $a \leftarrow b$
  $b \leftarrow \mathsf{parent}(b)$
end for
return (i, d, π, C)

$\underline{\mathsf{Verify}(pk, i, d, \pi, C)}$ ▷ *Client*

$e \leftarrow e_1, S \leftarrow 1$ ▷ $e_1 \leftarrow \mathsf{PrimeGen}(1)$
$L \leftarrow \mathsf{level}(i)$
$S \leftarrow S \cdot {S_1}^{t_i d_i}$ ▷ $S_1 \leftarrow g^{\prod_{j=1, j\neq 1}^{q+1} e_j}$
$a \leftarrow i$
$b \leftarrow \mathsf{parent}(i)$
for $h \in [L-1, 0]$ **do**
  $c \leftarrow \#\mathsf{child}(a)$
  $S \leftarrow S \cdot {S_c}^{t_b C_a}$ ▷ $S_c \leftarrow g^{\prod_{j=1, j\neq c}^{q+1} e_j}$
  if $\gcd(e, e_c) = 1$ **then**
    $e \leftarrow e \cdot e_c$ ▷ $e_c \leftarrow \mathsf{PrimeGen}(c)$
  end if
  $a \leftarrow b$
  $b \leftarrow \mathsf{parent}(b)$
end for
if $C \cdot {C_0}^{t_0} = S \cdot \pi^e$ **then return** d
else return $\perp$
end if

$\underline{\mathsf{Update}(pk, \mathrm{DB}, i, d')}$ ▷ *Client & Server*

Client:
send (i, d') to server

Server:
$(i, d, \tilde{\pi}) \leftarrow \mathsf{Query}(pk, \mathrm{DB}, i)$

Client:
$d/\perp \leftarrow \mathsf{Verify}(pk, i, d, \tilde{\pi})$

Client & Server:
if $d \leftarrow \mathsf{Verify}(pk, i, d, \tilde{\pi})$ **then**
  $(C'_i, U_i) \leftarrow \mathsf{DCUpdate}_{\mathsf{pp}}(C_i, 1, d, d')$
  $a \leftarrow i$
  $b \leftarrow \mathsf{parent}(i)$
  for $h \in [L-1, 0]$ **do**
    $c \leftarrow \#\mathsf{child}(a)$
    $(C'_b, U_b) \leftarrow \mathsf{DCUpdate}_{\mathsf{pp}}(C_b, c, C_a, C'_a)$
    $a \leftarrow b$
    $b \leftarrow \mathsf{parent}(b)$
  end for
end if
return $(C'_L, \ldots, C'_0)$

- $\mathsf{parent}(i) = \lfloor \frac{i-1}{q} \rfloor$ is the index of the parent of the node i.
- $\#\mathsf{child}(i) = ((i-1) \mod q) + 2$ is the position that the node i is inserted into its parent.
- $\mathsf{level}(i) = \lceil \log_q((q-1)(i+1)+1) - 1 \rceil$ is the level that the node i is appended in the tree.

Next, we give a brief description of our scheme VDS_1, the details of which are shown in Algorithm 1.

- $\mathsf{Setup}(1^\lambda, l, q)$: The client generates $(\mathsf{pp}, \mathsf{td}_1, \mathsf{td}_2) \leftarrow \mathsf{DCGen}(1^\lambda, l, q+1)$, initializes a counter $cnt \leftarrow 0$, and picks a random key $k \leftarrow \{0,1\}^\lambda$ for a secure pseudorandom function f. Then the client computes $r_0 \leftarrow f(k, 0)$ and the

Algorithm 2 Batch Query and Verify

BatchQuery(pk, DB, I) ▷ *Server*

$\mathsf{Pos} \leftarrow \emptyset, C \leftarrow 1$
for $i \in I$ **do**
  $L_i \leftarrow \mathsf{level}(i)$
  $\pi_{i,1} \leftarrow \mathsf{DCOpen}_{\mathsf{pp}}(1, d_i, aux_i)$
  if $\mathsf{Pos} = \emptyset$ **then**
    $\pi \leftarrow \pi_{i,1}$
  else
    $\pi \leftarrow \mathsf{DCAggCross}_{\mathsf{pp}}(\mathsf{Pos}, \pi, (C_i, 1, d_i, \pi_{i,1}))$
  end if
  $\mathsf{Pos} \leftarrow \mathsf{Pos} \cup \{1\}$
  $a \leftarrow i$
  $b \leftarrow \mathsf{parent}(i)$
  for $h \in [L_i - 1, 0]$ **do**
    $c \leftarrow \#\mathsf{child}(a)$
    $\pi_{b,c} \leftarrow \mathsf{DCOpen}_{\mathsf{pp}}(c, C_a, aux_b)$
    $\pi \leftarrow \mathsf{DCAggCross}_{\mathsf{pp}}(\mathsf{Pos}, \pi, (C_b, c, C_a, \pi_{b,c}))$
    $C \leftarrow C \cdot {C_a}^{t_a}$ ▷ $t_a \leftarrow H(a)$
    $\mathsf{Pos} \leftarrow \mathsf{Pos} \cup \{c\}$
    $a \leftarrow b$
    $b \leftarrow \mathsf{parent}(b)$
  end for
end for
return $(I, \vec{d}_I, \pi, C)$

BatchVerify$(pk, I, \vec{d}_I, \pi, C)$ ▷ *Client*

$e \leftarrow e_1, S \leftarrow 1$ ▷ $e_1 \leftarrow \mathsf{PrimeGen}(1)$
for $i \in I$ **do**
  $L_i \leftarrow \mathsf{level}(i)$
  $S \leftarrow S \cdot {S_1}^{t_i d_i}$ ▷ $S_1 \leftarrow g^{\prod_{j=1, j\neq 1}^{q+1} e_j}$
  $a \leftarrow i$
  $b \leftarrow \mathsf{parent}(i)$
  for $h \in [L_i, 0]$ **do**
    $c \leftarrow \#\mathsf{child}(a)$
    $S \leftarrow S \cdot {S_c}^{t_b C_a}$ ▷ $S_c \leftarrow g^{\prod_{j=1, j\neq c}^{q+1} e_j}$
    if $\gcd(e, e_c) = 1$ **then**
      $e \leftarrow e \cdot e_c$ ▷ $e_c \leftarrow \mathsf{PrimeGen}(c)$
    end if
    $a \leftarrow b$
    $b \leftarrow \mathsf{parent}(b)$
  end for
end for
if $C \cdot ({C_0}^{t_0})^{|I|} = S \cdot \pi^e$ **then return** $\vec{d}_I$
else return $\perp$
end if

tree root $(C_0, \mathsf{aux}_0) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(0, \ldots, 0; r_0)$. Finally, it outputs the secret key $sk \leftarrow (\mathsf{td}_1, \mathsf{td}_2, cnt, k)$ and public key $pk \leftarrow (\mathsf{pp}, C_0)$.

- $\mathsf{Append}(pk, sk, d)$: When appending a new data entry d, the client first parses $sk = (\mathsf{td}_1, \mathsf{td}_2, cnt, k)$ and obtains the index of the new data entry $i \leftarrow cnt + 1$, the index of its parent node $p \leftarrow \mathsf{parent}(i)$, the position $j \leftarrow \#\mathsf{child}(i)$ that this data entry will be inserted into its parent, and $cnt \leftarrow cnt + 1$. Next, the client computes a new node $(C_i, \mathsf{aux}_i) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(0, \ldots, 0; r_i)$ where $r_i \leftarrow f(k, i)$. To insert a new data entry d into the new node C_i, the client finds a collision by computing $\mathsf{aux}'_i \leftarrow \mathsf{DCCol}_{\mathsf{pp}}(C_i, 1, 0, d, \mathsf{td}_1, \mathsf{td}_2, \mathsf{aux}_i)$. To append the new node C_i in the tree, the client recomputes the parent node $(C_p, \mathsf{aux}_p) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}(0, \ldots, 0; r_p)$ where $r_p \leftarrow f(k, p)$ and runs $\mathsf{aux}'_p \leftarrow \mathsf{DCCol}_{\mathsf{pp}}(C_p, j, 0, C_i, \mathsf{td}_1, \mathsf{td}_2, \mathsf{aux}_p)$ to insert C_i as the j-th element of C_p. Finally, the client sends (i, d, C_i, r'_i, r'_p) to the server.
- $\mathsf{Query}(pk, \mathrm{DB}, i)$: When querying the i-th data entry, the server first obtains the level $L \leftarrow \mathsf{level}(i)$ of the node i, and then generates an aggregated proof (π, C) along the authentication path by running algorithm (see Algorithm 1 for details), finally sends the data and its proof (i, d, π, C) to the client.
- $\mathsf{Verify}(pk, i, d, \pi, C)$: The verifier (including the client) parses $pk = (\mathsf{pp}, C_0)$, verifies (i, d, π, C) (see Algorithm 1). If $v = 1$ output d, otherwise output $\perp$.
- $\mathsf{Update}(pk, \mathrm{DB}, i, d')$: The client and the server perform the update protocol.

  1. To update the data entry d to d' with the index i, the client first retrieves the data entry d. Concretely, the client sends the index i and a new data entry d' to the server. Then the server and the client run $\mathsf{Query}(pk, \mathrm{DB}, i) \rightarrow (i, d, \pi, C)$ and $\mathsf{Verify}(pk, i, d, \pi, C) \rightarrow d/\perp$ respectively.

2. When the query is validated, the client determines the level $L \leftarrow \mathsf{level}(i)$ of the updated node and then computes a new root C_0' as shown in Algorithm 1. Finally, it updates the public key $pk = (\mathsf{pp}, C_0')$.
3. The server writes the new data entry d' into DB and runs the same algorithm to update all commitments $(C_L', \ldots, C_0')$ along the path.

Batch Query: Now, we show that VDS_1 supports batch query and verifying.

- $\mathsf{BatchQuery}(pk, \mathrm{DB}, I)$: When performing query on the index set $I = \{i_1, \ldots, i_l\}$ sent from the client, the server obtains the level $L_i \leftarrow \mathsf{level}(i)$ of the node $i \in I$ and then generates an aggregated proof by running Algorithm 2. Finally, the server sends the data and its proof $(I, \vec{d}_I, \pi, C)$ to the client.
- $\mathsf{BatchVerify}(pk, I, \vec{d}_I, \pi, C)$: The verifier (including the client) parses $pk = (\mathsf{pp}, C_0)$ and verifies the proof as Algorithm 2. If the equation holds then output $\vec{d}_I$, otherwise output $\perp$.

4.3 Security Analysis

Theorem 1 (Secure VDS). *If f is a pseudorandom function and* DCVC *is position binding, then our scheme* VDS_1 *is secure.*

Proof. The proof of the theorem proceeds through hybrid games [11]. It starts with the real game $\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_1}(\lambda)$ and ends with a hybrid game where the pseudorandom function f is replaced by a random function, then these two games are computationally indistinguishable.

Game G_0: This is the real VDS security game $\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_1}(\lambda)$, so we have $\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_1}(\lambda) = 1] = \Pr[\mathsf{G}_0 = 1]$.

Game G_1: This game is identical to G_0 except the pseudorandom function f is replaced with a random function. By assuming f is pseudorandom, we immediately get that $\Pr[\mathsf{G}_0 = 1] - \Pr[\mathsf{G}_1 = 1] = \mathsf{negl}(\lambda)$.

In this game, we proceed by the contradiction. Assume there exists an adversary $\mathcal{A}$ that can win with non-negligible advantage in the game G_1. Then we can construct an efficient reduction $\mathcal{B}$ that uses $\mathcal{A}$ to break the security of DCVC. The reduction $\mathcal{B}$ proceeds in two case.

Let $(i^*, d^*, \hat{\pi}, \hat{C})$ be the tuple returned by the adversary at end of the game. If the game G_1 outputs 1, then it must hold that $\mathsf{Verify}(pk, i^*, d^*, \hat{\pi}, \hat{C}) = d^*$, $d^* \neq \perp$ and $d^* \neq d$, where d is the value with index i^* currently stored in the database. The honest authentication path of (i^*, d) computed by $\mathcal{B}$ is $(\tilde{\pi}, \tilde{C})$. Observe that both authentication paths $(\hat{\pi}, \hat{C})$ and $(\tilde{\pi}, \tilde{C})$ must end up at the public root and they must deviate at some node in the path from i^* up to the root. Then, we define the event Diffdata that the two authentication path deviate exactly at i^*, which means $C_{i^*} = C_{i^*}^*$. Obviously, $\overline{\mathsf{Diffdata}}$ means that the adversary may return a valid authentication path that deviates from the correct path at the internal node. Thus, we have

$$\Pr[\mathsf{G}_1 = 1] = \Pr[\mathsf{G}_1 = 1 \wedge \mathsf{Diffdata}] + \Pr[\mathsf{G}_1 = 1 \wedge \overline{\mathsf{Diffdata}}].$$

We show that $\Pr[\mathsf{G}_1 = 1 \wedge \mathsf{Diffdata}]$ and $\Pr[\mathsf{G}_1 = 1 \wedge \overline{\mathsf{Diffdata}}]$ are negligible in two cases Diffdata and $\overline{\mathsf{Diffdata}}$ respectively if our scheme DCVC is position binding.

Case $\mathsf{Diffdata}$: In this case, the authentication path $(\hat{\pi}, \hat{C})$ returned by the adversary deviates from the correct authentication path $(\tilde{\pi}, \tilde{C})$ at i^*, which means $C_{i^*} = C^*_{i^*}$.

The reduction $\mathcal{B}$ takes as input pp, computes the root $(C_0, \mathsf{aux}_0) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}$ $(0, \ldots, 0; r_0)$ by sampling a randomness, and sets the counter $cnt \leftarrow 0$. Then, it sets $pk \leftarrow (\mathsf{pp}, C_0)$ and runs $\mathcal{A}(pk)$ by simulating the game G_1.

To answer the append queries of $\mathcal{A}$, e.g., appending a data entry d, $\mathcal{B}$ runs $\mathsf{Append}(pk, sk, d)$ algorithm except that pseudorandom values are replaced by random values by sampling. Note that $\mathcal{B}$ does not know the trapdoors of DCVC, but it can directly compute the new node $(C_i, \mathsf{aux}_i) \leftarrow \mathsf{DCCom}_{\mathsf{pp}}$ $(d, \ldots, 0; r_i)$ for the data entry d instead of performing collision at the first position of the vector and use its collision oracle to append new nodes into the tree only when necessary. Note that $\mathcal{B}$ never uses its collision oracle at the position 1 in *Case* $\mathsf{Diffdata}$.

To answer the update queries of $\mathcal{A}$, e.g., updating the data entry d at position i to a new entry d', $\mathcal{B}$ simply runs Update algorithm. Note that this not require the trapdoors of DCVC.

The adversary outputs the tuple $(i^*, d^*, \hat{\pi}, \hat{C})$ at the end of the game. $\mathcal{B}$ computes the honest proof $(\tilde{\pi}, \tilde{C})$ for the data entry d at the position i^*, parses $(\hat{\pi}, \hat{C}) = (\pi^*_{i^*}, C^*_{i^*}, \ldots)$ and $(\tilde{\pi}, \tilde{C}) = (\pi_{i^*}, C_{i^*}, \ldots)$, and outputs $(C_{i^*}, 1, d, d^*, \pi_{i^*}, \pi^*_{i^*})$. We know that $C_{i^*} = C^*_{i^*}$ when $\mathsf{Diffdata}$ happens, and $\pi^*_{i^*}$ must pass the verification correctly for d^* since $\mathcal{A}$ wins in this game. Therefore, one can see that the tuple $(C_{i^*}, 1, d, d^*, \pi_{i^*}, \pi^*_{i^*})$ breaks the position binding of DCVC. Thus,

$$\Pr[\mathsf{G}_1 = 1 \wedge \mathsf{Diffdata}] \leq \Pr[\mathsf{PosBdg}^{\mathsf{DCVC}}_{\mathcal{B}}(\lambda) = 1] = \mathsf{negl}(\lambda).$$

Case $\overline{\mathsf{Diffdata}}$: In this case, the authentication path $(\hat{\pi}, \hat{C})$ returned by the adversary deviates from the correct authentication path $(\tilde{\pi}, \tilde{C})$ at the internal node.

The reduction $\mathcal{B}$ takes as input pp. It then chooses the tree depth $l = \lambda$ to set an upper limit on the number of data entry and builds an l size DCVC tree from bottom to up, where in each DCVC every position which does not point to a child (especially the first position in the internal node or all the positions in the leaf node) is set to 0. Let C_0 be the root. Then, $\mathcal{B}$ sets the counter $cnt \leftarrow 0$, sets $pk \leftarrow (\mathsf{pp}, C_0)$, and runs $\mathcal{A}(pk)$ by simulating the game G_1.

To answer the append queries of $\mathcal{A}$, e.g., appending a data entry d, $\mathcal{B}$ obtains the index $i \leftarrow cnt + 1$ for the new data entry, sets $cnt \leftarrow cnt + 1$, and inserts the new data entry into the tree by finding a collision in the position 1 of node n_i using collision oracle. Note that $\mathcal{B}$ never uses its collision oracle at the position $j > 1$ in *Case* $\overline{\mathsf{Diffdata}}$. If the adversary $\mathcal{A}$ exceeds the upper limit of the number of data entries, $\mathcal{B}$ stops the adversary and starts again by setting $l \leftarrow l \cdot \lambda$.

To answer the update queries of $\mathcal{A}$, e.g., updating the data entry d at position i to a new entry d', $\mathcal{B}$ simply runs Update algorithm.

At the end of the game the adversary returns the tuple $(i^*, d^*, \hat{\pi}, \hat{C})$. $\mathcal{B}$ parses $(\hat{\pi}, \hat{C}) = (\ldots, \pi^*_1, C^*_1, \pi^*_0, C^*_0)$ and finds the largest k such that $C^*_k = C_k$, that is, the authentication path $\hat{\pi}$ is still equal to the actual tree from C^*_k to the root. $\mathcal{B}$ computes the honest authentication path $(\tilde{\pi}, \tilde{C}) = (\ldots, \pi_1, C_1, \pi_0, C_0)$ from i^*

to the root if i^* is in the tree (i.e., i^* does not exceed the upper limit), otherwise computes one from the deepest ancestor of i^* to the root.

If C_k is the deepest node in the honest path $(\tilde{\pi}, \tilde{C})$ computed by $\mathcal{B}$, the j-th element committed in C_k by $\mathcal{B}$ is 0. Thus, $\mathcal{B}$ can compute an honest proof π_k that 0 is the j-th element committed in C_k, and output $(C_k, j, 0, C^*_{k+1}, \pi_k, \pi^*_k)$.

If C_k is not the deepest node in $\tilde{\pi}$, there exists a node $C^*_i = C_k$ and a proof π_k that C_{k+1} is the j-th element committed in C_k. Thus, $\mathcal{B}$ outputs $(C_k, j, C_{k+1}, C^*_{k+1}, \pi_k, \pi^*_k)$.

Therefore, the tuple $(C_k, j, 0, C^*_{k+1}, \pi_k, \pi^*_k)$ or $(C_k, j, C_{k+1}, C^*_{k+1}, \pi_k, \pi^*_k)$ breaks the position binding of DCVC. Thus,

$$\Pr[\mathsf{G}_1 = 1 \wedge \overline{\mathsf{Diffdata}}] \leq \Pr[\mathsf{PosBdg}^{\mathsf{DCVC}}_{\mathcal{B}}(\lambda) = 1] = \mathsf{negl}(\lambda).$$

In conclusion, the overall advantage of the adversary winning the game is negligible because it is negligible in both cases. Thus, our scheme VDS_1 is secure.

5 Verifiable Data Streaming from KVC

In this section, we propose our second scheme VDS_2 with optimal query communication and server storage overhead in the standard model from KVC and DCH. In the following, we will describe this scheme in detail.

5.1 High-level Description

Our second scheme VDS_2 explores a new approach to construct the unbounded VDS scheme with optimal query communication and server storage overhead in the standard model. Our main idea is to use KVC to guarantee the verifiability of the database in the standard model, and use DCH to make the public verification key unchanged when data is appended continuously. Concretely, as shown in Fig.2, the client first initialize a key-value commitment C for an empty database and a DCH hash value Ch of a pair $(0, \hat{r})$, and make this hash value Ch serve as the public verification key. When appending a new data entry d_i, the client appends this data entry to the key-value commitment by updating the commitment C and finding a collision (C, r_i) s.t. $\mathsf{DCh}(0, \hat{r}) = \mathsf{DCh}(C, r_i)$. Naturally, when updating a data entry d_i to d'_i, the client updates the commitment and the public verification key. For single query (or batch query), the server can generate a constant-size proof that d_i (or d_I) is the i-th (or I-subvector) committed data entry in the database.

5.2 Our Construction

In the following, we give a brief description of our scheme $\mathsf{VDS}_2 = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Update})$, the details of which are shown in Algorithm 3.

- $\mathsf{Setup}(1^\lambda, l)$: The client generates $(\mathsf{pp}_1, C) \leftarrow \mathsf{KGen}(1^\lambda, l)$ and $(\mathsf{pp}_2, Y, x) \leftarrow \mathsf{DCHKGen}(1^\lambda)$, sets the counter $cnt \leftarrow 0$. Then the client generates the one-time hash/trapdoor $(\hat{K}, \hat{k}) \leftarrow \mathsf{DCHTGen}(\mathsf{pp}_2)$, picks a random number $\hat{r} \leftarrow$

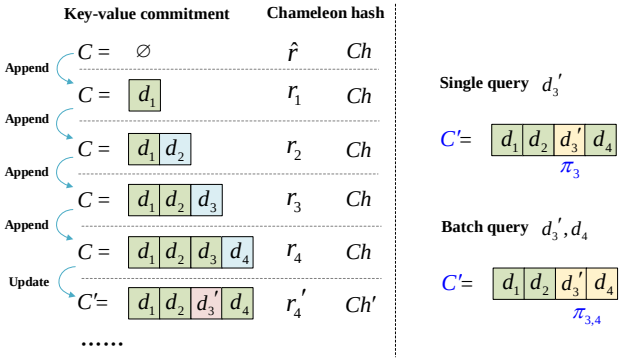


Fig. 2. Overview of VDS_2

$\{0,1\}^\lambda$, and computes the hash value $Ch \leftarrow \mathsf{DCH}_{\mathsf{pp}_2}(Y, \hat{K}, 0, \hat{r})$. Finally, the algorithm outputs the secret key $sk \leftarrow (cnt, x, \hat{r}, C)$ and the public key $pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch)$.

- $\mathsf{Append}(sk, d)$: When appending a new data entry d, the client first parses $sk = (cnt, x, \hat{r}, C)$, obtains the index $i \leftarrow cnt + 1$ of the new data entry d, updates the commitment $C \leftarrow \mathsf{KAppend}_{\mathsf{pp}_1}(C, i, d)$, determines a one-time hash/trapdoor key pair $(K, k) \leftarrow \mathsf{DCHTGen}(\mathsf{pp}_2)$, finds a collision (C, r) s.t. $\mathsf{DCh}(C, r) = \mathsf{DCh}(0, \hat{r})$ by running $r \leftarrow \mathsf{DCHCol}_{\mathsf{pp}_2}(x, k, 0, \hat{r}, C)$, and then increases the counter $cnt \leftarrow cnt + 1$. Finally, the client sends (i, d, C, K, r) to the server and the server stores (i, d) in DB as well as updates (C, K, r).
- $\mathsf{Query}(pk, \mathrm{DB}, i)$: When performing query on the index i sent from the client, the server computes a proof $\pi_i \leftarrow \mathsf{KOpen}_{\mathsf{pp}_1}(i, d, \mathrm{DB})$ that d is i-th data entry in DB, and sends the data and its proofs (i, d, π_i) and (C, K, r) to the client.
- $\mathsf{Verify}(pk, i, d, \pi_i, C, K, r)$: The verifier (including the client) parses $pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch)$, and then verifies the correctness of (C, K, r) and (i, d, π_i) by checking $Ch \stackrel{?}{=} \mathsf{DCH}_{\mathsf{pp}_2}(Y,K,C,r)$ and running the algorithm $\mathsf{KVer}_{\mathsf{pp}_1}(C,i,d,\pi_i)$. If both are true, then output d, otherwise output $\perp$.
- $\mathsf{Update}(pk, sk, \mathrm{DB}, i, d')$: The update protocol is run by the client and server.
  1. To update the data entry d to d' with the index i, the client first retrieves d with the index i from the server. Concretely, the client sends the index i and the new data entry d' to the server. Then the server and the client runs $\mathsf{Query}(pk, \mathrm{DB}, i) \rightarrow (i, d, \pi_i)$ and $\mathsf{Verify}(pk, i, d, \pi_i, C, K, r) \rightarrow d/\perp$ respectively.
  2. When the query is validated, the client first updates the commitment C' by replacing d to d'. Then the client choose a randomness $\hat{r}' \leftarrow \{0,1\}^\lambda$, produces the new hash value Ch', generates a one-time hash/trapdoor pair (K', k'), and finds a collision r' for the updated commitment C'. Finally, the client updates the secret and public keys $sk \leftarrow (cnt, x, \hat{r}', C')$ and $pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch')$ and sends (C', K', r') to the server.
  3. The server writes the new data d' into DB and updates (C, K, r) to (C', K', r').

Batch Query: We show that VDS_2 supports batch query and verifying.

Algorithm 3 VDS from KVC (VDS_2)

$\underline{\mathsf{Setup}(1^\lambda, l)}$ ▷ *Client*

$(\mathsf{pp}_1, C) \leftarrow \mathsf{KGen}(1^\lambda, l)$
$(\mathsf{pp}_2, Y, x) \leftarrow \mathsf{DCHKGen}(1^\lambda)$
$cnt \leftarrow 0$
$(\hat{K}, \hat{k}) \leftarrow \mathsf{DCHTGen}(\mathsf{pp}_2)$
$\hat{r} \leftarrow \{0,1\}^\lambda$
$Ch \leftarrow \mathsf{DCH}_{\mathsf{pp}_2}(Y, \hat{K}, 0, \hat{r})$
$sk \leftarrow (cnt, x, \hat{r}, C)$
$pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch)$
return (sk, pk)

$\underline{\mathsf{Append}(sk, d)}$ ▷ *Client*

$i \leftarrow cnt + 1$
$C \leftarrow \mathsf{KAppend}_{\mathsf{pp}_1}(C, i, d)$
$(K, k) \leftarrow \mathsf{DCHTGen}(\mathsf{pp}_2)$
$r \leftarrow \mathsf{DCHCol}_{\mathsf{pp}_2}(x, k, 0, \hat{r}, C)$
$cnt \leftarrow cnt + 1$
return (i, d, C, K, r)

$\underline{\mathsf{Query}(pk, \mathrm{DB}, i)}$ ▷ *Server*

$\pi_i \leftarrow \mathsf{KOpen}_{\mathsf{pp}_1}(i, d, \mathrm{DB})$
return $((i, d, \pi_i), (C, K, r))$

$\underline{\mathsf{Verify}(pk, i, d, \pi_i, C, K, r)}$ ▷ *Client*

$v \leftarrow Ch \stackrel{?}{=} \mathsf{DCH}_{\mathsf{pp}_2}(Y, K, C, r)$
$\quad \wedge\ \mathsf{KVer}_{\mathsf{pp}_1}(C, i, d, \pi_i)$
if $v = 0$ **then return** $\perp$
else return d
end if

$\underline{\mathsf{Update}(pk, sk, \mathrm{DB}, i, d')}$ ▷ *Client & Server*

Client:
send (i, d') to server
Server:
$(i, d, \pi_i) \leftarrow \mathsf{Query}(pk, \mathrm{DB}, i)$
Client:
$v \leftarrow \mathsf{Verify}(pk, i, d, \pi_i, C, K, r)$
Client:
if $v = 0$ **then return** $\perp$
else
  $C' \leftarrow \mathsf{KUpdate}_{\mathsf{pp}_1}(C, i, d, d')$
  $(\hat{K}', \hat{k}') \leftarrow \mathsf{DCHTGen}(\mathsf{pp}_2)$
  $\hat{r}' \leftarrow \{0,1\}^\lambda$
  $Ch' \leftarrow \mathsf{DCH}_{\mathsf{pp}_2}(Y, \hat{K}', 0, \hat{r}')$
  $(K', k') \leftarrow \mathsf{DCHTGen}(\mathsf{pp}_2)$
  $r' \leftarrow \mathsf{DCHCol}_{\mathsf{pp}_2}(x, k', 0, \hat{r}', C')$
  return $(C', \hat{r}', Ch', K', r')$
end if

$\underline{\mathsf{BatchQuery}(pk, \mathrm{DB}, I)}$ ▷ *Server*

$\pi_I \leftarrow \mathsf{KBatchOpen}_{\mathsf{pp}_1}(I, \vec{d}_I, \mathrm{DB})$
return $(I, \vec{d}_I, \pi_I)$

$\underline{\mathsf{BatchVerify}(pk, I, \vec{d}_I, \pi_I, C, K, r)}$ ▷ *Client*

$v \leftarrow Ch \stackrel{?}{=} \mathsf{DCH}_{\mathsf{pp}_2}(Y, K, C, r)$
$\quad \wedge\ \mathsf{KBatchVer}_{\mathsf{pp}_1}(C, I, \vec{d}_I, \pi_I)$
if $v = 0$ **then return** $\perp$
else return $\vec{d}_I$
end if

- $\mathsf{BatchQuery}(pk, \mathrm{DB}, I)$: The client sends the query index set I to the server. The server computes a proof $\pi_I \leftarrow \mathsf{KBatchOpen}_{\mathsf{pp}_1}(I, \vec{d}_I, \mathrm{DB})$ for the data entry set $\vec{d}_I$, and sends the proof $(I, \vec{d}_I, \pi_I)$ and (C, K, r) to the client.
- $\mathsf{BatchVerify}(pk, I, \vec{d}_I, \pi_I, C, K, r)$: The verifier (including the client) parses $pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch)$, and then verifies $v \leftarrow Ch \stackrel{?}{=} \mathsf{DCH}_{\mathsf{pp}_2}(Y, K, C, r) \wedge \mathsf{KBatchVer}_{\mathsf{pp}_1}(C, I, \vec{d}_I, \pi_I)$. If $v = 1$, then output $\vec{d}_I$, otherwise output $\perp$.

5.3 Security Analysis

Theorem 2 (Secure VDS). *If* KVC *is a key-binding key-value commitment and* DCH *is a collision-resistant double-trapdoor chameleon hash, then our scheme* VDS_2 *is secure.*

Proof. The proof of the theorem is conducted by executing the game $\mathsf{VDSsec}^{\mathsf{VDS}_2}_{\mathcal{A}}(\lambda)$. The adversary $\mathcal{A}$ may win the game in two ways, either by finding a collision in the double-trapdoor chameleon hash, or by breaking the key binding of the key-value commitment. We will show that the advantage of the adversary in both case is negligible.

The proof is proceeded by contradiction. Let $((i^*, d^*, \pi^*), (C^*, K^*, r^*))$ is the tuple returned by the adversary $\mathcal{A}$ at end of the game $\mathsf{VDSsec}^{\mathsf{VDS}_2}_{\mathcal{A}}(\lambda)$. If

the adversary wins in this game, recall that $Ch = \mathsf{DCH}_{\mathsf{pp}_2}(Y, K^*, C^*, r^*)$, $1 \leftarrow \mathsf{KVer}_{\mathsf{pp}_1}(C^*, i^*, d^*, \pi^*)$, $d^* \neq \perp$ and $(i^*, d^*) \notin \mathrm{DB}$. Consider the correct tuple $((i^*, d, \pi), (C, K, r))$ with index i^*. Then we define DCHCol as the event that $C^* \neq C$ such $(C^*, K^*, r^*) \neq (C, K, r)$. Obviously, we have

$$\begin{aligned}&\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1]\\ =\ &\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1 \wedge \mathsf{chcol}] + \Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1 \wedge \overline{\mathsf{chcol}}].\end{aligned}$$

Case chcol: In this case, $\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1 \wedge \mathsf{chcol}]$ is negligible under the assumption that DCH is collision-resistant. To this end, we construct an efficient reduction $\mathcal{B}_{\mathsf{DCH}}$ that uses $\mathcal{A}$ to break the collision-resistance of the double-trapdoor chameleon hash scheme DCH. The reduction $\mathcal{B}_{\mathsf{DCH}}$ proceeds as follows.

On input a hash public parameter pp_2, a hash key Y, and a hash value Ch, the reduction $\mathcal{B}_{\mathsf{DCH}}$ computes $(\mathsf{pp}_1, C) \leftarrow \mathsf{KGen}(1^\lambda, l)$ and sets the counter $cnt \leftarrow 0$. Then, the reduction $\mathcal{B}_{\mathsf{DCH}}$ runs $\mathcal{A}(pk)$ on the public key $pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch)$ by simulating the game $\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda)$. Note that $\mathcal{B}_{\mathsf{DCH}}$ does not know the full secret key sk, i.e., it does not know the trapdoors of DCH, therefore it has to access to a collision oracle.

To answer the append queries of $\mathcal{A}$, e.g., appending a data entry d, $\mathcal{B}_{\mathsf{DCH}}$ sets $i \leftarrow cnt + 1$, computes the commitment $C \leftarrow \mathsf{KAppend}_{\mathsf{pp}_1}(C, i, d)$ and the proof $\pi_i \leftarrow \mathsf{KOpen}_{\mathsf{pp}_1}(i, d, \mathrm{DB})$, sends C to its collision oracle, and forwards the response (K, r) together with the opening proof π_i of (i, d) to the adversary $\mathcal{A}$.

To answer the update queries of $\mathcal{A}$, e.g., updating the data entry d at position i to a new entry d', $\mathcal{B}_{\mathsf{DCH}}$ runs Update algorithm by accessing to its collision oracle, which is similar to the append queries.

The adversary outputs the tuple $((i^*, d^*, \pi^*), (C^*, K^*, r^*))$ at the end of the game. $\mathcal{B}_{\mathsf{DCH}}$ computes the honest proof $((i^*, d, \pi), (C, K, r))$ for the data entry d at the position i^* after the T queries, and outputs (C, C^*, r, r^*).

Observe that $\mathcal{B}_{\mathsf{DCH}}$ perfectly simulates the view of $\mathcal{A}$ as in the game $\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda)$. We know that $(C, C^*) \neq (r, r^*)$ when chcol happens. Since $\mathcal{A}$ wins in this game $Ch = \mathsf{DCH}_{\mathsf{pp}_2}(Y, K^*, C^*, r^*) = \mathsf{DCH}_{\mathsf{pp}_2}(Y, K, C, r)$. Therefore, one can see that

$$\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1 \wedge \mathsf{chcol}] \leq \Pr[\mathsf{Hashcol}_{\mathcal{B}_{\mathsf{DCH}}}^{\mathsf{DCH}}(\lambda) = 1] = \mathsf{negl}(\lambda).$$

Case $\overline{\mathsf{chcol}}$: In this case, $\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1 \wedge \overline{\mathsf{chcol}}]$ is negligible under the assumption that KVC is key-binding. To this end, we construct an efficient reduction $\mathcal{B}_{\mathsf{KVC}}$ that uses $\mathcal{A}$ to break the key-binding of the key-value commitment scheme KVC. The reduction $\mathcal{B}_{\mathsf{KVC}}$ proceeds as follows.

On input the public parameter pp_1 and the initialed commitment C, the reduction $\mathcal{B}_{\mathsf{KVC}}$ computes $(\mathsf{pp}_2, Y, x) \leftarrow \mathsf{DCHKGen}(1^\lambda)$ and a chameleon hash value Ch for $(0, \hat{r})$ where $\hat{r} \leftarrow \{0,1\}^\lambda$, and sets the counter $cnt \leftarrow 0$. Then, the reduction $\mathcal{B}_{\mathsf{KVC}}$ runs $\mathcal{A}(pk)$ on the public key $pk \leftarrow (\mathsf{pp}_1, \mathsf{pp}_2, Y, Ch)$ by simulating the game $\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda)$.

To answer the append queries of $\mathcal{A}$, e.g., appending a data entry d, $\mathcal{B}_{\mathsf{KVC}}$ sets $i \leftarrow cnt + 1$, computes the commitment $C \leftarrow \mathsf{KAppend}_{\mathsf{pp}_1}(C, i, d)$ and the proof $\pi_i \leftarrow \mathsf{KOpen}_{\mathsf{pp}_1}(i, d, \mathrm{DB})$, and finds a collision (C, r) with respect to the hash

value Ch by using the chameleon-hash trapdoors. The reduction $\mathcal{B}_{\mathsf{KVC}}$ sends $((i, d, \pi_i), (C, K, r))$ to the adversary $\mathcal{A}$.

To answer the update queries of $\mathcal{A}$, e.g., updating the data entry d at position i to a new entry d', $\mathcal{B}_{\mathsf{KVC}}$ runs Update algorithm, which is similar to the append queries.

The adversary outputs the tuple $((i^*, d^*, \pi^*), (C^*, K^*, r^*))$ at the end of the game. $\mathcal{B}_{\mathsf{KVC}}$ computes the honest proof $((i^*, d, \pi), (C, K, r))$ for the data entry d at the position i^* after the T queries, and outputs (i, d, d^*, π, π^*).

Observe that $\mathcal{B}_{\mathsf{KVC}}$ perfectly simulates the view of $\mathcal{A}$ as in the game $\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda)$. We know that $(C, C^*) = (r, r^*)$ when $\overline{\mathsf{chcol}}$ happens. Since $\mathcal{A}$ wins in this game, $\mathsf{KVer}_{\mathsf{pp}_1}(C, i^*, d^*, \pi^*) = \mathsf{KVer}_{\mathsf{pp}_1}(C, i^*, d, \pi) = 1$. Therefore, one can see that

$$\Pr[\mathsf{VDSsec}_{\mathcal{A}}^{\mathsf{VDS}_2}(\lambda) = 1 \wedge \overline{\mathsf{chcol}}] \leq \Pr[\mathsf{KeyBdg}_{\mathcal{B}_{\mathsf{KVC}}}^{\mathsf{KVC}}(\lambda) = 1] = \mathsf{negl}(\lambda).$$

Therefore, the overall advantage of the adversary winning the game is negligible. Thus, our scheme VDS_2 is secure.

6 Performance Evaluation

In this section, we report a comprehensive evaluation of our proposed two schemes VDS_1 and VDS_2. In the following, we first provide the description of experiment environment and parameter setting, and then discuss the comparison on communication, storage and time overhead between our two schemes with CVC/ACC-based VDS schemes [11], and VADS scheme [19].

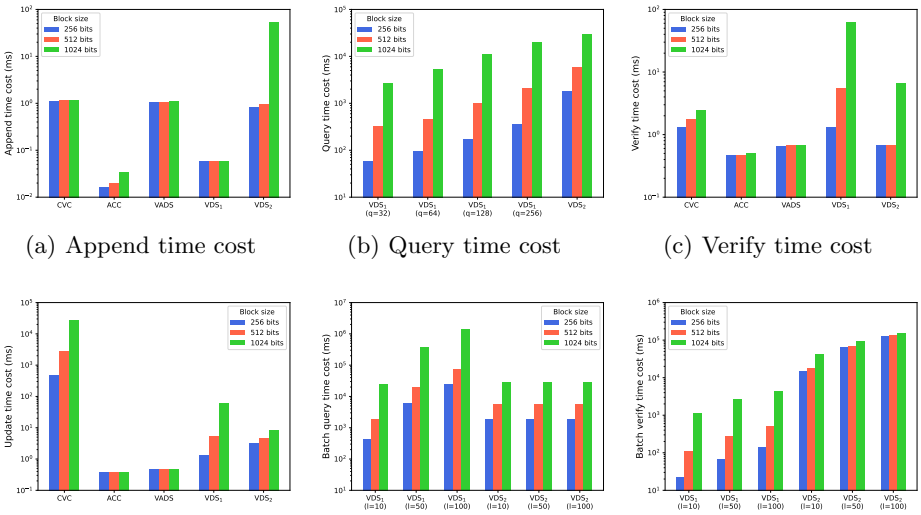
6.1 Implementation Setup

We implement in Python five VDS schemes including our two schemes VDS_1 (Section 4) and VDS_2 (Section 5), two VDS schemes in [11], and VADS scheme [19]. We use PBC-0.5.14 library with a type A elliptic curve for pairing-based and RSA-based cryptographic primitives, and SHA-256 hash function. We deploy our experiments on the machine with Intel(R) Core(TM) i9-11900K @ 3.50 GHz RAM 128GB and Ubuntu 20.04 LTS.

In the following experiment, we first determine the security parameter to 128 bits. To evaluate the time cost of five schemes, we set the database size to n = 4096, 2048, 1024, the branching number of the tree to q = 32, 64, 128, 256, the block size (i.e., the size of a data entry) to 256 bits, 512 bits and 1024bits, the number of queried data in batch query to l = 10, 50, 100. To completely quantify the query communication and server storage overhead, we perform separate experiments by setting the database size to $n = 2^{10}$, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20} and the batch query size to l = 10, 50, 100, 500, 1000, 2000.

6.2 Evaluation

Time Cost. We evaluate the time cost of five schemes in terms of the append time, the query time, the verify time, the update time, the batch query time, and



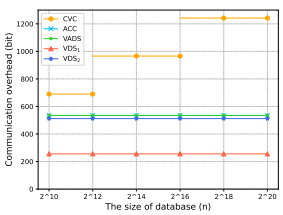
(a) Append time cost (b) Query time cost (c) Verify time cost

(d) Update time cost (e) Batch query time cost (f) Batch verify time cost

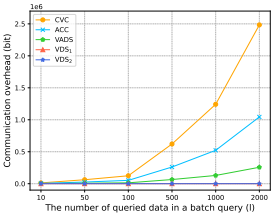
Fig. 3. Time cost comparison

the batch verify time. The experiment results show that, as detailed in Fig. 3, our two schemes are efficient but not optimal. First, we can observe from Fig. 3(a) and 3(d) that our scheme VDS_1 has better append and update time cost than CVC-based VDS scheme [11] which is also based on a tree structure. This result reveals that transferring the proof generation from the client to the server not only optimizes server storage but also improves client and server computation efficiency. Then we review Table 1 in Section 1 and further observe Fig. 3(a), 3(c), and 3(d). Although our scheme VDS_2 has constant client-side append time, verify time and update time independent of the size of the database, the block size plays a significant role. The reason is that in our scheme VDS_2 the time cost is dominated by the length of the prime determined by the block size. In addition, we show the time cost of single query, batch query, and batch verify in Fig. 3(b), 3(e), and 3(f). The query time cost of our two schemes is determined by server-side proof computation. According to Fig. 3(b) and 3(e), both single query and batch query time increase with the block size. Specially, Fig. 3(e) illustrates that the cost of batch query of VDS_2 is independent of the number of queried data l, while that of VDS_1 increases linearly with l. Fig. 3(f) shows that the cost of batch verify is dominated by block size and batch size l. Generally speaking, it is necessary to sacrifice computation efficiency to achieve better query communication and server storage as well as stronger security.

Communication Overhead. The communication overhead is mainly incurred by retrieving a data entry or multiple data entries in a single query or batch query. As shown in Fig. 4(a) and 4(b), both our schemes VDS_1 and VDS_2 reach the optimal communication overhead $O(1)$ in single query and batch query and



(a) Single query



(b) Batch query

Fig. 4. Communication overhead comparison

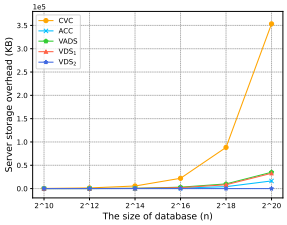


Fig. 5. Storage comparison

are superior to all existing scheme. Particularly, the query communication of VDS_1 and VDS_2 consists of only two elements and three elements respectively.

Storage Overhead. The server storage overhead in a VDS scheme mainly stems from the proof storage. The server storage of most existing VDS schemes is at least $O(n)$, where n is the database size. The ideal server storage is constant $O(1)$. Our scheme VDS_1 reduces the storage overhead to $O(n)$, and even our scheme VDS_2 achieves the optimal server storage overhead $O(1)$ by eliminating the proof storage and maintaining only constant-size auxiliary information (consisting of just three elements). As shown in Fig. 5, the server storage of our scheme VDS_2 outperforms that of the other four schemes.

7 Conclusion

In this paper, we explore new approaches to build unbounded VDS schemes for batch query with optimal query communication and server storage. To this end, we first introduce a new cryptographic primitive DCVC. Then, we propose an unbounded VDS scheme VDS_1 in the random oracle model from an aggregatable cross-commitment variant of DCVC, which has optimal communication cost $O(1)$ and better server storage $O(n)$. Further, we present the first unbounded VDS scheme VDS_2 with optimal communication overhead $O(1)$ and storage overhead $O(1)$ in the standard model. Both of our schemes enjoy constant-size public key. Compared with the state-of-the-art [11,19], our two schemes reach optimal communication and storage overhead, however, the computational performance is not optimal, so we leave constructing an overall optimal VDS scheme for batch query to be the future work.

Acknowledgments. This work was supported by the National Natural Science Foundation of China (Nos. 6196026014 and 62072357), the Fundamental Research Funds for the Central Universities (Nos. YJS2212 and ZDRC2204), and the Open Foundation of Henan Key Laboratory of Cyberspace Situation Awareness (No. HNTS2022012).

References

1. Agrawal, S., Raghuraman, S.: Kvac: Key-value commitments for blockchains and beyond. In: ASIACRYPT 2020, Daejeon, South Korea, December 7-11, 2020.

LNCS, vol. 12493, pp. 839–869. Springer (2020)
2. Ateniese, G., de Medeiros, B.: On the key exposure problem in chameleon hashes. In: SCN 2004, Amalfi, Italy, September 8-10, 2004. LNCS, vol. 3352, pp. 165–179. Springer (2004)
3. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to iops and stateless blockchains. In: CRYPTO 2019, Santa Barbara, CA, USA, August 18-22, 2019. LNCS, vol. 11692, pp. 561–586. Springer (2019)
4. Campanelli, M., Fiore, D., Greco, N., Kolonelos, D., Nizzardo, L.: Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In: ASIACRYPT 2020, Daejeon, South Korea, December 7-11, 2020. LNCS, vol. 12492, pp. 3–35. Springer (2020)
5. Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC 2013, Nara, Japan, February 26 - March 1, 2013. LNCS, vol. 7778, pp. 55–72. Springer (2013)
6. Chen, C., Wu, H., Wang, L., Yu, C.: Practical integrity preservation for data streaming in cloud-assisted healthcare sensor systems. Computer Networks **129**, 472–480 (2017)
7. Chen, X., Zhang, F., Kim, K.: Chameleon hashing without key exposure. In: ISC 2004, Palo Alto, CA, USA, September 27-29. LNCS, vol. 3225, pp. 87–98. Springer (2004)
8. Chen, X., Zhang, F., Susilo, W., Mu, Y.: Efficient generic on-line/off-line signatures without key exposure. In: ACNS 2007, Zhuhai, China, June 5-8, 2007. LNCS, vol. 4521, pp. 18–30. Springer (2007)
9. Chen, X., Zhang, F., Tian, H., Wei, B., Susilo, W., Mu, Y., Lee, H., Kim, K.: Efficient generic on-line/off-line (threshold) signatures without key exposure. Information Sciences **178**(21), 4192–4203 (2008)
10. Gennaro, R.: Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks. In: CRYPTO 2004, Santa Barbara, California, USA, August 15-19, 2004. LNCS, vol. 3152, pp. 220–236. Springer (2004)
11. Krupp, J., Schröder, D., Simkin, M., Fiore, D., Ateniese, G., Nürnberger, S.: Nearly optimal verifiable data streaming. In: PKC 2016, Taipei, Taiwan, March 6-9, 2016. LNCS, vol. 9614, pp. 417–445. Springer (2016)
12. Lai, R.W.F., Malavolta, G.: Subvector commitments with application to succinct arguments. In: CRYPTO 2019, Santa Barbara, CA, USA, August 18-22, 2019. LNCS, vol. 11692, pp. 530–560. Springer (2019)
13. Miao, M., Wei, J., Wu, J., Li, K., Susilo, W.: Verifiable data streaming with efficient update for intelligent automation systems. International Journal of Intelligent Systems **37**(2), 1322–1338 (2022)
14. Schröder, D., Schröder, H.: Verifiable data streaming. In: CCS 2012, Raleigh, NC, USA, October 16-18, 2012. pp. 953–964 (2012)
15. Schröder, D., Simkin, M.: Veristream - A framework for verifiable data streaming. In: FC 2015, San Juan, Puerto Rico, January 26-30, 2015. pp. 548–566. Springer (2015)
16. Shamir, A.: On the generation of cryptographically strong pseudorandom sequences. ACM Transactions on Computer Systems **1**(1), 38–44 (1983)
17. Sun, Y., Liu, Q., Chen, X., Du, X.: An adaptive authenticated data structure with privacy-preserving for big data stream in cloud. IEEE Transactions on Information Forensics and Security **15**, 3295–3310 (2020)

18. Tsai, I., Yu, C., Yokota, H., Kuo, S.: VENUS: verifiable range query in data streaming. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2018, Honolulu, HI, USA, April 15-19, 2018. pp. 160–165. IEEE (2018)
19. Wei, J., Tian, G., Shen, J., Chen, X., Susilo, W.: Optimal verifiable data streaming protocol with data auditing. In: ESORICS 2021, Darmstadt, Germany, October 4-8, 2021. LNCS, vol. 12973, pp. 296–312. Springer (2021)
20. Xu, J., Meng, Q., Wu, J., Zheng, J.X., Zhang, X., Sharma, S.: Efficient and lightweight data streaming authentication in industrial control and automation systems. IEEE Transactions on Industrial Informatics **17**(6), 4279–4287 (2021)
21. Xu, J., Wei, L., Wu, W., Wang, A., Zhang, Y., Zhou, F.: Privacy-preserving data integrity verification by using lightweight streaming authenticated data structures for healthcare cyber-physical system. Future Generation Computer Systems **108**, 1287–1296 (2020)

A Correctness of DCVC

Correctness. The correctness of our scheme DCVC can be verified as follows.

$$\begin{aligned} S_i^{m_i} \cdot \pi_i^{e_i} &= S_i^{m_i} \cdot \left(\sqrt[e_i]{\prod_{j=1, j\neq i}^{q} S_j^{m_j}} \cdot r^{\prod_{j=1, j\neq i}^{q} e_j} \right)^{e_i} \\ &= S_i^{m_i} \cdot \prod_{j=1, j\neq i}^{q} S_j^{m_j} \cdot r^{\prod_{j=1}^{q} e_j} \\ &= \prod_{j=1}^{q} S_j^{m_j} \cdot r^{\prod_{j=1}^{q} e_j} \\ &= C \mod N \end{aligned}$$

The correctness after updates also holds. Similarly, $C' = {S_j}^{m_j} \cdot {\pi'_j}^{e_j}$ for the commitment $C' = C \cdot S_i^{m'-m}$ and proofs $\pi'_j = \pi_j \cdot \sqrt[e_j]{S_i^{m'-m}}$ after the message m at position i is updated to m'.

Cross-Commitment Aggregation Correctness. The correctness of cross-commitment aggregation follows from the correctness of DCVC and Shamir's trick.

– Case1: When $k_l \notin \{k_j\}_{j\in[l-1]}$, $\gcd(e_{k_1}, \ldots, e_{k_l}) = 1$ and $\frac{\prod_{j\in[l]} e_{k_j}}{\gcd(e_{k_1}, \ldots, e_{k_l})} = \prod_{j\in[l]} e_{k_j} = e_K e_{k_l}$.

$$\rho_K = \hat{\pi} \cdot (\pi_{l,k_l})^{t_l e_{k_l}/e_K} = \left(\hat{\pi}^{e_K} \cdot (\pi_{l,k_l})^{t_l e_{k_l}}\right)^{1/e_K} \mod N$$

$$\rho_l = \hat{\pi}^{e_K/e_{k_l}} \cdot (\pi_{l,k_l})^{t_l} = \left(\hat{\pi}^{e_K} \cdot (\pi_{l,k_l})^{t_l e_{k_l}}\right)^{1/e_{k_l}} \mod N$$

$$
\begin{aligned}
&\prod_{j\in[l]} S_{k_j}^{t_j m_{j,k_j}} \pi^{\prod_{j\in[l]} e_{k_j}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot \left(\mathsf{ShamirTrick}(\rho_K, \rho_l, e_K, e_{k_l})\right)^{\prod_{j\in[l]} e_{k_j}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot \left(\left(\hat{\pi}^{e_K} \cdot (\pi_{l,k_l})^{t_l e_{k_l}}\right)^{1/e_K e_{k_l}}\right)^{\prod_{j\in[l]} e_{k_j}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot \hat{\pi}^{e_K} \cdot (\pi_{l,k_l})^{t_l e_{k_l}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot \hat{\pi}^{e_K} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot (\pi_{l,k_l})^{t_l e_{k_l}} \\
&= \prod_{j\in[l-1]} {C_j}^{t_j} \cdot {C_l}^{t_l} \\
&= \prod_{j\in[l]} {C_j}^{t_j} \mod N
\end{aligned}
$$

– Case2: When $k_l \in \{k_j\}_{j\in[l-1]}$, $\gcd(e_{k_1}, \ldots, e_{k_l}) = e_{k_l}$ and $\frac{\prod_{j\in[l]} e_{k_j}}{\gcd(e_{k_1},\ldots,e_{k_l})} = \prod_{j\in[l-1]} e_{k_j} = e_K$.

$$
\begin{aligned}
&\prod_{j\in[l]} S_{k_j}^{t_j m_{j,k_j}} \pi^{\prod_{j\in[l-1]} e_{k_j}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot \left(\hat{\pi} \cdot {\pi_{l,k_l}}^{t_l e_{k_l}/e_K}\right)^{\prod_{j\in[l-1]} e_{k_j}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot \hat{\pi}^{e_K} \cdot (\pi_{l,k_l})^{t_l e_{k_l}} \\
&= \prod_{j\in[l-1]} S_{k_j}^{t_j m_{j,k_j}} \cdot \hat{\pi}^{e_K} \cdot S_{k_l}^{t_l m_{l,k_l}} \cdot (\pi_{l,k_l})^{t_l e_{k_l}} \\
&= \prod_{j\in[l-1]} {C_j}^{t_j} \cdot {C_l}^{t_l} \\
&= \prod_{j\in[l]} {C_j}^{t_j} \mod N
\end{aligned}
$$

B Security Proof of DCVC

Theorem 3 (Position Binding). *If the strong RSA assumption holds, the scheme* DCVC *is position binding.*

Proof. Suppose there exists an adversary $\mathcal{A}$ who wins the game $\mathsf{PosBdg}_{\mathcal{A}}^{\mathsf{DCVC}}(\lambda)$ by producing two valid proofs to two different messages at the same position. We build a simulator $\mathcal{B}$ that may break the strong RSA assumption. The simulator

$\mathcal{B}$ takes a strong RSA problem instance (N, z) as input. The simulator $\mathcal{B}$ will use $\mathcal{A}$ to compute (y, e) s.t. $z = y^e \mod N$ as follows.

First, $\mathcal{B}$ selects a random $i \leftarrow \{1, \ldots, q\}$ as a guess for the index i on which $\mathcal{A}$ will break the position binding.

Next, $\mathcal{B}$ sets $e_i \leftarrow \mathsf{PrimeGen}(i)$ and $g \leftarrow z$. For $j = 1, \ldots, q, j \neq i$ the rest of the public parameters and trapdoors is computed as described by DCGen algorithm.

The adversary $\mathcal{A}$ is supposed to output (C, m, m', j, π, π') such that $m \neq m'$ and both π, π' are valid proofs at position j. If $j \neq i$, the simulator $\mathcal{B}$ aborts the simulation. Otherwise $\mathcal{B}$ proceeds as follows. Indeed,

$$S_i{}^m \cdot \pi^{e_i} = S_i{}^{m'} \cdot \pi'^{e_i} \quad \Longrightarrow \quad S_i{}^{m-m'} = (\pi'/\pi)^{e_i}.$$

Let $\Delta = m - m'$ and $\Lambda = \pi'/\pi$, the equation above can be rewritten as

$$g^{\Delta \prod_{j \neq i} e_j} = (\Lambda)^{e_i}.$$

Clearly, the absolute value of Δ is less than l bits and $e_1, \ldots, e_q$ are $(l+1)$-bit primes, it follows that $\gcd(\Delta \prod_{j \neq i} e_j, e_i) = 1$. We can get an e_i-root of g by use the Shamir's trick [16]. Concretely, we can compute two integers α and β such that $\alpha \Delta \prod_{j \neq i} e_j + \beta e_i = 1$ using the extended Euclidean algorithm. Thus,

$$g = g^{\alpha \Delta \prod_{j \neq i} e_j + \beta e_i} = (g^{\Delta \prod_{j \neq i} e_j})^{\alpha} \cdot g^{\beta e_i} = (\Lambda^{\alpha})^{e_i} \cdot (g^{\beta})^{e_i} = (\Lambda^{\alpha} g^{\beta})^{e_i}.$$

Therefore, if $\mathcal{A}$ succeeds in the game $\mathsf{PosBdg}_{\mathcal{A}}^{\mathsf{DCVC}}(\lambda)$ with probability ϵ, then $\mathcal{B}$ successfully breaks the strong RSA assumption with probability ϵ/q.

Theorem 4 (Indistinguishable Collisions). *The scheme* DCVC *has indistinguishable collisions.*

Proof. According to the definition of indistinguishable collisions, any PPT adversary cannot distinguish between a random value and the output of DCCol.

Concretely, in the game $\mathsf{CollInd}_{\mathcal{A}}^{\mathsf{DCVC}}(\lambda)$ it holds:

(1) The case of collision finding:

$$C_0 = S_1^{m_1} \ldots S_i^{m_i} \ldots S_q^{m_q} r_0^{\prod_{i=1}^{q} e_i},$$

$$aux^* = (m_1, \ldots, m_i, \ldots, m_q; r_0),$$

and

$$C_0 = S_1^{m_1} \ldots S_i^{m'_i} \ldots S_q^{m_q} r_0'^{\prod_{i=1}^{q} e_i},$$

$$aux_0 = (m_1, \ldots, m'_i, \ldots, m_q; r'_0),$$

where $r'_0 = r_0 \cdot t_{e_i}^{m_i - m'_i}$.

(2) The case of no collision finding:

$$C_1 = S_1^{m_1} \ldots S_i^{m'_i} \ldots S_q^{m_q} r_1^{\prod_{i=1}^{q} e_i},$$

$$aux_1 = (m_1, \ldots, m'_i, \ldots, m_q; r_1).$$

The aux_0 and aux_1 consist of all messages and r'_0 and r_1 respectively. Since r_0 is a uniformaly random element in $\mathbb{Z}_p$, r'_0 is also a uniformaly random element in $\mathbb{Z}_p$. Thus, r'_0 and r_1 are both uniformly random element in $\mathbb{Z}_p$. Therefore, the probability of any adversary winning the game $\mathsf{CollInd}_{\mathcal{A}}^{\mathsf{DCVC}}(\lambda)$ is exactly $1/2$.

Theorem 5 (Key Exposure Freeness). *The scheme* DCVC *based on RSA is key exposure freeness.*

Proof. Given a collision $(m_1, \ldots, m_i, \ldots, m_q; r)$ and $(m_1, \ldots, m'_i, \ldots, m_q; r')$, we can get

$$S_i{}^{m_i} \cdot \pi_i^{e_i} = S_i{}^{m'_i} \cdot \pi_i'^{e_i} \quad \Longrightarrow \quad (g^{d_i})^{m_i - m'_i} = r'/r.$$

From the equation above, the information of g^{d_i} may be recovered. However, it is impossible for anyone to compute the trapdoor d_i from g^{d_i}. Therefore, our scheme DCVC based on RSA is key exposure freeness.