Zhang *et al. J Surveill Secur Saf* 2024;5:213-33
DOI: 10.20517/jsss.2024.30

Journal of Surveillance, Security and Safety

Research Article

Open Access

Cloud-edge-end based key management for decentralized applications

Jie Zhang[1], Futai Zhang[2]

[1]School of Advanced Technology, Xi'an Jiaotong-Liverpool University, Suzhou 215123, Jiangsu, China.
[2]Fujian Provincial Key Laboratory of Network Security and Cryptology, College of Computer and Cyber Security, Fujian Normal University, No.18 Middle Wulongjiang Avenue, Shangjie, Minhou, Fuzhou 350117, Fujian, China.

Correspondence to: Prof. Futai Zhang, Fujian Provincial Key Laboratory of Network Security and Cryptology, College of Computer and Cyber Security, Fujian Normal University, No.18 Middle Wulongjiang Avenue, Shangjie, Minhou, Fuzhou 350117, Fujian, China. E-mail: futai@fjnu.edu.cn

How to cite this article: Zhang J, Zhang F. Cloud-edge-end based key management for decentralized applications. *J Surveill Secur Saf* 2024;5:213-33. http://dx.doi.org/10.20517/jsss.2024.30

Received: 15 Oct 2024 **First Decision:** 18 Nov 2024 **Revised:** 28 Nov 2024 **Accepted:** 4 Dec 2024 **Published:** 17 Dec 2024

Academic Editor: Qiong Huang **Copy Editor:** Ting-Ting Hu **Production Editor:** Ting-Ting Hu

Abstract

How to securely manage and use private keys of digital signature schemes is a pivotal problem for blockchain-based decentralized applications (DApps), as they determine the ownership of data contents and digital assets. A well-recognized approach for key management is threshold cryptography which distributes private keys to different nodes such that the whole system can tolerate a certain number of failures or corruptions. Motivated by the key management need of DApps, threshold signature has attracted widespread attention, with a number of new constructions being proposed in recent years. However, a practical one for DApps' end users is still lacking, as existing schemes are hard to deploy due to the high cost of communication and distributed features.

In this paper, we combine $(2,3)$-threshold signature with the widely deployed cloud-edge-end (CEE) paradigm. We first propose the CEE key management framework which involves an end node, an edge node and a cloud node maintaining $(2,3)$ shares of end users' private keys. Then, following the framework, we construct a $(2,3)$-threshold elliptic curve digital signature algorithm (ECDSA) scheme with provable security based on the ECDSA assumption. Compared with representative constructions of generalized threshold ECDSA, the new scheme is easy to deploy in applications developed following the CEE framework. Additionally, it addresses share tampering and requires only three messages for distributively issuing a signature, which is lower than the best practice of $(2,2)$-threshold ECDSA (four messages) that cannot tolerate tampering.

Keywords: Key management, ECDSA, threshold digital signature, blockchain, secret sharing, DApps

1. INTRODUCTION

1.1. Background and problem statement

Advances of the blockchain technique promote the development of decentralized applications (DApps), which replace the server of traditional applications with decentralized storage[1,2]. A remarkable feature of DApps is that they return the ownership of data produced by end users to themselves, which is controlled by the server in traditional applications. One of the key technologies behind this is digital signature. A user of a DApp possesses a pair of public and private keys, where the public key is published and the private key is known only by the user. The private key is used to issue a signature for every data produced by the user in the DApp system, such that everyone can verify the signature via the public key. A well-known example of DApps is the bitcoin[3], where owning the private key is equivalent to owning all the bitcoins under an address derived from the corresponding public key.

Therefore, how to securely manage and use the private key of digital signature schemes is a pivotal problem for DApps. To address this problem, a variety of software- and hardware-based approaches have been proposed, such as crypto wallets installed in users' terminals, online crypto exchanges managing private keys on behalf of users, and so on[4]. Among various key management methods, a noteworthy technique is threshold cryptography which distributes the private key into multiple distributed nodes in a way that a threshold of them can collaboratively issue a signature or decrypt a ciphertext without recovering the private key. Threshold-based key management systems can tolerate a certain number of failures and corruptions, leaving the system sufficient time to recover from security instances and thereby reducing users' losses caused by the leakage or tampering of private keys.

1.2. Related work

Motivated by the key management need of DApps, threshold digital signature has attracted widespread attention from both academic and industrial communities. As the most widely adopted signature scheme in blockchain platforms and applications, threshold construction for elliptic curve digital signature algorithm (ECDSA) signature has become a hot topic in recent years. Below we review some milestones and representative constructions.

To secure bitcoin wallets, in 2016, Gennaro *et al.*[5] proposed a threshold-optimal (EC)DSA signature scheme that requires the participation of the threshold t nodes to issue a signature. Before that, the best construction[6] requires $2t$ participants in the threshold signing phase. Instead of distributing the private key, Gennaro *et al.*[5] encrypted the private key through an additively homomorphic encryption scheme, and distributed the decryption key in (t, n)-threshold manner. However, the threshold signing protocol involves six rounds of interactions among at least t nodes, and each round requires multiple broadcasting and peer-to-peer communications. Besides, their scheme does not have a specific share renewal algorithm to update all shares after a leaking or tampering attack.

The first series of efficient threshold ECDSA schemes were proposed concurrently by Lindell and Nof[7] and Gennaro and Goldfeder[8] in 2018. The former uses ElGamal decryption "in the exponent" to realize multiplicative-to-additive (MtA) share conversion which is a major building block for threshold ECDSA. The latter instantiates the MtA protocol using the additively homomorphic encryption scheme of Paillier. After the two schemes, new constructions of threshold ECDSA have been continuously proposed every year in top journals and leading conferences in the field of information security and cryptography[9–12]. Till now, improving the practicability of threshold ECDSA remains a hot topic.

Compared to common (t, n)-threshold schemes with large t and n, two-party ECDSA is much more practical for real-world deployment, because it only needs two participants for issuing a signature and is thereby easy to implement. In 2017, Lindell[13] constructed an efficient $(2, 2)$-threshold ECDSA signature which involves three messages in an offline pre-signing phase before the message arrives and one message in the online signing phase. Although this scheme is much more practical than common (t, n) ones, it can only address the leakage problem of private key shares. Once one of the two shares is tampered with or broken, it will be unable to issue a signature anymore. Doerner *et al.*[14] presented a $(2, n)$-threshold ECDSA from ECDSA assumption in 2018. Their 2-out-of-n signing protocol is run with eight messages exchanged among two nodes. This communication cost is too high for common end users of DApps. Tu *et al.*[15] proposed a fast two-party signature based on the combinatorial ECDSA. However, the verification operation of their scheme is different from the original ECDSA.

In summary, reducing communication rounds and improving practicability are the main challenges in the area of threshold ECDSA. While $(2, 2)$-threshold ECDSA schemes have higher practicability compared to (t, n)-threshold ones, the best construction so far still needs eight messages. Besides, $(2, 2)$-threshold schemes only address leakage problem of private key shares. Once one of the two shares is tampered with or broken, the two-party ECDSA will be unable to issue a signature anymore. The minimum threshold setting for addressing both leaking and tampering issues is $(2, 3)$.

1.3. Our work

This paper addresses the key management problem for end users of DApps by combining $(2, 3)$-threshold signature with the cloud-edge-end (CEE) framework. The CEE framework originates from the Internet of Things (IoT), edge computing and cloud computing. It has eventually developed as a widely adopted framework for online computing products and services. Specifically, we present the CEE key management framework consisting of three protocols: the initial splitting protocol which distributes the user's private key via $(2, 3)$-threshold secret sharing into three shares held by the end node, edge node and cloud respectively, the two-party signing protocol which is run by the end and edge nodes to collaboratively generate a signature without recovering the private key, and the update protocol which renews the three shares so that they are incompatible with all previous ones.

Based on the CEE key management framework, we construct a concrete $(2, 3)$-threshold ECDSA scheme with provable security under the ECDSA assumption. To overcome the challenge of distributively computing $s = r^{-1} \cdot (e + sk \cdot r_x) \mod q$, where r is a secure instance from Z_q, $e = H(m)$, and $r_x = x \mod q$ is derived from $(x, y) = r \cdot P$, we adopt the following strategies. To generate r_x in a distributed way, we run a key agreement protocol among the end and edge nodes to generate $(x, y) = r^{-1} \cdot P = (r_1 \cdot r_2)^{-1} \cdot P$ by sampling r_1 in the end node and r_2 in the edge node and then exchanging $r_1^{-1} \cdot P$ and $r_2^{-1} \cdot P$. To compute s from partial signatures $s_1 = r_1 \cdot (e + sk_1 \cdot r_x)$ and $s_2 = r_2 \cdot (e + sk_2 \cdot r_x)$, we first convert $r_1 \cdot r_2 = r$ into $\alpha_1 + \alpha_2 = r$ via the MtA protocol, and convert the $(2, 3)$-threshold shares sk_1 and sk_2 of sk into $(2, 2)$-additive shares $s\hat{k}_1$ and $s\hat{k}_2$ by multiplying them with the Lagrangian coefficients, so that each party can compute s_i as $\alpha_i \cdot e + \alpha_i \cdot sk_i \cdot r_x$ for $i = 1, 2$. Then, by running the MtA protocol again, we output β_1 and γ_1 on the end node and β_2 and γ_2 on the edge node such that $\beta_1 + \beta_2 = \alpha_1 \cdot s\hat{k}_2$ and $\gamma_1 + \gamma_2 = \alpha_2 \cdot s\hat{k}_1$. Finally, the end node computes $s_1 = \alpha_1 \cdot e + (\alpha_1 \cdot s\hat{k}_1 + \beta_1 + \gamma_1) \cdot r_x$, and edge node computes $s_2 = \alpha_2 \cdot e + (\alpha_2 \cdot s\hat{k}_2 + \beta_2 + \gamma_2) \cdot r_x$, which are additive shares of s. Additionally, we present an update protocol for the $(2, 3)$-threshold ECDSA, which is often missing in related work of threshold ECDSA. This protocol can renew all these three shares even if one of them is tampered with. To recover the tampered share, we construct a share recovery protocol based on secure multi-party computation.

The main contributions of this paper are summarized as follows. First, the proposed CEE key management framework provides a practical pattern for securely managing and using private keys in blockchain-based applications. It addresses key management problems of DApp users and demonstrates a practical deployment

Table 1. Notation

Symbol	Meaning
F_p	A finite field with prime order p
$\mathcal{E}$	An elliptic curve defined over F_p
E	The elliptic curve group consists of all points on $\mathcal{E}$ plus the point $\mathcal{O}$ at infinity
G	A subgroup of E with a prime order q and a generator P
q	The order of G
P	The generator of G
Z_q	The prime field with q elements $\{0, \cdots, q-1\}$
H	A hash function $\{0,1\}^* \to Z_q$
$r \leftarrow Z_q$	Generating a random value r from finite field Z_q
$\{m\}_{pk}$	The encryption of message m under key pk
$m \xleftarrow{sk} \{m\}_{pk}$	Recovering m by decrypting $\{m\}_{pk}$ with sk

and application pattern of threshold signature. Although we only illustrate its implementation in ECDSA key management; it can be applied more easily for other signature schemes, given that constructing threshold ECDSA is more challenging than other signature schemes such as Schnorr and Boneh-Lynn-Shacham (BLS) signatures. Second, the constructed $(2, 3)$-threshold ECDSA has a much lower communication cost than parallel schemes with a $(2, n)$ threshold setting where $n \geq 2$, as it only requires three messages exchanged between the end and edge node for issuing a signature, while the best result of $(2, 2)$-threshold ECDSA [13] is four messages. Moreover, the $(2, 3)$-threshold setting can tolerate tampering on one of the private key shares, which is not achieved by the $(2, 2)$-threshold one.

1.4. Organization

The remaining sections of this paper are organized as follows. Section 2 reviews the ECDSA signature scheme and cryptographic tools used in our design. Section 3 presents the CEE key management framework and its threat and security models. Sections 4, 5, and 6 construct the CEE-based $(2, 3)$-threshold ECDSA scheme, prove its security, and evaluate its performance, respectively. Section 7 discusses the application prospect of the CEE-based key management framework and illustrates a case of its use. Finally, Section 8 concludes this paper and proposes future work.

2. PRELIMINARIES

This section reviews the ECDSA scheme and cryptographic tools for our design. Symbols used throughout this paper are defined in Table 1.

2.1. ECDSA

2.1.1 ECDSA signature scheme

Let F_p be a finite field with prime order p, $\mathcal{E}$ be an elliptic curve defined over F_p, E be the elliptic curve group consisting of all points on $\mathcal{E}$ plus the point $\mathcal{O}$ at infinity, G be an elliptic curve group of prime order q, P be a generator of G, $Z_q = \{0, \cdots, q-1\}$, and $H : \{0,1\}^* \to Z_q$ be a cryptographic secure hash function. The ECDSA scheme ($\mathbf{KeyGen}$, $\mathbf{Sign}$, $\mathbf{Verify}$) is reviewed as follows:

- $\mathbf{KeyGen}(\mathbf{1^k})$: outputs a random private key $sk \in Z_q$ and a public key $pk = sk \cdot P$.
- $\mathbf{Sign}_{sk}(m)$ where $m \in \{0,1\}^*$:
  1. Compute $e = H(m)$
  2. Select a random integer $r \in Z_q$
  3. Compute $(x, y) = r \cdot G$
  4. Compute $r_x = x \mod q$
  5. Compute $s = r^{-1} \cdot (e + sk \cdot r_x) \mod q$
  6. Output $\sigma = (r_x, s)$
- $\mathbf{Verify}_{pk}(m, \sigma)$ where $m \in \{0,1\}^*, \sigma = (r_x, s)$:

1. Compute $e = H(m)$
2. Compute $w = s^{-1}$
3. Compute $v_1 = e \cdot w \mod q$
4. Compute $v_2 = r_x \cdot w \mod q$
5. Compute $(x, y) = v_1 \cdot P + v_2 \cdot pk$
6. Compute $v_x = x \mod q$
7. Output 1 if $v_x = r_x$ or 0 otherwise

2.1.2 Security of ECDSA

The existential unforgeability against chosen-message attacks (EU-CMA) is a widely used security model for digital signature schemes. It models the security by the EU-CMA security experiment in which an attacker $\mathcal{A}$ against a digital signature scheme plays with a challenger C who intends to solve a difficult problem. Specifically, the EU-CMA security experiment is executed as follows.

- **Initialization**. C initializes system public parameters and keys.
- **Queries**. $\mathcal{A}$ queries C signatures on messages chosen at will.
- **Forgery**. $\mathcal{A}$ returns a forged signature σ^* on some new message m^*.

$\mathcal{A}$ wins the experiment if σ^* is a valid signature on the message m^* and a signature of m^* has not been queried in the query phase.

Definition 1 (EU-CMA Security) *A digital signature scheme is existentially unforgeable under chosen-message attacks if the advantage for any Probabilistic Polynomial Time (PPT) adversary $\mathcal{A}$ winning the EU-CMA security experiment is negligible.*

The EU-CMA security of ECDSA has been proved by Brown [16]. We will use it as an assumption in the security proof of our $(2, 3)$-threshold ECDSA.

2.2. Cryptographic toolbox

Below we review cryptographic tools used in our scheme.

2.2.1. Additively homomorphic encryption

Additively homomorphic encryption provides an operation that produces the encryption of the sum of two numbers, given only the encryptions of the numbers. Let $\mathbf{M}$ be the message space, $\mathbf{E}$ be the ciphertext space, $\mathbf{Z}$ be the set of integers, and $\mathbf{k}$ be a security parameter. An additively homomorphic encryption scheme consists of three algorithms $(\mathbf{KeyGen}, \mathbf{Enc}, \mathbf{Dec})$. For any pair of $(pk, sk) \leftarrow \mathbf{KeyGen}(\mathbf{1^k})$, messages $m_1, m_2, m \in \mathbf{M}$, and integer $z \in \mathbf{Z}$, there exists two group operations $\oplus : \mathbf{E} \times \mathbf{E} \to \mathbf{E}$ and $\odot : \mathbf{Z} \times \mathbf{E} \to \mathbf{E}$ such that:

- $m_1 + m_2 = \mathbf{Dec}_{sk}(\mathbf{Enc}_{pk}(m_1) \oplus \mathbf{Enc}_{pk}(m_2))$
- $z \cdot m = \mathbf{Dec}_{sk}(z \odot \mathbf{Enc}_{pk}(m))$

The additively homomorphic encryption is used in the MtA technology (reviewed in Section 2.2.4). Existing schemes such as Paillier [17] encryption, ElGamal encryption "in-the-exponent" [7], and linear homomorphic encryption schemes over a class group [18] are all compatible with the MtA protocol and the specific scheme of this paper. Our specific scheme will use the elliptic curve ElGamal encryption "in-the-exponent" [17] as an instantiation of the additive homomorphic encryption. Below we briefly review the scheme:

- $\mathbf{KeyGen}(\mathbf{1^k})$: generates a random private key $sk \in Z_q$ and computes the public key $pk = sk \cdot P$.
- $\mathbf{Enc}_{pk}(m)$ where $m \in Z_q$:
  1. Generate random $r \in Z_q$
  2. Compute $c_1 = r \cdot P$ and $c_2 = r \cdot pk + m \cdot P$
  3. Output $c = (c_1, c_2)$.

- $\mathbf{Dec}_{sk}(c)$ where $c = (c_1, c_2)$:
  1. Compute $M = c_2 - sk \cdot c_1$.
  2. Recover m from M.

Note that as with other homomorphic encryption schemes, the ElGamal encryption "in-the-exponent" also has its limitations. The decryption can only be achieved if m is relatively small, given that solving the discrete logarithm is hard. However, since our work focuses on key management of digital signatures, we simply use ElGamal encryption "in-the-exponent" as an instantiation of additively homomorphic encryption.

2.2.2. Threshold secret sharing

A (t, n)-threshold secret sharing scheme $(\mathbf{Dis}, \mathbf{Rec})$ distributes a secret a_0 into n shares in a way that a_0 can be reconstructed from any t shares, and no information about a_0 can be leaked from up to $t-1$ shares. We review Shamir's secret sharing scheme over prime field $(Z_q, +, \cdot)$ [19] as follows:

- $\mathbf{Dis}(a_0, t, n)$: take a secret $a_0 \in Z_q$ and a threshold setting (t, n) as inputs, the dealer $\mathcal{D}$:
  1. Generate $a_1, \cdots, a_{t-1} \in Z_q$ and construct a distribution polynomial

$$f(x) = a_0 + a_1 x + \cdots + a_{t-1} x^{t-1} \mod q \tag{1}$$

  2. For $i = 1, \cdots, n$, compute

$$y_i = f(i) \mod q \tag{2}$$

  and distribute y_i to participant $\mathcal{P}_i$ securely.

  Rigorously, $y_i = f(x_i)$ for some x_i. Here, we use $x_i = i$ for conciseness as many leading papers do.
- $\mathbf{Rec}(Q)$ where Q represents a set of at least t shares: without losing generality, let $Q = \{y_1, \cdots, y_t\}$, and this algorithm reconstructs a_0 as

$$a_0 = \sum_{i=1}^{t} \lambda_i y_i \mod q \tag{3}$$

  where

$$\lambda_i = \prod_{j=1, j\neq i}^{t} \frac{j}{j-i} \tag{4}$$

2.2.3. Share renewal

The share renewal protocol after the distribution of a secret updates its shares so that all previous shares are incompatible with the new ones. A representative approach is to update each share by adding a share of 0 to it. Here, we briefly review the basic share renewal protocol of Herzberg et al [20]. Suppose after the $\mathbf{Dis}(a_0, t, n)$, each participant $\mathcal{P}_i$ $(i = 1, \cdots, n)$ possesses a share y_i. They cooperatively update their shares without the participation of a dealer as follows:

1. For $i = 1, \cdots, n$, each $\mathcal{P}_i$
   (1) Generate $b_{i1}, \cdots, b_{i,t-1} \in Z_q$ and construct a distribution polynomial for 0:

$$g_i(x) = b_{i1} x + \cdots + b_{i,t-1} x^{t-1} \mod q \tag{5}$$

   (2) For $j = 1, \cdots, n$, compute

$$u_{ij} = g_i(j) \mod q \tag{6}$$

   and secretly send u_{ij} to $\mathcal{P}_j$ $(j \neq i)$.
2. For $i = 1, \cdots, n$, every $\mathcal{P}_i$ updates its share y_i into

$$y_i + \sum_{j=1}^{n} u_{ji} \mod q \tag{7}$$

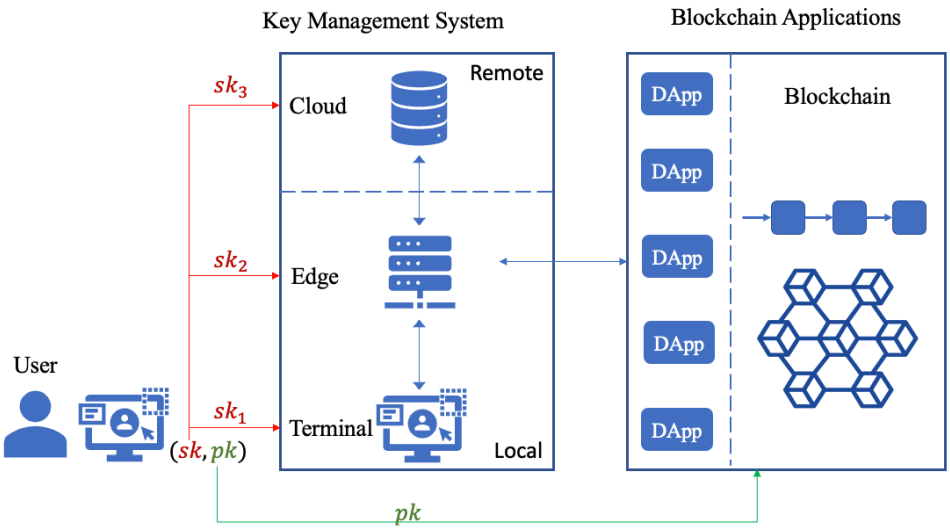

Figure 1. The CEE key management system and its interaction with Blockchain applications. The terminal for splitting sk and for holding sk_1 can be the same one. CEE: cloud-edge-end.

2.2.4. Multiplicative to additive share conversion

The MtA protocol[10,21] converts multiplicative shares into additive ones. It is used as a building block in common constructions of threshold ECDSA. Suppose party $\mathcal{P}_i$ possesses a and $\mathcal{P}_j$ possesses b, the MtA protocol outputs α for $\mathcal{P}_i$ and β for $\mathcal{P}_j$ such that $\alpha + \beta = a \cdot b$, without leaking a to $\mathcal{P}_j$ and b to $\mathcal{P}_i$. Below we briefly review the MtA protocol realized via additively homomorphic encryption:

1. $\mathcal{P}_i$ encrypts a under its public key pk_i:

$$C_a = \mathbf{Enc}_{pk_i}(a) \tag{8}$$

and sends C_a to $\mathcal{P}_j$.

2. $\mathcal{P}_j$ generates a random integer β, encrypts $-\beta$ under pk_i:

$$C'_\beta = \mathbf{Enc}_{pk_i}(-\beta) \tag{9}$$

computes the encryption of α under pk_i as

$$C_\alpha = (b \odot C_a) \oplus C'_\beta \tag{10}$$

and sends C_α to $\mathcal{P}_i$.

3. $\mathcal{P}_i$ decrypts and outputs α as

$$\alpha = \mathbf{Dec}_{sk_i}(C_\alpha) \tag{11}$$

3. THE CEE KEY MANAGEMENT FRAMEWORK

This section presents the framework of the CEE key management system and its threat and security models.

3.1. System overview

As shown in Figure 1, the CEE key management system involves three kinds of nodes: the cloud, edge, and end nodes. Their responsibilities in application scenarios are introduced as follows:

- **Cloud node** maintains a $(2, 3)$ share of user's private key. It is responsible for recovery or update of the private key in case of security incidents threatening one of the other two shares in the edge and end nodes.

In real-world applications, the cloud node can be a semi-trusted service provider that provides private key safeguarding services.

- **Edge node** also maintains a $(2, 3)$ share of user's private key. It participates in the signing protocol collaboratively with the end node. In the real world, the edge node can be the edge server or gateway of an institute such as a company, a university, etc. Another example is the smart home scenario where the edge node can be a router that connects end devices in the home to the internet. In these cases, the edge node can be regarded as a trusted user-side device.
- **End node** takes the original private key as input and distributes it into three shares. It also maintains a $(2, 3)$ share of the private key to issue signatures collaboratively with the edge node. It is the user-side terminal interacting with the user and, thereby, is trusted.

The number of each type of node depends on the needs and restrictions of real-world application environments. Each user may have one or more pairs of private and public keys, which again depends on the application scenario. For a pair of private and public keys (sk, pk), a cloud node, an edge node, and an end node will be involved to securely manage the private key. Specifically, there are three protocols for the key management framework:

- **Initial Splitting**. On input of a private key sk in the end node, this protocol uses $(2, 3)$-threshold secret sharing to split sk into three shares sk_1, sk_2, and sk_3 which will be securely stored in the end node, the edge node, and the cloud, respectively.
- **Two-Party Signing**. To issue a signature, the end node initiates the two-party signing protocol with the edge node. Specifically, each node will use its private key share to generate a partial signature, and the two partial signatures will be used to construct the signature of sk.
- **Update**. The update protocol can be run periodically or after a recovery from security incidents. Theoretically, it can be run with any two shares, even if the third one is leaked or tampered with. Here we focus on a situation in which the cloud is not compromised, and one of the user-side devices, i.e., the edge node or end node, is compromised. The cloud is normally more powerful in terms of computing and storing capabilities than the edge and end nodes. Moreover, it is maintained by service providers with professional security managers and technicians. Therefore, it is less likely to be compromised by security incidents in practice.

3.2. Threat model and security model

3.2.1. Threat model

We assume the cloud in the remote is semi-trusted, meaning it honestly follows the protocol specification but seeks to learn as much information as possible. We also assume that user-side devices, i.e., the edge and end nodes in the local area, are trusted. The cloud, edge, and end nodes may be compromised by security incidents, but they are not compromised simultaneously. Specifically, we assume at most one of them is compromised before an instance of the update protocol renews all shares. The threshold setting leaves the system sufficient time to identify and stop security incidents happening in one device before a second one is compromised.

Additionally, we assume the initial splitting protocol is run in a secure environment, where security channels exist among the three nodes, and none of them is compromised. This assumption is reasonable and easy to realize as this protocol is only executed once. During the other time, security incidents may occur and compromise one of the three nodes at a time.

3.2.2. Security model

Based on the threat model above and the EU-CMA security model of digital signature, we present the security model of $(2, 3)$-threshold signature under the CEE framework, denoted as $(2, 3)$-TEU-CMA. We first describe the $(2, 3)$-TEU-CMA security experiment and then give the security definition.

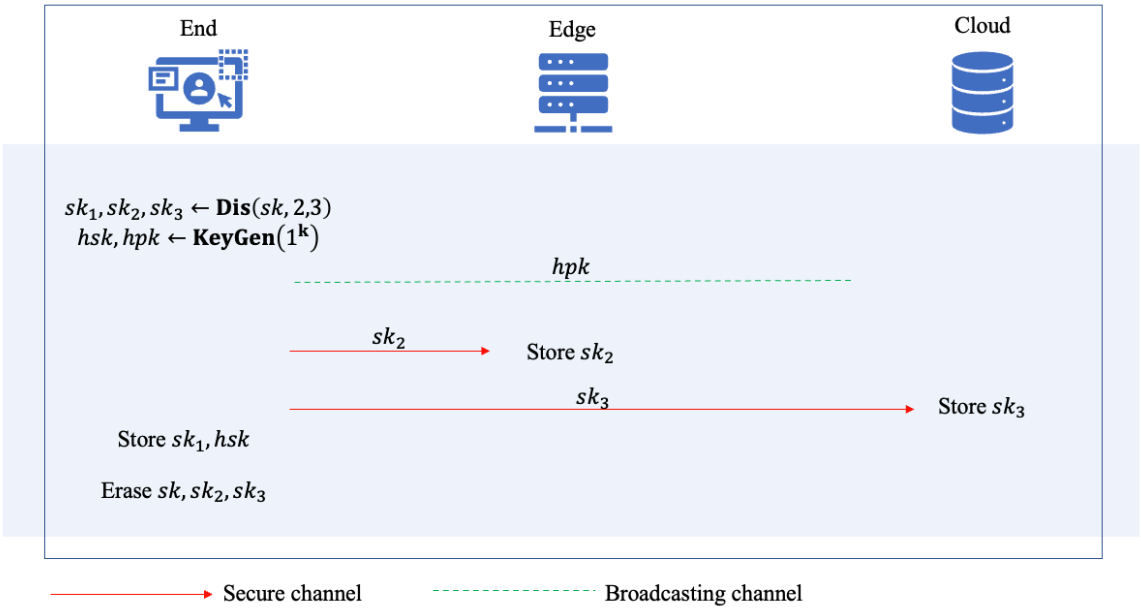

Figure 2. The initial splitting protocol. The secure channels can be established via public-key encryption or key agreement protocol.

Let $\mathcal{A}$ be an attacker against a $(2, 3)$-threshold signature scheme and C be the challenger who intends to solve a difficult problem. The $(2, 3)$-TEU-CMA security experiment is executed by $\mathcal{A}$ and C as follows:

- **Initialization**. C initializes system public parameters, and generates a pair of public and private keys (pk, sk) and three $(2, 3)$-threshold shares sk_1, sk_2, and sk_3 of sk. The private key and its shares are kept by C. The public key and public parameters are provided to $\mathcal{A}$.
- **Queries**.
  - Signature queries. $\mathcal{A}$ submits a message m chosen at will to C and queries the signature σ.
  - Corruption queries. $\mathcal{A}$ submits an index number i to C and queries sk_i and other secret parameters related to sk_i.
- **Forgery**. $\mathcal{A}$ returns a forged signature σ^* on some new message m^* that has never been submitted in the query phase.

The signature query is inherited from the EU-CMA security experiment. $\mathcal{A}$ can make the signature queries for many times. The corruption query is introduced to formulate the adversary's ability to corrupt a participant in the $(2, 3)$-threshold setting. $\mathcal{A}$ can make this type of query many times, but at most one such query for each batch of shares output by the initial splitting protocol or update protocol.

Definition 2 ((2,3)-TEU-CMA Security) *A $(2, 3)$-threshold digital signature scheme is existentially unforgeable under chosen-message attacks if the advantage for any PPT adversary $\mathcal{A}$ winning the $(2, 3)$-TEU-CMA security experiment is negligible.*

4. CEE-BASED $(2, 3)$-THRESHOLD ECDSA

Assume the user already has a pair of private and public keys (sk, pk) which are generated via the **KeyGen** algorithm of ECDSA. Therefore, $sk \in Z_q$ and $pk = sk \cdot P$. The public key is published to the distributed applications (DApps) in the blockchain ecosystem. Below we will show how to realize secure management for the private key via the CEE key management framework. Specifically, we will present a concrete $(2, 3)$-threshold ECDSA scheme that involves only the end node and the edge node for the two-party signing protocol.

4.1. Initial splitting

The initial splitting protocol is illustrated in Figure 2. Assume secure channels have been established among the end node, the edge node and the cloud. The user inputs sk in the end device and starts the initial splitting protocol which outputs three shares of sk and a pair of public and private keys for an additively homomorphic encryption scheme as follows:

1. Split sk into sk_1, sk_2, and sk_3 via the $\mathbf{Dis}(sk, 2, 3)$ algorithm of Shamir's threshold secret sharing.
2. Generate a pair of private and public keys (hsk, hpk) via the $\mathbf{KeyGen}(\mathbf{1^k})$ of an additively homomorphic encryption scheme and publish hpk
3. Securely send sk_2 to the edge node and sk_3 to the cloud.
3. Keep sk_1, hsk in the end node and erase sk, sk_2, sk_3.

In addition to the three private key shares, the above protocol also outputs and publishes a pair of keys (hsk, hpk) which will be used for implementing additively homomorphic encryption in the MtA procedure of the two-party signing protocol. Although it is possible to reuse sk_1 and $pk_1 = sk_1 \cdot P$ for the homomorphic encryption, here we have two reasons for generating dedicated key pairs. First, the additively homomorphic encryption scheme can be one established on a different mathematical problem rather than the discrete logarithm problem over elliptic curve groups of ECDSA, which means that (sk_1, pk_1) is not a pair of keys for the homomorphic encryption scheme. Secondly, publishing pk_1 in the system will reduce the security of the $(2, 3)$-threshold ECDSA scheme. We will explain this point in detail in Section 5.2 after proving the unforgeability of the scheme.

4.2. Two-party signing

The two-party signing protocol is run jointly by the end and edge nodes to output a signature $\sigma = (r_x, s)$ for a message m, where r_x is derived from the x coordinate of $R = r \cdot P$ for some randomly generated $r \in Z_q$, and $s = r^{-1} \cdot (e + sk \cdot r_x)$. In our scheme, R is established as $r_1^{-1} \cdot r_2^{-1} \cdot P$, where r_1 and r_2 are securely generated by the end and edge nodes. Therefore, the end and edge nodes can compute partial signatures as

$$s_i = r_i \cdot e + r_i \cdot sk_i \cdot r_x, \; i = 1, 2 \tag{12}$$

where r_1 and r_2 are $(2, 2)$-multiplicative shares of r^{-1}, and sk_1 and sk_2 are $(2, 3)$-Shamir shares of sk. However, we still cannot compute s from s_1 and s_2 as they are neither $(2, 2)$-multiplicative shares nor $(2, 3)$-Shamir shares of s. To construct s from s_1 and s_2, we need to convert s_1 and s_2 into additive shares of s. Specifically, we address the following three objectives in an offline pre-signing procedure before the input of m:

- Converting the multiplicative shares r_1 and r_2 of r^{-1} into additive shares α_1 and α_2, that is, outputting α_1 on the end node and α_2 on the edge node such that $\alpha_1 + \alpha_2 = r_1 \cdot r_2 = r^{-1}$. This is addressed via the MtA protocol.
- Converting the $(2, 3)$-threshold shares sk_1 and sk_2 of sk into additive shares $s\hat{k}_1$ and $s\hat{k}_2$, that is, outputting $s\hat{k}_1$ on the end node and $s\hat{k}_2$ on the edge node such that $s\hat{k}_1 + s\hat{k}_2 = sk$. This is addressed by multiplying sk_i with the Lagrange coefficient λ_i.
- Converting $\alpha_1 \cdot s\hat{k}_2$ into $\beta_1 + \beta_2$ and $\alpha_2 \cdot s\hat{k}_1$ into $\gamma_1 + \gamma_2$, where β_1 and γ_1 are output on the end node, and β_2 and γ_2 on the edge node. This is again realized via the MtA protocol.

By addressing the three objectives above, the end node will be able to compute $s_1 = \alpha_1 \cdot e + (\alpha_1 \cdot s\hat{k}_1 + \beta_1 + \gamma_1) \cdot r_x \mod q$, and the edge node can compute $s_2 = \alpha_2 \cdot e + (\alpha_2 \cdot s\hat{k}_2 + \beta_2 + \gamma_2) \cdot r_x \mod q$. Now, s_1 and s_2 are additive

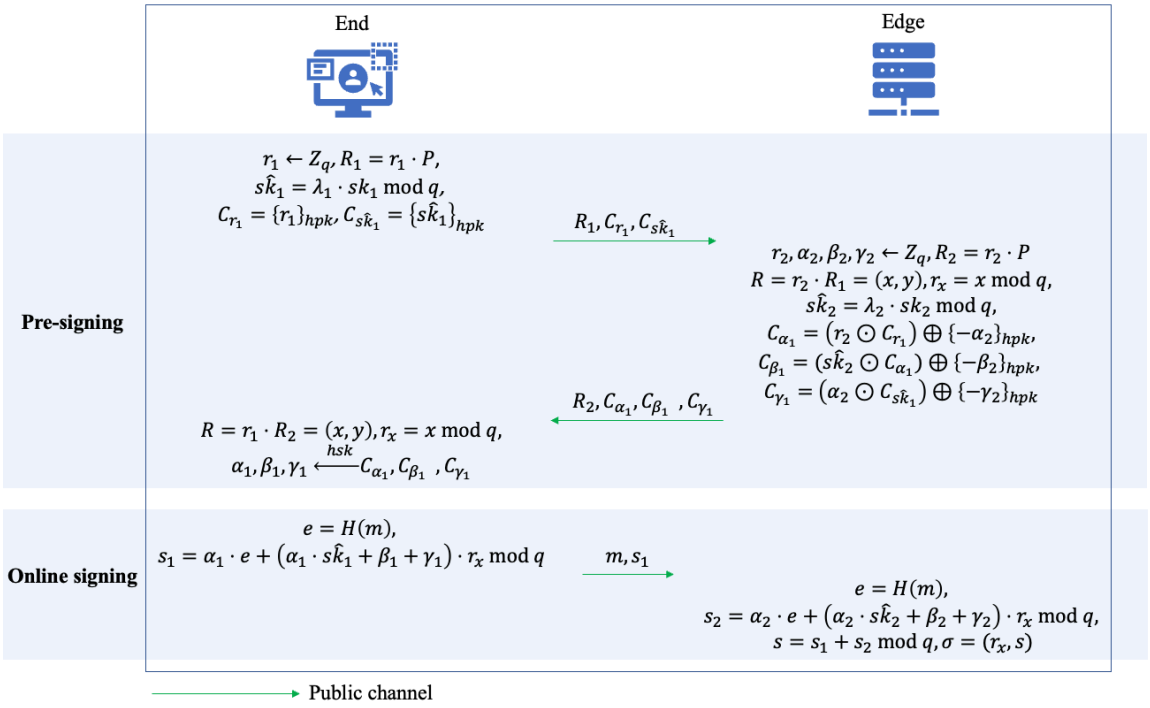

Figure 3. The two-party signing protocol.

shares of s, since

$$\begin{aligned} s_1 + s_2 &= \alpha_1 \cdot e + (\alpha_1 \cdot s\hat{k}_1 + \beta_1 + \gamma_1) \cdot r_x + \alpha_2 \cdot e + (\alpha_2 \cdot s\hat{k}_2 + \beta_2 + \gamma_2) \cdot r_x \\ &= (\alpha_1 + \alpha_2) \cdot e + (\alpha_1 \cdot s\hat{k}_1 + \beta_1 + \gamma_1 + \alpha_2 \cdot s\hat{k}_2 + \beta_2 + \gamma_2) \cdot r_x \\ &= r^{-1} \cdot e + (\alpha_1 \cdot s\hat{k}_1 + \alpha_2 \cdot s\hat{k}_2 + \alpha_1 \cdot s\hat{k}_2 + \alpha_2 \cdot s\hat{k}_1) \cdot r_x \\ &= r^{-1} \cdot e + (\alpha_1 + \alpha_2) \cdot (s\hat{k}_1 + s\hat{k}_2) \cdot r_x \\ &= r^{-1} \cdot e + r^{-1} \cdot sk \cdot r_x \\ &= s \end{aligned} \tag{13}$$

The above method requires running a key agreement instance to output R and three MtA instances to output (α_1, α_2), (β_1, β_2) and (γ_1, γ_2) among the end and edge nodes. This involves eight messages. To reduce the round of interactions, we integrate them together in the pre-signing protocol and reduce the total number of messages to three.

The two-party singing protocol is illustrated in Figure 3. It consists of an offline pre-signing protocol and an online singing protocol. Below we introduce the two protocols in detail.

4.2.1. Pre-signing protocol

1. The end node:
   (1) Generate a random value $r_1 \leftarrow Z_q$ and compute

$$R_1 = r_1 \cdot P \tag{14}$$

(2) Convert sk_1 into

$$\begin{aligned} s\hat{k}_1 &= \lambda_1 \cdot sk_1 \mod q \\ &= \frac{2}{2-1} \cdot \frac{3}{3-1} \cdot sk_1 \mod q \\ &= 3 \cdot sk_1 \mod q \end{aligned} \tag{15}$$

(3) Encrypt r_1 and $s\hat{k}_1$ under hpk

$$C_{r_1} = \mathbf{Enc}_{hpk}(r_1) \tag{16}$$

$$C_{s\hat{k}_1} = \mathbf{Enc}_{hpk}(s\hat{k}_1) \tag{17}$$

(4) Send $(R_1, C_{r_1}, C_{s\hat{k}_1})$ to the edge node via public channel.

2. The edge node:

(1) Generate random numbers $r_2, \alpha_2, \beta_2, \gamma_2 \leftarrow Z_q$ and compute

$$R_2 = r_2 \cdot P \tag{18}$$

(2) Convert sk_2 into

$$\begin{aligned} s\hat{k}_2 &= \lambda_2 \cdot sk_2 \mod q \\ &= \frac{1}{1-2} \cdot \frac{3}{3-2} \cdot sk_2 \mod q \\ &= -3 \cdot sk_2 \mod q \end{aligned} \tag{19}$$

(3) Compute the shared instance as follows:

$$R = r_2 \cdot R_1 = (x, y) \tag{20}$$

$$r_x = x \mod q \tag{21}$$

(4) Compute the ciphertext of $\alpha_1 = r_1 \cdot r_2 - \alpha_2$ under hpk as

$$C_{\alpha_2'} = \mathbf{Enc}_{hpk}(-\alpha_2) \tag{22}$$

$$C_{\alpha_1} = (r_2 \odot C_{r_1}) \oplus C_{\alpha_2'} \tag{23}$$

(5) Compute the ciphertext of $\beta_1 = \alpha_1 \cdot s\hat{k}_2 - \beta_2$ under hpk as

$$C_{\beta_2'} = \mathbf{Enc}_{hpk}(-\beta_2) \tag{24}$$

$$C_{\beta_1} = (s\hat{k}_2 \odot C_{\alpha_1}) \oplus C_{\beta_2'} \tag{25}$$

(6) Compute the ciphertext of $\gamma_1 = \alpha_2 \cdot s\hat{k}_1 - \gamma_2$ under hpk as

$$C_{\gamma_2'} = \mathbf{Enc}_{hpk}(-\gamma_2) \tag{26}$$

$$C_{\gamma_1} = (\alpha_2 \odot C_{s\hat{k}_1}) \oplus C_{\gamma_2'} \tag{27}$$

(7) Send $(R_2, C_{\alpha_1}, C_{\beta_1}, C_{\gamma_1})$ to the end node via public channel.

3. The end node:

(1) Compute the shared instance as follows:

$$R = r_1 \cdot R_2 = (x, y) \tag{28}$$

$$r_x = x \mod q \tag{29}$$

(2) Decrypt the ciphertexts $C_{\alpha_1}, C_{\beta_1}, C_{\gamma_1}$ using hsk:

$$\alpha_1 = \mathbf{Dec}_{hsk}(C_{\alpha_1}) \tag{30}$$

$$\beta_1 = \mathbf{Dec}_{hsk}(C_{\beta_1}) \tag{31}$$

$$\gamma_1 = \mathbf{Dec}_{hsk}(C_{\gamma_1}) \tag{32}$$

4.2.2. Online signing protocol

1. On input a message $m \in \{0,1\}^*$, the end node computes

$$e = H(m) \tag{33}$$

$$s_1 = \alpha_1 \cdot e + (\alpha_1 \cdot s\hat{k}_1 + \beta_1 + \gamma_1) \cdot r_x \tag{34}$$

and sends s_1, m to the edge node via the public channel.

2. The edge node computes

$$e = H(m) \tag{35}$$

$$s_2 = \alpha_2 \cdot e + (\alpha_2 \cdot s\hat{k}_2 + \beta_2 + \gamma_2) \cdot r_x \tag{36}$$

$$s = s_1 + s_2 \tag{37}$$

and outputs the signature as $\sigma = (r_x, s)$.

In the above online signing protocol, the signature is output by the edge node. The reason is that in the real-world environment, especially the IoT scenario, usually the edge node is networking equipment such as a gateway that connects the local end nodes to remote servers by forwarding traffic between them. Even if the signature is output by the end node, the end node has to send it through the edge node to the intended application server.

4.3. Update

The update protocol is initiated by the user via the end device, either periodically or after recovery from security incidents. We consider the following two cases. If a share is leaked but not tampered with, then the three parties can run the share renewal protocol in Section 2.2.3 to update all three shares. If a share sk_k ($k \in \{1,2\}$ according to our system model) is tampered with, then before the share renewal protocol, the update protocol needs to recover it via the other two shares sk_i and sk_j as follows:

$$sk_k = f(k) = \frac{k-j}{i-j} \cdot sk_i + \frac{k-i}{j-i} \cdot sk_j \tag{38}$$

where $i, j, k \in \{1,2,3\}$ and $i \neq j \neq k$ for a $(2,3)$ setting.

Theoretically, the update protocol can securely update all three shares as long as at least two of them are not tampered with and at most one of them is leaked. In our scenario, we assume the share sk_3 in the cloud is secure. In the recover procedure, to guarantee that only the party $\mathcal{P}_k$ can compute sk_k without knowing sk_i and sk_j, we use secure multi-party computation as follows:

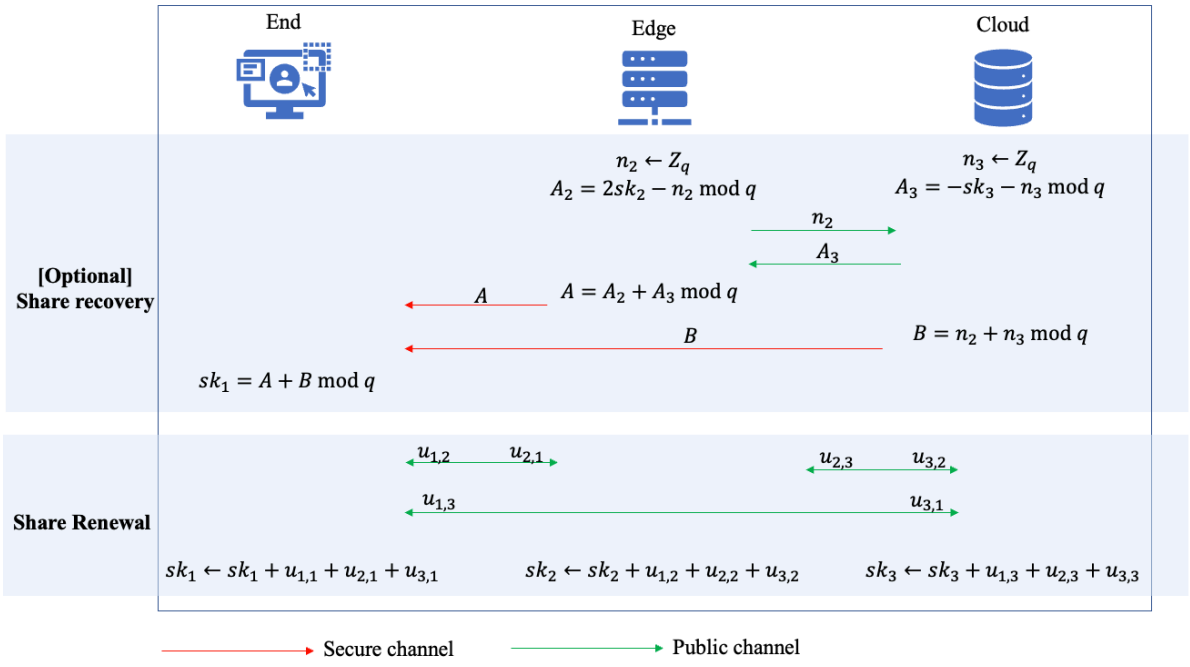

Figure 4. The update protocol, assuming sk_1 in the end node is tampered with.

- [Optional] Share recovery:
  1. $\mathcal{P}_i$ generates a random n_i, computes

$$A_i = \frac{k-j}{i-j} \cdot sk_i - n_i \mod q \tag{39}$$

  and sends n_i to $\mathcal{P}_j$ via public channel.
  2. $\mathcal{P}_j$ generates a random n_j, computes

$$A_j = \frac{k-i}{j-i} \cdot sk_j - n_j \mod q \tag{40}$$

  and sends A_j to $\mathcal{P}_j$ via public channel.
  3. $\mathcal{P}_i$ computes

$$A = A_i + A_j \mod q \tag{41}$$

  and send A to $\mathcal{P}_k$ via the secure channel.
  4. $\mathcal{P}_j$ computes

$$B = n_i + n_j \mod q \tag{42}$$

  and sends B to $\mathcal{P}_k$ via secure channel.
  4. $\mathcal{P}_k$ computes

$$sk_k = A + B \mod q \tag{43}$$

- **Share renewal**: The three participants run the share renewal protocol in Section 2.2.3 to refresh their private key shares.

The update protocol is illustrated in Figure 4, which assumes that sk_1 in the end node is tampered with.

5. SECURITY

5.1. Unforgeability

We first claim the unforgeability of our $(2, 3)$-threshold ECDSA in the following theorem and then prove it.

Theorem 1 *The $(2, 3)$-threshold ECDSA in Section 4 is $(\tau, q_s, q_c, \epsilon)$-secure in the $(2, 3)$-TEU-CMA model, given that the standard ECDSA signature scheme is EU-CMA secure. Specifically, the advantage of any PPT adversary $\mathcal{A}$ winning the $(2, 3)$-TEU-CMA experiment is*

$$Adv^{(2,3)-TEU-CMA}_{(2,3)-thresholdECDSA}(\mathcal{A}) \leq Adv^{EU-CMA}_{ECDSA}(\mathcal{B})$$

where $Adv^{EU-CMA}_{ECDSA}(\mathcal{B})$ is the advantage of $\mathcal{B}$ winning the EU-CMA game.

Let $\mathcal{A}$ be an adversary against the $(2, 3)$-threshold ECDSA, and $\mathcal{B}$ be the challenger running the $(2, 3)$-TEU-CMA experiments with $\mathcal{A}$. Meanwhile, $\mathcal{B}$, as an adversary, runs the EU-CMA experiment with an ECDSA challenger $\mathcal{C}$. Therefore, $\mathcal{B}$ works as a connector between the $(2, 3)$-TEU-CMA experiment and the EU-CMA experiment. Below we illustrate how to embed the EU-CMA experiment into the $(2, 3)$-TEU-CMA experiment and reduce the $(2, 3)$-TEU-CMA security of the $(2, 3)$-threshold ECDSA to the EU-CMA security of ECDSA.

In the $(2, 3)$-TEU-CMA experiment, $\mathcal{B}$ initializes system public parameters and keys and interacts with $\mathcal{A}$. Meanwhile, in the EU-CMA experiment, $\mathcal{B}$ interacts with $\mathcal{C}$ who initializes the system parameters and keys of ECDSA. Specifically,

- **Initialization**.
  - Initialization of ECDSA. $\mathcal{C}$ initializes the system public parameters $\mathbf{pp} = \{G, P, q, Z_q\}$ and a pair of public and private keys (pk, sk) of ECDSA. The system public parameters $\mathbf{pp}$ and public key pk are given to $\mathcal{B}$.
  - Initialization of $(2, 3)$-threshold ECDSA. $\mathcal{B}$ initializes the three shares of sk and keys of additively homomorphic encryption scheme as follows:
    1. Generate random $sk_1, sk_2, sk_3 \in Z_q$.
    2. Generate (hsk, hpk) as specified in the initial splitting protocol.
    3. Provide system public parameters and public keys (pk, hpk) to $\mathcal{A}$.
- **Queries**.
  - Signature queries. $\mathcal{A}$ submits a message m chosen at will to $\mathcal{B}$ and queries the signature. $\mathcal{B}$ queries $\mathcal{C}$ the signature on m. $\mathcal{C}$ runs the **Signing** algorithm of ECDSA and returns the signature σ to $\mathcal{B}$. Finally, $\mathcal{B}$ forwards σ to $\mathcal{A}$.
  - Corruption queries. $\mathcal{A}$ submits an index $i \in \{1, 2, 3\}$ to $\mathcal{B}$ to make a corruption query. $\mathcal{B}$ replies with (sk_1, hsk) if $i = 1$ and sk_i otherwise. $\mathcal{B}$ also replies s_i to $\mathcal{A}$ if $i \neq 3$.
- **Forgery**. $\mathcal{A}$ returns a forged signature σ^* on some new message m^* that has never been queried in the query phase. $\mathcal{B}$ returns m^* and σ^* to $\mathcal{C}$.

After q_s signature queries and q_c corruption queries, if $\mathcal{A}$ wins the $(2, 3)$-TEU-CMA experiment, then σ^* is a valid signature on m^* and $\mathcal{B}$ wins the TEU-CMA security experiment. Therefore, the advantage of $\mathcal{A}$ winning the $(2, 3)$-TEU-CMA experiment no more than the advantage of $\mathcal{B}$ winning the EU-CMA experiment, that is,

$$Adv^{(2,3)-TEU-CMA}_{(2,3)-thresholdECDSA}(\mathcal{A}) \leq Adv^{EU-CMA}_{ECDSA}(\mathcal{B}).$$

Since ECDSA is EU-CMA secure, $Adv^{EU-CMA}_{ECDSA}(\mathcal{B})$ is negligible. Therefore, $Adv^{(2,3)-TEU-CMA}_{(2,3)-thresholdECDSA}(\mathcal{A})$ is negligible. According to Definition 2, the $(2, 3)$-threshold ECDSA scheme is secure under the $(2, 3)$-TEU-CMA model.

Table 2. Evaluation of communication cost

Protocol	Messages over spc	Messages over ppc	Messages over bc
Phase I: Initial splitting	2	0	1
Phase II-1: Offline pre-signing	0	2	0
Phase II-2: Online signing	0	1	0
Phase III-1: [Optional] Share recovery	2	2	0
Phase III-2: Share renewal	0	6	0

5.2. Discussion

The initial splitting protocol in Section 4.1 generates dedicated private and public keys for the additively homomorphic encryption, rather than reusing (sk_1, pk_1). Now we explain why reusing (sk_1, pk_1) can negatively influence the security of the scheme.

If (sk_1, pk_1) is used for the additively homomorphic encryption scheme, then the adversary $\mathcal{A}$ will be given pk_1 in addition to pk in the $(2, 3)$-TEU-CMA security experiment. With pk_1 and pk, $\mathcal{A}$ can derive the other two public keys $pk_2 = sk_2 \cdot P$ and $pk_3 = sk_3 \cdot P$ as follows:

$$pk_j = \frac{1}{\lambda_j} \cdot (pk - \lambda_1 \cdot pk_1), \text{ for } j = 2, 3 \tag{44}$$

as

$$sk_j = \frac{1}{\lambda_j} \cdot (sk - \lambda_1 \cdot sk_1) \mod q, \text{ for } j = 2, 3 \tag{45}$$

Note that $\mathcal{B}$ does not have sk and the discrete logarithms of pk_2 and pk_3. By submitting an index 2 or 3 in the corruption query, $\mathcal{A}$ can distinguish the experiment from a real execution of the $(2, 3)$-threshold ECDSA, which will result in failure of the $(2, 3)$-TEU-CMA security experiment.

6. EVALUATION

We first evaluate the communication cost of the $(2, 3)$-threshold ECDSA as it is the major concern when deploying the scheme for DApps. Then we estimate the computing cost to provide a reference for real-world deployment.

6.1. Communication cost

The communication cost is estimated by the number of messages transmitted among secure and public peer-to-peer channels and broadcasting channels in each protocol. Let **spc** denote secure peer-to-peer channel, **ppc** denote public peer-to-peer channel, and **bc** denote broadcasting channel. The estimated communication cost is listed in Table 2.

The initial splitting protocol involves two messages over secure channels from the end node to the edge and the cloud and one broadcasting channel to publish the homomorphic encryption key. Notably, the two-party signing protocol uses merely two messages in an offline phase and one message in the online phase, and all are over public channels among the end and edge nodes in the local area. This brings very light communication costs for the application scenario. The update protocol involves two messages for an optional share recovery phase and six for the share renewable phase, and all of them are over public peer-to-peer channels.

To study the reduction in communication complexity achieved by the proposed $(2, 3)$-threshold ECDSA

Table 3. Comparison of communication complexity of threshold signing in $(2, 3)$-threshold setting

Scheme	Offline Pre-signing	Online Signing
Gennaro & Goldfeder 18 [8]	-	$8t$ bc + $4t(t-1)$ spc
Wong *et al.* [11]	10 bc + 2 spc	2 bc
Gagol *et al.* [22]	12 bc + 2 spc	6 bc
Gennaro and Goldfeder 20 [23]	-	12 bc + 4 spc
Castagnos *et al.* [12]	-	14 bc + 2 spc
Our (2,3)-threshold ECDSA	2 ppc	ppc

Table 4. Evaluation of computing cost

Protocol	End node	Edge node	Cloud node
Phase I: Initial splitting	1 **exp**	0	0
Phase II-1: Pre-signing	11 **exp**	11 **exp**	0
Phase II-2: Online signing	0	0	0
Phase III-1: (Optional) Share recovery	0	0	0
Phase III-2: Share renewal	0	0	0

Table 5. Experimental environment

Device/Parameter	Specification
End node	Smartphone with 12 GB RAM, Octa-core Max 3.1 GHz CPU, Android 14 OS
Edge node	Laptop with 16 GB RAM, Intel i7-1260P 2.10 GHz CPU, Windows 11 OS
ECC Curve	FIPS approved standard curve P-256

scheme, the number of messages involved in the threshold issuance of an ECDSA signature was compared with those of representative schemes, as shown in Table 3. The results indicate that this scheme significantly reduces communication complexity.

6.2. Computing cost

The computing cost is evaluated via the number of exponentiation (i.e., scalar multiplication over elliptic curve group G), and the key generation, encryption and decryption algorithms of additively homomorphic encryption scheme. The additively homomorphic encryption scheme is instantiated as the elliptic curve ElGamal encryption "in-the-exponent". The estimated computing cost is summarized in Table 4, where **exp** denotes exponentiation.

In Table 4, the estimated computing cost for most cells is 0. This does not mean the corresponding protocols bring no computing task to the node. There are still some lightweight operations such as integer addition, integer multiplication, random number generation, etc. Their costs are ignored compared to time-consuming operations such as exponentiation, encryption, etc.

Notably, in Table 4, most computing costs are from the pre-signing phase. In practice, this phase can be executed offline before the application's message is generated. Therefore, the scheme will not influence user experience of DApps given that the online signing phase introduces very low computing costs.

To further estimate the computing cost for real-world applications, we test the running time of exponentiation in common end and edge devices (a smartphone and a laptop). Details of the hardware and software environment are explained in Table 5. The average time for computing one exponentiation is 0.0273 seconds on the smartphone and 0.0116 seconds on the laptop. Based on the results, we evaluate the running time for initial splitting as 0.0273 seconds on the end node, and for pre-signing as 0.2998 seconds on the end node and 0.1280 seconds on the edge node. The results are acceptable for real-world applications.

7. APPLICATION

The CEE key management demonstrates the integration of $(2, 3)$-threshold signature and the CEE computing paradigm. It has wide application prospects for the DApp market and beyond. Below we illustrate the application of CEE key management in a promising blockchain-empowered sector, DePIN.

7.1. Background and key management challenges of DePIN

DePIN refers to decentralized physical infrastructure networks utilizing blockchain technology to return the ownership and commercial rights of data back to the users, allowing everyone to benefit from their own digital footprint. As a promising sector, DePIN has been attracting growing interest and investment recently. The leading credit ratings agency, Moody's Ratings, has highlighted the potential of DePIN to transform physical infrastructure networks[24]. Individuals holding physical devices (e.g., mobile phones, personal computers, smart cars, etc.) can participate in DePIN and get cryptocurrency rewards by providing data or services via their devices, such as providing real-time noise pollution information through a smartphone, supporting navigation services by uploading real-time traffic information through an intelligent vehicle while driving, etc. In these cases, the ownership of data is enforced by signatures issued by the private key of the device. The rewards are paid to an address derived from the corresponding public key.

Considering the scenario of real-time noise pollution monitoring for a residential community through DePIN. To monitor noise pollution in the community, the related department can collect noise data via DePIN instead of deploying physical equipment itself. An individual user can participate in the DePIN network by installing some DApps for noise pollution monitoring and registering the public key to DePIN. Then, he or she can use the sound recorder of his or her smartphone to collect noise data and submit them to the data requester (e.g., server of the noise pollution monitoring DApps). The noise data are signed by the private key of the user or the smartphone, so that a reward can be paid to the cryptocurrency address derived from the corresponding public key.

However, managing the private keys of physical devices is challenging for common users in DePIN. Existing solutions store and use the private key in the physical device. If the physical device is lost, broken, or attacked, the private key will be leaked or unavailable anymore, leading to two serious consequences:

- the reward under the cryptocurrency address will be lost, and
- a new pair of private and public keys need to be generated, and the user has to register the new public key to DePIN.

7.2. Application of CEE key management in DePIN

Now we show how to use the proposed CEE key management to address the key management problem for DePIN users.

As shown in Figure 5, the CEE key management service can be provided by a professional security company, a smartphone manufacturer, etc. To use this service, the DePIN users install client software on their smartphones. By running the software of the CEE key management system in the smartphone, the private key is distributed into three shares kept in the smartphone, the router in the home, and the cloud server of the service provider. When uploading noise pollution data, the smartphone and router run the two-party signing protocol to issue a signature and attach it to the data. The key management cloud is only involved in the key distribution and update phase.

With the CEE key management service, when the physical device (e.g., the smartphone) is lost, broken, or attacked, the private key share stored in it is leaked or unavailable anymore. However, the private key remains secure and available as long as the other two shares in the router and cloud server are secure; therefore, the

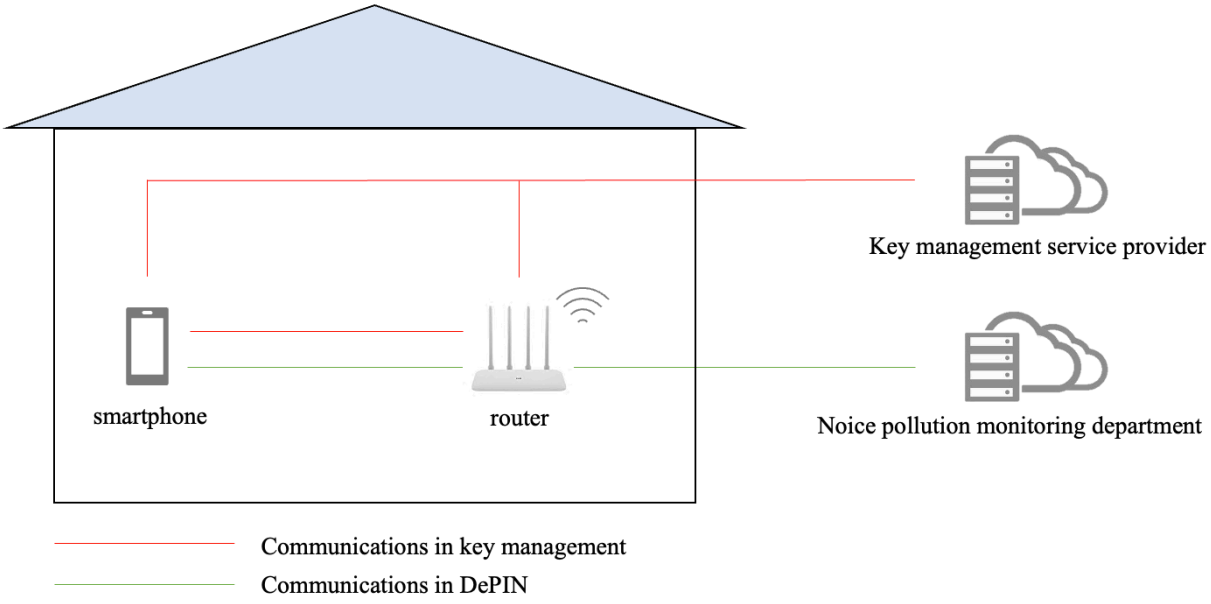

Figure 5. CEE key management for DePIN. CEE: cloud-edge-end.

first consequence in Section 7.1 will not happen. Further, by running the update protocol, the three shares are refreshed while the public key remains unchanged, which means the second consequence will not occur.

8. CONCLUSION

The CEE paradigm has been widely adopted in many online computing products and services, such as smart homes, intelligent transportation, e-health, etc. Combining $(2, 3)$-threshold key management with CEE provides a very convenient way for implementing key management for online products and services, where the end and edge nodes near the end users are responsible for daily uses of the private key, and the cloud serves in updating the storage of private key shares. The CEE key management framework is anticipated to have wide application in various digital signature and public-key encryption schemes. The proposed $(2, 3)$-threshold ECDSA demonstrates the application of the CEE key management framework in the widely used ECDSA signature. It also advances the theory of threshold ECDSA by further reducing the communication cost compared to the best practice of parallel schemes.

DECLARATIONS

Authors' contributions

Made substantial contributions to conception and design of the study: Zhang J, Zhang F
Performed security proof and evaluation: Zhang J
Provided administrative and technical support: Zhang F

Availability of data and materials

The supporting data for this article is available upon request directly from the authors.

Financial support and sponsorship

This work was partially supported by the National Natural Science Foundation of China under Grant No. 62172096; the XJTLU Research Development Fund under Grant No. RDF-21-02-014; the XJTLU Teaching Development Fund under Grant No. TDF2223-R25-207; and the Suzhou Municipal Key Laboratory for Intel-

ligent Virtual Engineering (SZS2022004).

Conflicts of interest
All authors declared that there are no conflicts of interest.

Ethical approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Copyright

REFERENCES
1. Yue K, Zhang Y, Chen Y, et al. A survey of decentralizing applications via blockchain: the 5G and beyond perspective. *IEEE Commun Surv Tutorials* 2021;23:2191–217. DOI
2. Zheng P, Jiang Z, Wu J, Zheng Z. Blockchain-based decentralized application: A survey. *IEEE Open J Comput Soc* 2023;4:121–33. DOI
3. Nakamoto S. Bitcoin: a peer-to-peer electronic cash system. *Satoshi Nakamoto* 2008. Available from: https://bitcoin.org/bitcoin.pdf. [Last accessed on 12 Dec 2024]
4. Houy S, Schmid P, Bartel A. Security aspects of cryptocurrency wallets—a systematic literature review. *ACM Comput Surv* 2023;56:1–31. DOI
5. Gennaro R, Goldfeder S, Narayanan A. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In: Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings 14. Springer; 2016. pp. 156–74. DOI
6. Gennaro R, Jarecki S, Krawczyk H, Rabin T. Robust threshold DSS signatures. In: Advances in Cryptology—EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings 15. Springer; 1996. pp. 354–71. DOI
7. Lindell Y, Nof A. 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; 2018. pp. 1837–54. DOI
8. Gennaro R, Goldfeder S. Fast multiparty threshold ECDSA with fast trustless setup. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security; 2018. pp. 1179–94. DOI
9. Zhang H, Xie G, Zou X, et al. Asynchronous threshold ECDSA with batch processing. *IEEE Trans Comput Soc Syst* 2023;11:566–75. DOI
10. Xue H, Au MH, Liu M, et al. Efficient multiplicative-to-additive function from Joye-Libert cryptosystem and its application to threshold ECDSA. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security; 2023. pp. 2974–88. DOI
11. Wong HW, Ma JP, Yin HH, Chow SS. Real threshold ECDSA. In: NDSS; 2023. DOI
12. Castagnos G, Catalano D, Laguillaumie F, Savasta F, Tucker I. Bandwidth-efficient threshold EC-DSA. In: IACR International Conference on Public-Key Cryptography. Springer; 2020. pp. 266–96. DOI
13. Lindell Y. Fast secure two-party ECDSA signing. In: Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part II 37. Springer; 2017. pp. 613–44. DOI
14. Doerner J, Kondi Y, Lee E, Shelat A. Secure two-party threshold ECDSA from ECDSA assumptions. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE; 2018. pp. 980–97. DOI
15. Tu B, Chen Y, Cui H, Wang X. Fast two-party signature for upgrading ECDSA to two-party scenario easily. *Theor Comput Sci* 2024;986:114325. DOI
16. Brown DR. Generic groups, collision resistance, and ECDSA. *Des Codes Cryptogr* 2005;35:119–52. DOI
17. Paillier P. Public-key cryptosystems based on composite degree residuosity classes. In: International conference on the theory and applications of cryptographic techniques. Springer; 1999. pp. 223–38. DOI
18. Castagnos G, Laguillaumie F, Tucker I. Practical fully secure unrestricted inner product functional encryption modulo p. In: International Conference on the Theory and Application of Cryptology and Information Security. Springer; 2018. pp. 733–64. DOI
19. Shamir A. How to share a secret. *Commun ACM* 1979;22:612–13. DOI
20. Herzberg A, Jarecki S, Krawczyk H, Yung M. Proactive secret sharing or: How to cope with perpetual leakage. In: Advances in Cryptology—CRYPT0'95: 15th Annual International Cryptology Conference Santa Barbara, California, USA, August 27–31, 1995 Proceedings 15. Springer; 1995. pp. 339–52. DOI
21. Aumasson JP, Hamelink A, Shlomovits O. A survey of ECDSA threshold signing. *Cryptology ePrint Arch* 2020. Available from: https://eprint.iacr.org/2020/1390. [Last accessed on 12 Dec 2024]

22. Gągol A, Kula J, Straszak D, Świętek M. Threshold ECDSA for decentralized asset custody. *Cryptology ePrint Arch* 2020. Available from: https://ia.cr/2020/498. [Last accessed on 12 Dec 2024]
23. Gennaro R, Goldfeder S. One round threshold ECDSA with identifiable abort. *Cryptology ePrint Arch* 2020. Available from: https://ia.cr/2020/540. [Last accessed on 12 Dec 2024]
24. Ratings M. How DePINs could build the future of physical infrastructure one token at a time. 2024. Available from: https://www.moodys.com. [Last accessed on 12 Dec 2024]