SCIENCE CHINA
Information Sciences

• **Supplementary File** •

DBKEM-AACS: A Distributed Key Escrow Model in Blockchain with Anonymous Authentication and Committee Selection

Axin XIANG[1,2], Hongfeng GAO[1,3*], Youliang TIAN[1,2,4*] & Liang WAN[1,5*]

[1]*College of Computer Science and Technology, Guizhou University,Guiyang* 550025, *China;*
[2]*Institute of Cryptography and Data Security, Guizhou University, Guiyang* 550025, *China;*
[3]*Network and Information Management Center, Guizhou University, Guiyang* 550025, *China;*
[4]*State Key Laboratory of Public Big Date, Guizhou University, Guiyang* 550025, *China;*
[5]*Institute of Computer Software and Theory, Guizhou University, Guiyang* 550025, *China*

Appendix A Preliminaries

Appendix A.1 CHURP scheme

The CHURP scheme is proposed by Maram et al. [1]. On the basis of the previous proactive secret sharing schemes, its core thought is to introduce the idea of the "dimension-switching" to prevent the destruction of whole proactive secret sharing scheme due to the number of corrupted nodes in the process of share updating exceeding the preset threshold value. The handoff in CHURP that achieve the update of shares from the old committee to the new committee is shown in Figure A1, in which three phases of the handoff process about the above idea are as follows:

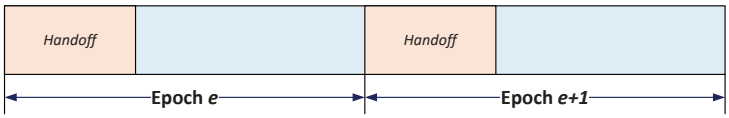

Figure A1 The Handoff during the epoch.

(1) *Share reduction.* Introducing a binary polynomial $B(x, y)$, which is of order $\langle t, 2t\rangle$, and nodes switch from the t-order dimension of $B(x, y)$ to the $2t$-order dimension, so that each node in the new committee gains a reduced share $B(x, j)$, which is an univariate polynomial of variable x.

(2) *Share proactivization.* The new committee generates a 0-sharing polynomial $Q(x, y)$, which satisfies $Q(0, 0) = 0$ and is of order $\langle t, 2t\rangle$, and calculates $B'(x, j) = B(x, j) + Q(x, j)$, so each new committee node gains a new reduced share $B'(x, j)$, which is independent of old reduced share $B(x, j)$.

(3) *Full-share distribution.* nodes switch from the $2t$-order dimension of $B(x, y)$ to the t-order dimension, and generate full shares $B'(h, y)$ through new reduced shares $\{B'(x, j)\}_j$, which are an univariate polynomial of variable y.

Appendix A.2 Smart contract

Smart contract is an executable code deployed on blockchain, which does not rely on any central organization to automatically execute the contract on behalf of all participants and has the characteristics of enforcement, tamper-proof and verifiability. When a certain precondition is met, the contract will automatically execute. By encoding the rules in real world, encrypting and then storing it on blockchain, it is ensured that all participants can run on the synchronized version without being unilaterally tampered. We introduce the contract structure of a smart contract verifiable framework proposed by Dorsala et al. [2] into the DBKEM-AACS to construct multiple contracts, and take solidity development documentation (https://docs.soliditylang.org/en/latest) as a reference to compile smart contracts for performance analysis based on the designed contract pseudocode.

Appendix A.3 Physically Unclonable Functions

Physically Unclonable Functions (PUFs) were proposed by Lim et al. [3]. Silicon PUF is an input-output mapping $\gamma : \{0, 1\}^m \to \{0, 1\}^n$, where m is the bit length of an input, n is the bit length of an output, and a n-bits output value clearly identified by a m-bits input value, which has the characteristics of unclonable, unpredictability. The most important thing is that its function is only related to the embedded object. Based on the above characteristics, Krzywiecki et al. [4] propose anonymous authentication scheme using PUF for the first time. Along that line, we introduce the idea of a PUF based anonymous authentication scheme proposed by Chatterjee et al. [5] into the DBKEM-AACS to construct an anonymous authentication scheme suitable for blockchain in the paper, which have the strictly-secure anonymity.

* Corresponding author (email: hfgao@gzu.edu.cn, youliangtian@163.com, wanliangtr@163.com)

Appendix B The pseudocodes for all contract

ACC-Contract

Function Init(): Set $state := Iint$, $A := 0$, $tx := Search()$, $wk := Search()$, $tx'_i := Search()$, $CND := Perform(CNSC - Contract)$, $wk'_i := Search()$

Function Create(): Upon receiving ("Create", tx_i, wk_i, pk_i, e) from BN_i:

- assert $state = Iint$, epoch is e, tx and wk
- assert $tx'_i == tx_i$ and $wk'_i == wk_i$
- assert $tx_i \leqslant tx$, $wk_i \leqslant wk$ and pk_i
- set $state := Created$

Function ADefine(): Upon receiving ("Initializ activeness", pk_i) from BN_i:

- assert $state = Created$
- If check ($tx_i! = 0$ and $wk_i == 0$):
  - set $A := \min(1, 0.5 + tx_i/tx)$
- If check ($tx_i == 0$ and $wk_i! = 0$):
  - set $A := \min(1, 0.5 + wk_i/wk)$
- If check ($tx_i! = 0$ and $wk_i! = 0$):
  - set $A := \min(1, 0.5 + wk_i/wk + tx_i/tx)$
- Else:
  - set $A := 0.5$
- set $state := Initializd\ activeness$

Function AUpdate(): Upon receiving ("Update activeness", A, pk_i, $e + 1$) from BN_i:

- assert $state = Initializd\ activeness$
- assert epoch is $e + 1$
- If check (pk_i in CND):
  - set $A := Perform\ formula\,(1)$
- Else:
  - set $A := Perform\ formula\,(2)$
- set $state := Updated\ activeness$

Figure B1 Activeness calculation contract.

CNSC-Contract

Function Init(): Set $state := Iint$, $\$reward := 0$, $\$deposit := \{NULL\}$, $PCND := \{NULL\}$, $END := \{NULL\}$, $CND := \{NULL\}$

Function Create(): Upon receiving ("Create", pk_{en}, d_{en}, N, r, e, $A_i(e)$, pk_i, d_{bni}) from EN and BN_i:

assert $state = Iint$, and $r \geqslant N * d_{bni}$

assert epoch is e and there are n candidate nodes

set $\$reward := r$, $\$deposit := \$deposit \cup (d_{bni},\ pk_i)$

set $state := Created$

Function Sort(): Upon receiving ("Sort node", pk_{en}, $\$deposit$, pk_i) from EN:

assert $state = Created$

assert $ledger\,[pk_i] \geqslant d_{bni}$ // where $ledger$ is the balance of an account

If check $(ledger\,[CNSC] == d_{en} + n * d_{bni})$:

set $PCND := BubbleSort(pk_i)$ // where $BubbleSort$ is a sorting algorithm

Else:

stop $CNSC{-}Contract$

set $state := Sorted\ node$

Function ECJ(): Upon receiving ("Select committee", pk_{en}, $PCND$, N) from EN:

assert $state = Sorted\ node$

assert N, $PCND$ and its node number $|PCND|$

If check $(|PCND| \geqslant N)$:

For search $(pk_i \in PCND)$:

set $CND := Selection(PCND,\ N)$ // where $Selection$ is a selection algorithm

Else:

stop $CNSC{-}Contract$

set $state := Selected\ committee$

Function Return(): Upon receiving ("Return", pk_{en}, CND, END) from EN:

assert $state = Selected\ committee$

assert $|CND| == N$

assert CND, END

transfer d_{bni} to the address of $pk_i \in END$

set $ledger[pk_i] := ledger[pk_i] + r\ (pk_i \in CND)$

set $state := Returned$

Figure B2 Committee node selection contract.

NAC-Contract

Function Init(): Set $state := Iint$, $k_i' := 0$, function H, random N_i, $d_i := 0$, $GM_i := 0$, $Vv := FALSE$, $h := 0$, $NDL := \{NULL\}$, $GD := \{NULL\}$

Function Create(): Upon receiving ("Create", pk_i, $H(r_i)$, $H(N_i)$), k_i, $r_i \oplus m_i$, $H(r_i \oplus m_i)$) from EN_i:

assert $state = Iint$, k_i and k_i'

set $d_i := H(N_i) \| pk_i \| (H(r_i) \oplus N_i)$

set $GM := d_i \| r_i \oplus m_i \| H(r_i \oplus m_i) \| "Grant" \| k_i$

set $state := Created$

Function Login(): Upon receiving ("Login", pk_i, d_i) from EN_i:

assert $state = Created$

set $NLD := NLD \cup (pk_i,\ d_i)$

set $state := Logined$

Function Grant(): Upon receiving ("Grant", pk_i, GM_i) from EN_i:

assert $state = Logined$, d_i and Vv

Go to VerifyD()

assert $state = Verified\ authorization\ message$

set $Vv := VerifyD()$

If check $(Vv == TRUE)$:

set $GD := GD \cup (pk_i, d_i, (r_i \oplus m_i) \| H(r_i \oplus m_i) \| "Grant" \| k_i)$

Else:

stop $NAC - Contract$

set $state := Granted$

Function VerifyD(): Upon receiving ("Verify authorization message", GM_i, d_i) from function Grant():

assert $state = Pre - Grant$, Vv, GM_i and $H(r_i \oplus m_i)$

set $h := H(r_i \oplus m_i)$

assert CND, END

If check $(h == H(r_i \oplus m_i))$:

set $Vv := TRUE$

RETURN Vv

set $state := Verified\ authorization\ message$

Figure B3 Node authentication contract.

NAC-Contract(continue)

Function Authenticate(): Upon receiving ("Authenticate", $H(r)$, d, k_i') from EN_i:

assert $state = Granted$

assert d_i, k_i'

If check ($k_i' \leqslant k_i$):

Go to VerifyH()

assert $state = Verified\ authorization$

set $Vv := VerifyH()$

If check ($Vv == TRUE$):

set $k_i' := k_i' + 1$

send (d_i, GM_i) to EN_i

set $GM_i := d_i || r_i \oplus m_i || H(r_i \oplus m_i) \| "Grant" \| k_i'$

Else:

stop $NAC - Contract$

Else:

Go to Revoke()

set $state := Authenticated$

Function VerifyH(): Upon receiving ("Verify authorization", d_i, $H(r_i)$, pk_i) from function Authenticity():

assert $state = Pre - Authenticate$

assert pk_i, $H(N_i)$ and $(H(r_i) \oplus N_i)$ in d_i

set $Vv := FALSE$

If check ($pk_i == pk_i$ and $H(H(r_i) \oplus (H(r_i) \oplus N_i)) == H(N_i)$):

set $Vv := TRUE$

RETURN Vv

set $state := Verified\ authorization$

Function Revoke(): Upon receiving ("Revoke permission", d_i, pk_i, k_i, k_i') from function Authenticity():

assert $state = Authenticated$

If check ($k_i' > k_i$):

assert $(pk_i, d_i || r_i \oplus m_i \| "Revoke" \| k_i')$

set $state := Revoked\ permission$

Figure B4 Node authentication contract(continue).

KEC-Contract

Function Init(): Set $state := Iint$, $k := 0$, $C := \{NULL\}$, $Epoch := 0$, $s := \{NULL\}$,$c := 0$, $SI := 0$,
$w := 0$, $CND := call\,(CNSC - Contract)$, $U' := \{NULL\}$, $C' := \{NULL\}$, $SIH := 0$,
$h := 0$, $CND' := call\,(CNSC - Contract)$, hash function H, $Cv := TRUE$, $SJ := 0$,
$A := call(NAC - Contract)$, $CB := 0$, $S := 0$, $k_i := 0$

Function Create(): Upon receiving ("Create", k', e, CND, CND', pk_i) from EN_i:
assert $state = Iint$
set $Epoch := e$
set $C := CND$, $C' := CND'$
set $k := k'$
set $state := Created$

Function Distribute(): Upon receiving ("Distribute", pk_i, C, $r_h \| pk_h$, $B_i(h, y)$) from EN_i and C_i:
assert $state = Created$
assert epoch is $Epoch$, C and k
set $w := H(sk_i)$
set $C := C \cup r_h \| pk_h$
For search($pk_h \in C$):
set $c := B_i(h,\ y) \oplus r_h$
set $s := s \cup (c \| H(c) \| pk_h \| pk_i)$
set $state := Distributed$

Function Update(): Upon receiving ("Update", C', $r_j \| pk_j$, s_j, s_i', s_i^h) from EN_i, C_i and C_i':
assert $state = Distributed$
set $Epoch := Epoch + 1$
assert epoch is $Epoch$
set $U' := \{C'\}_{j \in \{2t+1\}}$
set $Cv := CVerify(s_j)$
assert $state = Verified$
If check ($Cv == TRUE$):
set $SJ := s_j$
Else:
stop $KEC - Contract$
set $Cv := CVerify(s_i')$
assert $state = Verified$
If check ($Cv == TRUE$):
set $SI := s_i'$
Else:
stop $KEC - Contract$
set $Cv := CVerify(s_i^h)$
assert $state = Verified$
If check ($Cv == TRUE$):
set $SIH := s_i^h$
Else:
stop $KEC - Contract$
set $state := Updated$

Figure B5 Key escrow contract

KEC-Contract(continue)

Function Recover(): Upon receiving ("Recover", r_i, $B_i'(h,y)\|H(B_i'(h,y))\|pk_h\|pk_i$) from EN_i and C_i':

assert $state = Pre-Recover\ and\ Distributed$

set $Cv := CVerify(B_i'(h,y)\|H(B_i'(h,y))\|pk_h)$

assert $state = Verified$

If check $(Cv == TRUE)$:

store $B_i'(h_i,y)\|H(B_i'(h_i,y))\|pk_{h_i}$ in local memory

Else:

stop $KEC-Contract$

If check $(k_i \leqslant k)$:

For search $(pk_{h_i} \in C')$:

set $CB := (B_i'(h_1,y)\|B_i'(h_2,y)\|\ldots\|B_i'(h_{t+1},y)) \oplus r_i$

set $S := CB\|H(CB)\|pk_i$

set $k_i := k_i + 1$

Else:

stop $KEC-Contract$

set $state := Recovered$

Function CVerify(): Upon receiving ("Verify", $s = c_i'\,||H(c_i')||\,pk_i$) from function Update() and Recover():

assert $state = Pre-Verify$

assert Vv, GM and $H(r \oplus m)$

set $h := H(c_i')$, $c := c_i'$

If check $(h == H(c))$:

set $Cv := TRUE$

Else:

set $Cv := FALSE$

RETURN Cv

set $state := Verified$

Figure B6 Key escrow contract(continue)

Appendix C Key sub-share update

This part uses the idea of the "dimension-switching" [1]. Assuming that the ability of adversary to corrupt nodes in the old and new committees is: the adversary can corrupt t nodes at most in one epoch to ensure the security of keys in the handoff process, that is, it is strong enough to resist the collusion attack of $2t$ corrupted nodes. Actually, when the epoch e ends, and then go to the handoff phase of next epoch $e+1$, in which EN_i calls CNSC-Contract to complete the selection of a new committee $C_i^{(e+1)} = \{P_1^{e+1}, P_2^{e+1}, ..., P_n^{e+1}\}$ and public, where new committee nodes are also sorted according to lexicographic order of their public keys. Furthermore, the handoff process is divided into three stages including share reduction, share proactivezation, and full-share distribution. the execution details are as follows:

a. Share reduction.

• Firstly, KEC-Contract randomly selects $2t+1$ nodes from $C_i^{(e+1)}$ to make up a temporary new committee $U' = \{C_i^{(e+1)}\}_{j\in\{2t+1\}}$, and obtains their response values r_j, where $j = 1, 2, ..., 2t+1$.

• Secondly, the nodes in $C_i^{(e)}$ calculate the temporary reduction share $B_i(h,y)$, the secure temporary reduction share $c_i^j = B_i\,(h,j) \oplus r_j$ and the certificate of temporary reduction share $H(c_i^j)$ for each node in U', and connect them with public key pk_j by using "$\|$" to obtain s_j=$c_i^j\|H(c_i^j)\|pk_j$, and then input s_j into KEC-Contract, so as to publicly publish it on the blockchain, where h=$1, 2, ..., n$.

• Finally, each node U_j' in U' retrieves s_j by using their public key pk_j and completes the correctness verification of s_j through a hash function H, c_i^j, and $H(c_i^j)$. And then U_j' obtains $B_i(h,j)$ by calculating the formula $c_i^j \oplus r_j$. Furthermore, constructing the reduced share $B_i(x,j)$ of node pk_j by collecting $t+1$ temporary reduction shares that have been correctly verified, such as

$\{(h_k, B_i(h_k, j))\,|k = 1, 2, \cdots t+1\}$. The calculation formula is as follows:

$$\begin{cases} a_0 + a_1 h_1 + \cdots + a_t (h_1)^t + f(j) + ch_1 j = B_i(h_1, j) \\ a_0 + a_1 h_2 + \cdots + a_t (h_2)^t + f(j) + ch_2 j = B_i(h_2, j) \\ \vdots \\ a_0 + a_1 h_{t+1} + \cdots + a_t (h_{t+1})^t + f(j) + ch_{t+1} j = B_i(h_{t+1}, j) \end{cases} \tag{C1}$$

where a_i $(i = 1, 2, ..., t)$ is unknown, $f(j)$ is an unknown constant of the $2t$-order univariate polynomial $f(y)$ for y, and $ch_k j$ $(k = 1, 2, ..., t+1)$ is also unknown. Therefore, aiming at $t+1$ unknowns a_i that are all different, $ch_k j$ and $f(y)$, $B_i(x, j)$ can be constructed based on the following Lagrange interpolation formula:

$$\begin{aligned} B_i(x, j) &= \textstyle\sum_{k=1}^{t+1} B_i(h_k, j) \prod_{k \neq l,\ l=1}^{t+1} \frac{(x - h_l)}{(h_k - h_l)} \pmod q \\ &= (a_0 + f(j)) + (a_1 + cj)x + a_2 x^2 + \cdots + a_t x^t \pmod q \end{aligned} \tag{C2}$$

where $B_i(0, j) = a_0 + f(j)$, the coefficient $a_1 + cj$ of x, and the coefficient a_i $(i = 2, 3, ..., t)$ are all known.

b. Share proactivization.

• Firstly, KEC-Contract randomly selects $2t+1$ nodes from new committee $C_i^{(e+1)}$ to form the temporary new committees $U' = \{C_i^{(e+1)}\}_{j \in \{2t+1\}}$. Subsequently, each node U_j' in U' randomly selects a $\langle t, 2t\rangle$-order 0-shared binary polynomial $Q_i(x, y) = c_1^i x + c_2^i x^2 + \cdots + c_n^i + d_1^i y + d_2^i y^2 + \cdots + d_{2t}^i y^{2t}$, which satisfies $Q_i(0, 0) = 0$.

• Secondly, U_j' calculates the new reduction share $B_i'(x, j) = B_i(x, j) + Q_i(x, j)$, where the equation satisfies $B_i'(0, 0) = B_i(0, 0) + Q_i(0, 0)$, the secure new reduction share $c_i' = B_i{}'(x, j) \oplus r_j$, the certificate of new reduction share $H(c_i')$, and connects them with the public key pk_j to obtain $s_i' = c_i' \,\|H(c_i')\,\|pk_j$, and then inputs s_i' to KEC-Contract.

• Finally, KEC-Contract executes the function CVerify() to complete the correctness verification of s_i' through a hash function H, c_i', and $H(c_i')$. If the verification passes, it will be publicly published on the blockchain, otherwise ignored.

c. Full-share distribution.

• Firstly, for any node P_h' $(h{=}1, 2, ..., n)$ in the new committee $C_i^{(e+1)}$, each node U_j' in U' that has performed the operation of share proactivization calculates the temporary full-share $B_i'(h, j)$, the secure temporary full-share $c_i^h = B_i{}'(h, j) \oplus r_h$, the certificate of temporary full-share $H(c_i^h)$, and connects them with the public key pk_h to obtain $s_i^h = c_i^h \| H(c_i^h) \| pk_h$, and then inputs s_i^h to KEC-Contract, so as to publicly publish on the blockchain.

• Secondly, P_h' obtains s_i^h from the blockchain through his (or her) pk_h and completes the correctness verification of s_i^h through a hash function H, c_i^h, and $H(c_i^h)$, if the verification passes, the XOR operation will be used to obtain $B_i'(h, j)$ by combining with r_h, otherwise ignored.

• Finally, P_h' collects $2t+1$ temporary full shares that have been correctly verified, such as $\{(j_k, B_i'(h, j_k)|$ $k = 1, 2, \cdots, 2t+1\}$, to construct the full share $B_i'(h, y)$ of node P_h'. The calculation formula is as follows:

$$\begin{cases} e_0 + g(h) + e_1 j_1 + e_2 (j_1)^2 + \cdots + e_{2t} (j_1)^{2t} + chj_1 = B_i'(h, j_1) \\ e_0 + g(h) + e_1 j_2 + e_2 (j_2)^2 + \cdots + e_{2t} (j_2)^{2t} + chj_2 = B_i'(h, j_2) \\ \vdots \\ e_0 + g(h) + e_1 j_1 + e_2 (j_{2t+1})^2 + \cdots + e_{2t} (j_{2t+1})^{2t} + chj_{2t+1} = B_i'(h, j_{2t+1}) \end{cases} \tag{C3}$$

where e_0, $e_i = (b_i + d_i)$ $(i = 1, 2, ..., 2t)$ are all unknown, $g(h)$ is an unknown constant of the $2t$-order univariate polynomial $g(x)$ for x, and chj_k $(k = 1, 2, ..., t+1)$ is unknown. Therefore, aiming at $2t{+}1$ unknowns e_i that are all different, $g(h)$ and chj_k, $B_i'(h, y)$ can be constructed based on the following Lagrange interpolation formula:

$$\begin{aligned} B_i'(h, y) &= \textstyle\sum_{k=1}^{2t+1} B_i(h, j_k) \prod_{k \neq l,\ l=1}^{2t+1} \frac{(y - j_l)}{(j_k - j_l)} \pmod q \\ &= (e_0 + g(h)) + (e_1 + ch)y + e_2 y^2 + \cdots + e_{2t} y^{2t} \pmod q \end{aligned} \tag{C4}$$

where $B_i'(h, 0) = e_0 + g(j)$, the coefficient $e_1 + ch$ of y, and the coefficient e_i $(i = 2, 3, ..., 2t)$ are all known.

Appendix D Security analysis

Appendix D.1 Adversary model

In this section, we consider the details of the adversary model [1, 6] of the DBKEM-AACS, which can resist the active attack. On the one hand, we suppose a powerful active adversary $\mathcal{A}$, which can whenever perform attack on the committee nodes that in view of a $(t+1,\ n)$ threshold secret sharing scheme, there are at most t nodes corrupted by $\mathcal{A}$ in the old and new committees, respectively. On the other hand, aiming at an anonymous authentication scheme, assuming that $\mathcal{A}$ is a semi-honest adversary in the process of identity authentication.

Definition 1. In the case that adversary $\mathcal{A}$ only has the ability to corrupt no more than t nodes during the handoff phase, the DBKEM-AACS satisfies the following properties:

Confidentiality: if adversary $\mathcal{A}$ can only corrupt no more than t nodes in any epoch, $\mathcal{A}$ will not get any information about the secret key.

Integrity: if adversary $\mathcal{A}$ can only corrupt no more than t nodes during the process of handoff, after the handoff, the key shares for honest node still can be calculated correctly, and the secret key remains unchanged.

Definition 2. In the case that adversary $\mathcal{A}$ only has the ability to perform the active attack, the DBKEM-AACS satisfies the following property:

Correctness: if adversary $\mathcal{A}$ is semi-honest during the process of authentication, $\mathcal{A}$ cannot prevent the node from being correctly authenticated and the secret key can be correctly restored.

Appendix D.2 Security analysis

We analyze the security of the proposed model (DBKEM-AACS) in this article based on the definition of threat model in Section 4.2. On the basis of the assumption that an anonymous authentication scheme is designed in Section 4.3.2 is based on a secure channel, so as to the security of the proposed scheme in Section 4.3.3 is mainly to analyze the attack capability of an active adversary $\mathcal{A}$. Therefore, according to Definition 1 and Definition 2 in Section 4.2, in this section, we ensure that the security of the DBKEM-AACS by analyzing the security of the proposed scheme in Section 4.3.3. The key point of our analysis is to closely focus on the idea of "dimension-switching" in [1], so that while ensuring the security of the process of key share update, the DBKEM-AACS must meet the confidentiality, integrity and correctness. The proof process is as follows:

Confidentiality. The meaning of confidentiality in this article is that given the attack capability of the active adversary $\mathcal{A}$, he cannot obtain any information about the secret key sk_i based on the known information obtained. In order to prove the confidentiality of the DBKEM-AACS, we propose the following lemmas.

Lemma 1. If adversary $\mathcal{A}$ corrupts no more than t nodes in the old committee and no more than t nodes in the temporary new committee, the information obtained by $\mathcal{A}$ in the stage of share reduction cannot be used to obtain any information for sk_i.

Proof. Supposing that $\mathcal{A}$ obtains $2t$ reduced-shares $B_i(x, j)$ and t full-shares $B_i(h, y)$ in the new and old committees. If $\mathcal{A}$ wants to obtain the EN_i' s sk_i, the binary polynomial $B_i(x, y)$ for EN_i firstly needs to be recovered, so the system of $2t + 1$-variables linear congruence equations and the system of $t + 1$-variables linear congruence equations are established which same as formula (5) and formula (7), but the numbers of their equations are $2t$ and t, respectively. If there are $2t + 1$ unknowns, there is no unique solution to the establishment of a system of equations containing t equations. Moreover, the establishment of a system of equations containing $2t$ equations cannot uniquely solve $t + 1$ unknown number. And then we can conclude that the capability of $\mathcal{A}$ is not enough to obtain any information for sk_i in the share reduction phase.

In addition, based on the collision-resistance [**?**,**?**] of a hash function and the negligible probability attack [**?**] of a XOR function, what its definition is that let $X = A \oplus B$, where X, A, B are k-bit numbers, if X is known, the probability that A successfully produce X in combination with B is $P(k) = 2^{-k}$, in which as long as k is large enough, $P(k)$ is negligible, the PPT $\mathcal{A}$ cannot obtain any additional information from the distribution process of reduced-shares and full-shares.

Lemma 2. If adversary $\mathcal{A}$ corrupts no more than t nodes in the temporary new committee, then $\mathcal{A}$ cannot destroy the process to proactivizate the key sub-shares in the stage of share proactivization.

Proof. Given that a $\langle t, 2t\rangle$-order 0-shared binary polynomial $Q_i(x, y)$ with confidentiality and integrity, any $2t+1$ t-order univariate polynomial $Q_i(x, j)$ can calculate an $\langle t, 2t\rangle$-order $Q_i(x, y)$ by establishing a system of $2t + 1$-variables linear congruence equations. Therefore, in a temporary new committee with at least $2t + 1$ nodes but there are t corrputed nodes, as long as there is an honest node, $Q_i(x, y)$ can be restored to cover up $B_i(x, j)$.

In addition, based on the collision-resistance of a hash function and the negligible probability attack of a XOR function, the PPT $\mathcal{A}$ cannot obtain any additional information from the distribution process of reduced-shares.

Lemma 3. If adversary $\mathcal{A}$ corrupts no more t nodes in the new committee, the information obtained by $\mathcal{A}$ in the distribution phase of full-shares cannot be used to obtain any information for sk_i.

Proof. According to Lemma 1, assuming that $Q_i(x, y)$ can be generated correctly and randomly, so based on the formula $B'_i(x, y) = B_i(x, y) + Q_i(x, y)$, we can get $B'_i(x, j)$ is completely independent of $B_i(x, j)$. No matter how many nodes $\mathcal{A}$ corrupts in the temporary new committee, $\mathcal{A}$ can only get t $B'_i(h, y)$ in the distribution phase of full-shares. Because the order of the selected binary polynomial is $\langle t, 2t\rangle$, aiming at $t+1$ unknown numbers, there is no unique solution to establish a system of equations with t equations, then it can be concluded that the capability of $\mathcal{A}$ is not enough to get any information for sk_i in the distribution phase of full-share.

In addition, based on the collision-resistance of a hash function and the negligible probability attack of a XOR function, the PPT $\mathcal{A}$ cannot obtain any additional information from the distribution phase of full-shares.

In summary, by Lemma 1, 2 and 3, we can conclude that $\mathcal{A}$ cannot obtain any information for sk_i during the update phase of key sub-shares, thus ensuring the confidentiality of the DBKEM-AACS.

Integrity. The meaning of integrity in this article is that the nodes in the new committee can correctly calculate the sub-shares of the same sk_i. In order to prove the integrity of the DBKEM-AACS, we propose the following lemmas.

Lemma 4. If adversary $\mathcal{A}$ can corrupt no more than t nodes in the old committee and no more than t nodes in the temporary new committee, then after the phase of share reduction, at least $t + 1$ nodes in temporary new committee correctly reconstruct $B_i(x, j)$.

Proof. Given that the number of nodes in the old committee is n, which satisfies $n \geqslant 2t + 1$, the number of $\mathcal{A}$ to corrupt nodes in the old committee is no more than t, and at least $t+1$ nodes correctly send $B_i(h, j)$ to the temporary new committee. In addition, given that the number of nodes in the temporary new committee is $2t + 1$, and the maximum number of $\mathcal{A}$ to corrupt nodes in the temporary new committee is t. Therefore, based on the collision-resistance of a hash function and the negligible probability attack of a XOR function, there are at least $t + 1$ nodes correctly obtain at least $t + 1$ $B_i(h, j)$, respectively. Due to the order of $B(x, y)$ is $\langle t, 2t\rangle$, so at least $t + 1$ $B_i(x, j)$ can be reconstructed correctly.

Lemma 5. If the adversary $\mathcal{A}$ corrupts no more than t nodes in the temporary new committee, then at least $t+1$ nodes in the temporary new committee will correctly reconstruct $B'_i(x, j)$ after the stage of share proactivization.

Proof. Given that the number of nodes in the temporary new committee is $2t+1$, assuming that a $\langle t, 2t\rangle$-order 0-shared binary polynomial $Q(x, y)$ with confidentiality and integrity and the number of $\mathcal{A}$ to corrupt nodes in the old committee is no more than t. Under the premise of ensuring the confidentiality of the process of share proactivization, this process can always be executed correctly. Therefore, according to the formula $B'(x, y) = B(x, y) + Q(x, y)$, On the basis of the collision-resistance of a hash function and the negligible probability attack of a XOR function, it is ensured that at least $t+1$ temporary new committee nodes correctly reconstruct $B'_i(x, j)$, which satisfies $B'(0, 0) = B(0, 0) = sk_i$

Lemma 6. If adversary $\mathcal{A}$ corrupts no more than t nodes in the new committee, then after the full-share is distributed, at least $t+1$ new committee nodes correctly reconstruct $B'_i(h, y)$.

Proof. Given that the number of new committee nodes is n', which satisfies $n' \geqslant 2t + 1$, the maximum number of $\mathcal{A}$ to corrupt nodes in the new committee is t. Based on the Proof of Lemma 5, at least $t+1$ new committee nodes can correctly reconstruct $B'_i(h, y)$.

In summary, by Lemma 4, 5 and 6, $\mathcal{A}$ cannot prevent the new committee nodes from correctly calculating the sub-shares of the same sk_i during the phase of key share update, thereby ensuring the integrity of the DBKEM-AACS.

Correctness. The meaning of correctness in this article is that EN_i can be correctly authenticated by the smart contract and sk_i can be correctly restored. In order to prove the correctness of the DBKEM-AACS proposed in this paper, we propose the following lemmas.

Lemma 7. If $\mathcal{A}$ is semi-honest adversary, EN_i can be correctly authenticated.

Proof. Assuming that $\mathcal{A}$ is not malicious, that is, it will not carry out purposeless or meaningless attacks. In the case of based on the secure channel, it is guaranteed that the anonymous authentication process will not suffer from various active or passive attacks. Therefore, EN_i can be correctly authenticated by judging whether the equation $H(N_i) == H(H(r) \oplus (H((r) \oplus N_i)$ is established.

Lemma 8. If active attack capability of adversary $\mathcal{A}$ is t nodes at most, EN_i will have ability to correctly recover sk_i.

Proof. Based on the above proof of confidentiality and integrity, the collision-resistance of a hash function, the negligible probability attack of a XOR function, Lemma 7 and the characteristics $B'(0,0) = B(0,0) + Q(0,0) = B(0,0)$ of the formula $B'(x,y) = B(x,y) + Q(x,y)$, under the premise of ensuring that at least $t+1$ $B'_i(h,y)$ can be accurately obtain, EN_i can establish a system of equations with $t+1$ equations to acquire $B'_i(x,y)$, thus correctly recovering sk_i, which satisfies $sk_i = B'_i(0,0) = B_i(0,0)$.

In summary, by Lemma 7 and 8, $\mathcal{A}$ is not enough to affect the correct authentication of the nodes and the correct recovery of sk_i during the process of key escrow, thereby ensuring the correctness of the DBKEM-AACS.

Appendix E performance analysis

We implement our experiment in solidity 0.8.0, Python 3.7 and JavaScript VM. The contracts designed in Section 4.3 are encoded by using the Solidity language, and the Gas consumption generated by each function in all contracts is counted in the form of a table to show the calculation cost and economy cost, where numerical value is accurate to three digits after the decimal point, and using the Python language to visualize the Gas consumption of the supplementary function. In order to unify the conversion ratio between Gas and $, we will define Gas price as 1 Gas=1 Gwei and 1 ETH=10^9 Gwei. In addition, the current conversion price between Ether and $ is 1 ETH=$3556.0, which is real world costs in November 2021. However, because smart contracts have many deficiencies in programming, we may make appropriate adjustments to each contract in actual compilation to facilitate the implementation.

In table E1, we list the calculation cost and the economic cost of main functions in ACC-Contract. The contract is specifically used to calculate a node activeness. Observe that the cost of the function Iint() is relatively large (Other contracts in our paper have the same phenomenon), but the information that it has been initialized can be used over multiple times between the hosting node and the committee. In addition, in order to facilitate the design and implementation of ACC-Contract, we redefine the type of activeness in the actual encoding, because the current smart contract compiled by the solidity language does not support the assignment and modification of the float type, and can only be simply defined. Therefore, we define the original floating-point type bounded by 1 as an integer bounded by 10, what happens that if the upgrade of smart contracts overcomes the defects of floating-point type, we will redefine it as a floating-point type bounded by 1.

Table E1 Costs of ACC-Contract

Function	Caller	Gas Consumption	$ Consumption
Iint()	Blockchain node	472224+cost(Search())	1.679+ cost(Search())
Create()	Blockchain node	92283	0.328
ADefine()	Blockchain node	52746	0.188
AUpdate()	Blockchain node	31035+cost(Search())	0.110+ cost(Search())

Table E2 Costs of Search() and its supplementary functions

Function	Caller	Gas Consumption	$ Consumption
Search()	ACC-Contract	473709	1.685
SHA256() (compute 70 times)	Search()	149490	0.532

In ACC-Contract, in order to solve the problem of querying the number of related transaction and mining, we perform a simulation experiment of a function Search() for block query, and show its performance overhead in table E2, where the structure of which has a total input of 64 transactions and 6 blocks. Aiming at the problem that the judgment symbol "==" in the smart contract is not compatible with the type of string, because the type of transaction or block defined in code are all string types, we input the hash calculation result by using SHA256() to make judgement. Therefore, the function Search() also includes the performance overhead of the hash calculation using SHA256(), where the function of SHA256() is to call the library function of the SHA256 algorithm has been defined in smart contracts: Taking string type data as input to obtain the hash value of bytes32 type. Figure E1 shows how the function Search() increases over the number of blocks and the number of transactions. It can be concluded that the growth of Gas consumption after 20 transactions and 2 blocks has stabilized, which can guarantee the normal operation of ACC-Contract.

The table E3 shows the calculation cost and economic cost of main functions in CNSC-Contract, where assuming that there are a committee with 5 nodes and 4 nodes in it that need to be selected. We select the bubble sort algorithm that is stable and has a computational complexity of $O(n^2)$ to carry out the sorting process in this article, and the gas consumption of the bubble sort algorithm varies with the input size as showed in Figure E2. In view of the problem that smart contracts cannot achieve flexible deletion of node information, we set the node state value, such as $state = 0$ or $state = 1$, to determine whether a node in the PCND has been selected as a committee node.

In table E4, we list the calculation cost and economic cost of main functions in NAC-Contract. From the table, we can know that due to the existence of the function Login(), the Gas consumption of the originally designed function Create() at compile time is the function Login() at compile time, the Gas consumption of the function Grant() includes the Gas consumption of the function VerifyD(), and then the Gas consumption of the function Authenticate() includes the Gas consumption of the function VerifyH().

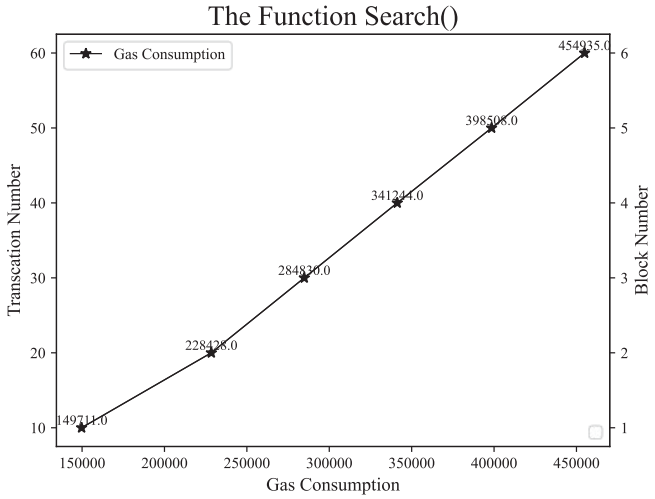

Figure E1 The Gas consumption of Search()

Table E3 Costs of CNSC-Contract

Function	Caller	Gas Consumption	$ Consumption
Iint()	Escrow Node	983780	3.498
Create()	Escrow Node	21567	0.077
Rank()	Escrow Node	114197	0.406
ECJ()	Escrow Node	349373	1.242
Return()	Escrow Node	35115	0.125

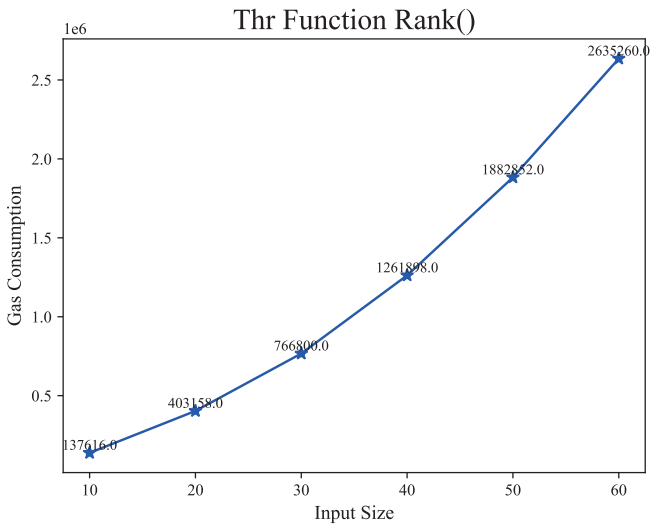

Figure E2 The Gas consumption of Rank()

The table E5 shows the calculation cost and economic cost of the supplementary functions in NAC-Contract. In order to perform the hash calculation of SHA256() accurately, we compile the function uint2str() in NAC-Contract, the function of which is to convert an integer type to a string type as the input of the function SHA256(). The Gas consumption of the function VerifyH() includes the Gas consumption of the function uint2str(). In order to facilitate the comparison of judgment equations, the function SHA256() is introduced for type conversion. Figure E3 shows the Gas consumption of the function uint2str(). It can be seen from the figure that the Gas consumption for a single input value tends to be stable.

In table E6, we list the calculation cost and economic cost of main functions in KEC-Contract. Since KEC-Contract is not a contract that is continuously executed until the final result is obtained, the correlation between its main functions in it is weaker than other contracts. Therefore, when compiling KEC-Contract, the Gas consumption of each main function includes the part

Table E4 Costs of NAC-Contract

Function	Caller	Gas Consumption	$ Consumption
Iint()	Escrow Node	1799384	6.399
Login() or Create()	Escrow Node	94462	0.336
Grant()	Escrow Node	143409	0.510
Authenticate()	Escrow Node	50593	0.180
VerifyD()	Escrow Node	27161	0.097
VerifyH()	Escrow Node	33339	0.119
Revoke()	Escrow Node	44424	0.158
Retrun()	Escrow Node	38658	0.137

Table E5 Costs of Search() and its supplementary functions

Function	Caller	Gas Consumption	$ Consumption
uint2str() (compute 1 time)	NAC-Contract	24363	0.087
SHA256() (compute 2 times)	NAC-Contract	22967+22871	0.082+0.0.081

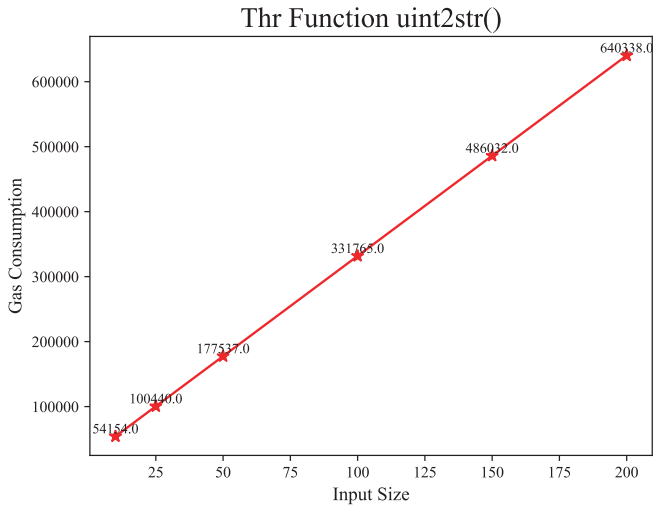

Figure E3 The Gas consumption of uint2str()

of Gas consumption of Create(). It can be seen from the table that the Gas consumption of function Update() and function CVerify() includes Gas consumption of three stages, such as share reduction, share automation, and full-share distribution. Similar to table E5, table E7 shows the calculation cost and economic cost of the supplementary functions of KEC-Contract.

Table E6 Costs of KEC-Contract

Function	Caller	Gas Consumption	$ Consumption
Iint()+2*CNSC+1*NAC (include Create())	Escrow Node	1537192+2*CNSC+1*NAC	5.466+2*CNSC+1*NAC
Distribute() (include Create()) or Create()	Escrow Node	524418	1.865
Update() (include Create())	Escrow Node	129100+115100+115100	0.495+0.409+0.409
Recover() (include Create())	Escrow Node	739589	2.630
CVerify()	Escrow Node	25369+24031+24031	0.090+0.085+0.085

Table E7 Costs of the supplementary functions of KEC-Contract

Function	Caller	Gas Consumption	$ Consumption
uint2str() (compute 1 time)	KEC-Contract	52253	0.186
SHA256() (compute 2 times)	KEC-Contract	36235	0.129

The table E8 conducts the comparison of function between our paper and other papers such as reference [1] and reference [6]. We can conclude from the table that our paper proposes a key escrow model in blockchain by combining smart contracts (SC), anony-

mous authentication (AA) and committee selection (CS) compared to the [1] and [6], meanwhile, the communication complexity (CC) in our paper is approximately equal to the [1] and less than the [6], where $t \approx n$.

Table E8 Function analysis of DBKEM-AACS

	key management	in Blockchain	SC	AA	CS	CC
In [6]	NO	NO	NO	NO	NO	$O(n^4)$
In [1]	YES	YES	NO	NO	NO	$O\left(n^2\right) \sim O\left(n^3\right)$
In our aper	YES	YES	YES	YES	YES	$O(n^3)$

To sum up, the Gas consumption of the function Iint() in each contract is relatively large, but on the whole, all values after the initial execution can be used multiple times. Therefore, the average Gas consumption per time is acceptable. In addition, why the Gas consumption of other main functions in KEC-Contract except the function Iint() is relatively high is because the Gas consumption of the function Create() is also included.

References

1 Maram S K D, Zhang F, Wang L, et al. CHURP: Dynamic-Committee Proactive Secret Sharing. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, 2010. 2369-2386

2 Dorsala M R, Sastry V N, Chapram S. Fair payments for verifiable cloud services using smart contracts. Computers & Security, 2020, 90: 101712

3 Lim D, Lee J W, Gassend B, et al. Extracting secret keys from integrated circuits. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2005, 13: 1200-1205

4 Chatterjee U, Mukhopadhyay D, Chakraborty R S. 3PAA: A Private PUF Protocol for Anonymous Authentication. IEEE Transactions on Information Forensics and Securitys, 2021, 16: 756-769

5 Krzywiecki L. Anonymous Authentication Scheme Based on PUF. In: Proceedings of International Conference on Information Security and Cryptology, Seoul, 2015: 359-372

6 Schultz D A, Liskov B, Liskov M D. MPSS: Mobile Proactive Secret Sharing. ACM Transactions on Information and System Security, 2010, 13: 34:1-34:32