Vol.2021-DPS-186 No.34
Vol.2021-CSEC-92 No.34
2021/3/15

Threshold ECDSA for securing digital assets in combination with blockchain

Wang Zhaobo[1,a)] Atsuko Miyaji[1,2,b)]

Abstract: Because of the explosion of cryptocurrencies and blockchain technology, decentralized security technology also brings many new challenges. One of the big challenges is to construct a decentralized protocol that is secure and usable. For this challenge, Threshold ECDSA signatures have received much attention in recent years due to the widespread use of ECDSA in cryptocurrencies. Threshold ECDSA signature enhances the decentralization property and can be used to protect digital assets. Although many threshold signature protocols have been proposed, they are unsuitable for deployment scenarios. This is because their protocols are largely lacking in robustness and are structurally unable to integrate into existing blockchains. We propose a threshold ECDSA signature scheme with basic robustness in the signature phase based on the ElGamal commitments. In addition to that, the scheme can be integrated in other blockchains to improve better usability.

Keywords: ECDSA, threshold, blockchain

1. Introduction

With the development of blockchain technology, digital signatures have also received enough attention as the core of many blockchain application authentication mechanism such as Bitcoin[13]. In cryptocurrencies, digital signatures are used to authenticate transactions, and the ability to generate a signature is equivalent to the ability to spend one's money. The commonality between all of these applications is that the theft or loss of the signing key can be catastrophic, and a key difficulty is how to store signing keys in a manner that is both easy to use and resilient to theft and loss.

Threshold cryptography, and threshold signature in particular, has greatly reduced the risk of losing a single signature. In a threshold signature scheme, signing keys are distributed among several parties which need to jointly generate a digital signature. More specifically, a (t, n) threshold signature scheme means that a key is split into n shares and a parameter t is defined such that an adversary which compromises t or fewer parties is unable to generate a valid signature and learn no information about others secrets. On the other hand, a set of $t+1$ parties can directly jointly generate a signature without reconstructing signing key. This is very effective against the risk of the signing key being controlled by an adversary, and even if the adversary controls a small number of parties, the remaining parties can still issue a vaild signature.

The most popular signature algorithm in deployment is the Elliptic Curve Digital Signature Algorithm (ECDSA)[11], which has been used in many applications especially Bitcoin. More specifically, each Bitcoin account is associated with an ECDSA public key, and each transaction sent from such an account is confirmed by an appropriate ECDSA signature. Build threshold ECDSA is a very challenging task for over two decades[10], in recent years, this research has been picking up again in order to more effectively protect the assets of cryptocurrencies. The early threshold ECDSA is not able to achieve threshold optimality (i.e. a protocol that for a prespecified threshold $1 \leq t \leq N$ allows any $t+1$ parties to sign, at the same time being resistant to an adversary controlling t parties), it requires at least $2n/3$ out of n parties to sign a message. There has been a significant achievement in 2016, threshold optimal were obtain for the general case [9]. This work was followed by Gennaro and Lindell[7, 12] in 2018 to reach a more general threshold optimum using different methods. Thus, the study of threshold signatures has again started to be studied in many different ways.

Another interesting idea is to create bridges between the various blockchains. For example, it is more safe and convenient to allow to deposit BTC on a Ethereum and move it there freely, Technically, the problem with building such a bridge from limitations on the Bitcoin side, for there is not enough scripting support in the Bitcoin mechanism. One particular approach to build such a bridge between Bitcoin and Ethereum is wBTC[16] which uses a trusted central authority to hold custody over the corresponding Bitcoin account. However, such a solution is not quite satisfactory though, as Bitcoin and Ethereum were built to avoid central authorities in the first place, and thus having a decentralized bridge would be preferred. For Bitcoin, threshold ECDSA is a good method of decentralizing assets, but the existing threshold ECDSA protocols[7, 9, 12, 4] are not quite suitable for this pur-

[1] Graduate School of Engineering, Osaka University, 2-1 Yamadaoka, Suita, Osaka, 565-0871 Japan
[2] Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, Ishikawa 923-1292 Japan
[a)] wangz@cy2sec.comm.eng.osaka-u.ac.jp
[b)] miyaji@comm.eng.osaka-u.ac.jp

pose. For it needs to firstly elect a committee of t honest parties, if any party of them crushes or compromised by an adversary, then the protocol fails and need to restart.

In this work, we present a new threshold ECDSA protocol that is designed to doesn't require a choice of an "honest committee". Additionally, we have split the threshold ECDSA into two parts, online and offline to reduce the online computation resources, in particular, the online part of our protocol can be combined with Ethereum's smart contracts for better robustness and usability.

The paper is structured as follows. Section 2. describes the basic definitions and knowledge required. Section 3. describes some of the prior research relevant to our study. Section 4. describes the ours work in this paper. Section 5. provides a summary.

2. Preliminary

The protocol is described with respect to a a cyclic group $\mathbb{G}$ of prime order q, a fixed generator g, a hash Function $F : \mathbb{G} \to \mathbb{Z}_q$.

2.1 The Digital Signature Standard

The Digital Signature Algorithm (DSA) was proposed by Kravitz in 1991, and adopted by NIST in 1994 as the Digital Signature Standard (DSS). ECDSA[11], the elliptic curve variant of DSA, has become quite popular in recent years, especially in cryptocurruencies. Our results can be applied to DSA or ECDSA, where we briefly describe a generic DSA scheme as follows.

- **Key-Gen** On input the security parameter λ, randomly uniformly selects a private key $x \in \mathbb{Z}_q$, and compute public key $y = g^x$ in $\mathbb{G}$.
- **Sig** On input an message m
  - select $k \in Z_q$ randomly uniformly
  - compute $R = g^{k^{-1}}$ in $\mathbb{G}$ and $r = H(R)$ in Z_q
  - compute $s = k^{-1}(m + xr)$ *mod* q - output signature (r, s)
- **Ver** On input $m, (r, s), y$
  - check that $r, s \in \mathbb{Z}_q$
  - compute $R' = g^{ms^{-1} \bmod q} y^{rs^{-1} \bmod q} \in \mathbb{G}$
  - Accept iff $H(R') = r$

2.2 Threshold Signature scheme (TSS)

Definition 2.1. *A (t, n)-threshold secret sharing of a secret x consists of n shares $x_1, \cdots, x_n$ such that an efficient algorithm exists that takes as input $t+1$ of these shares and outputs the secret, but t or fewer shares do not reveal any information about the secret.*

TSS likes a common signature scheme, a (t, n)-TSS enables the signing among a group of n players such as any group of at least $t + 1$ of these players can jointly generate a signature, whereas groups of size t or fewer cannot. More formally, TSS mainly consist of two protocols:

- **Thresh-KeyGen**
  Thresh-KeyGen is a distributed key generation(DKG) protocol, with no previously shared key material, but only the parties public identities. Parties can share keys through this protocol and when the protocol successfully completes, each party P_i will get their private shares k_i of the secret key sk. This protocol will also output a public key pk corresponding to the secret key sk and all parties will be aware of the public key.
- **Thresh-Sig**
  Thresh-Sig is the distributed signing protocol which takes as public input a message m to be signed as well as a private input sk_i from each player. It outputs a valid signature.

In addition, there are two sub-protocols **Thresh-Presig** and **Thresh-Reshare**, which are not necessarily included in TSS, but used to calculate the required values for **Thresh-Sig** in advance and to redistribute keys respectively.

2.3 Shamir Secret Sharing

Shamir Secret Sharing scheme is a key part of the composition of TSS. In Shamir Secret Sharing scheme, to share a secret $\sigma \in \mathbb{Z}_q$, the dealer generates a polynomial $p(\cdot)$ of degree t over $\mathbb{Z}_q$ such that $p(0) = \sigma$ and the coefficients a_1 to a_t are random values that are not zero.

$$p(x) = \sigma + a_1 x + a_2 x^2 + \cdots + a_t x^t \bmod q \qquad (1)$$

Each party P_i receive a share $\sigma_i = p(i) \bmod q$.

2.4 Ethereum

Ethereum is a decentralized, open-source blockchain featuring smart contract functionality. Ether (ETH) is the native cryptocurrency of the platform. More specifically, Ethereum is a permissionless, non-hierarchical network of computers (nodes) which build and come to consensus on an ever-growing series of "blocks", or batches of transactions, known as the blockchain. Each block contains an identifier of the block that it must immediately follow in the chain if it is to be considered valid. Whenever a node adds a block to its chain, it executes the transactions therein in their order, thereby altering the ETH balances and other storage values of Ethereum accounts. These balances and values, collectively known as the state, are maintained on the node's computer separately from the blockchain, in a Merkle Patricia tree.

Definition 2.2. *A smart contract is a computer program or a transaction protocol which is intended to automatically execute, control or document legally relevant events and actions according to the terms of a contract or an agreement.*

Ethereum implements a Turing-complete language on its blockchain, a prominent smart contract framework.

2.4.1 Ethereum as a Broadcast Channel

In our paper, each party of the online sign protocol actively monitors the Ethereum blockchain. In particular, the client can watch all transactions to the address of the pre-deployed online sign contracts. A message is broadcast by issuing an Ethereum transaction, which effectively executes a function within the online sign smart contract when the transaction is mined within a block in the Ethereum network.

2.5 Zero Knowledge Proof

TSS protocols use zero-knowledge proof (ZKP) to ensure that parties do not deviate from the protocol. ZKP appear most commonly in the following type of situation: a party holds some secret piece of data x that along with a public part d is to be applied

in a function $F(\cdot,\cdot)$ that used both the secret and the corresponding public part. In order to make sure the value y gotten $F(w,d)$ is indeed the result of this computation, this party should publish a "certification" π along with y that proves the party holds a secret w such that $F(w,d) = y$. The method of zero-knowledge proofs used in this paper is mainly derived from [6, 15], and which is discussed in further in Lindell's work[12].

2.6 Additively Homomorphic Encryption

Additively homomorphic encryption does the important transformation of the product form of share to the additive form in TSS. An additively homomorphic encryption scheme consists of three algorithms **KeyGen, Enc, Dec**, such that:

- $(pk, sk) \leftarrow KGen(\lambda)$, λ are security parameters.
- $Enc(pk, m) = c$ is a probabilistic algorithm, where $m \in \mathcal{M}$ is a message in the message space $\mathcal{M}$ and $c \in C$ is the corresponding ciphertext in the ciphertext space $mathcalC$.
- $Dec(sk, c) = m$ is a deterministic algorithm.
- There exist two operations $\oplus : C \times C \to C$ and $\otimes : \mathbb{Z} \times C \to C$ such that:

$$m_1 + m_2 = Dec(sk, (Enc(sk, m_1) \oplus Enc(pk, m_2))) \quad (2)$$

$$k \cdot m = Dec(sk, k \otimes Enc(pk, m)) \quad (3)$$

2.7 ElGamal Commitments

ElGamal Commitments is also an important components of the whole protocol, they are also called "ElGamal in-the-Exponent".

Given a private key $d \in \mathbb{Z}_q$, the public key for ElGamal commitments h is g^d. With the respect to this public key, we define an ElGamal commitment to $x \in \mathbb{Z}_q$ as

$$E(x) = (g^r, h^r g^x) \in \mathbb{G}^2 \quad (4)$$

$r \in Z_q$ is uniformly randomness.

A important property of ElGamal commitments is that they are addtively homomorphic, more precisely, for $x1, x2, r1, r2 \in \mathbb{Z}_q$ we have

$$E(x1, r1) \cdot E(x2, x2) = (x1 + x2, r1 + r2) \in \mathbb{G}^2 \quad (5)$$

where $\cdot$ denotes the coordinate-wise multiplication in $\mathbb{G}^2$. This property can be used to update the commitments and can also be used to calculate values on the exponent.

2.8 Multiplicative-to-additive share conversion protocol (MtA)

Multiplicative-to-additive share conversion protocol (MtA) aims to convert the multiplicative form of secrets to the additive form of secrets. In the threshold ECDSA, this protocol is generally used to convert the secret form of the product sharing k, x to a, b such that $kx = a + b$ in Z_q. We briefly describe this protocol as follows:

Algorithm 1 MtA [7]

Input: secret a, b; additive homomorphic encryption algorithm E_{pk_i}
Output: secret α, β, s.t. $\alpha + \beta = ab$
Parties: P_i, P_j

1: P_i initiates the protocol
   - sends $c_A = E_{pk_i}(a)$ to P_j
   - does ZKP with a
2: P_j chooses $\beta' \leftarrow \mathbb{Z}_N$
   - computes $c_B = b \times c_A + E_{pk_i}(\beta')$
   - does ZKP with b
   - sends c_B to P_i
   - sets $\beta = -\beta'$
3: P_i decrypts c_B to obtain α'
   - sets $\alpha = \alpha \ mod \ q$
4: **return** α for P_i, β for P_j

3. Related Work

In this section, we focus on some of the important components of our protocol, which come from [12, 8]. We will also give a brief introduction to the protocols in [12], as this protocol the inspired our work.

3.1 Lindell's scheme in 2018 [12]

In 2018, Lindell presented another ECDSA TSS that has practical distributed key generation and fast signing. The difference with Gennaro's work[7] is that they replace the Paillier additively homomorphic encryption with ElGamal in-the-exponent that also supports additive homomorphism. This makes that it can compute an encrypted signature in a similar way to that of [7], except that upon decryption the parties are only able to receive $s \cdot G$ (where G is the generator point of the Elliptic curve group) and not s itself, where s is the desired portion of the signature. This is due to the fact that we use ElGamal in-the-exponent and so it only obtain the result "in the exponent".

According to ElGamal in-the-exponent, we can use the property of additive homomorphism to compute two ciphertext in ElGamal. If there are two ciphertexts (A, B), (C, D) are encrypted from $E(m)$ and $E(m')$ respectively by ElGamal encryption. More specifically, If $(A, B) = (r \cdot G, r \cdot P + m \cdot G)$ and $(C, D) = (s \cdot G, s \cdot P + m' \cdot G)$, we can calculate $(A + C, B + D) = ((r + s) \cdot G, (r + s) \cdot P + (m + m') \cdot G) = E(m + m')$ In addition, the scalar multiplication $(c \cdot A, c \cdot B) = (cr \cdot G, cr \cdot P + cm \cdot G) = E(cm)$ also can be calculated by homomorphic property.

On the basis of this property, Lindell constructs a threshold ecdsa available that achieves threshold optimality. This work is based on universal composability (UC) model[3], In order not to lose the original meaning, we follow the rules of the UC model to describe this scheme in detail.

3.1.1 $\mathcal{F}_{mult}$ functionality

The most important point of Lindell's work is that it defined a secure multiplicative subprotocol $\mathcal{F}_{mult}$ which private the functionality that each party P_i provide a_i and b_i for input, and then returing c_i to P_i where $c_1, \cdots, c_n$ are random under the constraint that $\sum c_i = \sum a_i \sum b_i \ mod \ q$. Now we give the details of the $\mathcal{F}_{mult}$ functionality as follows:

1. $\mathcal{F}_{mult}$ works with parties $P_1, \cdots, P_n$, and we defines the set

of indices of corrupted parties $C \in [1, \cdots, n]$, $(\mathbb{G}, G, q)$ for a group $\mathbb{G}$ of order q with generator G, and sid is for private session id of parties.

2. upon receiving (**init**, $(\mathbb{G}, G, q)$) from all parties, $\mathcal{F}_{mult}$ stores $(\mathbb{G}, G, q)$) and ignores some $(\mathbb{G}, G, q)$) has already been stored.
3. upon receiving (**input**, sid, a_i) from a party $P_i \in C$, if no value $(sid, i, \cdot)$ is stored, then $\mathcal{F}_{mult}$ stores (sid, i, a_i) else ignores the message.
4. upon receiving (**input**, $sid, \cdot$) from a party $P_i \notin C$, if no value $(sid, i, \cdot)$ is stored, then $\mathcal{F}_{mult}$ chooses a random $a_i \leftarrow \mathbb{Z}_q$ and stores (sid, i, a_i) and returns (**input**, sid, a_i) to P_i otherwise ignores the message.
5. if some $(sid, i, \cdot)$ has beed stored for all $i \in [1, \cdots, n]$ then $\mathcal{F}_{mult}$ computes $a = \sum a_i \ mod \ q$ and stores the value a, then sends (**input**, sid) to all parties.
6. upon receiving (**mult**, sid_1, sid_2) from the all parties, $\mathcal{F}_{mult}$ checks (sid_1, a) and (sid_2, b) values have been stored. If yes, then $\mathcal{F}_{mult}$ sets $c = ab \ mod \ q$ and sends (**mult**, sid_1, sid_2, c) to all parties.
7. upon receiving (**affine**, sid_1, sid_2, x, y) from the honest party P_i with $x, y \in \mathbb{Z}_q$, $\mathcal{F}_{mult}$ checks (sid_1, a) has been stored. If yes, $\mathcal{F}_{mult}$ computes $b = a \cdot x + y \ mod \ q$, stores (sid_2, b) and sends ($affine$, sid_1, sid_2, x, y) to P_i.
8. upon receiving (**element-out**, sid) from all parties, $\mathcal{F}_{mult}$ checks that some (sid, a) has been stored. If yes, $\mathcal{F}_{mult}$ computes $A = A \cdot G$ and sends (**element-out**, sid, A) to all parties.

With the definition of $\mathcal{F}_{mult}$ subprotocol, we can continue to **Thresh-KeyGen** and **Thresh-Sig** in Lindell's work.

Algorithm 2 Thresh-KeyGen [12]

Input: $(\mathbb{G}, G, q)$;
Output: P;
Parties: $P_1, \cdots, P_n$

1: each P_i sends (**init**, $(\mathbb{G}, G, q)$) to $\mathcal{F}_{mult}$ to run the initialization phase.
2: each P_i sends (**input**, sid_{gen}) to $\mathcal{F}_{mult}$ and receives back (**input**, sid_{gen}, x_i). Denote $x = \sum x_i$ and $P = x \cdot G$.
3: P_i waits to receives (**input**, 0). Note that identifier 0 is associated with the private key x.
4: P_i sends (**element-out**, 0) to $\mathcal{F}_{mult}$.
5: P_i receives (**element-out**, $0, P$) from $\mathcal{F}_{mult}$.
6: **return** P_i saves P locally for public key.

Algorithm 3 Thresh-Sig [12]

Input: sid, m, P;
Output: (r, s);
Parties: $P_1, \cdots, P_n$

1: P_i sends (**input**, sid_1) and (**input**, sid_2) to $\mathcal{F}_{mult}$, and receives back (**input**, sid_1), k_i and (**input**, sid_2), p_i. Denote $\sum k_i = k$ and $\sum p_i = p$.
2: after receiving (**input**, sid_1) and (**input**, sid_2) from $\mathcal{F}_{mult}$.P_i sends (**mult**, sid_1, sid_2) and (**element-out**, sid_1) to $\mathcal{F}_{mult}$.
3: P_i receives (**mult**, sid_1, sid_2, τ) and (**mult**, sid_1, R) from $\mathcal{F}_{mult}$. Note that $\tau = kp$ and $R = k \cdot G$.
4: P_i computes $r = R \ mod \ q$.
5: P_i sends (**affine**, $0, sid_3, r, m$) to $\mathcal{F}_{mult}$. (recall that identifier 0 is associated with the private key x, and thus sid_3 will be associated with $m + x \cdot r \ mod \ q$).
6: P_i sends (**mult**, sid_2, sid_3) to $\mathcal{F}_{mult}$.
7: P_i receives (**mult**, sid_2, sid_3, β) from $\mathcal{F}_{mult}$. (note that $\beta = p \cdot (m' + x \cdot r)$).
8: P_i computes $s = \tau^{-1} \cdot \beta \ mod \ q$
9: **return** P_i outputs (r, s).

Lindell's work looks complicated, but we can simply verify it's correctness. In the $\mathcal{F}_mult$ subprotocol, we can observe that τ is the product of $k = \sum_{k_i}$ and $p = \sum_{p_i}$, and that β is the product of the same p with $(m' + xr)$. According to the $\mathcal{F}_mult$ subprotocol, each party P_i will have an secret share $\alpha_i = \frac{m'}{n} + rx_i$, and $\alpha = \sum \alpha_i = (m + xr)$. Hence $s = \tau^{-1} \cdot \beta = k^{-1}p^{-1}p\alpha^{-1} = k^{-1}(m + xr)$, thus (r, s) is a valid ECDSA signature with secret key x.

4. Proposal

In this section, we would modify Lindell's work[12] to adding some new functionality . Like previous works, our protocol consists 3 subprotocol **Thresh-KeyGen**,**Thresh-PreSig** and **Thresh-Sig**, which we will descibe in detail below. In the protocol, we also use commitment scheme and **MtA**(Algorhitm 1) to ensure secure key distribution. Our protocols have the following three properties:

- **Identifiable and attributable aborts.**
  - TSS can effectively identity the identity of potential attackers and abort current protocols.
- **Off/Online Processing.**
  - In the offline phase, TSS processes and saves the results that are not related to message m
  - In the online phase, tss loads the previously saved results and uses them to sign message m, which has the advantage of significantly reducing the online processing time.
- **Integration into Ethernet.**
  - The online processing can be done using Ethernet's smart contracts.

4.1 Subprotocols

We proceed to formulate and discuss some auxiliary subprotocols that are used in our scheme. These includes 3 auxiliary subprotocols **MtA**, **ElGamal-KeyGen** and **Thresh-Reshare**. we omit **MtA** and **ElGamal-KeyGen** (Algorhitm 1) for clarity from previous section and start a brief description of **Thresh-Reshare** in this section. **Thresh-Reshare** is a very important sub-protocol in our proposed TSS, whose main purpose is to convert additive

secret sharing into Shamir's secret sharing. To ensure the security of this subprotocol, it is necessary to use ElGamal commitments in the protocol, the specific steps of this subprotocol are as follows:

Algorithm 4 Thresh-Reshare

Input: g,secret $a_1, \cdots, a_n$, randomness $r_1, \cdots, r_n$, public ElGamal commitments $(c_1, \cdots, c_n)$ and $c_i = E(a_i, r_i)$;

Output: secret share $\hat{a_1}, \cdots, \hat{a_n}$ and commitments string $c_{a_1}, \cdots, c_{a_2}$

Parties: $P_1, \cdots, P_n$

1: Each P_i publish a ZKP of a_i, r_i such that $E(a_i, r_i) = c_i$.
2: Each P_i pick a random degree $t+1$ polynomial $f_i(\cdot)$ and set $f_i(0) = a_i$.
3: Each P_i make the ElGamal commitment of coefficients of $f_i(\cdot)$: $c_{f_i} = E(f_{ij}, r_{ij})$ (f_{ij} is coefficients of $f_i(\cdot)$, $j \in [t]$ and r_{ij} is randomness) and broadcasts c_{f_i}.
4: After receiving commitments from all other parties, all parties do ZKP of f_{ij} and cf_0.
5: Each P_i decommits other's commitments string and checks correntness.
6: Each P_i computes $E(f_i(j), r_{i\to j}) = \prod E_{k\in[t]}(f_{ik}, r_{ik})^{j^k}$ for $j \in [N]$.
7: Each P_i recommits to $f_i(j)$ by sampling a fresh randomizing element $r'_{i\to j}$ and publishs previous value $E(f_i(j), r_{i\to j})$ and makes ZKP to check $f_i(j)$ from $E(f_i(j), r_{i\to j})$ and $E(f_i(j), r'_{i\to j})$.
8: Each P_i sends $(f_i(j), r'_{i\to j})$ privately to P_j.
9: After receiving the shares from other parties, P_i check consistency from previous commitments string.
10: If all the proof are corrent, P_i computes $\hat{a_i} = \sum_{j\in[N]} f_j(i)$, $r'_i = \sum_{j\in[N]} r'_{j\to i}$.
- P_i also computes $E(\hat{a_i}, r'_i) = \prod_{j\in[N]} E(f_j(i), r'_{j\to i})$.
11: Each P_i recommits to s_i by sampling a fresh randomizing element $r''_k \in \mathbb{Z}_q$ and does ZKP to check consistency with previous commitments.
12: **return** $\hat{a_i}$ as Shamir's secret share and $E(\hat{a_i}, r''_i)$ as the public commitment of P_i.

The main idea behind the subprotocol is that Each P_i generates a random polynomial $f_i(\cdot)$ of degree t such that $f_i(0) = a_i$. This idea comes from Feldman's VSS[5] and we would like to use the linear property of polynomial to generate a hidden polynomial $f(\cdot) = f_1(\cdot) + \cdots + f_N(\cdot)$ to share secret a. Moreover, the adversary cannot generate its polynomials based on the ones by honest parties, and $f(\cdot)$ and a is also uniformly random subject to this condition. In order for each party P_i to learn its $\hat{a_i} = f(i)$, after generating $f_i(\cdot)$, P_i will send $f_i(j)$ to another party P_j. Consequently, P_i receives $N-1$ values $f_j(k)$ for each $j \in [N]$ and can compute $f(k) = \sum_{j\in[N]} f_j(i)$.

4.2 Protocol Description

This section is devoted to presenting our mail protocol of our work. Before we dive into that, we refer to the definition of some notation here again, $\mathbb{G}$ represents an elliptic curve group with order q and a generator g. $E(\cdot,\cdot)$ stands for ElGamal encryption algorithm. In the offline part of our protocol(**Thresh-KeyGen**,**Thresh-PreSig**), we assume that there is a reliable public broadcast channel and a point-to-point private channel during the communication.

4.2.1 Thresh-KeyGen

In Thresh-KeyGen, it can be divided into three main parts, they are generating keys for ECDSA, the keys for ElGamal and the secret parameters for Paillier encryption scheme. we now present the details of the protocol.

Algorithm 5 Thresh-KeyGen

Input: generator g;

Output: public key shares $(g^{x_1}, \cdots, g^{x_n})$; secret key shares $(x_1, \cdots, x_n)$; ElGamal encryption algorithms $(Ep_1, \cdots, Ep_i)$; ElGamal public key h; public key y

Parties: $P_1, \cdots, P_n$

1: Each P_i randomly selects $x_i \in \mathbb{Z}_q$ and make commitment of g^{x_i} and broadcasts the commitment string.
- Each P_i does the ZKP of x_i and decommit the commitment string to g^{x_i}.
- If all the proof are corrent, then parties set g^{x_i} as public share and form the public key $y = \prod g^{x_i}$ for public key.
2: Parties call the **ElGamal-KeyGen** subprotocol (Algorhitm **??**) and receive h and the corresponding public information.
3: P_i randomly selects two prime number p_i, q_i and let $N_i = p_i q_i$ be the RSA modulus associated with public key Ep_i, and does ZKP of p_i, h_i.
4: **return** y is public key for ECDSA, h is public key for ECDSA, Ep_i is public key for Paillier encryption[14], g^{x_i} is public share for P_i, x_i is secret share for P_i.

4.2.2 Thresh-PreSig

We have divided the signature phase into two parts, the presignature phase is the part that does not contain the required signature information m. Therefore, this part can be executed "offline", which means that parties can save the information they need in advance and load it up when they need it, the protocol is as follows:

Algorithm 6 Thresh-PreSig

Input: generator g; public key shares $(g^{x_1}, \cdots, g^{x_n})$; secret key shares $(x_1, \cdots, x_n)$, ElGamal encryption algorithms $(Ep_1, \cdots, Ep_i)$; ElGamal public key h; public key y;

Output: private secret shares $(\hat{\delta}_i, \hat{\gamma}_i, \hat{\sigma}_i)$ and corresponding commitments.

Parties: $P_1, \cdots, P_n$

1: Each P_i randomly selects $\gamma_i \in \mathbb{Z}_q$ and $k_i \in \mathbb{Z}_q$. we note that $\sum_{i\in[N]} k_i = k$ and $\sum_{i\in[N]} \gamma_i = \gamma$.
- Each P_i commits γ_i and gets $E(\gamma_i)$ and broadcasts it.
- Each P_i computes $E(\gamma) = \prod_{i\in N} E(\gamma_i)$.

2: Every Pair parties P_i, P_j engages in **MtA** as follows:
- P_i, P_j run **MtA** with shares k_i, γ_j respectively. Let a_{ij}(resp. b_{ij}) be the share received by P_i(resp. P_j) at the end of protocol, i.e. $k_i\gamma_j = a_{ij} + b_{ij}$.
- P_i, P_j run **MtA** with shares x_i, γ_j respectively. Let u_{ij}(resp. v_{ij}) be the share received by P_i(resp. P_j) at the end of protocol, i.e. $x_i\gamma_j = u_{ij} + v_{ij}$.

3: P_i sets $\delta_i = k_i\gamma_i + a_{ij} + b_{ij}$. Note that $\sum_{i\in[N]} \delta_i = k\gamma$.
- P_i commits δ_i and does ZKP of it and broadcasts it.
- P_i chooses a random r_i and publish $E(\gamma \cdot k_i) = E(\gamma_i)^{(k_i)} \cdot E(0; r_i)$ and does ZKP of it.
- If all the proof are corrent, each P_i computes $E(\gamma \cdot k)$ and $E(\delta) = \prod_{i\in[N]} E(\delta_i)$, then P_i checks consistency between $E(\delta)$ and $E(\gamma \cdot k)$(detail in Appendix). Accept if result is correct, otherwise abort.

4: P_i sets $\sigma_i = x_i\gamma_i + u_{ij} + v_{ij}$. Note that $\sum_{i\in[N]} \sigma_i = x\gamma$.
- P_i commits σ_i and does ZKP of it and broadcasts it.
- P_i chooses a random r_i' and publish $E(\gamma \cdot x_i) = E(\gamma_i)^{(x_i)} \cdot E(0; r_i')$ and does ZKP of it.
- If all the proof are corrent, each P_i computes $E(\gamma \cdot x)$ and $E(\sigma) = \prod_{i\in[N]} E(\sigma_i)$, then P_i checks consistency between $E(\sigma)$ and $E(\gamma \cdot x)$(detail in Appendix). Accept if result is correct, otherwise abort.

5: P_i commits δ_i with ElGamal commitment and get $c_{\delta_i} = E(\delta_i, r_{1i})$.
- commits σ_i with ElGamal commitment and get $c_{\sigma_i} = E(\sigma_i, r_{2i})$.
- commits γ_i with ElGamal commitment and get $c_{\gamma_i} = E(\gamma_i, r_{3i})$.

6: P_i runs **Thresh-Reshare** with input $(c_{\delta_i}, \delta_i, r_{1i})$ and get output $(\hat{\delta}_i, c_{\hat{\delta}_i})$.
- P_i runs **Thresh-Reshare** with input $(c_{\gamma_i}, \gamma_i, r_{2i})$ and get output $(\hat{\gamma}_i, c_{\hat{\gamma}_i})$.
- P_i runs **Thresh-Reshare** with input $(c_{\sigma_i}, \sigma_i, r_{3i})$ and get output $(\hat{\sigma}_i, c_{\hat{\sigma}_i})$.

7: **return** P_i saves all shares and commitments $(\hat{\delta}_i, \hat{\gamma}_i, \hat{\sigma}_i)$ and corresponding commitments $com(\hat{\delta}_i), com(g^{\hat{\gamma}_i}).com(g^{\hat{\sigma}_i})$ under **presig[p]**.

After the parties execute the **Thresh-PreSig** protocol, each of them stores private shares of the protocol and corresponding commitments, which will be used later in the online signing phase.

4.2.3 Thresh-Sig

Before the online signature protocol is executed, parties hold the Shamir's secret sharing $(\hat{\delta}_i, \hat{\gamma}_i, \hat{\sigma}_i)$ and the corresponding ElGamal commitments $com(\hat{\delta}_i), com(g^{\hat{\gamma}_i}).com(g^{\hat{\sigma}_i})$ in **presig[p]**. We note that in Ether's smart contracts, all values are public and there are no one-to-one private channels. So, in the online signature protocol, we will use the discrete logarithm problem to hide that certain values are exposed, and we need to ensure that all operations do not pass through a private channel. we describe the protocol in detall:

Algorithm 7 Thresh-Sig

Input: security parameter g, h, y; Shamir's secret shares $(\hat{\delta}_i, \hat{\gamma}_i, \hat{\sigma}_i)$, ElGamal commitments $com(\hat{\delta}_i), com(g^{\hat{\gamma}_i}), com(g^{\hat{\sigma}_i})$.

Output: signature (r, s)

Parties: $P_1, \cdots, P_n$

1: Set the set of all potential attackers to S. If $|S| > N - t$ abort.

2: Wait for $t + 1$ parties and then loading all information from **presig[p]**.

3: Each P_i recommits $\hat{\delta}_i$ and does ZKP with $com(\hat{\delta}_i)$.
- If all the proofs are correct, P_i publishs $\lambda_{i,\delta}\hat{\delta}_i(\lambda_{i,\delta}$ is appropriate Lagrangian coefficient) and computes $\sum_{i\in T} \lambda_{i,\delta}\delta_i = k\gamma$.

4: Each P_i recommits $g^{\hat{\gamma}_i}$ and does ZKP with $com(g^{\hat{\gamma}_i})$.
- If all the proofs are correct, P_i publishs $g^{\lambda_{i,\gamma}\hat{\gamma}_i}(\lambda_{i,\gamma}$ is appropriate Lagrangian coefficient) and computes $\prod_{i\in T} g^{\lambda_{i,\gamma_i}\hat{\gamma}_i} = g^{\gamma}$.

5: Each P_i computes $r = g^{\gamma^{-}\delta} = g^{-k}$.

6: Each P_i recommits $\hat{\sigma}_i$ and does ZKP of it with $com(g^{\hat{\sigma}_i})$.
- If all the proofs are correct, P_i computes $s_i = \delta^{-1}(\lambda_{i,\gamma}\hat{\gamma}_i H(m) + \lambda_{i,\sigma}\hat{\sigma}_i r)$ and broadcasts it.

7: Each P_i commits s_i and does ZKP of it.
- If all the proofs are correct, computing $s = \sum_{i\in T} s_i = (k\gamma)^{-1}(\gamma H(m) + r\gamma x) = k^{-1}(H(m) + rx)$.

8: **return** if (r, s) is a valid signature then return (r, s), otherwise abort.

4.2.4 Thresh-Sig in Ethereum

According to the **Thresh-Sig** protocol, it is simple to see that the values that participants need to publish are $\hat{\delta}_i, g^{\hat{\gamma}_i}, g^{\hat{\sigma}_i}$ and commitments $com(\hat{\delta}_i), com(g^{\hat{\gamma}_i}), com(g^{\hat{\sigma}_i})$. In fact, only $\hat{\delta}_i$ will be directly exposed to the parties, The remaining values are publicly available based on the discrete logarithm or ZKP. For ZKP, we can use a common non-interactive ZKP (NIZK) technique[1, 2] to show the correctness of each secret value.

The utilization of a smart contract platform such as Ethereum also enables us to readily implement dynamic participation strategies. If the choice is made to employ this protocol feature, the set of parties N which run the sign protocol is not defined a priori, but rather obtained in an additional registration phase, executed at the beginning of the sign protocol. For this purpose, the creator of the corresponding smart contract specifies a set of participation rules at the time of contract creation. A participation rule specifies under which condition a particular Ethereum account is allowed to "join" the set N. Briefly, this means that we do not have to define a committee in advance. Within the limitations of the Ethereum platform, arbitrary smart contract code can be used to define some participation rules. In the following, we provide basic examples for participation rules as follows:

- **First come, First serve**: Only the first $t+1$ parties to register are allowed to join the **Thresh-Sig** protocol.
- **Security deposit**: Only parties, which provide a security deposit of at least X Ether are allowed to join the protocol.
- **Highest bidding**: The N parties, which provided the highest amount of security deposit are allowed to join the protocol.

For conditions 1 and 2 the participation rules are checked as soon as a registration transaction is included in an Ethereum block. Only upon success is the issuer of the transaction added to the set N, tracked within the smart contract. The implementation of condition 3 is rendered slightly more complex, It is not a necessary mechanism and conflicts with condition 1, but it can increase the cost of attacks. In this case, the smart contract keeps track of the

set N consisting of up to $t + 1$ parites and their provided security deposits. And we accept only the $t + 1$ parites with the most deposits for the online **Thresh-Sig** protocol during the registration phase.

5. Related Work

We propose a new threshold ECDSA protocol with the more features. From a general point of view, our results can be summarized as follows. First, we added an offline processing subprotocol based on Lindell's work[12], the intermediate values calculated by offline pre-processing can effectively reduce the time of online computation. Second, The architecture of our protocol can be signed without having to generate a committee in advance, this will enhance the flexibility of the protocol. Finally and most importantly is the online phase of our protocol can be implemented in a smart contract with Turing completeness (e.g. Ethereum), This allows the protocol to achieve more functionality for protecting digital assets through smart contracts.

Acknowledgments This work is partially supported by enPiT(Education Network for Practical Information Technologies) at MEXT, and Innovation Platform for Society 5.0 at MEXT

References

[1] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[2] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. *Technical Report/ETH Zurich, Department of Computer Science*, 260, 1997.

[3] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[4] I. Damgard, Thomas P. Jakobsen, J. Nielsen, J. Pagter, and Michael Baksvang stergard. Fast threshold ecdsa with honest majority. *IACR Cryptol. ePrint Arch.*, 2020:501, 2020.

[5] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[6] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[7] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1179–1194, 2018.

[8] Rosario Gennaro and Steven Goldfeder. One round threshold ecdsa with identifiable abort. *IACR Cryptol. ePrint Arch.*, 2020:540, 2020.

[9] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security. In *International Conference on Applied Cryptography and Network Security*, pages 156–174. Springer, 2016.

[10] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold dss signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 354–371. Springer, 1996.

[11] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[12] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1837–1854, 2018.

[13] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[14] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[15] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[16] WBTC.Network. Wrapped tokens a multi-institutional framework for tokenizing any asset. January 2019.