SkyEye: A Traceable Scheme for Blockchain

Tianjun Ma
Haixia Xu*
matianjun@iie.ac.cn
xuhaixia@iie.ac.cn
State Key Laboratory of Information Security, Institute of
Information Engineering, CAS, Beijing, China
School of Cyber Security, University of Chinese Academy
of Sciences, Beijing, China
Data Assurance and Communication Security Research
Center, Chinese Academy of Sciences, Beijing, China

Peili Li
lipeliliie.ac.cn
State Key Laboratory of Information Security, Institute of
Information Engineering, CAS, Beijing, China
Data Assurance and Communication Security Research
Center, Chinese Academy of Sciences, Beijing, China

ABSTRACT
Many studies focus on the blockchain privacy protection. Unfortunately, the privacy protection brings some issues (e., money-laundering problem). Tracing users’ identities is a critical step in addressing these issues. When each user’s identity in the blockchain data is determined, the regulator can do some regulatory operations (such as Big Data analysis) to decide who should be punished or who should own the lost data. In this paper, we propose SkyEye, a traceable scheme for blockchain, that can be applied to a class of blockchain application. SkyEye enables the regulator to trace users’ identities. Moreover, we demonstrate the security of SkyEye under specific cryptographic assumptions. Finally, we implement two prototypes of SkyEye, and evaluate the running time and related data storage requirements by performing the aforementioned prototypes.

KEYWORDS
traceable scheme, blockchain, chameleon hash, zero knowledge

1 INTRODUCTION
The blockchain was first introduced in Bitcoin[35], and quickly became the supporting technology of decentralized cryptocurrencies such as PPcoin[25] and Litecoin[1]. The blockchain integrates multiple technologies (e.g., cryptography and peer-to-peer networking) and includes a variety of features: distributed, decentralized, anonymity, transparency, and so on. Today, the blockchain is not only applied in decentralized cryptocurrencies, but also has broad applications in other fields, including defense, finance, and smart contract.

The blockchain can be considered a distributed database that only appends data (e., transactions). The data is stored in the block that contains the block header and block body. Every block header includes the hash of the previous block, forming a chain. The strategy of appending a block to the blockchain uses a consensus mechanism such as proof of work (PoW)[35], proof of stake (PoS) [5, 14, 24], or practical byzantine fault tolerance (PBFT)[10]. In many blockchain applications, every user generally has public/private information (e., the public key address and the signature private key for each user in Bitcoin[35], more details about the public/private information are described in Section 2.3).

Many studies focus on the blockchain privacy protection [7, 38]. Unfortunately, the privacy protection brings regulatory issues. On one hand, if a user’s private information is lost or stolen, the user loses control of the data corresponding to the private information forever. For example, if the Bitcoin user loses the signature private keys in his wallet, there is no way to recover the coins in this wallet. In other words, the user loses the coins controlled by these signature private keys forever. On the other hand, strong privacy protection in the blockchain facilitates many criminal activities (e.g., ransomware[2], money laundering). CipherTrace’s second quarter 2019 cryptocurrency anti-money laundering report shows that the total amount of funds that cybercriminals directly steal, scam, and misappropriate from users and trading platforms is approximately $4.4 billion in aggregate for 2019. These regulatory issues not only present a serious threat to the interests of users, but also have seriously hindered the development and application of the blockchain.

We stress that tracing users’ identities is a critical step in addressing blockchain regulatory issues. When each user’s identity in the blockchain data is determined, the regulator can conduct some regulatory operations (such as Big Data analysis) to decide who should be punished or who should own the lost data. Although there has been progress in designing traceable mechanisms, such as zkLedger[36] and several others[4, 15, 20, 23], these approaches are designed for specific application environment and do not seem to have been extended to other applications. More details about these approaches are described in Section 9.

Our contributions. In this paper, we present a traceable scheme that can be applied to a class of blockchain application. The main contributions of this paper are as follows.

First, we introduce the notion of a traceable scheme for blockchain and formalize the security properties to be satisfied, namely identity proof indistinguishability and identity proof unforgeability.

Second, we propose SkyEye, a traceable scheme for blockchain. SkyEye can be applied to the blockchain applications that satisfy the following conditions: (I) The users have public and private information, where the public information is generated by the private information; (II) The users’ public information is disclosed in the blockchain data. These blockchain applications are called SkyEye-friendly blockchain applications; see Section 2.3 for

*The corresponding author.
details. SkyEye requires the user to register only once, and enables the regulator to trace users’ identities. In our design strategy, we add identity proofs, associated with the users’ private information, to the blockchain data. The design of SkyEye leverages some cryptographic primitives (e.g., zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs)) [21]. In addition, we demonstrate the security of SkyEye under specific cryptographic assumptions.

Finally, we implement two prototypes of SkyEye: SkyEyeHT and SkyEyeSG. These correspond to the two primary ways of generating public and private information in the blockchain applications. The first way is through a pseudorandom function, and the second way is using elliptic curve scalar multiplication. We evaluate the running time and related data storage requirements by performing SkyEyeHT and SkyEyeSG. Our evaluation results illustrate that using an i7 processor, a 16 GB RAM desktop machine, and a Merkle tree depth of 34, the time taken by a verifier to verify a user’s identity proof is nearly 4.6 ms in the first way and less than 25 ms in the second way.

Paper organization. The remainder of this paper is organized as follows. Section 2 provides the background. Section 3 describes our implementation and the evaluation results. We present the potential applications of SkyEye in Section 7. We discuss the related work in Section 8 and summarize this paper in Section 9.

2 BACKGROUND

2.1 Cryptographic Preliminaries

The cryptographic building blocks in our construction include the following: chameleon hash scheme, zk-SNARKs, and public key encryption. Below, we informally describe these notions.

Chameleon hash scheme. Compared with the traditional hash scheme, the chameleon hash scheme has a special property: the user who knows the trapdoor can easily find collision. A chameleon hash scheme $\text{Chash} = (\text{gChash}, \text{Kchash}, \text{Hchash}, \text{CFchash})$ is described below:

- $\text{gChash}(\lambda) \rightarrow ppchash$: Given a security parameter $\lambda$, $\text{gChash}$ returns the public parameters $ppchash$.
- $\text{Kchash}(ppchash) \rightarrow (pkchash, skchash)$: Given the public parameters $ppchash$, $\text{Kchash}$ returns a pair of public/private keys $(pkchash, skchash)$, where $skchash$ is also known as the trapdoor.
- $\text{Hchash}(pkchash, m, r) \rightarrow CH$: Given the public key $pkchash$, a message $m$, and a random number $r$, $\text{Hchash}$ returns a chameleon hash value $CH$ of $m$.
- $\text{CFchash}(skchash, m, m', r) \rightarrow r'$: Given the trapdoor $skchash$, two messages $m, m'$, and the random number $r$, $\text{CFchash}$ returns $r'$ such that $\text{Hchash}(pkchash, m, r) = \text{Hchash}(pkchash, m', r')$.

A chameleon hash scheme satisfies three secure properties: (i) collision resistance; (ii) trapdoor collision; and (iii) semantic security. More details are available in [3, 27].

There is a relationship between the public key $pkchash$ and the trapdoor $skchash$, which we refer to as the generation relationship. As in [27], the public parameters $ppchash = (p, q, g)$, where $p, q$ are prime numbers such that $p = kq + 1$, and the order of $g$ is $q$ in $\mathbb{Z}_p^*$. The public key $pkchash = h$ is computed as follows: $h = g^r \mod p$, where $x \in \mathbb{Z}_q^*$ is the trapdoor. Let equation $pkchash = \text{chash} \_\text{gen}(skchash)$ describe this relation, where $\text{chash} \_\text{gen}(\cdot)$ denotes the generation algorithm between $pkchash$ and $skchash$.

Zero-knowledge succinct non-interactive arguments of knowledge. Let $RAC = \{(x, w) \in \mathbb{F}^n \times \mathbb{F}^t | AC(x, w) = 0^t\}$ be an NP relation, where $F$ denotes a finite field, and $AC : \mathbb{F}^n \times \mathbb{F}^t \rightarrow \mathbb{F}^t$ denotes an $F$-arithmetic circuit. The language for $RAC$ is $\mathcal{L}_{AC} = \{x \in \mathbb{F}^n | \exists w \in \mathbb{F}^t \ s.t. \ AC(x, w) = 0^t\}$. A zk-SNARK scheme $\text{NIZK} = (\text{Kzizk}, \text{Pzizk}, \text{Vzizk})$ corresponds to the language $\mathcal{L}_{AC}$, which is described below:

- $\text{Kzizk}(\lambda, AC) \rightarrow (pk, ok)$. Given a security parameter $\lambda$ and an $F$-arithmetic circuit $AC$, $\text{Kzizk}$ returns a proving/verification key pair $(pk, ok)$.
- $\text{Pzizk}(pk, x, w) \rightarrow \pi$. Given the proving key $pk$, a statement $x$, and a witness $w$, $\text{Pzizk}$ returns a proof $\pi$ for a statement $x$ using a witness $w$.
- $\text{Vzizk}(ok, x, \pi) \rightarrow \{0, 1\}$. Given the verification key $ok$, the statement $x$, and the proof $\pi$, $\text{Vzizk}$ returns 1 if verification succeeds, or 0 if verification fails.

A zk-SNARK scheme satisfies five secure properties: (i) completeness; (ii) soundness; (iii) succinctness; (iv) proof of knowledge; and (v) perfect zero knowledge. More details are available in [7, 9, 21].

Public key encryption. A public key encryption scheme $\text{Enc} = (\text{Genp}, \text{Encp}, \text{Encv}, \text{Decp})$ is described below:

- $\text{Genp}(\lambda) \rightarrow ppenc$: Given a security parameter $\lambda$, $\text{Genp}$ returns the public parameters $ppenc$.
- $\text{Encp}(ppenc) \rightarrow (pkenc, skenc)$. Given the public parameters $ppenc$, $\text{Encp}$ returns a pair of public/private keys $(pkenc, skenc)$.
- $\text{Encv}(pkenc, m) \rightarrow c$. Given the public key $pkenc$ and a message $m$, $\text{Encv}$ returns a ciphertext $c$.
- $\text{Decv}(skenc, c) \rightarrow m | \perp$. Given the private key $skenc$ and the ciphertext $c$, $\text{Decv}$ returns a message $m$, or returns $\perp$ if decryption fails.

The public encryption scheme $\text{Enc}$ satisfies a security property: ciphertext indistinguishability under adaptive chosen ciphertext attack (IND-CCA2 security). More details are provided in [11].

2.2 Notation

We use $u$ to denote a user, $id_u$ to denote the identity of the user, and $\text{proof}_{id_u}$ to denote the user’s identity proof. Let $(pkchash_u, skchash_u)$ denote the user’s chameleon hash public/private key pair and $CH_{id_u}$ denote the chameleon hash value of identity $id_u$. We denote the user’s public/private information as $(pub_u, priv_u)$.

We use $pkchash_u || CH_{id_u}$ to denote the concatenation of $pkchash_u$ and $CH_{id_u}$, where $||$ denotes the concatenate symbol. Let $MT = (rt; pkchash_1 || CH_{id_1}, \ldots, pkchash_n || CH_{id_n})$ denote a Merkle tree, where $rt$ denotes the root of the Merkle Tree and $(pkchash, || CH_{id})$ denotes one leaf node in the Merkle tree. Let $(pkreq, skreq)$ denote the encryption public/private key pair of the regulator. Moreover, we use $\text{B}_u$ to denote SkyEye-friendly blockchain applications and $\text{B}_e$ to denote $\text{B}_u$ using SkyEye.

2.3 SkyEye-friendly blockchain applications

Here, we describe the blockchain data in $\text{B}_u$ and an overview of $\text{B}_e$. 
2.3.1 The Blockchain Data in $B_s$. $B_s$ satisfies two conditions: (I) The users have public and private information, where the public information is generated by the private information; (II) The users’ public information is disclosed in the blockchain data.

We use $pub = gen(prio)$ to describe the generation relation in the condition (I), where $pub$ denotes the public information, $prio$ denotes the private information, and $gen(\cdot)$ denotes a one-way function, i.e., it is easy to compute $pub$ using the private information $prio$ but is hard to invert. In many blockchain applications, every user generally has private information that corresponds to public information. For example, the public key address and the signature private key in Bitcoin[35] are the user's public/private information. In Zerocash[7], $(sn, (a_{sk}, \rho))$ is the user’s public/private information, where $sn$ is the serial number, $a_{sk}$ is the address private key, and $\rho$ is the random number used to generate the serial number.

According to the condition (II), the blockchain data in $B_s$ can be divided into two parts: one part is the users’ public information, such as the input/output addresses in Bitcoin[35], and the other part is the data contents, such as the payment amount and the executable contract code. Therefore, the blockchain data in $B_s$ can be represented as $\text{data}_{B_s} = [(pub_i)_{i \in \{1, \ldots, n\}}, C]_{\text{crytool}}$, where $(pub_i)_{i \in \{1, \ldots, n\}}$ denotes the set of the users’ public information, $n$ is the number of the users’ public information in the blockchain data such as the number of public key addresses in a Bitcoin transaction, $C$ denotes the data contents, and $\text{crytool}$ denotes the cryptographic tools (e.g., digital signature) that guarantee blockchain features such as tamper-resistance and privacy protection.

For example, Bitcoin[35], Ethereum[41], and RSCoin[13] are the applications that belong to $B_s$. In these blockchain applications, the public key address and the signature private key are the user’s public/private information, where the public key address is generated by the signature private key. Moreover, the user’s public key address is disclosed in the blockchain data. We briefly describe how to use SkyEye in these three applications in Section 7.

2.3.2 An overview of $B_s$. As shown in Figure 1, in (1) and (2), the user $u$ generates $(pub_u, priv_u)$, and publishes $pub_u$ to the node network. In (3), for creating data, the user $u$ obtains others’ public information $(pub_i)_{i \in \{1, \ldots, n-1\}}$ from the node network. In (4), the user $u$ creates $\text{data}_{B_s} = [(pub_i)_{i \in \{1, \ldots, n-1\}}, C]_{\text{crytool}}$, where $pub_n$ denotes the public information $pub_u$, and publishes $\text{data}_{B_s}$ to the node network. In (5) and (6), a verifier receives $\text{data}_{B_s}$ from the node network and verifies data contents. If the verification is successful, $\text{data}_{B_s}$ is valid and is added to the block generated by the verifier. In (7), the block is broadcast to the node network by the verifier. In (8), according to a consensus mechanism, the nodes in the network select a final block and add it to the blockchain.

3  KEY IDEAS AND SKYEYE OVERVIEW

In this section, we provide key ideas in SkyEye design and an overview of SkyEye.

3.1 Key Ideas

As shown in Figure 2, the design idea of SkyEye is to add identity proofs to $\text{data}_{B_s}$. The blockchain data in $B_{se}$ can be represented as $\text{data}_{B_{se}} = [(pub_i, proof_id)_{i \in \{1, \ldots, n\}}, C]_{\text{crytool}}$, where $proof_id$ denotes the identity proof of the user whose identity is $id_i$, and the other variables are the same as those in $\text{data}_{B_s}$, $(pub_i, proof_id)$ can be viewed as the new public information $pub'_i$ of the user whose identity is $id_i$.

The identity proof is the core of SkyEye. The two purposes of the identity proof are to prove the user’s legitimacy and to achieve tracing. Next, we briefly describe the identity proof according to the above two purposes. More details are described in Section 5.

3.1.1 Proving the user’s legitimacy. We assume that the user $u$ has generated $(pub_u, priv_u)$, $(pk_{hash_u}, sk_{hash_u})$, and $CH_{du} = H_{\text{chash}}(pk_{hash_u}, id_u, r)$, where $r$ is the random number sampled by $u$.

Step 1: user registration. To prove the user’s legitimacy, there must be something (similar to a certificate) that can indicate the user’s legitimacy. In SkyEye, this is done through user registration. Here, we briefly introduce user registration in SkyEye. More details on user registration appear in Section 5.1.2.

As shown in Figure 3, the user $u$ sends registration information $(C_{\text{info}}, \pi_{\text{info}})$ to the regulator, where $C_{\text{info}}$ is the ciphertext that is the encryption of the plaintext $(pk_{hash_u}, id_u, CH_{id_u})$ under the public key $pk_{req}$ and $\pi_{\text{info}}$ is the zk-SNARK proof that is used to prove: "I know $(sk_{hash_u}, r)$ which can generate $pk_{hash_u}$ and $CH_{id_u}$.”

If the verification of $(C_{\text{info}}, \pi_{\text{info}})$ is successful, the regulator stores $(pk_{hash_u}, id_u, CH_{id_u})$, and adds $pk_{hash_u}$ to $CH_{id_u}$ to the Merkle tree $MT$. The regulator publishes the Merkle tree $MT$ at the right time. The registration of $u$ is successful only if $pk_{hash_u}$ appears in the Merkle tree $MT$.

The Merkle tree $MT$ can be regarded as a credential of proving the user’s legitimacy. In other words, to prove the legitimacy of $u,$
the user $u$ must prove that $pk_{chash_u}||CH_{id_u}$ appears in the Merkle tree $MT$. Therefore, $proof_{id_u}$ generated by the user $u$ must be able to prove the following.

"I know $(sk_{chash_u}, id_u, r)$ that can generate $pk_{chash_u}$ and $CH_{id_u}$, and $pk_{chash_u}||CH_{id_u}$ appears as a leaf of the Merkle tree $MT$ with the root $rt$".

Step 2: establishing the binding relationship between $pub_u$ and $proof_{id_u}$. Although $proof_{id_u}$ described above can prove the legitimacy of $u$, an issue remains. It can be seen from Figure 4 that $(pub_u, proof_{id_u})$ needs to be published in the node network. The adversary who has registered with the regulator can generate an identity proof $proof_{id_u}$, and publish $(pub_u, proof_{id_u})$ to the node network. At this time, $pub_u$ corresponds to two different users’ identity proofs (i.e., $proof_{id_u}$ and $proof_{id_u'}$). This presents a great obstacle to trace. We need to establish a relation between $pub_u$ and $proof_{id_u}$ to ensure that only the user $u$ can generate the identity proof $proof_{id_u}$ corresponding to $pub_u$.

The key idea behind establishing this relation is that we establish the binding relationship between $pri_u$ and $proof_{id_u}$. Because $pub_u$ is generated by $pri_u$, there is a binding relationship between $pub_u$ and $proof_{id_u}$.

We leverage the special property of chameleon hash scheme (i.e., the user who knows the trapdoor can easily find collision) to establish the binding relationship between $pri_u$ and $proof_{id_u}$. Given the private information $pri_u$, the user $u$ who knows $sk_{chash_u}$ can easily find $r'$ such that $CH_{id_u} = H_{chash}(pk_{chash_u}, pri_u, r')$. To achieve the binding of $pri_u$ and $proof_{id_u}$, we require the identity proof $proof_{id_u}$ to prove the following.

- The public information $pub_u$ is generated by the private information $pri_u$.
  - I know $(sk_{chash_u}, pri_u, r')$ that generates $pk_{chash_u}$ and $CH_{id_u}$.
  - $pk_{chash_u}||CH_{id_u}$ appears as a leaf of a Merkle tree with the root $rt$.

This binding relationship between $proof_{id_u}$ and $pri_u$ will ensure that only the user $u$ who knows the private information $pri_u$ can generate the identity proof $proof_{id_u}$, and others cannot forge the identity proof corresponding to $pub_u$.

Moreover, the special property of chameleon hash scheme makes the user $u$ just register once, and then can generate identity proofs without involving the regulator.

3.1.2 Achieving tracing. To achieve tracing, we add $C_{id_u}$, which is the ciphertext of $pk_{chash_u}$ under the regulator public key $pk_{reg}$, to $proof_{id_u}$, and require $proof_{id_u}$ to prove that the plaintext corresponding to the ciphertext $C_{id_u}$ is $pk_{chash_u}$. Because the regulator has the record $(pk_{chash_u}, id_u, CH_{id_u})$, the regulator can obtain $pk_{chash_u}$ through decrypting $C_{id_u}$ and determine the identity $id_u$ of the user $u$ based on the record.

3.2 SkyEye Overview

As can be seen from Figure 4, SkyEye’s application strategy in $B_u$ is that the user $u$ generates the identity proof $proof_{id_u}$ corresponding to the public information $pub_u$ so that every verifier can verify the legitimacy of $u$ and the regulator can trace $data_{ver}$.

From Figure 4, it can also be seen that the SkyEye scheme mainly has the following operations between the regulator, users, and verifiers.

- **User registration.** The user generates registration information and sends it to the regulator. The regulator is responsible for the verification of registration information.
- **Generating identity proof.** The user who registers successfully can generate the identity proof. There is a binding relationship between the identity proof and the private information.
- **Verifying identity proof.** Different from traditional verification process in the blockchain, the verifier (e.g., the miner) verifies identity proofs in addition to verifying data contents. If the data contents and identity proofs are simultaneously verified successfully, the data will be added to the block generated by the verifier.
- **Tracing.** The regulator traces the users’ true identities in the blockchain data.

4 DEFINITION OF A TRACEABLE SCHEME

4.1 Definition

A traceable scheme for blockchain is a tuple of polynomial-time algorithms $\Pi = (\text{Setup}, \text{GenInfo}, \text{VerInfo}, \text{GenProof}, \text{VerProof}, \text{Trace})$ described below:

- **Setup$(\lambda)$** $\rightarrow$ pp. Given a security parameter $\lambda$, Setup returns public parameters pp. This algorithm is executed by a trusted party

![Figure 3: User registration.](image1)

![Figure 4: Overview of $B_u$. The red lines 1*, 2*, 3* and 4* represent the operations of the SkyEye scheme. The red line 1* denotes user registration. The red line 2* denotes generating identity proof. The red line 3* denotes verifying identity proof. The red line 4* denotes tracing.](image2)
and is done only once. The public parameters $pp$ are published and made available to all parties.

- $Gen_{info}(pp, id) \rightarrow reginfo$. Given the public parameters $pp$ and a user identity id, this algorithm returns the registration information reginfo.

- $Ver_{info}(pp, reginfo, sk_{reg}) \rightarrow b$. Given the public parameters $pp$, the registration information $reginfo$ and the regulator’s private key $sk_{reg}$, this algorithm returns a bit $b$. If verification succeeds, this algorithm returns 1; otherwise, it returns 0.

- $Gen_{proof}(pp, pub, pri, id, CH_{id}, pk_{chash}, sk_{chash}; r, rt, path_{id}) \rightarrow proof_{id}$. Given the public parameters $pp$, a public/private information $(pub, priv)$, the identity id, the chameleon hash value $CH_{id}$ of id, the chameleon hash public/private key $(pk_{chash}, sk_{chash})$, the random element $r$ for computing $CH_{id}$, the Merkle tree root $rt$, and the path $path_{id}$ from $pk_{chash}|CH_{id}$ to $rt$, $Gen_{proof}$ returns the user’s identity proof $proof_{id}$.

- $Ver_{proof}(pp, pub, proof_{id}) \rightarrow b$. Given the public parameters $pp$, the user’s public/private information $pub$ and the user’s identity proof $proof_{id}$, $Ver_{proof}$ returns a bit $b$. If the verification of $proof_{id}$ succeeds, this algorithm returns 1; otherwise, it returns 0.

- $Trace(data_{reg}, sk_{reg}) \rightarrow ID$. Given the blockchain data $data_{reg}$ and the regulator’s private key $sk_{reg}$, $Trace$ returns the identity set ID for $data_{reg}$.

4.2 Security

We assume that in $B_{s}$ relevant cryptographic techniques (e.g., digital signatures) have been used to ensure that the blockchain data generated by the users cannot be tampered with. Therefore, the identity proofs added to the blockchain data also cannot be tampered with. We also assume that the regulator is trusted and has an efficient way of verifying user identity (such as, face recognition, identity card, or short message service (SMS) verification). Therefore, the goals of the adversary are to forge the identity proof and to distinguish two distinct identity proofs. The security of a traceable scheme must satisfy two properties: identity proof indistinguishability and identity proof unforgeability.

Definition 4.1. A traceable scheme $[\ ] = (Setup, Gen_{info}, Ver_{info}, Gen_{proof}, Ver_{proof}, Trace)$ is secure if it satisfies identity proof indistinguishability and identity proof unforgeability.

Below, we briefly describe each property, and defer formal definition of each property to Appendix A.

- Identity proof indistinguishability. This property requires that even if the adversary can adaptively induce honest parties to perform operations of his choice, the identity proof reveals no information except for some public information, such as public addresses and serial numbers. In other words, even if the adversary queries two different honest parties (one identity is $id_{0}$, and the other identity is $id_{1}$), no polynomial-time adversary can distinguish between the identity proofs $proof_{id_{0}}$ and $proof_{id_{1}}$. The meaning of this property is that if the blockchain is indistinguishable, adding the identity proofs to the blockchain data does not affect the indistinguishability of the blockchain.

- Identity proof unforgeability. This property requires that even if the adversary can adaptively induce honest parties to perform operations of his choice, no polynomial-time adversaries can forge the identity proof of honest parties. This property ensures that the adversary cannot forge the honest user’s identity proof to create blockchain data for evading tracing.

5 CONSTRUCTION

5.1 SkyEye Construction

5.1.1 Initialization. The public parameters $pp$ created by the Setup algorithm include the following information: the proving/verification key $(pk_{info}, sk_{info})$ used to generate and verify the zk-SNARK proof $\pi_{info}$ for the NP relation $R_{info}$ (see Section 5.1.2 for details), the proving/verification key $(pk_{proof}, sk_{proof})$ used to generate and verify the zk-SNARK proof $\pi_{proof}$ for the NP relation $R_{proof}$ (see Section 5.1.3 for details), the regulator public key $pk_{reg}$ for public key encryption, and the public parameters $pp_{chash}$ of the chameleon hash scheme. Because the regulator is trusted, Setup algorithm is performed by the regulator. (See Setup algorithm in Algorithm 1 for specific operations.)

5.1.2 User Registration. As shown in Algorithm 1, the $Gen_{info}$ algorithm is responsible for the generation of registration information and the $Ver_{info}$ algorithm is used to verify the registration information.

In the $Gen_{info}$ algorithm, a user generates the chameleon hash public-private pair $(pk_{chash}, sk_{chash})$ based on $pp_{chash}$ and computes the chameleon hash value $CH_{id}$ of identity id. Then the user stores $(id, pk_{chash}, sk_{chash}, r, CH_{id})$, where $r$ is a random number used to generate $CH_{id}$. At this point, the user can produce a zk-SNARK proof $\pi_{info}$ for the following NP relation, which we call $R_{info}$:

- Given $x_{info} = (id, pk_{chash}, CH_{id})$, I know $w_{info} = (sk_{chash}, r)$ such that:
  - The chameleon hash private key matches the chameleon hash public key: $pk_{chash} = chash_{gen}(sk_{chash})$.
  - The chameleon hash is computed correctly: $CH_{id} = H_{chash}(pk_{chash} \cdot id, r)$.

The $Gen_{info}$ algorithm outputs registration information $reginfo$, which consists of the ciphertext $C_{info}$ and zk-SNARK proof $\pi_{info}$. $C_{info}$ is the ciphertext of $x_{info}$ encrypted by $pk_{reg}$.

The verification operations in the $Ver_{info}$ algorithm include verifying the identity id and verifying the zk-SNARK proof $\pi_{info}$. If the above two operations are verified successfully, the regulator stores $(pk_{chash}, id, CH_{id})$, and adds $pk_{chash}|CH_{id}$ to the Merkle tree $MT$. Meanwhile, this algorithm returns 1. Finally, the regulator broadcasts $MT$ to the network at the right time.

5.1.3 Generating and Verifying Identity Proof. As shown in Algorithm 1, the $Gen_{proof}$ algorithm is used to generate the identity proof for each user.

In the $Gen_{proof}$ algorithm, assume a user has generated public/private information $(pub, pri)$. According to the known trapdoor $sk_{chash}$, the user can calculate a value $r'$ such that $CH_{id} = H_{chash}(pk_{chash} \cdot pri, r')$. Next, the user computes ciphertext $C_{id} = E_{enc}(pk_{reg}, pk_{chash}, r, n)$, where $pk_{reg}$ is the public key of the regulator, and $n$ is the random number used for encryption. Finally, the user produces a zk-SNARK proof $\pi_{proof}$ for the following NP relation, which we term $R_{proof}$:

- Given a statement $x_{proof} = (pub, rt, pk_{reg}, C_{id})$, I know $w_{proof} = (path_{id}, CH_{id}, pk_{chash}, pk_{chash} \cdot pri, r', n)$ such that:
Algorithm 1 SkyEye Construction

\textbf{Setup}

\textbf{Input}: security parameter \(\lambda\);
\textbf{Output}: public parameters \(pp\);
1. construct arithmetic circuit \(AC_{info}\) for relation \(R_{info}\) at security \(\lambda\);
2. construct arithmetic circuit \(AC_{proof}\) for relation \(R_{proof}\) at security \(\lambda\);
3. \((pk_{info}, ok_{info}) = K_{nizk}(\lambda, AC_{info})\);
4. \((pk_{proof}, ok_{proof}) = K_{nizk}(\lambda, AC_{proof})\);
5. compute \(pp_{enc} = G_{enc}(\lambda)\);
6. compute \((pk_{reg}, sk_{reg}) = K_{enc}(pp_{enc})\);
7. compute \(pp_{chash} = G_{chash}(\lambda)\);
8. return \(pp = (pk_{info}, ok_{info}, pk_{proof}, ok_{proof}, pk_{reg}, pp_{chash})\).

\textbf{Geninfo}

\textbf{Input}:
- public parameters \(pp\),
- user identity id;
\textbf{Output}:
- registration information \(\text{reginfo}\);
1. \((pk_{chash}, sk_{chash}) = \mathcal{K}_{chash}(pp_{chash})\);
2. randomly sample \(r\);
3. compute \(CH_{id} = \mathcal{H}_{chash}(pk_{chash}, id, r)\);
4. set \(x_{info} = (id, pk_{chash}, CH_{id}), w_{info} = (sk_{chash}, r)\);
5. \(\pi_{info} = \mathcal{P}_{nizk}(pk_{info}, x_{info}, w_{info})\);
6. set \(C_{info} = E_{enc}(pk_{reg}, x_{info})\);
7. store \((id, pk_{chash}, sk_{chash}, r, CH_{id})\);
8. return \(\text{reginfo} = (C_{info}, \pi_{info})\).

\textbf{Verinfo}

\textbf{Input}:
- public parameters \(pp\),
- registration information \(\text{reginfo}\),
- the regulator’s private key \(sk_{reg}\);
\textbf{Output}: bit \(b\);
1. parse \(\text{reginfo}\) as \((C_{info}, \pi_{info})\);
2. \(x_{info} = D_{enc}(sk_{reg}, C_{info})\);
3. parse \(x_{info}\) as \((id, pk_{chash}, CH_{id})\);
4. if \(id\) not valid then
5. return \(b=0\);
6. end if
7. if \(\mathcal{V}_{nizk}(sk_{info}, x_{info}, \pi_{info}) = 0\) then
8. return \(b=0\);
9. else
10. store \((pk_{chash}, id, CH_{id})\);
11. add \(pk_{chash}\)(id, CH_{id}) to the Merkle tree \(MT\);
12. return \(b=1\);
13. end if

\textbf{Genproof}

\textbf{Input}:
- public parameters \(pp\),
- user public/private information \((pub, priv)\),
- user identity \(id\),
- chameleon hash value \(CH_{id}\) of \(id\),
- chameleon hash public/private key \((pk_{chash}, sk_{chash})\),
- random number \(r\) for computing \(CH_{id}\),
- Merkle tree root \(rt\),
- path \(path_{id}\) from \(pk_{chash}\)(id) to \(rt\);
\textbf{Output}:
- identity proof \(proof_{id}\);
1. compute \(r’ = C’F_{chash}(sk_{chash}, id, priv, r)\);
2. randomly sample \(rn\) for encrypting;
3. compute \(C_{id} = E_{enc}(pk_{reg}, pk_{chash}, rn)\);
4. set \(u_{proof} = (rt, pk_{reg}, C_{id})\);
5. set \(x_{proof} = (pub, u_{proof})\),
6. compute \(\pi_{proof} = \mathcal{P}_{nizk}(pk_{proof}, x_{proof}, w_{proof})\);
7. return \(proof_{id} = (u_{proof}, \pi_{proof})\).

\textbf{Verproof}

\textbf{Input}:
- public parameters \(pp\),
- public information \(pub\),
- identity proof \(proof_{id}\);
\textbf{Output}:
- bit \(b\);
1. parse \(proof_{id}\) as \((u_{proof}, \pi_{proof})\)
2. set \(x_{proof} = (pub, u_{proof})\)
3. if \(\mathcal{V}_{nizk}(sk_{proof}, x_{proof}, \pi_{proof}) = 0\) then
4. return \(b=0\);
5. else
6. return \(b=1\);
7. end if

\textbf{Trace}

\textbf{Input}:
- blockchain data \(data_{se}\),
- the regulator’s private key \(sk_{reg}\);
\textbf{Output}:
- identity set \(ID\) for \(data_{se}\);
1. set \(ID = \emptyset\);
2. get the chameleon hash public key ciphertext set \(C\) from \(data_{se}\);
3. for each \((C_{id} \in C)\) do
4. compute \(pk_{chash} = D_{enc}(sk_{reg}, C_{id})\);
5. search \((pk_{chash}, id, CH_{id})\) records, get \(id_i\) corresponding to \(pk_{chash}\);
6. put \(id_i\) in \(ID\);
7. end for
8. return \(ID\);
The private information matches the public information: $pub = gen(prio)$.

The chameleon hash private key matches the chameleon hash public key: $pk_{chash} = \text{chash}\_gen(sk_{chash})$.

The chameleon hash is computed correctly: $CH_{id} = \mathcal{H}\text{chash}(pk_{chash}, prio, r^*)$.

The ciphertext $C_{id}$ corresponds to the plaintext $pk_{chash}: C_{id} = \mathcal{E}_{\text{enc}}(pk_{reg}, pk_{chash}, r\cdot n)$.

The $pk_{chash}||CH_{id}$ appears as a leaf of a Merkle tree with the root rt.

Finally, $Gen_{proof}$ outputs the identity proof $proof_{id} = (u_{proof}, \pi_{proof}) = ((rt, pk_{reg}, C_{id}), \pi_{proof})$.

The $Ver_{proof}$ algorithm in Algorithm 1 is used to verify the user’s identity proof $proof_{id}$. The verification operation verifies the zk-SNARK proof $\pi_{proof}$. This algorithm returns 1 if and only if the above operation verifies successfully.

### 5.1.4 Tracing

As shown in Algorithm 1, the Trace algorithm is used to trace the blockchain data $data_{\text{tree}}$. The regulator can obtain a chameleon hash public key set by decrypting each chameleon hash public key ciphertext in $data_{\text{tree}}$. Then the regulator can determine the true identities of the users in $data_{\text{tree}}$ according to the record that stores each user’s chameleon hash public key, chameleon hash value, and identity. This algorithm returns the identity set $ID$.

### 5.2 SkyEye Security

Theorem 5.1. Assuming that the Chash scheme is collision resistant, trapdoor collision and semantic security, the NIZK scheme is perfect zero-knowledge and proof of knowledge, the encryption scheme $Enc$ satisfies IND-CCA2 security, and $gen(\cdot)$ is a one-way function. Our scheme $\Pi = (\text{Setup}, \text{Geninfo}, \text{Verinfo}, \text{Genproof}, \text{Verproof}, \text{Trace})$ described in Algorithm 1 is a secure (cf. Definition 4.1) traceable scheme.

We provide the proof of Theorem 5.1 in Appendix B.

### 6 IMPLEMENTATION AND EVALUATION

#### 6.1 Implementation

There are two main ways of generating public and private information in blockchain applications. One is through the pseudorandom function (e.g., Zerocash[7], Hawk[26]), i.e. $pub = PRF_{\text{prio}}(s)$, where $PRF$ denotes the pseudorandom function, $pub$ is the pseudorandom number, $prio$ is the private key used to generate $pub$, and $s$ is the uniform seed. The other way is to use elliptic curve scalar multiplication (e.g., Bitcoin) to generate the public and private information, i.e., $pub = prio \cdot G$, where $prio$ is a scalar, $G$ is a base point on the elliptical curve, and $pub$ is a point on the elliptical curve. We use $SkyE_{\text{EY}}$ to represent the scheme that generates public and private information in the first way, and $SkyE_{\text{EY}}$ to represent the scheme that generates public and private information in the second way. We use the C++ programming language to implement the prototype of the above two different schemes based on the zk-SNARK library, libsnark[8].

There are some cryptographic building blocks in $SkyE_{\text{EY}}$: the pseudorandom function, chameleon hash scheme, hash function in the Merkle tree, public encryption scheme, and zk-SNARK scheme. For the chameleon hash scheme, we use the chameleon hash scheme proposed by Hugo Krawczyk and Tal Rabin[27]. For efficiency, we use the SHA256 compression function to implement the pseudorandom function and hash function in the Merkle tree, which is similar to the approach used in Zerocash[7]. We use the practical public key encryption scheme proposed by Cramer and Shoup[11], an IND-CCA2 secure public encryption scheme, as our encryption scheme. We use the scheme proposed by Parno et al.[37] as the zk-SNARK scheme. In the concrete implementation, we use the Baretto-Naehrig elliptic curve[6] that provides 128-bit security as the underlying curve of the zk-SNARK scheme. The implementation of the chameleon hash and public key encryption scheme is based on a prime field of 254 bits.

In $SkyE_{\text{EY}}$, the main cryptographic building block differs from the former in that the pseudorandom function is replaced by elliptic curve scalar multiplication. The chameleon hash scheme, public key encryption scheme, and zk-SNARK scheme are the same as those in the $SkyE_{\text{EY}}$. In the concrete implementation, we use the MNT4 elliptic curve[34] as the underlying curve of the zk-SNARK scheme. The implementation of elliptic curve scalar multiplication is based on the MNT6 elliptic curve[34]. We implement the chameleon hash scheme and public key encryption scheme in a prime field of 298 bits. To improve efficiency, in the formation of the Merkle tree, because the length of the leaf node is 298 bits, two leaf nodes together cannot form 512 bits. Therefore, the upper node is generated by the leaf node using the standard SHA256. In addition, the data length of the node above the leaf node is 256 bits, so each node that is not generated through the leaf node is generated by the SHA256 compression function.

### 6.2 Evaluation

We evaluate the performance of every algorithm in the two aforementioned schemes in two different configurations: configuration 1, with an Intel i5 processor and 4 GB memory laptop; and configuration 2, with an Intel i7 processor and 16 GB memory desktop computer. The depth of the Merkle tree in our evaluation is 10, 20, 30, and 34, respectively. In other words, the maximum number of users which the Merkle tree supports is $2^{10} \approx 1024$, $2^{20} \approx 1048576$, $2^{30} \approx 1073741824$, and $2^{34} \approx 268435456$. This fully meets demand, because the current global population is about 7.5 billion, and $2^{34}$ reaches more than 17 billion. Moreover, we evaluate the performance of the Trace algorithm under the condition that there are already 1024 successfully registered users at the regulator.

Table 1 and Table 2 illustrate the performance results of the $Setup, Gen_{\text{info}}, Ver_{\text{info}}, Gen_{\text{proof}}, Ver_{\text{proof}}$ and $Trace$ algorithms in $SkyE_{\text{EY}}$ and $SkyE_{\text{EY}}$, respectively (the time in the two tables is the average of 10 runs per algorithm). In the two tables, time represents the running time of the algorithm, and $\cdot$ represents the data length. For example, the $\lvert pk_{\text{info}} \rvert$ represents the length of the proving key in the registration. Without loss of generality, using an i7 processor, a 16 GB memory desktop machine, and with a tree depth of 34 in Table 1, we can obtain the results of the $SkyE_{\text{EY}}$ scheme:

- $Setup$ algorithm takes 94 s. The size of the proving key and verification key used for user registration are 480 KB and 574 B, respectively. And the size of the proving key and verification key used for user identity proof are 231 MB and 21 KB, respectively.
Table 1: Performance of SkyEyeH

<table>
<thead>
<tr>
<th>SkyEyeH</th>
<th>Tree depth</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Setup</td>
<td>69</td>
<td>114</td>
</tr>
<tr>
<td>$p_{info}$ (KB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sigma_{info}$ (B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$pk_{proof}$ (MB)</td>
<td>90</td>
<td>149</td>
</tr>
<tr>
<td>$zk_{proof}$ (KB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geninfo</td>
<td>time (ms)</td>
<td></td>
</tr>
<tr>
<td>$\tau_{info}$ (B)</td>
<td>475.7</td>
<td>494.3</td>
</tr>
<tr>
<td>Verinfo</td>
<td>time (ms)</td>
<td>15.3</td>
</tr>
<tr>
<td>Genproof</td>
<td>time (s)</td>
<td>29</td>
</tr>
<tr>
<td>$\tau_{proof}$ (B)</td>
<td>287</td>
<td></td>
</tr>
<tr>
<td>Verproof</td>
<td>time (ms)</td>
<td>10.1</td>
</tr>
<tr>
<td>Trace</td>
<td>time (ms)</td>
<td>0.13</td>
</tr>
</tbody>
</table>

Table 2: Performance of SkyEyes

<table>
<thead>
<tr>
<th>SkyEyes</th>
<th>Tree depth</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Setup</td>
<td>187</td>
<td>296</td>
</tr>
<tr>
<td>$p_{info}$ (KB)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sigma_{info}$ (B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$pk_{proof}$ (MB)</td>
<td>105</td>
<td>174</td>
</tr>
<tr>
<td>$zk_{proof}$ (KB)</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Geninfo</td>
<td>time (ms)</td>
<td>1398.0</td>
</tr>
<tr>
<td>$\tau_{info}$ (B)</td>
<td>337</td>
<td></td>
</tr>
<tr>
<td>Verinfo</td>
<td>time (ms)</td>
<td>51.8</td>
</tr>
<tr>
<td>Genproof</td>
<td>time (s)</td>
<td>56</td>
</tr>
<tr>
<td>$\tau_{proof}$ (B)</td>
<td>337</td>
<td></td>
</tr>
<tr>
<td>Verproof</td>
<td>time (ms)</td>
<td>47.8</td>
</tr>
<tr>
<td>Trace</td>
<td>time (ms)</td>
<td>0.14</td>
</tr>
</tbody>
</table>

- $Gen_{info}$ requires 272.4 ms, and the size of the zk-SNARK proof $\tau_{info}$ is 287 B.
- $Ver_{info}$ algorithm takes 6.9 ms.
- $Gen_{proof}$ algorithm takes 35 s to generate a user’s identity proof, and the size of the zk-SNARK proof $\tau_{proof}$ is 287 B.
- $Ver_{proof}$ algorithm takes 4.6 ms.
- $Trace$ algorithm takes 0.075 ms to trace a user’s identity.

The tables reveal the following:
- In each configuration, the time required for verification by the regulator and the verifiers is small and does not substantially change as the depth of the tree changes. As shown in Table 1, the regulator takes approximately 15 ms to verify the user registration information in configuration 1 and approximately 7 ms in configuration 2; and the time taken by a verifier to verify the user identity proof is approximately 10 ms in configuration 1 and approximately 5 ms in configuration 2. From Table 2, we can observe that the time taken for verifying the user registration information is approximately 53 ms in configuration 1 and approximately 28 ms in configuration 2; and the time taken for verifying the user identity proof is approximately 48 ms in configuration 1 and approximately 25 ms in configuration 2.
- Not all of the information in SkyEye must be on-chain. Only the information $proof_{id}$ generated by the $Gen_{proof}$ algorithm is added to the user data. Furthermore, the size of the user’s proof $\tau_{proof}$ in the $proof_{id}$ is dominant. As can be observed from the two
Although SkyEye is a centralized traceable scheme, SkyEye provides an alternative traceable strategy for the blockchain applications that belong to $B_s$. If a blockchain application that does not belong to $B_s$ wants to use SkyEye, this application can modify some rules to make it belong to $B_s$.

In this section, we briefly describe how to apply SkyEye to some applications: Bitcoin[35], Ethereum[41], and RSCoin[13]. It can be seen from Section 2.3 that these three applications belong to $B_s$.

### 7.1 Bitcoin and Ethereum

Although SkyEye is a centralized traceable scheme, SkyEye provides two functions: proving the user’s legitimacy and tracing. Therefore, blockchain applications can use SkyEye according to their requirements. If SkyEye’s tracing function is used in the Bitcoin or Ethereum, because the identity proofs are added to the blockchain data, the underlying rules of these applications need to be modified.

If Ethereum or Bitcoin only uses the function of proving the user’s legitimacy, the underlying rules of these applications do not be modified. For example, in Bitcoin or Ethereum, users wish to trade with certifiable merchants in many cases, which is more secure and more assured. At this point, SkyEye can be used to allow the merchants that need to be certified to register with the regulator. A successfully registered merchant shows the public key address and identity proof to the user. The user can determine whether the merchant is certifiable by verifying the identity proof. If so, the user generates a transaction to transfer coins to the merchant. This transaction does not contain the merchant’s identity proof. Therefore, all operations of SkyEye are performed off-chain. In this way, the user only needs to reach a consensus with the merchant. And Ethereum and Bitcoin do not need to modify the underlying rules.

### 7.2 A Regulatory Currency

Although existing private digital currency systems (such as Bitcoin) exhibit advantages such as cost savings, transaction transparency and high security, their problems (e.g., low transaction throughput and difficulty in supervision) have severely restricted the application of digital currency. To make better use of the advantages of digital currency and to prevent the risks and harms caused by private digital currency, many countries in the world such as China, Britain, and America have performed studies on digital currency, striving to issue legal digital currency as soon as possible to enhance their international competitiveness. The representative is RSCoin[13], the first digital currency framework. Because the transaction amount is disclosed in the blockchain in RSCoin, realizing identity tracing through traceable scheme achieves the purpose of supervision.

SkyEye can combine with the RSCoin framework to enable the central bank to trace transactions in the blockchain, thereby constructing a regulatory currency. As can be seen from Figure 5, the user first registers with the central bank. After the central bank passes the verification, the user can generate the transaction $tx$, which contains the data contents and the identity proofs, and send $tx$ to the mintettes that have been certified by the central bank. Mintettes are divided into two groups: group 1 verifies whether the user’s inputs are legal (such as whether the inputs belong to the unspent transaction output (UTXO) collection). In addition to the original verification of RSCoin, group 1 must also verify the identity proofs in the transaction $tx$. Finally, the group 2 provides several low-level blocks to the central bank. The central bank merges the blocks to produce a high-level block and adds it to the blockchain. At this point, the central bank can obtain the identities of the users through SkyEye to complete the tracing of the blockchain.

### 8 DISCUSSION AND FUTURE WORK

In SkyEye, the centralization of the regulator is a major issue. The regulator can arbitrarily trace the identity of blockchain data without any restrictions and oversight.

From the data tracing process of the regulator, it can be seen that the regulator must first use its private key $sk_{reg}$ to decrypt the ciphertext of each user’s chameleon hash public key in the blockchain data. Therefore, we can restrict the regulator through the distributed key generation (DKG) protocol[22]. Specifically, the public/private key pair $(pk_{reg}, sk_{reg})$ is generated by a committee with a threshold of $t$ through the DKG protocol. In this way, $pk_{reg}$ is made public, and each committee member has a share of $sk_{reg}$. The regulator submits the data and tracing evidence to the committee. If at least $t+1$ members of the committee accept the data and tracing evidence, the regulator will obtain $sk_{reg}$ from the committee.

However, this approach does not completely restrict the regulator. Even if the committee regularly updates the public/private
key pair, as long as the regulator obtains the private key $sk_{reg}$ in a cycle, it can trace not only the data submitted to the committee, but also all user data in this cycle. In future work, we will consider how to restrict the regulator to make the regulator only trace the data submitted to the committee.

### 9 RELATED WORK

Blockchain research focuses primarily on enhancing blockchain privacy protection [7, 12, 33, 38], improving blockchain scalability [16, 31, 42], analyzing blockchain security [17–19, 30], and applying blockchain to other areas [28, 29, 32, 40]. However, research on traceable mechanisms is limited.

Narula, Vasquez, and Virza proposed zkLedger [36], the first distributed ledger system, that provides strong privacy protection, public verifiability, and practical auditing. zkLedger uses table construction in the ledger. Each user identity corresponds to each column in the ledger. Therefore, the regulator can determine every user identity through the ledger. However, this traceable mechanism in zkLedger cannot be applied to environments with a large number of users and is used only for auditing digital asset transactions over some banks.

Defrawy and Lampkins [15] proposed a proactively-private digital currency (PDC) scheme that can provide privacy-preserving and accountability. In their scheme, the ledger is kept by a group of ledger servers. Every ledger server has a balance ledger that contains a share of every user identity. Therefore, the regulator can determine every user identity through those ledger servers. However, their traceable mechanism does not seem to have been extended to other applications.

Ateniese et al. [4] constructed a scheme that provides certified Bitcoin addresses to enable Bitcoin users to trade with certified users authenticated by the trusted certificate authority. The regulator can determine every user identity through the authority. However, if a user wants to use a new certified address for each transaction, the user must contact the certificate authority to obtain a certified address. This reduces the efficiency of the entire system and exerts considerable pressure on the certificate authority when the number of users is large. Moreover, their approach only applies to Bitcoin.

Garman et al. [20] designed new decentralized anonymous payment (DAP) systems to address the regulatory issue by adding privacy preserving policy-enforcement mechanisms that guarantee regulatory compliance, allow selective user tracing, and admit tracing of tainted coins. The regulator can determine every user identity through the identity escrow policy. However, the DAP system are based on Zerocash [7].

The traceable mechanisms proposed above can only be applied to specific application environments and do not seem to have been extended to other applications. We propose SkyEye, a traceable scheme for blockchain. Our scheme can be applied to a class of blockchain applications, which is denoted by $B_k$.

### 10 CONCLUSION

In this paper, we design SkyEye, a traceable scheme for blockchain. SkyEye can be applied to the blockchain applications that satisfy the following conditions: (I) The users have public and private information, where the public information is generated by the private information; (II) The users’ public information is disclosed in the blockchain data. SkyEye just requires the user to register only once, and enables the regulator to trace users’ identities. Moreover, we implement two different SkyEye prototypes: SkyEye$_1$ and SkyEye$_2$. Our evaluation results show that even if the number of users is very large, the registration information and identity proof are verified quickly.

### ACKNOWLEDGMENTS

This work was supported in part by the National Key R&D Program of China (2017YFB0802500), Beijing Municipal Science and Technology Project (No. Z191100007119007), and Shandong province major science and technology innovation project (2019JZZY020129).

### REFERENCES

A SECURITY OF THE TRACEABLE SCHEME

We describe identity proof indistinguishability and identity proof unforgeability. Every property is formalized as an experiment between an adversary \(A\) and a challenger \(C\). The behavior of the honest user with identity \(id\) is realized by the oracle \(O_{id}\), and the behavior of the regulator is realized by the oracle \(O_{reg}\). We assume that the honest users and adversary in the experiment have already registered successfully in the regulator, i.e., they can generate any identity proof. Below, we describe how \(O_{id}\) and \(O_{reg}\) work.

Oracles \(O_{id}\) and \(O_{reg}\) are initialized by challenger \(C\) using the public parameters \(pp\). \(O_{reg}\) stores: (1) \textbf{Record}, a set of information used to trace true identities of all registered users; (2) the encryption public/private key pair \((pk_{reg}, sk_{reg})\). \(O_{reg}\) accepts different queries, which are described below:

\[ Q = (\text{judge}, proof_{id},proof_{id'}) \]

\(O_{reg}\) verifies whether \(proof_{id}\) and \(proof_{id'}\) belong to the same user, and sends the result to the inquirer. \(O_{reg}\) has a secret hash set \(P_{hash}\) for the inquirer, where \(P_{hash}\) includes the chameleon hash value of the user who generates \(proof_{id}\).

Oracles \(O_{id}\) and \(O_{reg}\) work together to ensure that the identity proof is indistinguishable from random data and unforgeable. The behavior of the honest user with identity \(id\) is realized by the oracle \(O_{id}\), and the behavior of the regulator is realized by the oracle \(O_{reg}\). We assume that the honest users and adversary in the experiment have already registered successfully in the regulator, i.e., they can generate any identity proof. Below, we describe how \(O_{id}\) and \(O_{reg}\) work.

Oracles \(O_{id}\) and \(O_{reg}\) are initialized by challenger \(C\) using the public parameters \(pp\). \(O_{reg}\) stores: (1) \textbf{Record}, a set of information used to trace true identities of all registered users; (2) the encryption public/private key pair \((pk_{reg}, sk_{reg})\). \(O_{reg}\) accepts different queries, which are described below:

\[ Q = (\text{judge}, proof_{id},proof_{id'}) \]

\(O_{reg}\) verifies whether \(proof_{id}\) and \(proof_{id'}\) belong to the same user, and sends the result to the inquirer. \(O_{reg}\) has a secret hash set \(P_{hash}\) for the inquirer, where \(P_{hash}\) includes the chameleon hash value of the user who generates \(proof_{id}\).

Identity proof indistinguishability and identity proof unforgeability. Every property is formalized as an experiment between an adversary \(A\) and a challenger \(C\). The behavior of the honest user with identity \(id\) is realized by the oracle \(O_{id}\), and the behavior of the regulator is realized by the oracle \(O_{reg}\). We assume that the honest users and adversary in the experiment have already registered successfully in the regulator, i.e., they can generate any identity proof. Below, we describe how \(O_{id}\) and \(O_{reg}\) work.

Oracles \(O_{id}\) and \(O_{reg}\) are initialized by challenger \(C\) using the public parameters \(pp\). \(O_{reg}\) stores: (1) \textbf{Record}, a set of information used to trace true identities of all registered users; (2) the encryption public/private key pair \((pk_{reg}, sk_{reg})\). \(O_{reg}\) accepts different queries, which are described below:

\[ Q = (\text{judge}, proof_{id},proof_{id'}) \]

\(O_{reg}\) verifies whether \(proof_{id}\) and \(proof_{id'}\) belong to the same user, and sends the result to the inquirer. \(O_{reg}\) has a secret hash set \(P_{hash}\) for the inquirer, where \(P_{hash}\) includes the chameleon hash value of the user who generates \(proof_{id}\).
then generates the public information pub. Finally, the oracle $O_{id}$ calls the Gen\textsubscript{proof} algorithm to generate the identity proof proof\textsubscript{id}, and sends $(pub, proof\textsubscript{id})$ to the inquirer.

- $(genid\textsubscript{proof}, priv)$. The adversary knows the private information $priv$, and the oracle $O_{id}$ uses the $priv$ selected by the adversary to generate the public information $pub$ and then calls the Gen\textsubscript{proof} algorithm to generate the identity proof $proof\textsubscript{id}$. Finally, $O_{id}$ sends $(pub, proof\textsubscript{id})$ to the inquirer.

- $(genid\textsubscript{proof}, pub)$. Here, $pub_i \in T_{pub}$, and $T_{pub} = \{pub_i\}_{i(1..n)}$ is the public information set of the user whose identity is $id$. The oracle $O_{id}$ calls the Gen\textsubscript{proof} algorithm to generate the identity proof $proof\textsubscript{id}$, and sends $(pub, proof\textsubscript{id})$ to the inquirer.

### A.1 Identity proof indistinguishability

This property is formalized by $Exp^{IDP-IND}_{\mathcal{A}_{\Pi}}(\lambda)$, which is shown below:

1. The challenger $C$ randomly samples $b \in \{0, 1\}$, gets $pp$ by running Setup($\lambda$), and sends $pp$ to adversary $\mathcal{A}$. Next, $\mathcal{A}$ initializes two separate oracles $O_{id}$ and $O_{reg}$.

2. At each query phase, the adversary $\mathcal{A}$ issues a pair of queries $(Q, Q')$, where $(Q, Q')$ is one of the following:

   - $(Q, Q') = (\text{genidproof, priv}, \text{genidproof, priv'})$, where priv $\neq priv$. $C$ forwards $Q$ to $O_{id}$, and forwards $Q'$ to $O_{reg}$. $C$ replies to $\mathcal{A}$ with ($(pub, proof\textsubscript{id}), (pub_{1-b}, proof\textsubscript{id-1-b})$), which is the two oracle answer.

   - $(Q, Q') = (\text{genidproof, proof}, \text{genidproof, proof'})$, where proof $\neq proof$. $C$ forwards $Q$ to $O_{id}$, and forwards $Q'$ to $O_{reg}$. $C$ replies to $\mathcal{A}$ with ($(pub, proof\textsubscript{id}), (pub, proof\textsubscript{id-1-b})$), which is the two oracle answer.

3. At the end of the query, $\mathcal{A}$ sends $C$ a guess $b' \in \{0, 1\}$. If $b = b'$, $C$ outputs $1$; otherwise, $C$ outputs $0$.

Identity proof indistinguishability requires that the adversary $\mathcal{A}$ wins the above experiment with only negligible probability. Next, we formally define this property.

**Definition A.1.** A traceable scheme $\Pi$ satisfies identity proof indistinguishability if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there is a negligible function $\negl(\cdot)$ such that

$$\text{Adv}^{IDP-IND}_{\mathcal{A}_{\Pi}} \leq \negl(\lambda), \quad (1)$$

where $\text{Adv}^{IDP-IND}_{\mathcal{A}_{\Pi}} = Pr[Exp^{IDP-IND}_{\mathcal{A}_{\Pi}}(\lambda) = 1] - 1/2$ is $\mathcal{A}$’s advantage in the experiment $Exp^{IDP-IND}_{\mathcal{A}_{\Pi}}(\lambda)$.

### A.2 Identity proof unforgeability

This property is formalized by $Exp^{IDP-UNF}_{\mathcal{A}_{\Pi}}(\lambda)$, which is shown below:

1. The challenger $C$ obtains $pp$ by running Setup($\lambda$), and sends $pp$ to adversary $\mathcal{A}$. Next, $\mathcal{A}$ initializes two separate oracles $O_{id}$ and $O_{reg}$. Let $T_{pub} = \{pub_1, \ldots, pub_n\}$ be the public information set for the user whose identity is $id$.

2. The adversary $\mathcal{A}$ issues queries $q_1, \ldots, q_m$, where $q_i$ is $(\text{genidproof, pub})$, and $pub_i \in T_{pub}$. $C$, forwards $Q$ to $O_{id}$, $C$ replies to $\mathcal{A}$ with $(pub_i, proof\textsubscript{id})$, which is the oracle $O_{id}$’s answer.

3. At the end of the query, let $P = \{proof_1, \ldots, proof_m\}$ is the identity proof set that is generated by $O_{id}$. $\mathcal{A}$ sends $(pub, proof) \rightarrow C$. $C$ checks as follows:

   - If $proof_{id} \notin P \land \text{Ver}_{\text{proof}}(pp, pub, proof) = 1$, $C$ proceeds as follows; otherwise it aborts.

   - $C$ sends $(\text{judge, proof\textsubscript{id}}, proof) \rightarrow O_{reg}$, where $i \in [1, m]$. If $proof\textsubscript{id}$ and proof\textsubscript{i} belong to the user whose the identity is $id$, $O_{reg}$ sends $c = 1 \rightarrow C$; otherwise it returns $c = 0$.

   - If $proof\textsubscript{id} \notin P \land \text{Ver}_{\text{proof}}(pp, pub, proof) = 1 \land c = 1$, $C$ outputs $1$; otherwise, $C$ outputs $0$.

The adversary $\mathcal{A}$ wins the above experiment if $proof\textsubscript{id}$ such that: (i) $proof\textsubscript{id} \notin P$; (ii) $\text{Ver}_{\text{proof}}(pp, pub, proof) = 1$; (iii) $proof\textsubscript{id}$ belongs to the user whose identity is $id$. In other words, $\mathcal{A}$ can forge the identity proof of honest parties. Identity proof unforgeability requires that the adversary wins the above experiment only with negligible probability. Next, we formally define this property.

**Definition A.2.** A traceable scheme $\Pi$ satisfies identity proof unforgeability if for all probabilistic polynomial-time adversaries $\mathcal{A}$, there is a negligible function $\negl(\cdot)$ such that

$$\text{Adv}^{IDP-UNF}_{\mathcal{A}_{\Pi}} \leq \negl(\lambda), \quad (2)$$

where $\text{Adv}^{IDP-UNF}_{\mathcal{A}_{\Pi}} = Pr[Exp^{IDP-UNF}_{\mathcal{A}_{\Pi}}(\lambda) = 1] - 1/2$ is $\mathcal{A}$’s advantage in the experiment $Exp^{IDP-UNF}_{\mathcal{A}_{\Pi}}(\lambda)$.

### B PROOF OF THEOREM 5.1

#### B.1 Proof of identity proof indistinguishability

**Theorem B.1.** Assuming that the NIZK scheme is perfect zero-knowledge and proof of knowledge, the encryption scheme Enc satisfies IND-CCA2 security, and gen($\cdot$) is a one-way function, then our scheme $\Pi$ described in Algorithm 1 satisfies identity proof indistinguishability.

We prove the Theorem B.1 through a sequence of hybrid experiments. Let $q_m$ be the number of queries issued by the adversary $\mathcal{A}$.

**exp\textsubscript{real}.** The experiment $exp\textsubscript{real}$ is the same as the experiment $Exp^{IDP-IND}_{\mathcal{A}_{\Pi}}(\lambda)$.

**exp\textsubscript{1}.** This experiment is the same as the experiment $exp\textsubscript{real}$ except that the challenger $C$ simulates the NIZK. More precisely, $C$ calls a polynomial-time simulator $S_{\text{nizk}}(\lambda, AC_{\text{proof}})$ to obtain $(pk_{\text{proof}}, pk_{\text{proof}}, \text{tra})$, where $\text{tra}$ is the trapdoor, instead of invoking $K_{\text{nizk}}(\lambda, AC_{\text{proof}})$. When an oracle $O_{id}$ sends an identity proof to $C$, $C$ replaces the real zk-SNARK proof with a simulated zk-SNARK proof by invoking $S_{\text{nizk}}(pk_{\text{proof}}, x_{\text{proof}}, \text{tra})$, without using the witness.

Because the NIZK scheme is perfect zero-knowledge, the distribution of the simulated zk-SNARK proof is identical to that of the proof computed in $exp\textsubscript{real}$. Therefore, $Adv_{exp\textsubscript{real}} = Adv_{exp\textsubscript{1}}$.

**exp\textsubscript{2}.** This experiment is the same as the experiment $exp\textsubscript{1}$ except when $Q$ and $Q'$ sent by the adversary are both genidproof queries, the challenger $C$ replaces the public information by a random number. Because gen($\cdot$) is a one-way function, then no polynomial-time adversary can distinguish $exp\textsubscript{1}$ and $exp\textsubscript{2}$ except with negligible probability.
The experiment \( \exp_{\text{final}} \) is the same as the experiment \( \exp_2 \) except that the challenger \( C \) replaces \( C_{id} \) in \( \text{proof}_{id} \) by encrypting a random string. More precisely, when an oracle \( O_{id} \) sends an identity proof \( \text{proof}_{id} \) to \( C \), \( C \) replaces the \( C_{id} \) with a \( C_{id} \) generated by \( E_{\text{enc}}(pk_{\text{reg}}, r, m) \), where \( r \) is a random string sampled uniformly from the plaintext space of the encryption scheme. Because the responses to the adversary \( A \) in \( \exp_{\text{final}} \) are independent of the bit \( b \). Therefore, \( Adv_{\exp_{\text{final}}} = 0 \) in the experiment \( \exp_{\text{final}} \).

Next, we prove that no polynomial-time adversary can distinguish \( \exp_2 \) from \( \exp_{\text{final}} \) except with negligible probability (see below lemma).

**Lemma 1.** After \( q_m \) queries, \( |Adv_{\exp_{\text{final}}} - Adv_{\exp_2}| \leq q_m \cdot Adv_{\text{enc}}, \) where \( Adv_{\text{enc}} \) denotes the adversary’s advantage in the IND-CCA2 experiment.

**Proof sketch.** We construct an algorithm \( B \), using \( A \) as a subroutine, to win the IND-CCA2 experiment.

Let \( e = Adv_{\exp_{\text{final}}} - Adv_{\exp_2} \). For some \( i \in \{1, ..., q_m\} \), when \( A \) issues an \( i \)-th query, \( B \) uses the same method as \( \exp_2 \) to generate \( \text{proof}_{id} \) except for the ciphertext \( C_{id} \) generation method. \( B \) chooses a random string \( r \) that has the same length as plaintext \( m \) corresponding to \( C_{id} \). \( B \) sends \( (m_0, m_1) = (m, r) \) to the IND-CCA2 challenger and receives \( C^* = E_{\text{enc}}(pk_{\text{reg}}, m_0, m_1) \), where \( b^* \) is the bit chosen by the IND-CCA2 challenger. \( B \) replaces \( C_{id} \) included in \( \text{proof}_{id} \) with \( C^* \). \( B \) returns \( b^* \), which \( A \) outputs as the guess in the IND-CCA2 experiment. We know that when \( B = 0 \), \( A \)'s view of the interaction is distributed identically to that of \( \exp_2 \). And when \( B = 1 \), \( A \)'s view represents the \( \exp_2 \) in which one ciphertext \( C_{id} \) has been replaced. Based on a standard hybrid argument over each of the \( q_m \) ciphertexts, we can conclude that over the randomness of the experiment, \( B \) must succeed in the IND-CCA2 experiment with the advantage of at least \( e/q_m \). Therefore, \( |Adv_{\exp_{\text{final}}} - Adv_{\exp_2}| \leq q_m \cdot Adv_{\text{enc}} \).

**B.2 Proof of identity proof unforgeability**

**Theorem B.2.** Assuming that the Chash scheme is collision resistant, trapdoor collision and semantic security, the NIZK scheme is perfect zero-knowledge and proof of knowledge, and \( \text{gen}() \) is a one-way function, then, our scheme \( \Pi \) described in Algorithm 1 satisfies identity proof unforgeability.

From experiment \( Exp^{\text{DP-UNF}}(\lambda) \), we can observe that \( A \) succeeds only if it outputs \( (pub^*, \text{proof}_{id}^*) \) such that: (i) \( \text{proof}_{id}^* \notin P \); (ii) \( \text{Ver}_{\text{proof}}(pp, pub^*, \text{proof}_{id}^*) = 1 \); (iii) \( \text{proof}_{id}^* \) belongs to the user whose identity is \( id \). We define the two disjoint events which \( A \) succeeds: (i) \( \text{Event} \), \( A \) succeeds, and \( pub^* \notin T_{pub} \); (ii) \( \overline{\text{Event}} \), \( A \) succeeds, and \( pub^* \in T_{pub} \).

Obviously, \( Adv_{\text{DP-UNF}}^A = Pr[\text{Event}] + Pr[\overline{\text{Event}}] \). Define \( e_1 = Pr[\text{Event}] \) and \( e_2 = Pr[\overline{\text{Event}}] \).

When \( \text{Event} \) occurs, we construct the algorithm \( B \). It uses \( A \) as a subroutine, and solves the one-wayness of \( \text{gen}() \). Let \( \varepsilon \) be the NIZK extractor for \( A \). The algorithm \( B \) works as follows.

1. \( B \) randomly selects \( i \in \{1, ..., n\} \).
2. \( B \) performs the experiment \( Exp^{\text{DP-UNF}}(\lambda) \) with \( A \) to obtain \( (pub^*, \text{proof}_{id}^*) \).
3. \( B \) runs the \( \varepsilon(\text{ok}_{\text{proof}}, \pi_{\text{proof}}^*) \) to obtain \( w_{\text{proof}} = (\text{path}_{id}^*, CH_{id}^{*}, pk_{\text{chash}}^{*} \cdot sk_{\text{chash}}^{*} \cdot \text{prior}^{*} \cdot r^* \cdot r_n^* \).
4. If \( pub^* = pub_i \), then \( B \) outputs \( \text{prior}^* \); otherwise, \( B \) aborts.

Because the index \( i \) is selected at random, \( B \) succeeds with probability \( e_1/n \). Because of the one-wayness of the \( \text{gen}() \), \( e_1 \) must be negligible in \( \lambda \).

When \( \overline{\text{Event}} \) occurs, we construct the algorithm \( Z \). It uses \( A \) as a subroutine and finds collision for the chameleon hash scheme. \( Z \) sends \( (\text{chash}_{set}, \text{proof}_{id}) \) to \( O_{reg} \), and obtains \( P_{\text{chash}} = \{CH_{id}, ..., CH_{id}'\} \) from the oracle \( O_{reg} \), where \( k = \lambda \). The set \( P_{\text{chash}} \) includes the chameleon hash \( CH_{id} \) of the user whose identity is \( id \). The algorithm \( Z \) performs as follows.

1. \( Z \) randomly selects \( i \in \{1, ..., k\} \).
2. \( Z \) performs the experiment \( Exp^{\text{DP-UNF}}(\lambda) \) with \( A \) to obtain \( (pub^*, \text{proof}_{id}^*) \).
3. \( Z \) runs the \( \varepsilon(\text{ok}_{\text{proof}}, \pi_{\text{proof}}^*) \) to obtain \( w_{\text{proof}} = (\text{path}_{id}^*, CH_{id}^{*}, pk_{\text{chash}}^{*} \cdot sk_{\text{chash}}^{*} \cdot \text{prior}^{*} \cdot r^* \cdot r_n^* \).
4. If \( CH_{id}^{*} = CH_{id}' \), then \( Z \) outputs \( \text{prior}^* \); otherwise, \( Z \) aborts.

Because the index \( i \) is selected at random, \( Z \) succeeds with probability \( e_2/k \). Furthermore, because of the collision resistance of the chameleon hash scheme, \( e_2 \) must be negligible in \( \lambda \).