Front. Comput. Sci., 2024, 18(1): 181802
https://doi.org/10.1007/s11704-022-2288-x

RESEARCH ARTICLE

Non-interactive SM2 threshold signature scheme with identifiable abort

Huiqiang LIANG, Jianhua CHEN (✉)

School of Mathematics and Statistics, Wuhan University, Wuhan 430072, China

Abstract A threshold signature is a special digital signature in which the N-signer share the private key x and can construct a valid signature for any subset of the included t-signer, but less than t-signer cannot obtain any information. Considering the breakthrough achievements of threshold ECDSA signature and threshold Schnorr signature, the existing threshold SM2 signature is still limited to two parties or based on the honest majority setting, there is no more effective solution for the multiparty case. To make the SM2 signature have more flexible application scenarios, promote the application of the SM2 signature scheme in the blockchain system and secure cryptocurrency wallets. This paper designs a non-interactive threshold SM2 signature scheme based on partially homomorphic encryption and zero-knowledge proof. Only the last round requires the message input, so make our scheme non-interactive, and the pre-signing process takes 2 rounds of communication to complete after the key generation. We allow arbitrary threshold $t \leqslant n$ and design a key update strategy. It can achieve security with identifiable abort under the malicious majority, which means that if the signature process fails, we can find the failed party. Performance analysis shows that the computation and communication costs of the pre-signing process grows linearly with the parties, and it is only 1/3 of the Canetti's threshold ECDSA (CCS'20).

Keywords SM2 signature, secure multi-party computation, threshold signature, UC-secure, dishonest majority

1 Introduction

Due to the rapid development of Bitcoin and other cryptocurrencies, distributed key management has become increasingly important. Especially in cryptocurrency, it will cause irreversible economic losses if the user's private key is lost or stolen. Bitcoin initially solved this problem with multi-signature, but multi-signature has limited flexibility and does not support arbitrary and complex access structures. In contrast, threshold signature is a classic concept in cryptography introduced by Desmedt and Frankel [1,2]. It has more general properties than multi-signature. Therefore the threshold signature has gradually moved from theory to practice. We can get a common signature and do not need to count the number of signatures. It will reduce the transaction fee and time, meanwhile, we can set up more flexible access strategies, party joining and exit methods, and key update strategies relying on relevant research on threshold cryptography.

SM2 signature as an international standard (ISO/IEC 14888-3:2018) adapted by ISO (International Organization of Standardization) [3]. It has the same computing speed and security as ECDSA (Elliptic Curve Digital Signature Algorithm).

Unlike other signature schemes, such as RSA, ElGamal, or Schnorr [4], they can directly construct their threshold signatures. SM2 and ECDSA have resisted efficient threshold signatures because the additive sharing ab needs to be computed without reconstructing the secret information a and b. To solve this problem, MacKenzie and Reiter [5] use AHE (additive homomorphic encryption) to generate signatures for two-party in the honest majority setting. Then the two-party threshold ECDSA was proposed by Gennaro [6] using AHE techniques again, and it was improved and optimized by Lindell [7], meanwhile, Doerner [8] implements two-party threshold signatures with oblivious transfer technology. They all focus on the technique of security to turn the product of two numbers into an addition, such as Paillier encryption, oblivious transfer, Shamir secret share, Beaver's triple, garbled circuits, etc.

In the case where these techniques can be matured for the two-party threshold signature, Gennaro and Goldfeder [9] and others have extended to multi-party threshold ECDSA, which n-party to n-party. Pettit [10] uses Shamir secret sharing to fulfill multi-party case that is fast and easy to implement, but it still requires a weak security assumption of the honest majority. Canetti [11] makes the threshold ECDSA non-interactive, UC-secure, and security with identifiable aborts under the work of Gennaro [9] and Lindell et al. [12]. However, compared with the breakthrough of the threshold ECDSA, Shang [13] proposed the threshold SM2 scheme under the honest majority based on secret sharing, and Zhang [14] proposed a two-party threshold SM2 signature based on

Received May 25, 2022; accepted October 10, 2022

E-mail: chenjh_ecc@163.com

the Paillier encryption. Up to now, the effective multi-party threshold SM2 signature is still unfinished work.

1.1 Challenges and key technology

So far, there have been some suitable technologies to compute the additive sharing of ab [15]:

1) HE (homomorphic encryption).
2) OT (oblivious transfer).
3) SS (secret share).
4) GC (garbled circuits).

Among them, HE includes PHE (partially homomorphic encryption) and FHE (fully homomorphic encryption) [16]. Then we can build the multi-party threshold SM2 signature.

The challenges in this process are:

1) How to ensure that interact messages do not reveal any secret information, except for the build signature.
2) How to achieve security with identifiable abort.
3) Balance the security and efficiency of the protocol.

The key technologies of our soluation are:

1) Internal interact via PHE's ciphertext or nonce-marked message to prevent disclosure of secret information.
2) Using the zero-knowledge proof and the commitment function to verify whether each process of each party is executed according to the protocol to identify the fails party.
3) The pre-signing process is introduced in the threshold SM2 signing process, which makes the online stage of signature highly efficient.

1.2 Protocol overview

Here we briefly describe the basic methods and principles of the protocol. The main idea is derived from the work of Canetti etc. [9,11,12].

Firstly, the protocol contains two core steps.

1) The additive sharing of the public key.
2) The additive sharing of the random number.

Secondly, we modify the public-private key pair from d and dG to x and $(x^{-1}-1)G$ in Table 1 to improve the efficiency of the threshold SM2 signature, then r is unchanged, but $s=x(k+r)-r \bmod n$.

Thirdly, the public key $(x^{-1}-1)G$ is obtained by additive sharing γx and γG. At the same time, run verifiable serect sharing to turn the secret key d into (t,n)-share. Then, only t-party are required to share the random number and compute the additive sharing of signature. The main part of the protocol is shown in Fig. 1.

The structure of this paper is as follows: Section 2 gives the preliminaries, including the definitions of SM2 signature, Paillier encryption, multiplicative-to-additive share with check, verifiable secret sharing, and the ideal zero-knowledge proof functionality. Section 3 gives the ideal/real paradigm. Then Section 4 gives the protocol of non-interactive threshold SM2 signature. Section 5 constructs a simulator to prove the security of the threshold SM2 signature scheme. Its performance analysis is in Section 6 and conclusions and further work is in Section 7. At last, we give the relevant zero-knowledge proof protocol in the appendix.

Table 1 Modify the pk and sk of SM2

Sym.	SM2	modified SM2
sk	d_A	x
pk	d_AG	$(x^{-1}-1)G$
x_1	$(x_1,y_1)=kG$	$(x_1,y_1)=kG$
e	$H_v(Z_A\|\|M)$	$H_v(Z_A\|\|M)$
r	$e+x_1 \bmod n$	$e+x_1 \bmod n$
s	$(1+d_A)^{-1}(k-rd_A) \bmod n$	$x(k+r)-r \bmod n$

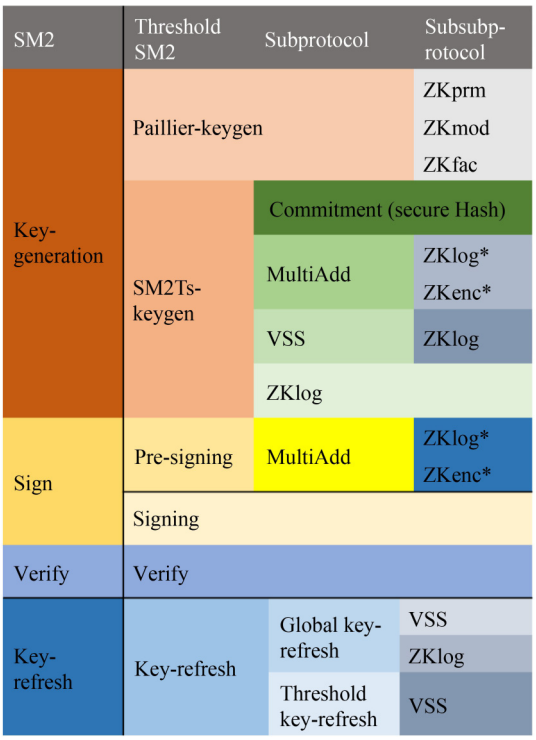


Fig. 1 Threshold SM2 signature scheme

2 Preliminaries

In this paper, the $x_1,x_2,...,x_n$ are the additive sharing of x denoted as $[x]$, where the $x=x_1+x_2+\cdots+x_n$. we use $H_v(\cdot)$ denote the Hash function, such as $H:\{0,1\}^* \to \{0,1\}^{256}$. $\phi(\cdot)$ denotes the Euler's totient function. $P(\cdot)$ is polynomial like $P(x)=\sum_{i=0}^{n-1}a_ix^i$, and $\lambda_i=\prod_{j=0,j\neq i}^{n-1}(x_i-x_j)^{-1}(x-x_j)$. κ denotes security parameters and $negl(\cdot)$ denotes the negligible probability.

2.1 SM2 signature algorithm

The SM2 signature algorithm was released by the State Cryptology Administration in 2010 and become China's national encryption standard (GB/T32918.2-2016) in 2016 [17] . We briefly describe it.

Alice has the private key d_A, additional information Z_A and message M. Chooses $k\in[1,n-1]$ randomly and computes $(x_1,y_1)=kG$, $e=H_v(Z_A\|M)$, $r=(e+x_1) \bmod n$, $s=(1+d_A)^{-1}(k-rd_A) \bmod n$, then sends the signature $\sigma=(r,s)$ to Bob.

Bob has the massage M', additional information Z_A, Alice's public key $P_A=d_AG$ and the signature $\sigma'=(r',s')$. The verification is to Compute $e'=H_v(Z_A\|M')$, $t=(r'+s') \bmod n$, $(x_1',y_1')=s'G+tP_A$, $R=(e'+x_1') \bmod n$, then check whether $R=r'$ is true. If it is true, the verification passed, otherwise, the verification failed.

2.2 Paillier encryption

Paillier encryption was first proposed by Paillier [18] in 1999, and many people did optimizations after that. The Paillier encryption in this paper is defined as follows:

1) Key generation: On the input of security parameter κ, generate two large equal length $\kappa/2$ prime numbers p,q as the private key, set $N=pq$ as the public key. Output (N,p,q).

2) Encryption: The input message $m\in Z_N$, and choose $r\in Z_N^*$ randomly. Output c, where

$$c=ENC_{pk}(m,r)=(1+N)^m r^N \bmod N^2. \tag{1}$$

3) Decryption: The input ciphertext $c\in Z_{N^2}$, compute $\mu=\phi(N)^{-1} \bmod N$. Output M', where

$$M'=DEC_{sk}(c)=\left(\frac{c^{\phi(N)} \bmod N^2-1}{N}\right)\mu \bmod N. \tag{2}$$

4) Additive homomorphism: Given two ciphertexts $c_1=ENC_{pk}(m_1)\in Z_{N^2}, c_2=ENC_{pk}(m_2)\in Z_{N^2}$, $k\in Z_N$, it holds that

$$DEC_{sk}(c_1\times c_2 \bmod N^2)=m_1+m_2 \bmod N. \tag{3}$$

$$DEC_{sk}(k\cdot c_1 \bmod N^2)=km_1 \bmod N. \tag{4}$$

2.3 Multiplicative-to-additive share with check

MtAwc (Multiplicative-to-additive share with check) is an improved version of MtA (Multiplicative-to-additive share). Proposed by Gennaro [9] and modified by Tymokhanov [19]'s attack. The MtAwc is to convert the product ab of the two private inputs a and b into $\alpha+\beta$.

In more detail [20]. The MtAwc takes the input of the unique identifier d_i, Alice has the secret value a, auxiliary input $B=bG$, Bob has the secret value b. we assume Bob has checked that Alice's Paillier public key is in the correct size.

1) Alice computes $c_A=ENC_A(a,v)$, sends c_A to Bob.

2) Alice sends $(prove,ZKenc,...,c_A;a,v)$ to $ZKenc$.

3) When receiving $(verify,ZKenc,...,c_A,\beta')$ from $ZKenc$ and c_A from Alice, Bob verifies β'.

4) Bob chooses a random number β in correct size, sets $\beta'=-\beta$.

5) Bob computes $c_B=b\cdot c_A\times ENC_A(\beta',u)$, sends c_B to Alice.

6) Bob sends $(prove,ZKlog*,...,(c_A,B,G);a,u)$ to $ZKlog*$.

7) When receiving $(verify,ZKlog*,...,(c_A,B,G),\beta')$ from $ZKlog*$ and c_B from Bob.

8) Alice verifies β' and computes $\alpha=DEC_A(c_B)$.

The relevant zero-knowledge proof is in Section 2.5. We emphasize that the range proofs and input check are essential for Paillier encryption, and inappropriate inputs can make paillier encryption insecure.

2.4 Verifiable secret sharing

The Shamir secret sharing scheme is based on a polynomial in that n distinct points correspond to a unique $n-1$ order polynomial. At the same time, the verifiable property is that each party can check whether a unique secret is shared. We use Feldman's VSS (verifiable secret sharing) scheme [21,22]. This scheme immediately reports abort and the party who had an exception makes our protocol identifiable abort. Gennaro and Lindell [9,23] also use this scheme in their threshold signature.

The VSS takes the input of the unique identifier d_i, secret value x_i for every $P_i\in P$ where $|P|=n$ and threshold m then output y_i for each party that satisfied (m,n)-share.

1) Each party P_i random choose $a_{i,1},...,a_{i,m-1}\leftarrow Z_N^*$, set $a_{i,0}=x_i$, compute $A_{i,0}=a_{i,0}G,...,A_{i,m-1}=a_{i,m-1}G$.

2) Compute $y_{i,j}=\sum_{k=0}^{m-1}(d_j)^k a_{i,k}, j=1,...,n$.

3) Compute $C_{i,j}=ENC_j(y_{i,j}),\ j=1,...,n$.

4) Boardcast $A_{i,0},...,A_{i,m-1},C_{i,j}$ to all parties.

5) When receiving $A_{j,0},...,A_{j,m-1},C_{j,i}$ from all P_j,

6) Compute $y'_{j,i}=DEC_i(C_{j,i}), j=1,...,n$.

7) Verify $y'_{j,i}G=\sum_{k=0}^{m-1}(d_i)^k A_{j,k}, j=1,...,n, j\neq i$.

8) When the verification passed, set $y_i=\sum_{j=1}^{n}y'_{j,i}$.

At the end of the protocol, the (m,n)-addition share of x consists of the appropriate lagrangian coefficients λ_i with y_i.

2.5 Zero-knowledge proof

In the zero-knowledge proof, the prover attempts to convince the skeptical verifier that a statement is true, except for a negligible probability of error, the verifier should be convinced if and only if the statement is true, and except nothing should be learned beyond the validity of the assertion [24].

We design an ideal zero-knowledge proof function for proving NP-relations in Fig. 2. The specific protocol is shown in the Appendix. It can be turned into a non-interactive zero-knowledge proof protocol using the fiat-shamir heuristic.

We summarize here the NP-relations that needs to be proofed.

$$prm=\{(N,s,t;v)|s=t^v \bmod N\},$$

$$mod=\left\{(N;p,q)\middle|\begin{matrix}N=pq,gcd(N,\phi(N))=1,\\ p,q=3 \bmod 4\end{matrix}\right\},$$

$$fac=\{(l,N_0;p,q)|p,q<\pm 2^l\sqrt{N_0}\},$$

$$enc=\left\{(l,N_0,K;k,\rho)\middle|\begin{matrix}k\in\pm 2^l,\\ K=(1+N_0)^k\rho^{N_0} \bmod N_0^2\end{matrix}\right\},$$

$$log=\{(X,Y;x)|X=xY\}$$

$$enc*=\left\{\begin{pmatrix}l,l',N_0,N_1,\\ D,C,Y,X;\\ x,y,\rho_x,\rho_y\end{pmatrix}\middle|\begin{matrix}x\in\pm 2^l, y\in\pm 2^{l'}, X=xG,\\ Y=(1+N_1)^y\rho_y{}^{N_1} \bmod N_1^2,\\ D=C^y(1+N_0)^x\rho_x{}^{N_0} \bmod N_0^2\end{matrix}\right\},$$

$$log*=\left\{\begin{pmatrix}l,N_0,\\ C,X,G;\\ x,\rho\end{pmatrix}\middle|\begin{matrix}x\in\pm 2^l, X=xG,\\ C=(1+N_0)^x\rho^{N_0} \bmod N_0^2\end{matrix}\right\},$$

Ideal zero-knowledge proof ZK_{Rel}

1) Upon receiving $(prove, ZK_{Rel}, x; w)$;
Set $\beta=Rel(x;w)$, send $(verify, ZK_{Rel}, x, \beta)$.

Fig. 2 Ideal zero-knowledge proof functionality

$$mul = \left\{ \begin{pmatrix} N, E, D, C; \\ x, \rho_x, \rho_y \end{pmatrix} \middle| \begin{matrix} (1+N)^x \rho_x{}^N = E, \\ C = D^x \rho_y{}^N \bmod N^2 \end{matrix} \right\}.$$

3 Ideal/real paradigm

Now we introduce the security goal of our protocol using the concept of the secure multi-party computation under the malicious model that defines security with identifiable abort [25]. For more details, please refer to the Foundations of Cryptography-Goldreich [26]. Informally, Suppose a malicious adversary can gain no more information by attacking the protocol in the real world than by attacking an ideal computation (an uncorrupted trusted party assists the computation) in the ideal world. In that case, the protocol can be at least as secure as the ideal function F under the malicious model.

Definition 1 An n-party functionality is a random process that maps vectors of n inputs to vectors of n outputs and can be computed in the polynomial- time. Set $f:(\{0,1\}^*)^n \to (\{0,1\}^*)^n$ as an n-party functionality.

In the real world, the n-party protocol $\Pi = (P_1, ..., P_n)$ is a probabilistic polynomial-time interactive Turing machine of n-tuple. P_i input $x_i \in \{0,1\}^*$ and random number $r_i \in \{0,1\}^*$, the input length is the security parameter κ. An adversary A is another interactive Turing machine describing the behavior of the corrupted parties, it begins execution with an input that contains the identities of the corrupted parties and their private inputs (possibly has additional auxiliary inputs). Each party executes the protocol in rounds in a synchronous network. Each round consists of two phases, a receive phase (where parties receive messages from other parties) and a send phase (where parties send messages to other parties).

The parties communicate in each round via a broadcast channel or a fully connected peer-to-peer network. In the authenticated channels, the adversary can only read messages sent between two honest parties but cannot modify them.

All honest parties follow the instructions of the prescribed protocol throughout the execution of the protocol, while the corrupted parties receive instructions from the adversary. An adversary is called malicious, which means it can instruct the corrupt parties to deviate from the protocol in arbitrary ways. At the end of the protocol execution, the honest parties output the output prescribed by the protocol. The corrupted parties do not output any information. The adversary A outputs its computation view (arbitrary) function, including the views of corrupted parties. The view of a party in the protocol execution consists of its random numbers, inputs, and massage it sees during execution.

Definition 2 Let $\Pi = (P_1, ..., P_n)$ be the n-party protocol, and the set of corrupted parties controlled by the adversary A is $I \subseteq [n]$. The joint execution of Π under(A, I) in the real model, auxiliary input z, security parameter κ, and input $x = (x_1, x_2, ..., x_n)$, denoted $REAL_{\Pi,I,A(z)}(x,\kappa)$, is defined as $P_1, ..., P_n$ and $A(z)$ resulting the interaction with the protocol, including messages computed by the adversary A for each corrupted party P_i and the messages computed by Π for each honest party P_j.

The ideal computation with identifiable abort of the n-party functionality f with $(P_1, ..., P_n)$ input $x = (x_1, x_2, ..., x_3)$ and a set of parties(corrupted parties $I \subseteq [n]$) that the ideal-model adversary A was controlling, proceeds via the following steps.

Sending inputs to the trusted party: The honest party P_i sends x_i inputs to the trusted party. The adversary A sends any input of the corrupted party to the trusted party and lets x_i' be the actual input of P_i.

Early abort: Adversary A chooses a corrupted party $i^* \in I$ and sends the message (abort,i^*) to the trusted party to abort the computation, in which case the trusted party sends (abort,i^*) to all parties and halts.

The trusted party replies to the adversary A: The trusted party computes $(y_1, ..., y_n) = f(x_1', ..., x_n')$ and sends y_i to each corrupted party P_i, i.e., $i \in I$.

Late abort: Adversary A chooses a corrupted party $i^* \in I$ and sends the message (abort,i^*) to the trusted party to abort the computation, in which case the trusted party sends (abort,i^*) to all parties and halts. Otherwise, the computation continues.

The trusted party replies to the honest parties: The trusted party sends y_i to each party P_i, $i \notin I$.

Outputs: The honest parties always output the message received from the trusted party, and the corrupt parties do not output anything. Adversary A outputs an arbitrary function of the message received by its auxiliary inputs and the corrupted parties from the trusted party.

Definition 3 N-party functionality $f:(\{0,1\}^*)^n \to (\{0,1\}^*)^n$, and $I \subseteq [n]$ denoted the set of corrupted parties. The joint execution of Π under (A, I) in the ideal model, on auxiliary input z, the security parameter κ, and input $x = (x_1, x_2, ..., x_n)$, denoted $IDEAL_{f,I,A(z)}^{id-abort}(x,\kappa)$, is defined as $P_1, P_2, ..., P_n$ and $A(z)$ resulting from the above described ideal process.

Definition 4 N-party functionality $f:(\{0,1\}^*)^n \to (\{0,1\}^*)^n$, Π is a probabilistic polynomial-time protocol for computing f. The protocol $\Pi(\delta, l)$ securely computes f with abort: if for every real model adversary A in probabilistic polynomial time, there exists an (expected) polynomial-time ideal model adversary S, at each $I \subseteq [n]$ of size at most l, it holds that

$$REAL_{\Pi,I,A(z)}(x,\kappa) \equiv_c^{\delta} IDEAL_{\Pi,I,S(z)}^{id-abort}(x,\kappa). \tag{5}$$

If δ is negligible, then we say that the Π protocol computes f is l-securely with abort under the assumption of computational security.

Theorem 1 (UC Secure Combination [27]) Let $l < n$, $m \in N$, $f_1, ..., f_m$, g are n-party functionality, we say that Π is n-party protocol l-securety computes the function g in $(f_1, ..., f_m)$-hybrid model, where the n-party protocol $\rho_1, ..., \rho_m$ are l-securely computes the $f_1, ..., f_m$ and no more than one ideal function is called at each round. Then Π is the n-party protocol l-securely evaluates g under $\rho_1, ..., \rho_m$.

4 Non-interactive threshold SM2 signature protocol

A map $x = (1 + d_A)^{-1} \bmod n$ from $[1, n-2]$ to $[2, n-1]$ is a bijection if n is a prime. Keeping d_A secret is equivalent to

keeping x secret, and using x as the "private key" does not affect the security properties of SM2 signature algorithm. So we modify the original algorithm of SM2 in Table 1 to make it more suitable for the threshold scheme.

The Threshold SM2 master protocol (Fig. 3) consists of four parts: Key-generation, Pre-signing, Signing and Key-refresh.

The Paillier-keygen step (Fig. 4) of the Key-generation generates the public and private keys of Paillier encryption for each party.

The SM2Ts-keygen step (Fig. 5) of the Key-generation is n parties cooperate to generate a "public key" $x^{-1}G-G$ and broadcast. Meanwhile, each party store its own "private key" $[x]$ and (t,n)-share value y_i. The more details are: each party chooses a random number as $[\gamma]$ and $[x]$, computes $[\gamma G]$, invokes the VSS step to obtain the (t,n)-share y_i, invokes the MultiAdd step to compute $[\gamma x]$, then recovers γx and computes $(\gamma x)^{-1}[\gamma G]$ as $[x^{-1}G]$, and then recovers $x^{-1}G$ and compute $X = x^{-1}G-G$.

The Pre-signing step (Fig. 6) is for t partis to choose a random number as $[k]$, compute $[kG]$, and invoke the MultiAdd step to compute kG and $[kx]$.

The Signing step (Fig. 7) is after the party receives the message msg to be signed, uses kG and $[kx]$ of the Pre-signing step to compute it's partial signatures $(r,[s])$ directly, and broadcast, where $[s] = [kx] + r[x]$. Then each party gets (r,s) and them verifies whether the signature $(r, s-r \bmod n)$ is a valid signature or not for the message msg.

4.1 MultiAdd and VSS protocols

There are two sub-protocols in our protocol. One is the MultiAdd step, we design it (Fig. 8) to multi-party invoking the MtAwc, and each party inputs their secret values a_i, b_i and gets the output c_i such that $c_1+c_2+\cdots+c_n = (a_1+a_2+\cdots+a_n)(b_1+b_2+\cdots+b_n)$. Then $c_1, c_2, ..., c_n$ can be obtained by calling the additive sharing of $[ab]$.

Another is the VSS step (Fig. 9), which takes the input of the unique identifier d_i and secret value x_i for every $P_i \in P$ where $|P| = n$ and threshold m then output y_i for each party that satisfied (m,n)-share. We add zero-knowledge proofs and

Threshold SM2 master protocol

Key-Generation: When receiving the input $(keygen, tig, t, i)$ from P_i, do:

1) Run the Paillier-keygen step and obtain $(tig, rtig, i, N, s, t)$, (p_i, q_i).
2) Upon receiving $(sm2tskeygen, tig, rtig, t, i)$ from P_i, run the SM2Ts-keygen step and obtain $(tig, \rho, i, A, X, (X)_i, (Y)_i, (x_i, y_i)$, set $sid = (tig, rtig, \rho, N, s, t, X, (X)_i, (Y)_i)$ and standby.

Pre-signing: Initialize $S = \varnothing$, on the input $(presign, sid, i)$ from P_i, add P_i to S, do:

1) Clear all Pre-signing data $(tig, ...)$.
2) If $|S| >= t$, random choose t parties from S record S', run the Pre-signing step, else output "not enough parties to signing." and standby.
   - Upon obtaining the output (sid, i, R), (w_i, χ_i) and standby.

Signing: On the iuput $(sign, sid, i, msg)$ from P_i, do:

1) Run the Signing step.
2) Upon obtaining the output $(signature, sid, msg, \sigma)$ and standby.

Key-refresh: Initialize $Q = \varnothing$, on the input $(keyrefresh, aux, sid, i)$ from P_i, add P_i to Q, do:

1) If $|Q| >= t$, run aux key-refresh step.
   Upon obtaining the output and standby.
2) Else output "not enough parties to aux key-refresh." and standby.

Fig. 3 Threshold SM2 master protocol

Paillier-keygen step
Round 1.
On the input $(keygen, tig, t, i)$ from P_i, interpret $tig = (..., P, \kappa, a, b, n, G, ID_P, ENTL_P)$, where ID_P is $(d_1||d_2||...||d_n)$, and $d_i \in Z_n^*$ is the unique identifier of P_i, do:
- Choose $rtig_i = \{0,1\}^\kappa$, two large equal length 4κ-bits primes (p_i, q_i) randomly, set $N_i = p_i q_i$.
- Choose $r \leftarrow Z_{N_i}^*, \nu \leftarrow Z_{\phi(N_i)}$ randomly, set $t_i = r^2 \bmod N_i$ and $s_i = t_i^\nu \bmod N_i$.
Send $(prove, ZK_{prm}, (tig, i), (N_i, s_i, t_i); \nu)$ to ZK_{prm}, $(prove, ZK_{mod}, (tig, i), N_i; (p_i, q_i))$ to ZK_{mod},
$(prove, ZK_{fac}, (tig, i), N_i; (p_i, q_i))$ to ZK_{fac}.
Send $(tig, i, rtig_i, N_i, t_i, s_i)$ to all P_j.
Output:
1. When receiving $(verify, ZK_{prm}, (tig, j), N_j, s_j, t_j, \beta_j)$ from ZK_{prm}, $(verify, ZK_{mod}, (tig, j), N_j, \beta'_j)$ from ZK_{mod}, $verify, ZK_{fac}, (tig, j), N_j, \hat{\beta}'_j)$ from ZK_{fac}. If some $\beta_j = 0$ or $\beta'_j = 0$ or $\hat{\beta}'_j = 0$, report the failed party and abort.
When receiving $(tig, j, rtig_j, N_j, t_j, s_j, \hat{\psi}_j, \psi_j, \phi_j)$ from P_j, do:
- Verify $N_j \geq 2^{8\kappa}$, set $rtig = rtig_1 + rtig_2 + ...$
Output $(tig, rtig, i, N = (N_j), s = (s_j), t = (t_j))$.
Errors. Report the failed party and halt when the verification step fails.
Stored state. Store $(tig, rtig, N, s, t)$, and (p_i, q_i).

Fig. 4 Paillier-keygen step

SM2Ts-keygen step
The stored state of P_i is $N_i = p_i q_i, p_i, q_i$.
Round 1.
On the input $(sm2tskeygen, tig, rtig, t, i)$ from P_i, interpret $tig = (..., P, params)$, do:
- Choose $x_i, \gamma_i \leftarrow Z_N^*$ randomly, set $X_i = x_i G$, $\Gamma_i = \gamma_i G$.
- Choose $\rho_i, u_i \leftarrow \{0,1\}^\kappa$ randomly, set $V_i = H(tig, rtig, i, X_i, \Gamma_i, \rho_i, u_i)$.
- Send $(tig, rtig, i, V_i)$ to all P_j.
Round 2.
When all (tig, j, V_j) is received, send $(tig, i, X_i, \Gamma_i, \rho_i, u_i)$ to all P_j.
Round 3.
1. When all $(tig, j, X_j, \Gamma_j, \rho_j, u_j)$ is received, do:
- Verify $H(tig, rtig, j, X_j, \Gamma_j, \rho_j, u_j) = V_j$.
2. When all P_j verifications are passed, set $\rho = \rho_1 + \rho_2 + ...$, do:
- Run VSS-$(tig, rtig, \rho, X_i, x_i, t, n)$ step, obtain the output (tig, i, A, Y_i), and the stored state is (A, Y_i), (x_i, y_i).
- Run MultiAdd-$(tig, rtig, \rho, X_i, \Gamma_i, x_i, \gamma_i)$ step, obtain the output (tig, i, Γ), and the stored state is $(x_i, \gamma_i, \delta_i)$.
- Set $\Delta_i = x_i \Gamma$. Send $(prove, ZK_{log}, (tig, rtig, \rho, i), (Y_i, G); y_i)$, $(prove, ZK_{log}, (tig, rtig, \rho, i), (\Delta_i, \Gamma); x_i)$ to ZK_{log}.
Send $(tig, i, Y_i, \Delta_i, \psi_i, \psi_i', \delta_i)$ to all P_j.
Output:
When receiving $(verify, ZKlog, (tig, rtig, \rho, j), (Y_j, G), \beta_j)$, $(verify, ZKlog, (tig, rtig, \rho, j), (\Delta_j, \Gamma), \beta_j')$ from ZK_{enc*}.
If some $\beta_j = 0$ or $\beta_j' = 0$, report the failed party and abort. When all $(tig, j, Y_j, \Delta_j, \psi_j, \psi_j', \delta_j)$ is received, do:
- Set $\delta = \sum \delta_j \bmod n$, verify $\delta G = \sum \Delta_i$, if this verification error, run the SM2Ts checker.
- Set $X = \delta^{-1} \Gamma - G$, and output (tig, i, X).
Clear all data except stored state.
Errors. Report the failed party and halt when the verification fails.
Stored state. Store $(tig, ..., A, X, (X)_i, (Y)_i)$, (p_i, q_i, x_i, y_i).
SM2Ts checker: Check the MultiAdd step data.
1. Send $(prove, ZK_{enc*}, (tig, rtig, \rho, i), E_{j,i}, ...)$, $j \neq i$ to ZK_{enc*}.
2. Compute $U_i = ENC_i(a_i b_i)$ and send $(prove, ZK_{mul}, (tig, rtig, \rho, i), (G_i, F_i, U_i); ...)$ to ZK_{mul}.
3. Send $(prove, ZK_{enc}, (tig, rtig, \rho, i), U_i \times \prod_{j \neq i}(E_{i,j} \times D_{j,i}); b_i a_i + \sum_{j \neq i}(\alpha_{i,j} + \beta_{i,j}))$ to ZK_{enc}.

Fig. 5 SM2Ts-keygen step

Pre-signing step
The stored state of P_i is $N_i = p_i q_i, p_i, q_i, y_i$.
Round 1.
On the input $(presign, sid, i)$ from P_i, interpret $sid = (..., \kappa, a, b, n, G, ID_P, ENTL_P, rtig, X, N, s, t)$, do:
- Choose $k_i \leftarrow Z_N^*$, set $\Lambda_i = k_i G$.
- Compute $w_i = \lambda_i y_i$, where $\lambda_i = \prod_{k=0, k \neq i}^{t-1} (d_i - d_k)^{-1}(-d_k)$.
- Verify $A = \sum_{j=1}^{t} W_j$, where $W_i = \lambda_j Y_j$, if this verification error, run presigning checker.
- Run MultiAdd-$(sid, W_i, \Lambda_i, w_i, k_i)$ step, obtain the output (sid, i, R), and the stored state is (w_i, k_i, χ_i).
Then output (sid, i, R), clear all data except stored state.
Errors. Report the failed party and halt when the verification fails.
Stored state. Store $(sid, ..., R)$ and $(p_i, q_i, x_i, y_i, w_i, \chi_i)$.
Presigning check: Check the VSS step data,
1. Check $Y_i = \sum_{j=1}^{n}(X_j + \sum_{k=1}^{m}(d_i)^k A_{j,k}), i = 1, .., n$.
2. Compute $C_i = \sum_{j=1}^{n} C_{j,i}$, send $(prove, ZK_{log*}, sid, (C_i, Y_i, G); ...)$ to ZK_{log*}.
3. Compute $Y_{i,j} = X_i + \sum_{k=1}^{m-1}(d_j)^k A_{i,k}$, send $(prove, ZK_{log*}, sid, (C_{i,j}, Y_{i,j}, G); ...)$ to ZK_{log*}.

Fig. 6 Pre-signing step

Signing step
Round 1.
On the input $(sign, sid, i, msg)$ from P_i, if (sid, i, R) and (w_i, χ_i) are recorded, do:
- Compute $Z = H(ID_P || ENTL_P || P || a || b || x_G || y_G || x_X || y_X)$, $e = H(Z || msg)$, set $r = (R_{x-axis} + e) \bmod n$.
- Compute $s_i = \chi_i + w_i r \bmod n$, send (sid, i, s_i) to all P_j.
Output:
When all (sid, j, s_j) is received, set $s = \sum s_j - r \bmod n$, do:
- Verify the $\sigma = (r, s)$ is a valid signature of msg and X, if this verification error, run the Signing checker,
Output $(signature, sid, m, \sigma)$, clear (sid, i, w_i, χ_i) from memory.
Errors. Report the fails party and halt when the verification failed.
Signing checker: Check the MultiAdd step data.
1. Send $(prove, ZK_{enc*}, sid, E_{j,i}, ...)$, $j \neq i$ to ZK_{enc*}.
2. Compute $U_i = ENC_i(a_i b_i)$ and send $(prove, ZK_{mul}, sid, (G_i, F_i, U_i); ...)$ to ZK_{mul}.
3. Send $(prove, ZK_{enc}, sid, U_i \times \prod_{j \neq i}(E_{i,j} \times D_{j,i}) \times r \cdot F_i; b_i a_i + \sum_{j \neq i}(\alpha_{i,j} + \beta_{i,j}) + w_i r)$ to ZK_{enc}.

Fig. 7 Signing step

MultiAdd-$(aux, A_i, B_i, a_i, b_i)$ **step**
The stored state of P_i is $N_i = p_i q_i, p_i, q_i$.
Round 1.
On the input $(multiadd, aux, i)$ from P_i, interpret $aux = (..., P, params)$, do:
- Choose $v_i \leftarrow Z^*_{N_i}$ randomly, set $G_i = ENC_i(a_i, v_i)$.
Send $(prove, ZK_{log*}, (aux, i), (G_i, A_i, G); a_i, v_i)$ to ZK_{log*}.
Send (aux, i, A_i, B_i, G_i) to all P_j, if A_i and B_i already know, ignore A_i and B_i.
Round 2.
When receiving $(verify, ZKlog*, (aux, j), (G_j, A_j, G), \beta_j)$ from ZK_{log*}. If some $\beta_j = 0$, report the failed party and abort, when receiving (aux, j, A_j, B_j, G_j) from P_j, do:
- Set $B = \sum B_i$, choose $f_{i,j}, g_{i,j} \leftarrow Z_{N_j}, \beta_{i,j} \leftarrow J$ randomly for every $j \neq i$, do:
- Compute $E_{j,i} = b_i \cdot G_j \times ENC_j(-\beta_{i,j}, f_{i,j})$, $D_{j,i} = ENC_i(\beta_{i,j}, g_{i,j})$.
Send $(prove, ZK_{enc*}, ..., (E_{j,i}, G_j, D_{j,i}, B_i); (b_i, \beta_{i,j}, f_{i,j}, g_{i,j}))$ to ZK_{enc*}.
Send $(aux, i, E_{j,i}, D_{j,i})$ to P_j.
Output:
1. When receiving $(verify, ZK_{enc*}, ..., (E_{i,j}, G_i, D_{i,j}, B_j), \beta_{i,j})$ from ZK_{enc*}. If some $\beta_{i,j} = 0$, report the failed party and abort. When all $(aux, j, E_{i,j}, D_{i,j})$, $P_j \in P$, do:
- Set $\alpha_{i,j} = DEC_i(E_{i,j})$ for every $j \neq i$.
- Compute $\delta_i = b_i a_i + \sum_{j \neq i}(\alpha_{i,j} + \beta_{i,j}) \bmod n$, output (aux, i, B).
Errors. Report the failed party and halt when the verification fails.
Stored state. Store (aux, N, s, t) and $(p_i, q_i, a_i, b_i, \delta_i)$.

Fig. 8 MultiAdd-$(aux, A_i, B_i, a_i, b_i)$ step

VSS-(aux, X_i, x_i, m, n) **step**
Round 1.
On the input $(vsstsgen, aux, m, i)$ from P_i, interpret $aux = (... ID_P, params)$, do:
- Choose $a_{i,1}, ..., a_{i,m-1} \leftarrow Z^*_N$ randomly, set $A_{i,1} = a_{i,1}G, ..., A_{i,m-1} = a_{i,m-1}G$.
- Compute $y_{i,j} = x_i + \sum_{k=1}^{m-1}(d_j)^k a_{i,k}.j = 1, ..., n$, $C_{i,j} = ENC_j(y_{i,j})$, $j = 1, ..., n$.
Send $(prove, ZK_{log}, (tig, i), (X_i, G); x_i)$ to ZK_{log}.
Send $(aux, i, A_{i,1}, ..., A_{i,m-1}, C_{i,j})$ to all P_j.
Output:
1. When receiving $(verify, ZKlog, (tig, j), (X_j, G), \beta_j)$ from ZK_{log}. If some $\beta_j = 0$, report the failed party and abort.
When receiving $(aux, j, A_{j,1}, ..., A_{j,m-1}, C_{j,i})$ from P_j,do:
2. When all P_j verifications are passed, set $A = \sum X_i$, do:
- Compute $y'_{j,i} = DEC_i(C_{j,i}), j = 1, ..., n$, verify $y'_{j,i}G = X_j + \sum_{k=1}^{m-1}(d_i)^k A_{j,k}, j = 1, ..., n, j \neq i$.
- Set $y_i = \sum_{j=1}^{n} y'_{j,i}$, $Y_i = y_i G$, output (aux, i, A, Y_i).
Errors. Report the failed party and halt when the verification fails.
Stored state. Store (A, Y_i), (x_i, y_i).

Fig. 9 VSS-(aux, X_i, x_i, m, n) step

additional information $X_i = x_i G$ to confirm the identity of each party to resist arbitrary input from malicious parties.

4.2 Adversary ability and communicate channel

The adversary is malicious, meaning it can instruct the corrupt parties to deviate from the protocol in arbitrary ways.

Our protocol communicates in authenticated channels. There are two communication modes, send $(aux, i, ...)$ to P_j represents a point-to-point communication network, and send $(aux, i, ...)$ to all P_j represents a broadcast channel that all parties in P.

4.3 Commitment

When each party randomly chooses the additive sharing of the public key X in Fig. 5 round 1, it needs to commit to the additive sharing to prevent malicious parties from replacing the public key. Likewise, the generation of random number R in Fig. 6 is used paillier commitments also, i.e., using the Paillier ciphertext to ensure that random number cannot be replaced, which can be found in $ZKlog*$ in Fig. 8.

4.4 Identify corrupt parties

In the identify corrupt parties process, the main idea is that each party has executed zero-knowledge proof at every step. For example, if aG needs to be broadcast, the zero-knowledge proof of aG is broadcast yet, so that the receiver can verify that the sender party has a secret a, and the broadcast message is aG.

So if an adversary tries to replace the public key, he cannot compute his current private key because he does not have the private keys of all parties, so cannot send the correct $ZKlog$.

It is not easy to find the failed party in the public key verification and the signature verification error, we set up additional checks for this. In VSS step, additional execution of $ZKlog*$ is required to find the corrupt party for messages sent by other parties. In MultiAdd step, additional execution of $ZKlog*$, $ZKmul$ and $Zkenc$ is required because it only the computation of δ_i is not proofed.

4.5 Key-refresh strategy

The Key-refresh step has different options: Global key-refresh

(Fig. 10) allows each party to refresh their secret value x_i and keeps the master public key unchanged. Threshold key-refresh (Fig. 11) is for each party to change their threshold t'.

The above key-refresh strategy is (t,n), so how to change their party's size n? i.e. remove and add party. It is easy to remove a party as long as declared party Q is invalid. However adding a party without changing the public key is not simple, We can be compatible with Li and Yu's [28,29] scheme in our solution.

5 Security analysis

We use the ideal/real paradigm to prove that the protocol realizes the ideal threshold signature functionality F. According to the ideal function F and the ideal random oracle in our security model, if the threshold SM2 protocol does not realize the ideal function F, there is an adversary A with non-negligible probability distinguishing Paillier ciphertext or forging an SM2 signature.

5.1 Ideal threshold signature functionality F

Key generation: When receiving the output $(keygen, tig, t, i)$ from party P_i,

1) Interpret $tig = (..., P, params)$, where $P = (P_1, P_2, ..., P_n)$.

- If $P_i \in P$, send $(keygen, tig, t, i)$ to S and record $(keygen, tig, t, P_i)$.
- Else, ignore this message.

2) When $(keygen, tig, t, P_j)$ is recorded from every $P_j \in P$, send $(publickey, tig, t)$ to S and do:

(a) When receiving the output $(publickey, tig, t, pk)$ from S, record $(publickey, tig, t, pk)$.

(b) When receiving the ouptut $(publickey, tig, t)$ from P_i.

- If $(publickey, tig, t, pk)$ is recorded, output it.
- Else, ignore this message.

Sign: Initialize $Q = \varnothing$, upon receiving the output $(sign, sid = (tig, ...), msg)$ from party P_i, add P_i to Q.

1) Send it to S and record $(sign, sid, msg, i)$.

2) When receiving the output $(sign, sid = (tig, ...), msg, j)$ from S, then record $(sign, sid, msg, j)$ if P_j is corrupted. Otherwise, ignore this message.

3) Upon $|Q| > t$, send $(sign, sid, msg)$ to S, do:

(a) When receiving the output $(signature, sid, msg, \sigma, C)$ from S, where $C \in P$:

- If $(sid, msg, \sigma, 0)$ is recorded, output error.
- Else if (msg, σ) is the valid signature of the public key pk, then record $(sid, msg, \sigma, 1)$.
- Else if C is corrupted, record C as an malicious party.
- Else record the first corrupted party to be the malicious party.

(b) When receiving $(signature, sid, msg)$ from $P_i \in P$:

- If $(sid, msg, \sigma, 1)$ is recorded, output $(signature, sid, msg, \sigma)$ to P_i,
- Otherwise, output the identity of the malicious party.

Verify: When receiving the output $(verify, sid, msg, \sigma, Q_A)$ from any party P^*, where Q_A is the public key.

1) Send $(verify, sid, msg, \sigma, Q_A)$ to S, do:

- If (msg, σ, b') is recorded, set $b = b'$.
- Else if P never signed the message msg, set $b = 0$.
- Else if (msg, σ) is the valid signature of pk, set $b = 1$.

Global-(n,n) key-refresh step

The store state of P_i is X_i, x_i.

Round 1.

On the input $(keyrefresh, n, n, sid, i)$ from P_i, interpret $sid = (..., \kappa, a, b, n, G, ID_P, ENTL_P, rtig, X, N, s, t)$, do:

- Run VSS-(sid, X_i, x_i, n, n) step, obtain the output (sid, i, A, Y_i), and the stored state is (A, Y_i), (x_i, y_i).

Send $(prove, ZK_{log}, (tig, rtig, \rho, i), (Y_i, G); y_i)$ to ZK_{log}, send (sid, i, Y_i, ψ_i) to all P_j.

Output:

When receiving $(verify, ZK_{log}, (tig, rtig, \rho, j), (Y_j, G), \beta_j)$ from ZK_{log}. If some $\beta_j = 0$, report the failed party and abort. When all (sid, j, Y_j, ψ_j) is received, do:

- Verify $A = \sum_{j=1}^{n} \lambda_j Y_j$, $\lambda_i = \prod_{k=0, k\neq i}^{n-1} (d_i - d_k)^{-1}(-d_k)$, if this verification error, run global key-refresh checker.
- Compute $x_i = \lambda_i y_i$, $X_i = \lambda_i Y_i$.
- Output (sid, i, X_i), clear all data except stored state.

Errors. Report the fails party and halt when the verification failed.

Stored state. Store (sid, X_i), and (p_i, q_i, x_i).

Global key-refresh checker: Check the VSS step data,

1. Check $Y_i = \sum_{j=1}^{n} (X_j + \sum_{k=1}^{m} (d_i)^k A_{j,k}), i = 1, .., n$.
2. Compute $C_i = \sum_{j=1}^{n} C_{j,i}$, send $(prove, ZK_{log*}, (C_i, Y_i, G); ...)$ to ZK_{log*}.
3. Compute $Y_{i,j} = X_i + \sum_{k=1}^{m-1} (d_j)^k A_{i,k}$, send $(prove, ZK_{log*}, (C_{i,j}, Y_{i,j}, G); ...)$ to ZK_{log*}.

Fig. 10 Global-(n,n) key-refresh step

Threshold-(t',n) key-refresh step

The store state of P_i is X_i, x_i.

Round 1.

On the input $(keyrefresh, t', n, sid, i)$ from P_i, interpret $sid = (..., \kappa, a, b, n, G, ID_P, ENTL_P, rtig, X, N, s, t)$, do:

- Run VSS-(sid, X_i, x_i, t', n) step, then obtain the output (sid, i, A, Y_i), and the stored state is (A, Y_i), (x_i, y_i).

Output (sid, i, Y_i), clear all data except stored state.

Errors. Report the fails party and halt when the verification failed.

Stored state. Store (sid, X_i, Y_i), and (p_i, q_i, x_i, y_i).

Fig. 11 Threshold-(t',n) key-refresh step

- Otherwise, set $b = 0$.

Record (msg, σ, b) and send $(isture, sid, msg, \sigma, b)$ to P_*.

Key-refresh: Initialize $Q = \varnothing$; when receiving the key-refresh from P_i, send it to S, add P_i to Q and do:

1) When $|Q| > t$,

- Refresh the public key, private key, Paillier key or threshold t for all parties.

5.2 UC security proof

Let us first briefly summarize the proof idea. If the threshold SM2 protocol satisfies the existential unforgeability, it can UC-realize the idea function F. At the same time, the threshold SM2 protocol is existential unforgeability if the SM2 signature existential unforgeability. So the point is the reducting process, which can be proved like a simple game series.

- Game 0: Simulator S samples the (Q_A, d_A) of SM2 and auxiliary information $Q'_A = (1 + d_A)^{-1} G$, rewind the adversary A, and fixes the public key to Q_A.
  - Extract the Paillier key of the corrupted party.
  - It does not decrypt the honest party's Paillier ciphertext but decrypts the corrupted parties' relevant messages.
  - Invoking zero-knowledge on honest parties to generate valid proofs, programming the random oracle.
- Game 1: Simulator S randomly selects an honest party, which is considered special.
  - Whenever the honest party instructs the corrupt party to encrypt a value with the special party Paillier key, the simulator sends random encryption of 0.
- Game 2: Simulate the situation where the special party's secret is not known using the SM2 oracle.

Game 0 and Game 1 are computationally secure under Paillier encryption is semantic security. Game 1 and Game 2 are perfectly secure because the adversary's view is indistinguishable if the simulator S sends random encryption of 0. Game 2 is equivalent to invoking the adversary A to forge the SM2 signature.

Theorem 2 Assuming the strong RSA, the semantic security of the Paillier encryption scheme, and the existential unforgeability of SM2, the protocol UC-realize the ideal function F under the ideal zero-knowledge functionality ZK_{Rel}.

Lemma 1 Suppose the protocol does not UC-realize the ideal function F under the ideal zero knowledge functionality ZK_{Rel}. In that case, an adversary A can forge an SM2 signature for previously unsigned messages when executing the protocol.

Prove the Lemma 1: It is clear since the definition of the ideal function F.

Lemma 2 If an adversary A outputs a forged SM2 signature using $Q'_A = (1 + d_A)^{-1} G$, then the adversary A can forge the SM2 signature.

Prove the lemma 2: Obviously that Q'_A cannot provide any valuable information to the adversary.

Lemma 3 Assuming the strong RSA, if there is an adversary A that can forge a threshold SM2 signature for previously unsigned messages when executing the protocol, then there are algorithms R1 and R2 that access an adversary A with black boxes under the ideal zero-knowledge functionality ZK_{Rel}, at least one is correct,

1. R1 wins Paillier's semantic security with a significant probability greater than 1/2.
2. R2 wins the SM2 existential unforgeable experiment with non-negligible probability.

Prove the Lemma 3: Let A as an adversary who can forge an unsigned message when executing the protocol, and $T \in poly$ denotes the upper limit of the auxiliary information operation before the forgery occurs. $N_1, ..., N_T$, (Q_A, d_A), and Q'_A, denote the Paillier public key, SM2 key-pair, and auxiliary information, sampled according to the protocol. Let R1 be the Paillier-distinguisher, and R2 be the SM2-Forger, consider the following three experiments:

Experiment A. R1 invokes adversary A with the parameters (Q'_A, Q_A, d_A) and $(N_k, c_k), k = 1, ..., T$, where $c_k = ENC_{N^k}(1)$.

Experiment B. R1 invokes adversary A with the parameters (Q'_A, Q_A, d_A) and $(N_k, c_k), k = 1, ..., T$, where $c_k = ENC_{N^k}(0)$.

Experiment C. R2 invokes adversary A with the parameter (Q'_A, Q_A).

Claim 1 Assuming the strong RSA, if the adversary A outputs a forged signature in the protocol with probability α in time τ, then R1 can forge the SM2 signature in experiment A with probability at least α^2 in time $nlog(n)\tau$.

Prove the claim 1: Time $nlog(n)\tau$ because the rewind step, α reduce to α^2 because it needs to rewind of the zero-knowledge proof.

Claim 2 Assuming the semantic security of the Paillier encryption scheme, if the adversary A outputs a forged signature in experiment A with probability α in time τ, then R1 can output a forged signature in experiment B with probability α in time τ.

Claim 3 If the adversary A outputs a forged signature in experiment B with probability α in time τ, R2 outputs a forged signature in experiment C with probability $\alpha + negl(\kappa)$ in time τ.

We formally describe the UC simulation and the reduction process for R1 and R2, we set up an independence simulator for each stage, and each simulator is queried through a query-answer list L for the random oracle, which already depending on the Reduction. For example, in R1, the reduction process provides a challenge (N, c) for the simulator. Similarly, in the key generation stage in R2, the reduction process provides a challenge (SM2 public key Q_A and Q'_A) for the simulator, trying to win the existential unforgeability game.

- **Paillier-distinguisher R1**

R1 takes $T \in poly(\kappa)$ as the upper bound of query quantity, Paillier public key and ciphertext are $N_1, ..., N_T, c_1, ..., c_T$, and SM2 key (Q_A, d_A) and $Q'_A = (1 + d_A)^{-1} G$. Set ctr is a counter variable, initialize $ctr = 0$. Let L denote the list of zero knowledge prove-verifies stored in memory by the simulator S

and initialized to an empty set. The algorithm R1 interacts with the adversary A by the following steps.

Zero konwledge query.

When receiving $(prove, ZK_{Rel}, aux', x; w)$ from adversary A, do:

1) If $aux' \neq aux$, send $(prove, ZK_{Rel}, x; w)$ to ZK_{Rel} and obtain $(verify, ZK_{Rel}, x, \beta)$, return $(verify, ZK_{Rel}, aux', x, \beta)$.

2) Else, if the rewinding step generates $Rel = fac$, then:

- Write the zero knowledge program and extract p, q, that $N = pq$. Add related tuples to L.

3) Else

(a) If $(ZK_{Rel}, x, \beta) \in L$, return $(verify, ZK_{Rel}, aux', x, \beta)$.

(b) Otherwise return $(verify, ZK_{Rel}, aux', x, \beta)$, $\beta = Rel(x; w)$, and add (ZK_{Rel}, x, β) to L.

Key generation.

The simulator S writes $(keygen, tig = (..., P, params, i)$ on the tape of each P_i (including the corrupted party $C \subsetneq P$). Random $N_0 = p_0 q_0$ according to the given distribution. Set $aux_0 = N_0$, then add $ctr := ctr + 1$ and set $aux = N_{ctr}$, invoke $S_1(tig, C, L, aux)$ to get output, then output $b, L, tig, rtig$, $\{N_j, s_j, t_j\}_{j \in P}$, retrieve $\{(p_j, q_j)\}_{j \neq b}$.

Then the simulator S writes $(sm2tskeygen, tig, rtig, t, i)$ on the tape of P_i (including the corrupted party $C \subsetneq P$).

Random choose $\omega \leftarrow F_n$, invoke $S_2(tig, C, L, Q'_A, Q_A, \omega)$ to get the output, then output $b, L, \rho, \{x_k, y_k\}_{k \neq b}$, set $x_b = (1 + d_A)^{-1} - \sum_{j \neq b} x_j \bmod n$.

Pre-signing.

Simulator S writes $(presign, sid, i)$ on the tape of each P_i (including the corrupted party $C \subsetneq P$).

Choose t parties in P randomly as S'. If $P_b \notin S'$, then rewind it until P_b included.

Choose a $k_b \leftarrow F_n$ randomly, and set $y^{\setminus b} = \{y_i\}_{i \neq b}$, $y_b = \lambda_b^{-1}((1 + d_A)^{-1} - \sum_{j=1, j \neq b}^{n} \lambda_j y_j)$, $aux = (c, k_b, y_b)$.

(a) Invoke S_3 with the relevant input.

Signing.

Simulator S writes $(sign, sid, i, msg)$ on the tape of $P_i \in S'$.

(a) Retrieve R, $(k_i, \chi_i)_{i \notin C}$, set $r = (R_{x-axis} + e) \bmod n$, $s_i = \chi_i + x_i r \bmod n$.

(b) Send $\{(sid, i, s_i)\}_{i \notin C}$ to adversary A.

Errors.

1) If the SM2Ts checker run.

- For $P_i \in H$, submit the relevant information to the adversary A according to the protocol.
- For P_b, set $\hat{U}_b = ENC_b(0)$, simulates S_{mul}, S_{enc} the messages $(verify, ZK_{mul}, (tig, rtig, \rho, b), (G_b, F_b, \hat{U}_b), 1)$, $(verify, ZK_{enc}, (tig, rtig, \rho, i), U_i \times \prod_{j \neq i}(E_{i,j} \times D_{j,i}), 1)$, add related tuples to L.

2) If the Presigning checker run.

- For $P_i \in H$, submit the relevant information to the adversary A according to the protocol.
- For P_b, compute Y_b, C_b according to the protocol, simulates S_{log*} the message $(verify, ZK_{log*}, sid, (Y_b, C_b, G), 1)$.
- Compute $Y_{b,j}, C_{b,j}$ according to the protocol, simulates S_{log*} the message $(verify, ZK_{log*}, sid, (Y_{b,j}, C_{b,j}, G), 1)$.

Add related tuples to L.

3) If the Signing step fails.

- For $P_i \in H$, submit the relevant information to the adversary A according to the protocol.
- For P_b, set $\hat{U}_b = x_b \cdot G_b \times ENC_b(0)$, simulates S_{mul}, S_{enc} the message $(verify, ZK_{mul}, sid, (G_i, F_i, U_i), 1)$, $(verify, ZK_{enc}, sid, U_i \times \prod_{j \neq i}(E_{i,j} \times D_{j,i}) \times r \cdot F_i, 1)$, add related tuples to L.

● SM2 Forger R2

SM2 is defined by the interaction of the adversary A and the SM2 signature with the public key Q_A and $Q'_A = (1 + d_A)^{-1} G$. Let L denote the list of prove-verifies stored in memory by the simulator S and initialized to an empty set. The algorithm R2 interacts with the adversary A by the following steps.

Zero konwledge query.

Same as the Paillier-disting- uisher R1 zero konwledge query.

Key generation.

The simulator S writes $(keygen, tig = (..., P, params, i)$ on the tape of each P_i (including the corrupted party $C \subsetneq P$).

Invoke $S_1(tig, C, L, \perp)$ get output, then output b, L, $tig, rtig$, $\{N_j, s_j, t_j\}_{j \in P}$, retrieve $\{(p_j, q_j)\}_{j \in P}$.

Then simulator S writes $(sm2tskeygen, tig, rtig, i)$ on the tape of each P_i (including the corrupted party $C \subsetneq P$).

Random choose $\omega \leftarrow F_n$, invoke $S_2(tig, C, L, Q'_A, Q_A, \omega)$ to get the output, then output $b, L, tig, rtig, \rho, \{x_k\}_{k \neq b}$, $X = (X_1, ...)$.

5.3 UC simulation

The simulator S_{Rel} represents ZK_{Rel}.

$Rel \in \{prm, mod, fac, log, log*, enc, enc*, mul\}$.

● Paillier-keygen simulator S_1

$S_1(tig, C, L, aux)$, $tig = (..., P, params)$, an identity information tig, a set of parties $C \subsetneq P$, a list L for prove-verifies, auxiliary input $aux = \perp$ or $aux = N_*$.

Round 1.

Randomly choose $P_b \leftarrow P \setminus C$, set $H = P \setminus C \cup \{P_b\}$.

Case 1: $aux = N_*$.

For each $P_i \in H$, execute according to the protocol and submit $(tig, i, rtig_i, N_i, t_i, s_i)$ to the adversary A.

For P_b, choose a $rtig_b \leftarrow \{0, 1\}^K$ randomly.

- Set $N_b = N_*$, simulates S_{mod} the message $(verify, ZK_{mod}, (tig, i), N_b, 1)$, simulates S_{fac} the message $(verify, ZK_{fac}, (tig, i), N_b, 1)$, and the other information executes according to the protocol.

Add related tuples to L, submit $(tig, b, rtig_b, N_b, t_b, s_b)$ to adversary A.

Case 2: $aux = \perp$.

For each $P_i \notin C$, executes according to the protocol and submits $(tig, i, rtig_i, N_i, t_i, s_i)$ to the adversary A.

Output:

When receiving $(prove, ZK_{prm}, (tig, i), (N_i, s_i, t_i); v)$ that A sends to ZK_{prm}, $(prove, ZK_{mod}, (tig, i), N_i; (p_i, q_i))$ that A sends to ZK_{mod}, $(prove, ZK_{fac}, (tig, i), N_i; (p_i, q_i))$ that A sends to ZK_{fac}. Check they are corrent. If not, sends the failed party to ZK_{prm} or ZK_{mod} or ZK_{fac}, and aborts.

When receiving all $(tig, j, rtig_j, N_j, t_j, s_j)$ from adversary A.

Execute according to the protocol, then output b, L, $tig, rtig$ and $\{N_j, s_j, t_j\}_{j \in P}$.

● SM2Ts-keygen simulator S_2

$S_2(tig, C, L, Q'_A, Q_A, \omega)$, $tig = (..., P, params)$, a list L for prove-verifies and a set of corrupted parties $C \subsetneq P$. Initialize $ext = 0$.

Round 1.

For each $P_i \in H$, sample V_i according to the protocol and submit (tig, i, V_i) to the adversary A.
For P_b, if $ext = 0$, set X_b, Γ_b according to the protocol, otherwise, set $X_b = Q'_A - \sum_{j \neq b} X_j$, $\Gamma_b = \omega(Q_A + G) - \sum_{j \neq b} \Gamma_j$.
Sample V_b according to the protocol and submit (tig, i, V_b) to the adversary A.
Round 2.
When receiving (tig, j, V_j) from adversary A, submit $(tig, i, X_i, \Gamma_i, \rho_i, u_i)$, $P_i \notin C$ to the adversary A.
Round 3.
When receiving $(tig, j, X_j, \Gamma_j, \rho_j, u_j)$ from adversary A, retrive $\{\gamma_j\}_{j \in C}$.
1. If $ext = 0$, execute according to the protocol then submit $(tig, i, Y_i, \Delta_i, \delta_i)$, $P_i \notin C$ to adversary A.
2. Else, verify $H(...) = V_j$ and compute ρ according to the protocol.
Invoke $Svss(tig, rtig, \rho, C, L, (X)_j)$, the store state is $\{x_j, y_j\}_{j \neq b}$ and output $L, A, (Y_j)_{j \in P}$.
Invoke $Smultiadd(tig, rtig, \rho, C, L, (X)_j, (\Gamma)_j, \perp)$, the store state is $\{x_j, \gamma_j\}_{j \neq b}$ and (γ_b, δ_b) and output L, Γ.
Set $\Delta_b = \omega G - \sum_{i \neq b} x_i \Gamma$, simulates S_{log} the message $(verify, ZK_{log}, (tig, rtig, \rho, i), (Y_i, G), 1)$, simulates S_{log} the message $(verify, ZK_{log}, (tig, rtig, \rho, i), (\Delta_i, \Gamma), 1)$.
Submit $(tig, i, Y_i, \Delta_i, \delta_i)$, $P_i \notin C$ to the adversary A, add related tuples to L.
Output:
When receiving $(prove, ZK_{log}, (tig, rtig, \rho, i), (Y_i, G); y_i)$, $(prove, ZK_{log}, (tig, rtig, \rho, i), (\Delta_i, \Gamma); x_i)$ from A to ZK_{log}. Check they are corrent. If not, sends the failed party to ZK_{log}, and aborts.
When receiving $(tig, j, Y_j, \Delta_j, \delta_j)$ from adversary A, execute according to the protocol and obtain X.
1. $ext = ext + 1$, if $ext = 1$, go back to round 1, delete the tuples added to L since S_2 round 1.
2. Else, extract $\{x_j, y_j\}_{j \notin C}$, output $L, b \in P, \{x_j, y_j\}_{j \neq b}$.

- **Pre-signing simulator** S_3

$S_3(sid, C, L, b, y^{\backslash b}, aux)$, input $sid = (..., X, N, s, t)$, a set of corrupted parties $C \subsetneq P$, a list L for prove-verifies, and auxiliary input $aux = R$, or $aux = (c^{ctr}, k_b, x_b)$.
Round 1.
Case 1: If $aux = (c^{ctr}, k_b, x_b)$.
For $P_i \in H$ choose Λ_i according to the protocol.
For P_b, set $\Lambda_b = k_b G$, invoke $Smultiadd(sid, C, L, (W)_j, (\Lambda)_i, aux)$, the store state is $\{k_i, \chi_i\}_{i \neq b}$, χ_b, and output L, R.
store $(w_i, \chi_i)_{j \notin C}$, output R.
Case 2: If $aux = R$, initialize $ext = 0$, do:
If $ext = 0$ output R according to the protocol, otherwise, set $\Lambda_b = R - \sum_{i \neq b} \Lambda_i$, compute w_i and verify A is corrent according to the protocol, invoke $Smultiadd(sid, C, L, (W)_j, (\Lambda)_i, aux)$, the store state are $\{k_i, \chi_i\}_{i \neq b}$, η_0, η_1, output L, R.
1. $ext = ext + 1$, if $ext = 1$, go back to round 1, delete the tuples added to L since S_3 round 1.
2. Else, store (η_0, η_1) and $(w_i, \chi_i)_{i \in b}$ then output R.

- **MultiAdd simulator** $Smultiadd$

$Smultiadd(aux, C, L, (A)_j, (B)_j, aux')$, $aux = sid$ or $(tig, rtig, \rho)$, a set of corrupted parties $C \subsetneq P$, a list L for prove-verifies and $aux' = \perp$, R or (c, k_b, w_b).
Set case 1 as $aux' = \perp$, case 2 as $aux' = (c, a_b, b_b)$, case 3 as $aux' = R$.
Round 1.
Each $P_i \in H$, execute according to the protocol and submit (aux, i, A_i, B_i, G_i) to the adversary A.
For P_b.
Case 1: set $G_b = ENC_b(0)$.
Case 2: set $G_b = a_b \cdot c$.
Case 3: set $G_b = ENC_b(0)$.
Simulates S_{log*} the message $(verify, ZK_{log*}, ..., (G_b, A_b, G), 1)$, and submit (aux, b, A_b, B_b, G_b) to the adversary A.
Add the related tuples to L.
Round 2.
When receiving $(prove, ZK_{log*}, (aux, i), (G_i, A_i, G); a_i, v_i)$ that A sends to ZK_{log*}. Check they are corrent. If not, sends the failed party to ZK_{log*} and aborts.
When receiving (aux, j, A_j, B_j, G_j) from adversary A, set $B = \sum B_i$, retrive $\{a_j, b_j\}_{j \in C}$.
For $P_i \in H$, execute according to the protocol and submit $(aux, i, E_{j,i}, D_{j,i}, \hat{\psi}_{j,i})$ to the adversary A.
For P_b, randomly choose $\{\alpha_{j,b} \leftarrow J\}_{j \neq b}$.
Case 1: set $E_{j,b} = ENC_j(\alpha_{j,b})$, $D_{j,b} = ENC_b(0)$.
Case 2: set $E_{j,b} = ENC_j(\alpha_{j,b})$, $D_{j,b} = (a_j b_b - \alpha_{j,b}) \cdot c$.
Case 3: set $E_{j,b} = ENC_j(\alpha_{j,b})$, $D_{j,b} = ENC_b(0)$.
Simulates S_{enc*} the message $(verify, ZK_{enc*}, (aux, i), (E_{j,i}, G_j, D_{j,i}, B_i), 1)$ for each $j \neq b$.
Submit $(aux, b, E_{j,b}, D_{j,b})$ to the adversary A, add the related tuples to L.
Output.
When receiving $(prove, ZK_{enc*}, ..., (E_{j,i}, G_j, D_{j,i}, B_i); (b_i, \beta_{i,j}, f_{i,j}, g_{i,j}))$ that A to ZK_{enc*}. Check they are corrent. If not, sends the failed party to ZK_{enc*} and aborts.
When receiving $(aux, j, E_{i,j}, D_{i,j}, \hat{\psi}_{i,j})$ from adversary A, retrieve $\{\alpha_{j,b}, \beta_{j,b}\}_{j \neq b}$.
Case 1: set $\delta_b = \omega - \sum_{j,i \neq b} a_i b_j - \sum_{j \neq b} (\alpha_{j,b} + \beta_{j,b})$.
Case 2: set $\delta_b = b_b a_b + \sum_{j \neq b} (b_j a_b - \alpha_{j,b}) + (b_b a_j - \beta_{j,b})$,
Case 3: set $\eta_0 = \sum_{j \neq b} a_j$, $\eta_1 = \sum_{j,i \neq b} a_i b_j + \sum_{j \neq b} (\alpha_{j,b} + \beta_{j,b})$.
Store $\{a_i, b_i\}_{i \neq b}$, and δ_b or (η_0, η_1), submit (aux, i, B) to the adversary A.

- **VSS simulator** $Svss$

$Svss(aux, C, L, (X)_j, aux')$, $tig = (..., P, params)$, a set of corrupted parties $C \subsetneq P$, a list L for prove-verifies, $aux = (tig, rtig, \rho)$ or $aux = sid$.
Round 1.
1. For $P_i \in H$, execute according to the protocol and submit $(aux, i, A_{i,1}, ..., A_{i,m-1}, C_{i,j})$ to adversary A.
2. For P_b, random choose $y_{b,1}, ..., y_{b,n} \leftarrow Z_N^*$, set $Y_{b,j} = y_{b,j} G$, $C_{b,j} = ENC_j(y_{b,j})$, $j = 1, .., n$.
Obtain the identify d_{j^*} of $P_j \in C$, where $|C| < m$. Then randomly choose $P_j \notin C$ until $m - 1$ parties.
Compute

$$\begin{bmatrix} A_{b,1^*} \\ \vdots \\ A_{b,m-1^*} \end{bmatrix} = \begin{bmatrix} d_{1^*} & \cdots & d_{m-1^*}{}^{m-1} \\ \vdots & \ddots & \\ d_{m-1^*} & & d_{m-1^*}{}^{m-1} \end{bmatrix}^{-1} \times \begin{bmatrix} Y_{b,1^*} - X_b \\ \vdots \\ Y_{b,m-1^*} - X_b \end{bmatrix},$$

Simulates S_{log} the message $(verify, ZK_{log}, aux, (X_b, G), 1)$.
Submit $(aux, i, A_{b,1}, ..., A_{b,m-1}, C_{b,j})$ to adversary A, add related tuples to L.
Output.
When receiving $(prove, ZK_{log}, aux, (X_i, G); x_i)$ that A sends to ZK_{log}. Check they are corrent. If not, sends the failed party to ZK_{log} and aborts. When receiving $(aux, j, A_{j,1}, ..., A_{j,m-1}, C_{j,i})$ from the adversary A.
- Set $A = \sum X_j$, $Y_b = A + \sum_{k=1}^{m-1}(d_b)^k \sum_{j=1}^{n} A_{j,k}$, store $\{x_i, y_i\}_{j \neq b}$, the other data according to the protocol.
Submit (aux, i, A, Y_i) to the adversary A.

In the ideal threshold function F, the threshold signature verification algorithm uses the SM2 signature verification algorithm, which is more suitable for the existential unforgeability of the signature.

6 Performance analysis

The computation and communication consumption are as follows.

G denotes the computing exponentiation in the elliptic curve group $\bar{G}$, N denotes the computing exponentiation in the rings Z_N, N_2 is the computing exponentiation in the Z_{N^2}. n is the number of parties, t is the number of threshold parties. k is the elliptic curve group element and μ as a Paillier plaintext ring element. Others are omitted. We count the total cost per party in Table 2.

Compared with other threshold SM2 schemes, Zhang et al.'s scheme [14] does not have UC security and only has the function of $(2, n)$-share. Our scheme achieves UC security and can perform threshold signature for any $t \leqslant n$.

Compared with the same type of threshold ECDSA [5–7,9,10,30,31], we use zero-knowledge proof to ensure the identifiable abort of the protocol, the price is the increasing amount of computation. Lindell's scheme [12] does not have the pre-signing step, and the signing step in Table 3 of our's scheme includes the pre-signing step, the keygen step includes the Paillier-keygen step.

Compared with the Canetti's threshold ECDSA scheme. Our scheme also achieves the non-interactive, identifiable abort and UC security. Then, it only invokes the MultiAdd step once in the pre-signing step with 2 rounds, while Canetti's scheme needs to invoke twice with 3 rounds at least. So our scheme reducing the computation and communication by almost 2/3 in the pre-signing stage (Table 4).

We use Golang on AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz to implement our protocol, the $\kappa_{sm2} = 256, \kappa_{paillier} = 2048$. Using parallelism in Paillier-keygen to speed up protocol execution, and others are implemented without any optimizations, Paillier-keygen and SM2Ts-keygen are both related to the party's size n, and Pre-signing is only related to the threshold t. We emphasize that parties are simulated by threads, and in a real distributed environment, each party only needs (current runtime/n) in Table 5. The most time-consuming protocol is the Paillier encryption and decryption operation, proper optimization can improve a lot efficiency.

7 Conclusions and further work

Threshold signature has a well-distributed generation property and high fault tolerance rate. This paper implements the UC security threshold SM2 signature, which can apply in distributed key management, smart contracts, and digital currency [32,33].

Our protocol can achieve security with identifiable abort under the malicious majority and has very low coupling, suitable for further expansion and maintenance. So it can be

Table 2 Computational and communication costs of the protocol

Step	Round	Computation	Communication
Paillier-keygen	1	$422N$	$n(2k + 334\mu)$
SM2Ts-keygen	6	$8N + (14 + t)G + 4N^2 + n(17N + 8G + 14N^2)$	$9k + 7\mu + n(18k + 13\mu)$
Pre-signing	2	$8N + 4G + 4N^2 + t(17N + 4G + 12N^2)$	$9k + 7\mu + t(17k + 12\mu)$
Signing	1	0	nk
MultiAdd	2	$8N + 3G + 4N^2 + n(16N + 3G + 12N^2)$	$9k + 7\mu + n(17k + 12\mu)$
VSS	1	$tG + n(N + 5G + 2N^2)$	$(t + 2)k + n\mu$

Table 3 Computational and communication costs of Lindell's scheme [12]

Step	Round	Computation	Communication
Keygen	5	$n(11G + 11N^2)$	$n(11k + 11\mu)$
Keygen(ours)	7	$14G + 434N^2 + n(9G + 31N^2)$	$9k + 7\mu + n(19k + 346\mu)$
Signing	8	$n(78G + 21N^2)$	$n(59k + 24\mu)$
Signing(ours)	3	$4G + 12N^2 + n(4G + 29N^2)$	$9k + 7\mu + n(18k + 12\mu)$

Table 4 Computational and communication costs of Canetti's scheme [11]

Pre-signing	Round	Computation	Communication
Three rounds	3	$n(56N + 12G + 33N^2)$	$n(57k + 54\mu)$
Six rounds	6	$n(49N + 29G + 33N^2)$	$n(67k + 51\mu)$
Lightweight	7	$n(38N + 26G + 19N^2)$	$n(67k + 30\mu)$
Ours	2	$8N + 4G + 4N^2 + t(17N + 4G + 12N^2)$	$9k + 7\mu + t(17k + 12\mu)$

Table 5 Runtime in PC

(n or t)/s	Paillier-keygen	SM2Ts-keygen	Pre-signing
2	15.85	17.3	16.35
3	21.27	46.28	43.27
4	29.38	86.89	83.68
5	35.79	141.82	134.93
6	44.9	209.24	203.48
7	52.27	288.51	283.35
8	65	386.76	378.03
9	71.72	490.02	484.11

used as an alternative to multi-signature in cryptocurrency. Furthermore, we have added the non-interactive properties that make our scheme practical.

In the next step, we will look for quantum-resistant distributed signature [34,35] and zero-knowledge proof, keeping our distributed signature scheme security in the quantum world.

Acknowledgements Thanks to my supervisor and our team for helpful discussions, thanks to the reviewers and editors for making this paper better.

Appendix

● **Ring-Pedersen parameters** ZK_{prm} [11]

Public input (N,s,t).

The prover has a witness ν, that $s=t^{\nu} \bmod N$.

1. Prover chooses a $\{a_i \leftarrow Z_{\phi(N)}\}_{i\in[m]}$, computes $A_i = t^{a_i} \bmod N$, and sents it to the Verifier.

2. Verifier answers $\{e_i \leftarrow \{0,1\}\}_{i\in[m]}$.

3. Prover computes $\{z_i = a_i + e_i\nu \bmod \phi(N)\}_{i\in[m]}$ and sends it to verifier.

Verification: Accept iff the following hold:

- $t^{z_i} = A_i s^{e_i} \bmod N$ for every $i\in[m]$.

● **Paillier-Blum modulus** ZK_{mod}

Public input N.

The prover has a witness p,q, that $N=pq$,

$gcd(N,\phi(N))=1$, $p,q=3 \bmod 4$.

1. Prover randomly chooses a $w \leftarrow Z_N$ of Jacobi symbol -1 and sents it to the verifier.

2. Verifier sends $\{y_i \leftarrow Z_N\}_{i\in[m]}$

3. For every $i\in[m]$, set $x_i = \sqrt[4]{v_i} \bmod N$, $z_i = y_i^{N^{-1} \bmod \phi(N)}$ where $v_i = (-1)^{a_i} w^{b_i} y_i$ for every $a_i,b_i \leftarrow \{0,1\}$.

Sends $\{(x_i,a_i,b_i),z_i\}_{i\in[m]}$ to the verifier.

Verification: Accept iff the following hold:

- N is an odd composite number.
- ${z_i}^N = y_i \bmod N$ for every $i\in[m]$.
- $x_i^4 = (-1)^{a_i} w^{b_i} y_i \bmod N$ and $a_i,b_i \leftarrow \{0,1\}$ for every $i\in[m]$.

● **No small factor** ZK_{fac}.

Auxiliary input the Ring-Pederson parameters $s,t\in Z_{\hat{N}}^*$ and a big safe prime $\bar{N}$.

Public input a RSA modulus N_0.

The prover has a secret p,q, that $p,q < \pm 2^l \sqrt{N_0}$.

1. Prover chooses

$$\begin{cases} \alpha,\beta \leftarrow \pm 2^{l+\epsilon}\sqrt{N_0}, \mu,\nu \leftarrow \pm 2^l \bar{N}, \\ \omega \leftarrow \pm 2^l N_0 \hat{N}, r \leftarrow \pm 2^{l+\epsilon} N_0 \hat{N}, \\ x,y \leftarrow \pm 2^l \hat{N}, \end{cases} \tag{6}$$

randomly, and computes

$$\begin{cases} P = s^p t^{\mu}, Q = s^q t^{\nu} \bmod \bar{N}, \\ D = s^{\alpha} t^{x}, C = s^{\beta} t^{y} \bmod \bar{N}, \\ T = Q^{\alpha} t^{r} \bmod \hat{N}, \end{cases} \tag{7}$$

Sends (P,Q,D,C,T,ω) to verifier.

2. Verifier answers $e \leftarrow \pm q$.

3. Prover sets $\hat{\omega} = \omega - \nu p$, and sents (z_1,z_2,m_1,m_2,v) to verifier, where

$$\begin{cases} z_1 = \alpha + ep, z_2 = \beta + eq, \\ m_1 = x + e\mu, m_2 = y + e\nu, \\ v = r + e\hat{\omega}, \end{cases} \tag{8}$$

Verification: Verifier sets $R = s^{N_0} t^{\omega}$, and checks

$$\begin{cases} s^{z_1} t^{m_1} = DP^e \bmod \bar{N}, \\ s^{z_2} t^{m_2} = CQ^e \bmod \bar{N}, \\ Q^{z_1} t^{v} = TR^e \bmod \bar{N}, \end{cases} \tag{9}$$

rang checks: $z_1,z_2 \in \pm\sqrt{N_0} 2^{l+\epsilon}$

The proof guarantees that each $p,q > 2^l$ (assuming $2^{2l+\epsilon} \approx \sqrt{N_0}$).

● **Paillier encryption in range** ZK_{enc}

Auxiliary input the Ring-Pederson parameters $s,t\in Z_{\hat{N}}^*$ and a big safe prime $\bar{N}$.

Public input the (N_0,E)

The prover has a witness (x,ρ), that $x\in\pm 2^l$, $E=(1+N_0)^x \rho^{N_0} \bmod N_0^2$.

1. Prover chooses

$$\begin{cases} \alpha \leftarrow \pm 2^{l+\epsilon}, \mu \leftarrow \pm 2^l \bar{N}, \\ r \leftarrow Z_N^*, \gamma \leftarrow \pm 2^{l+\epsilon} \bar{N}, \end{cases} \tag{10}$$

randomly, and computes

$$\begin{cases} A = s^x t^{\mu}, B = s^{\alpha} t^{\gamma} \bmod \bar{N}, \\ C = (1+N_0)^{\alpha} r^{N_0} \bmod N_0^2, \end{cases} \tag{11}$$

Sends (A,B,C) to verifier.

2. Verifier answers $e \leftarrow \pm n$.

3. Prover sends (z_1,z_2,z_3) to verifier, where

$$\begin{cases} z_1 = \alpha + ex, \\ z_2 = r\rho^e \bmod N_0, \\ z_3 = \gamma + e\mu, \end{cases} \tag{12}$$

Verification: Verifier checks

$$\begin{cases} (1+N_0)^{z_1} {z_2}^{N_0} = CE^e \bmod N_0^2, \\ s^{z_1} t^{z_3} = BA^e \bmod \bar{N}, \end{cases} \tag{13}$$

Rang checks: $z_1 \in \pm 2^{l+\epsilon}$

The proof guarantees that $x\in\pm 2^{l+\epsilon}$.

● **Paillier operation with Paillier commitment** ZK_{enc*}

Auxiliary input the Ring-Pederson parameters $s,t\in Z_{\hat{N}}^*$ and a big safe prime $\bar{N}$.

Public input the (N_0,N_1,D,C,Y,X)

The prover has a witness (x,y,ρ_x,ρ_y), that $x\in\pm 2^l$, $y\in\pm 2^{l'}$, $X=xG$, $D=C^x(1+N_0)^y {\rho_x}^{N_0} \bmod N_0^2$, $Y=(1+N_1)^y {\rho_y}^{N_1} \bmod N_1^2$.

1. Prover chooses

$$\begin{cases} \alpha \leftarrow \pm 2^{l\pm\epsilon}, \beta \leftarrow \pm 2^{l'\pm\epsilon}, \\ r_x \leftarrow Z^*_{N_0}, r_y \leftarrow Z^*_{N_1}, \\ \gamma \leftarrow \pm 2^{l+\epsilon}\bar{N}, m \leftarrow \pm 2^{l}\bar{N}, \\ \delta \leftarrow \pm 2^{l+\epsilon}\bar{N}, \mu \leftarrow \pm 2^{l}\bar{N}, \end{cases} \tag{14}$$

randomly and computes

$$\begin{cases} A = C^{\alpha}(1+N_0)^{\beta} r_x^{N_0} \bmod N_0^2, \\ B = \alpha G, \\ E = (1+N_1)^{\beta} r_y^{N_1} \bmod N_1^2, \\ F = s^{\alpha} t^{\gamma}, H = s^{x} t^{m} \bmod \bar{N}, \\ I = s^{\beta} t^{\delta}, J = s^{y} t^{\mu} \bmod \bar{N}, \end{cases} \tag{15}$$

Sends (A, B, E, F, H, I, J) to verifier.
2. Verifier answers $e \leftarrow \pm q$.
3. Prover sends$(z_1, z_2, z_3, z_4, w_x, w_y)$ to verifier, where

$$\begin{cases} z_1 = \alpha + ex, z_2 = \beta + ey, \\ z_3 = \gamma + em, z_4 = \delta + e\mu, \\ w_x = r_x \rho_x^e \bmod N_0, w_y = r_y \rho_y^e \bmod N_1, \end{cases} \tag{16}$$

Verification: Verifier checks

$$\begin{cases} C^{z_1}(1+N_0)^{z_2} w_x^{N_0} = AD^e \bmod N_0^2, \\ z_1 G = B + eX, \\ (1+N_1)^{z_2} w_y^{N_1} = EY^e \bmod N_1^2, \\ s^{z_1} t^{z_3} = FH^e \bmod \bar{N}, s^{z_2} t^{z_4} = IJ^e \bmod \bar{N}, \end{cases} \tag{17}$$

Range checks: $z_1 \in \pm 2^{l\pm\epsilon}$, $z_2 \in \pm 2^{l'\pm\epsilon}$
The proof guarantees that $x \in \pm 2^{l\pm\epsilon}$, $y \in \pm 2^{l'\pm\epsilon}$.

- **Dlog equality** ZK_{log}

Public input $(\bar{G}, n, A, B)$, $n = |\bar{G}|$
The prover has a witness x, that $A = xB$,
1. Prover chooses $\alpha \leftarrow Z_n$ randomly and computes $X = \alpha B$, sends X to verifier.
2. Verifier answers $e \leftarrow Z_n$.
3. Prover sends z to verifier where $z = \alpha + ex \bmod n$.
Verification: Verifier checks $zB = X + eA$.

- **Dlog vs Paillier encryption in ringe** ZK_{log*}

Auxiliary input the Ring-Pederson parameters $s, t \in Z^*_{\hat{N}}$ and a big safe prime $\bar{N}$.
Public input $(\bar{G}, n, N_0, C, X, G)$, $n = |\bar{G}|$, G is the generator of $\bar{G}$.
The prover has a witness (x, ρ), that $x \in \pm 2^{l}$, $C = (1+N_0)^{x}\rho^{N_0} \bmod N_0^2$, $X = xG \in \bar{G}$.
1. Prover chooses

$$\begin{cases} \alpha \leftarrow \pm 2^{l+\epsilon}, \mu \leftarrow \pm 2^{l}\bar{N}, \\ r \leftarrow Z^*_N, \gamma \leftarrow \pm 2^{l+\epsilon}\bar{N}, \end{cases} \tag{18}$$

randomly, and computes

$$\begin{cases} A = s^{x} t^{\mu}, D = s^{\alpha} t^{\gamma} \bmod \bar{N}, \\ B = (1+N_0)^{\alpha} r^{N_0} \bmod N_0^2, \\ Y = \alpha G \in \bar{G}, \end{cases} \tag{19}$$

Sends (A, B, Y, D) to verifier.
2. Verifier answers $e \leftarrow \pm n$.
3. Prover sends (z_1, z_2, z_3) to verifier, where

$$\begin{cases} z_1 = \alpha + ex, z_2 = r\rho^e \bmod N_0, \\ z_3 = \gamma + e\mu, \end{cases} \tag{20}$$

Verification: Verifier checks

$$\begin{cases} (1+N_0)^{z_1} z_2^{N_0} = BC^e \bmod N_0^2, \\ z_1 G = Y + eX \in \bar{G}, \\ s^{z_1} t^{z_3} = DA^e \bmod \bar{N}, \end{cases} \tag{21}$$

Rang checks: $z_1 \in \pm 2^{l+\epsilon}$
The proof guarantees that $x \in \pm 2^{l+\epsilon}$.

- **Paillier multiplication** ZK_{mul}

Public input (N, A, B, C).
The prover has a witness (x, ρ_x, ρ_y), that $(1+N)^{x}\rho_x^{N} = A$, $C = B^{x}\rho_y^{N} \bmod N^2$
1. Prover chooses $\alpha, g, f \leftarrow Z^*_N$ randomly and computes

$$\begin{cases} D = B^{\alpha} g^{N} \bmod N^2, \\ E = (1+N)^{\alpha} f^{N} \bmod N^2, \end{cases} \tag{22}$$

Sends (D, E) to verifier.
2. Verifier answers $e \leftarrow Z_n$.
3. Prover sends (z_1, z_2, z_3) to verifier, where

$$\begin{cases} z_1 = \alpha + ex, z_2 = g\rho_x^{\,e} \bmod N, \\ z_3 = f\rho_y^{\,e} \bmod N, \end{cases} \tag{23}$$

Verification: Verifier checks

$$\begin{cases} B^{z_1} z_2^{N} = DC^e \bmod N^2, \\ (1+N)^{z_1} z_2^{N} = EA^e. \end{cases} \tag{24}$$

To ensure 80-bits statistical securety and 128-bits computational security, we choose $m = 80$ in ZK_{mod} and ZK_{prm}. The costs of ZK proofs is show in Table 6.

The prm, mod are Completeness, Soundness and HVZK (honest verifier zero-knowledge).

The $mul, fac, enc, enc*, log, log*$ are Completeness, Special Soundness and HVZK.

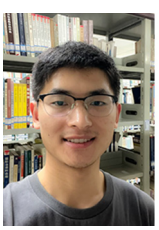
Table 6 Computational and communication costs of the ZK proofs

ZKproof	Computation (prover)	Computation (verifier)	Communication
prm	$80N$	$80N$	160μ
mod	$160N$	$80N$	160μ
fac	$10N$	$11N$	$2k{+}11\mu$
log*	$1G{+}5N{+}1N^2$	$2G{+}3N{+}2N^2$	$7k{+}6\mu$
enc	$5N{+}1N^2$	$3N{+}2N^2$	$6k{+}6\mu$
enc*	$1G{+}10N{+}3N^2$	$2G{+}6N{+}5N^2$	$17k{+}12\mu$
log	$1G$	$2G$	$2k$
mul	$2N^2$	$3N{+}4N^2$	$k{+}2\mu$

References

1. Desmedt Y. Society and group oriented cryptography: a new concept. In: Proceedings of the Advances in Cryptology. 1988, 120−127
2. Desmedt Y, Frankel Y. Threshold cryptosystems. In: Proceedings of the Advances in Cryptology. 1990, 307−315
3. ISO. ISO/IEC 14888-3:2018 IT security techniques-Digital signatures with appendix-part 3: discrete logarithm based mechanisms. Geneva, Switzerland: International Organization for Standardization, 2018
4. Nick J, Ruffing T, Seurin Y. MuSig2: simple two-round schnorr multi-signatures. In: Proceedings of the 41st Annual International Cryptology Conference on Advances in Cryptology. 2021, 189–221

5. MacKenzie P, Reiter M K. Two-party generation of DSA signatures. In: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology. 2001, 137−154
6. Gennaro R, Goldfeder S, Narayanan A. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In: Proceedings of the 14th International Conference on Applied Cryptography and Network Security. 2016, 156−174
7. Lindell Y. Fast secure two-party ECDSA signing. Journal of Cryptology, 2021, 34(4): 44
8. Doerner J, Kondi Y, Lee E, Shelat A. Secure two-party threshold ECDSA from ECDSA assumptions. In: Proceedings of 2018 IEEE Symposium on Security and Privacy (SP). 2018, 980–997
9. Gennaro R, Goldfeder S. Fast multiparty threshold ECDSA with fast trustless setup. In: Proceedings of 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018, 1179−1194
10. Pettit M. Efficient threshold-optimal ECDSA. In: Proceedings of the 20th International Conference on Cryptology and Network Security. 2021, 116−135
11. Canetti R, Gennaro R, Goldfeder S, Makriyannis N, Peled U. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Proceedings of 2020 ACM SIGSAC Conference on Computer and Communications Security. 2020, 1769−1787
12. Lindell Y, Nof A. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In: Proceedings of 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018, 1837−1854
13. Shang M, Ma Y, Lin J Q, Jing J W. A threshold scheme for SM2 elliptic curve cryptographic algorithm. Journal of Cryptologic Research, 2014, 1(2): 155–166
14. Zhang Y, He D, Zhang M, Choo K K R. A provable-secure and practical two-party distributed signing protocol for SM2 signature algorithm. Frontiers of Computer Science, 2020, 14(3): 143803
15. Keller M. MP-SPDZ: a versatile framework for multi-party computation. In: Proceedings of 2020 ACM SIGSAC Conference on Computer and Communications Security. 2020, 1575−1590
16. Acar A, Aksu H, Uluagac A S, Conti M. A survey on homomorphic encryption schemes: theory and implementation. ACM Computing Surveys, 2019, 51(4): 79
17. Administration S C. Information security technology—Public key cryptographic algorithm SM2 based on elliptic curves—Part 2: Digital signature algorithm. Beijing: State Cryptography Administration, 2016
18. Paillier P. Public-key cryptosystems based on composite degree residuosity classes. In: Proceedings of the International Conference on Advances in Cryptology. 1999, 223−238
19. Tymokhanov D, Shlomovits O. Alpha-rays: key extraction attacks on threshold ECDSA implementations. IACR Cryptology ePrint Archive, 2021, 2021:1621
20. Gennaro R, Goldfeder S. One round threshold ECDSA with identifiable abort. IACR Cryptology ePrint Archive, 2020, 2020:540
21. Shamir A. How to share a secret. Communications of the ACM, 1979, 22(11): 612–613
22. Feldman P. A practical scheme for non-interactive verifiable secret sharing. In: Proceedings of the 28th Annual Symposium on Foundations of Computer Science. 1987, 427−438
23. Lindell Y. Simple three-round multiparty schnorr signing with full simulatability. IACR Cryptology ePrint Archive, 2022, 2022:374
24. Cramer R, Damgård I, Schoenmakers B. Proofs of partial knowledge and simplified design of witness hiding protocols. In: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology. 1994, 174−187
25. Cohen R, Haitner I, Omri E, Rotem L. From fairness to full security in multiparty computation. Journal of Cryptology, 2022, 35(1): 4
26. Goldreich O. Foundations of Cryptography: Volume 2, Basic Applications. Cambridge: Cambridge University Press, 2009
27. Canetti R. Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science. 2001, 136−145
28. Li X, He M. A protocol of member-join in a secret sharing scheme. In: Proceedings of the 2nd Information Security Practice and Experience. 2006, 134−141
29. Yu J, Kong F, Hao R, Li X. How to publicly verifiably expand a member without changing old shares in a secret sharing scheme. In: Proceedings of the IEEE ISI 2008 International Workshops on Intelligence and Security Informatics. 2008, 138−148
30. Castagnos G, Catalano D, Laguillaumie F, Savasta F, Tucker I. Two-party ECDSA from hash proof systems and efficient instantiations. In: Proceedings of the 39th Annual International Cryptology Conference on Advances in Cryptology. 2019, 191–221
31. Doerner J, Kondi Y, Lee E, Shelat A. Threshold ECDSA from ECDSA assumptions: the multiparty case. In: Proceedings of 2019 IEEE Symposium on Security and Privacy (SP). 2019, 1051−1066
32. Wang C, Wang D, Tu Y, Xu G, Wang H. Understanding node capture attacks in user authentication schemes for wireless sensor networks. IEEE Transactions on Dependable and Secure Computing, 2022, 19(1): 507–523
33. Qiu S, Wang D, Xu G, Kumari S. Practical and provably secure three-factor authentication protocol based on extended chaotic-maps for mobile lightweight devices. IEEE Transactions on Dependable and Secure Computing, 2022, 19(2): 1338–1351
34. Wang Q, Wang D, Cheng C, He D. Quantum2FA: efficient quantum-resistant two-factor authentication scheme for mobile devices. IEEE Transactions on Dependable and Secure Computing, 2021, doi: 10.1109/TDSC.2021.3129512
35. Li Z, Wang D, Morais E. Quantum-safe round-optimal password authentication for mobile devices. IEEE Transactions on Dependable and Secure Computing, 2022, 19(3): 1885–1899



Huiqiang Liang is currently pursing his MS degree at School of Mathematics and Statistics, Wuhan University, China. His main research interests include cryptography and information security and SMPC.

Jianhua Chen received the MS and PhD degrees from School of Mathematics and Statistics, Wuhan University, China in 1989 and 1994, respectively. He is currently a professor of the Applied Mathematics Department, Wuhan University, China. His current research interests include number theory, information security, and network security.