Part I Aleo Technical Introduction

1. Aleo Overview

Blockchain is a distributed system where data is transparent and open for anyone to view, exposing users' private information. Protecting the privacy of data has become an urgent issue in blockchain. Zerocash, a privacy transaction protocol based on Bitcoin's UTXO model, and Zether, a privacy transaction protocol based on Ethereum's account model, use zero-knowledge proof technology to some extent to protect the privacy of transaction data, achieving complete concealment of transaction senders, receivers, and amounts. However, these protocols fall short of providing privacy at the smart contract/function level.

Aleo is a privacy-focused blockchain platform dedicated to offering highly privacy-protected smart contracts and decentralized applications. By utilizing cryptographic technologies such as zero-knowledge proofs, Aleo allows users to verify the validity and integrity of private data without revealing the original data. It achieves off-chain zero-knowledge proof generation and efficient verification of zero-knowledge proofs (transaction simplicity) on the blockchain.

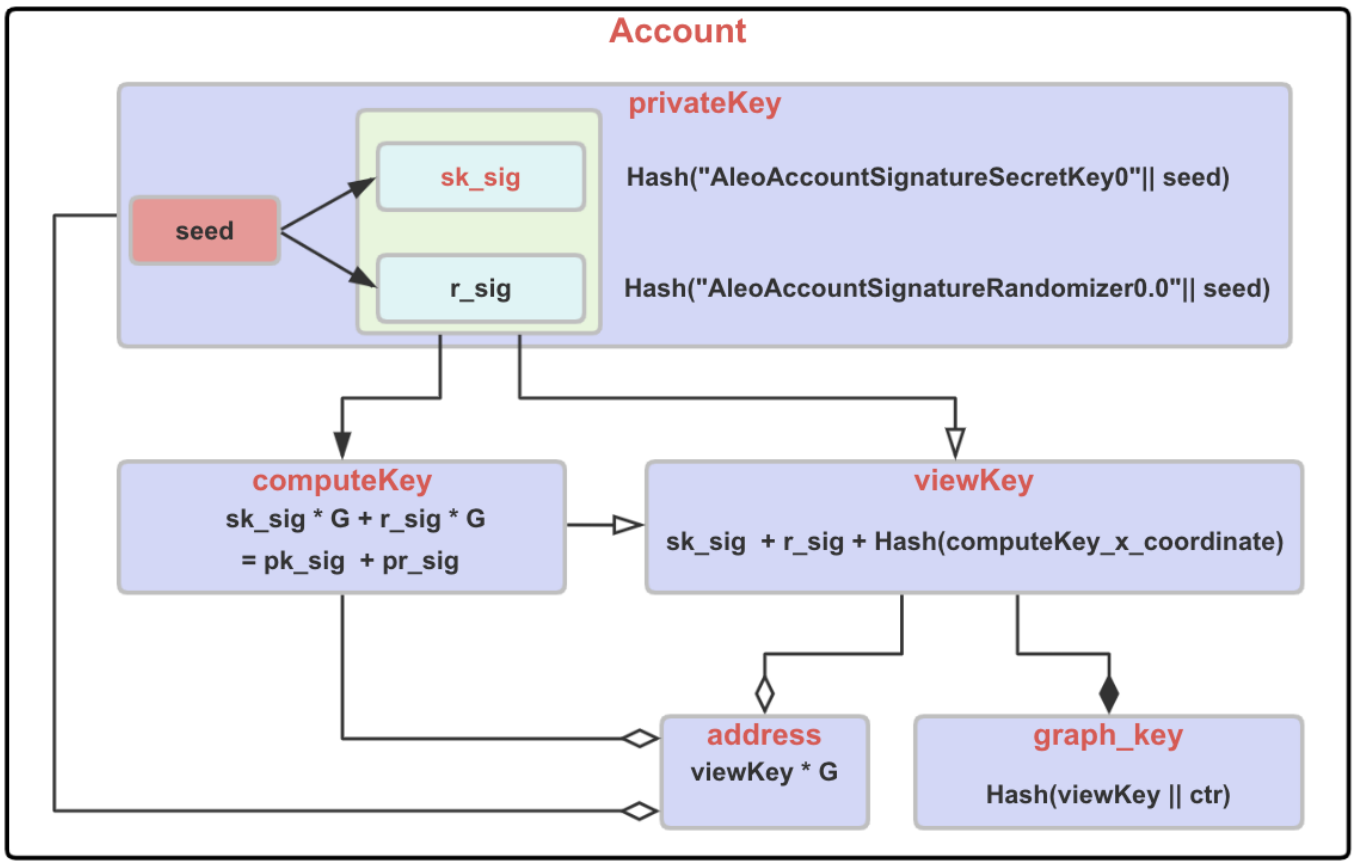
Aleo mainly consists of the following major modules:

- Smart Contracts: Similar to platforms like Ethereum, Aleo supports the development and execution of smart contracts. The key difference lies in Aleo's emphasis on providing a higher level of privacy.
- Leo Language: Aleo introduces a programming language called Leo, which is designed specifically for privacy protection. The Leo language enables developers to create smart contracts that support privacy while ensuring data integrity.
- Aleo Instructions: Aleo provides a set of instructions, offering developers an easy-to-use environment to write programs supporting privacy and integrity.

2. Account Model

Similar to the Ethereum account, Aleo's account manages the user's private key, from which the public key(address) can be derived and publicly disclosed. Any user can transact with others using their public key. Through the private key, users can access and control their funds. Therefore, the security of the private key is crucial; once lost, control over the account's funds is forfeited.

The account is responsible for signature and encryption processes. Aleo's account mainly includes privateKey, viewKey, and address. The following diagram illustrates the relationship between account keys.



2.1 privateKey

Aleo's privateKey represents the user's unique identity and plays a crucial role in:

- Deriving the public key
- Participating in transaction signing, i.e., authorizing the execution of a transaction with a zero-knowledge proof
- Engaging in certain cryptographic algorithm operations.

The privateKey mainly consists of three parts:

- seed: A randomly generated number from the domain. Its length depends on the domain size, usually 256 bits. It is a crucial seed for the private key.
- sk_{sig} : Used for signing transaction information (Schnorr Signature) and participating in address calculation.
  - `hash_to_scalar_psd2(EncodeToField("AleoAccountSignatureSecretKey0"), seed)`

- r_{sig} : Temporary private key (random number) randomizes sk_{sig} . The design concept is that the viewKey serves as the decryption key for the Record ciphertext. Even with knowledge of the viewKey, no information about the private key sk_{sig} can be obtained.
- `hash_to_scalar_psd2(EncodeToF("AleoAccountSignatureRandomizer0.0"), seed)`

2.2 computeKey

The computeKey is calculated from the privateKey using the generator of the elliptic curve group, denoted by G. This computeKey is a component of the signature structure and is used to generate and verify signatures for transactions made by the user. For more information, please refer to section 2.6.

$$sk_{sig} \cdot G + r_{sig} \cdot G) = (pk_{sig} + pr_{sig})$$

2.3 viewKey

The viewKey of an Aleo account is derived from its privateKey and is primarily used for encrypting and decrypting Aleo blockchain transaction input/output data. Refer to the Record model section for specific details.

$$sk_{sig} + r_{sig} + hash\_to\_scalar\_psd4(pk_{sig}|_x + pr_{sig}|_x)$$

2.4 Address

In Aleo, the address is the only public identity of a user. It is used to receive and send official tokens (Aleo credits) and interact with other users or applications. There are several ways to calculate the address:

- privateKey -> address: This method is self-explanatory and will not be elaborated.
- computeKey -> address: $Hash(\text{computeKey}|_x) + \text{computeKey}$
- viewKey -> address: $\text{viewKey} * G$

2.5 graphKey

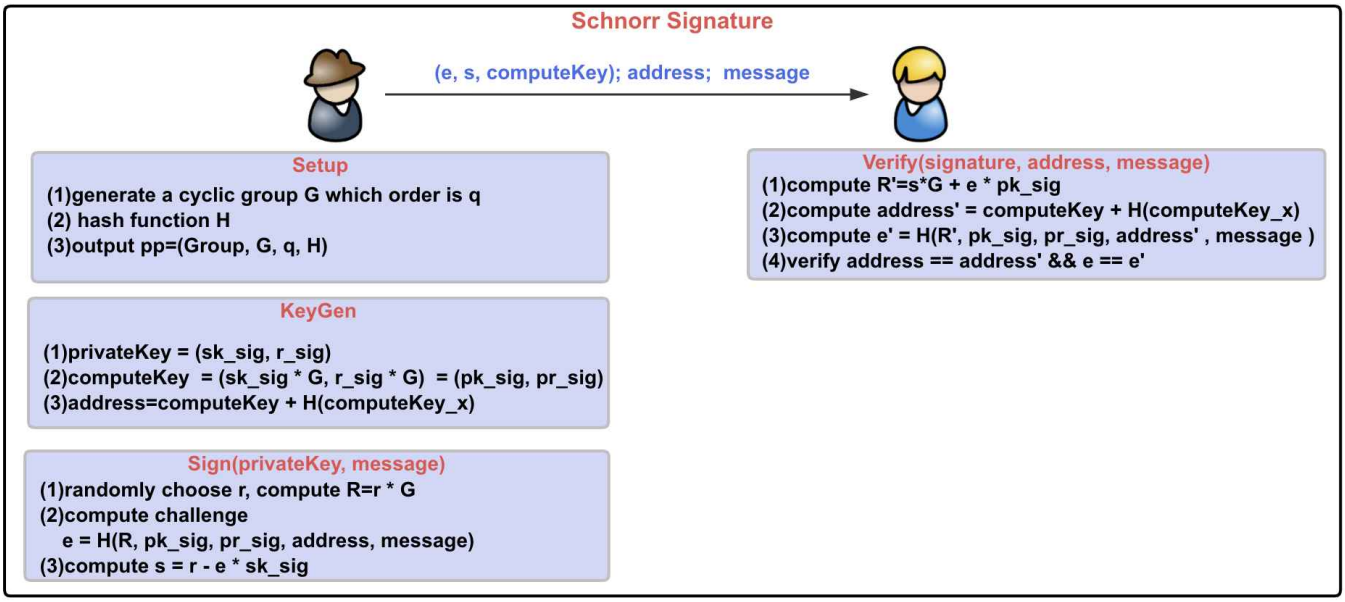
The graphKey is a value calculated from the viewKey. Its purpose is to generate a commitment/tcm (transaction commitment) tag for transactions. This tag is involved in generating the commit by privacy input/output, record.private. The tag is used to quickly verify the accuracy of the commit to the record.

> tag = Hash(graphKey, commit(record{owner, data, nonce}))

- `graphKey = sk_tag = hash_psd4("AleoGraphKey0",view_key, Field::zero())`

2.6 Signature Algorithm

Interaction between users requires confirmation of the sender's identity. The role of the signature is to confirm the sender's identity and the integrity of the message content. Therefore, Aleo introduces the Schnorr signature algorithm for signing transactions.



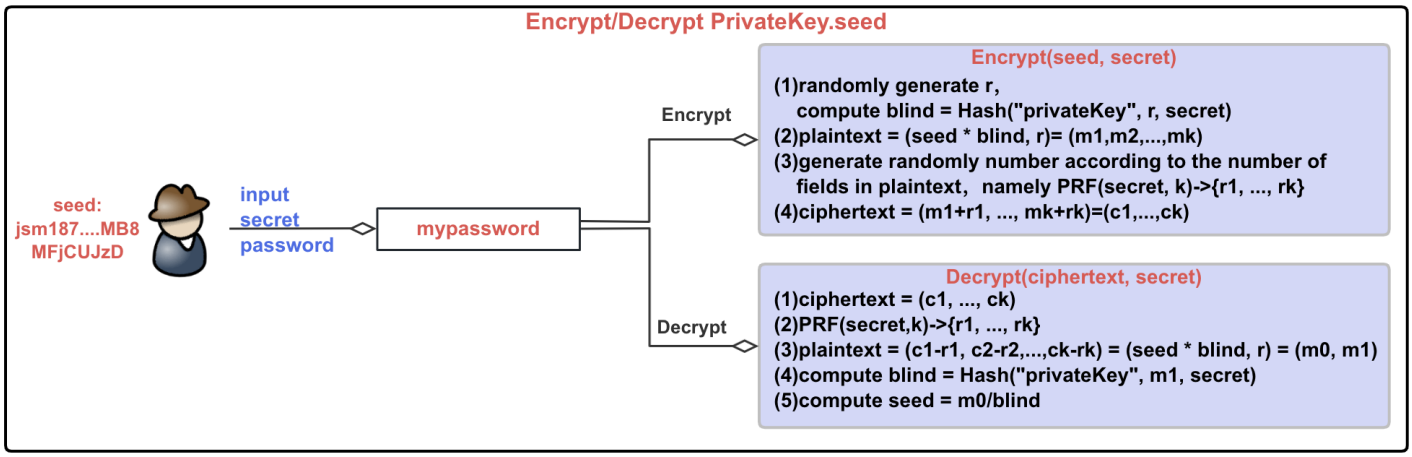
The signature algorithm typically consists of four primary algorithms:

- $Setup(1^\lambda) \rightarrow pp_{sig}$: Primarily generates public parameters, including group construction parameters and cryptographic functions like hash functions.
- $KeyGen(pp_{sig}) \rightarrow (sk, pk)$: Key generation, including public and private keys. In Aleo, due to its account construction, its key generation differs slightly from the Schnorr signature key generation.
  - `sk=privateKey, pk= (computeKey, address)`
- $Sign(sk, message) \rightarrow \sigma$: Signature algorithm. Aleo uses sk_{sig} from the privateKey to sign the message $\sigma = (s, e)$, and the signature algorithm aligns with the Schnorr signature algorithm.
- $Verify(\sigma, pk, message) \rightarrow b$: Verification algorithm, verifying the legitimacy of the signature.

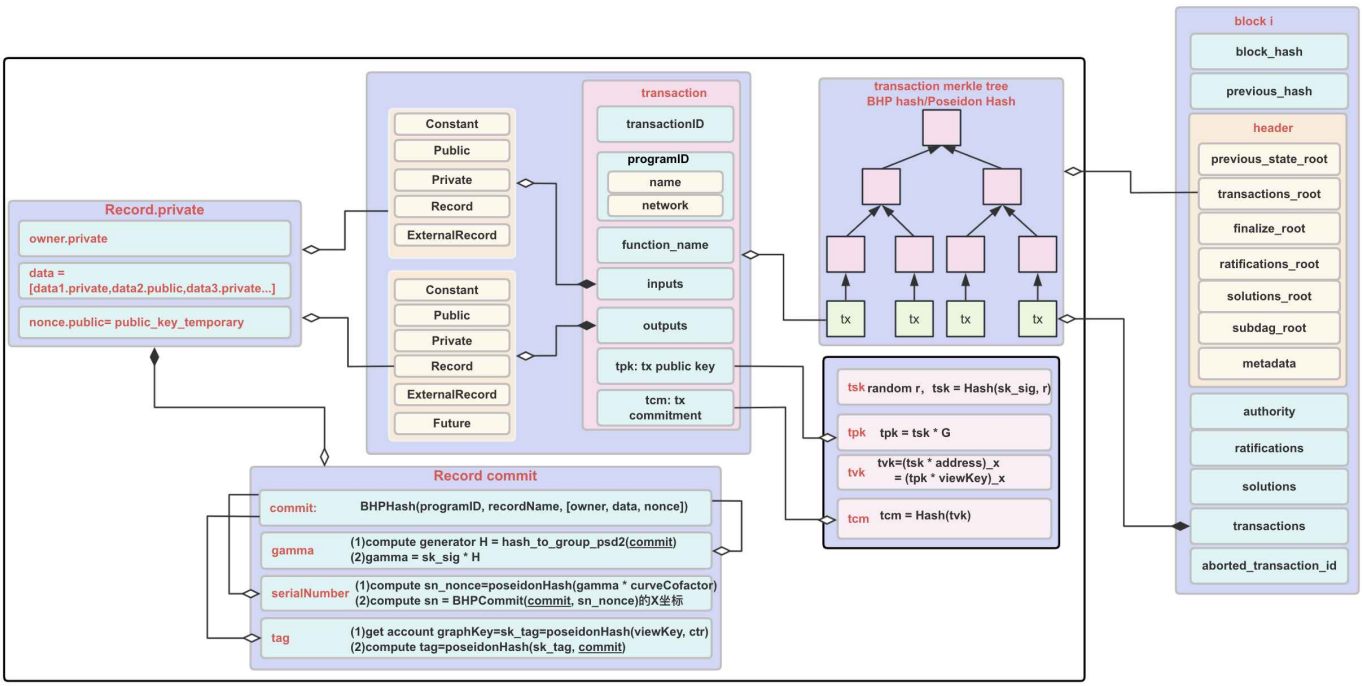
2.7 Encryption Storage of Private Key

Similar to mnemonic words, the private key is of utmost importance to users, granting absolute control over the account. Once lost, recovery becomes impossible. Based on Aleo's private key derivation algorithm, the seed is identified as crucial. The core of the privateKey lies in this random seed; with possession of the seed, one can fully generate the account's sk_{sig}, r_{sig} .

Typically, when creating an account, the generated private key is stored locally. For security reasons, the private key undergoes encryption during storage. The specific algorithm is illustrated in the diagram. The core idea is to introduce randomness to conceal/blind the seed, then employ a PRF to generate multiple random numbers deterministically. Each element in plaintext fields is further subjected to randomization/encryption processing.



3. Record Model



In a transaction, there are many inputs and outputs, and both inputs and outputs can be of the record type. The record is private by default. The Aleo record is similar to UTXO. If a record is used as an input to a function, it implies that the record will be consumed and cannot be used again, i.e., its unique identifier serial number (sn) is disclosed. In most cases, a function's output will create a new record.

3.1 Plaintext Fields of Record

Record is a crucial data structure in Aleo used to encode user assets and status data. It includes the following three fields:

- owner: The owner of the record.
- data: The data of the program.
- nonce: A unique identifier for a record, ensuring uniqueness for each record. Therefore, nonce values are unique.

An example is illustrated in the diagram below:

LEO CODE ALEO INSTRUCTIONS

```
program token.aleo;

record token:
owner as address.private;
token_amount as u64.private;

function compute:
input r0 as token.record;
add r0.token_amount r0.token_amount into r1;
output r1 as u64.private;

```

```
Record::from_str(&format!("{{
        owner: {caller}.private,
        token_amount: 100u64.private,
        _nonce: 0group.public}}"
```

3.2 Encrypted Fields of Record

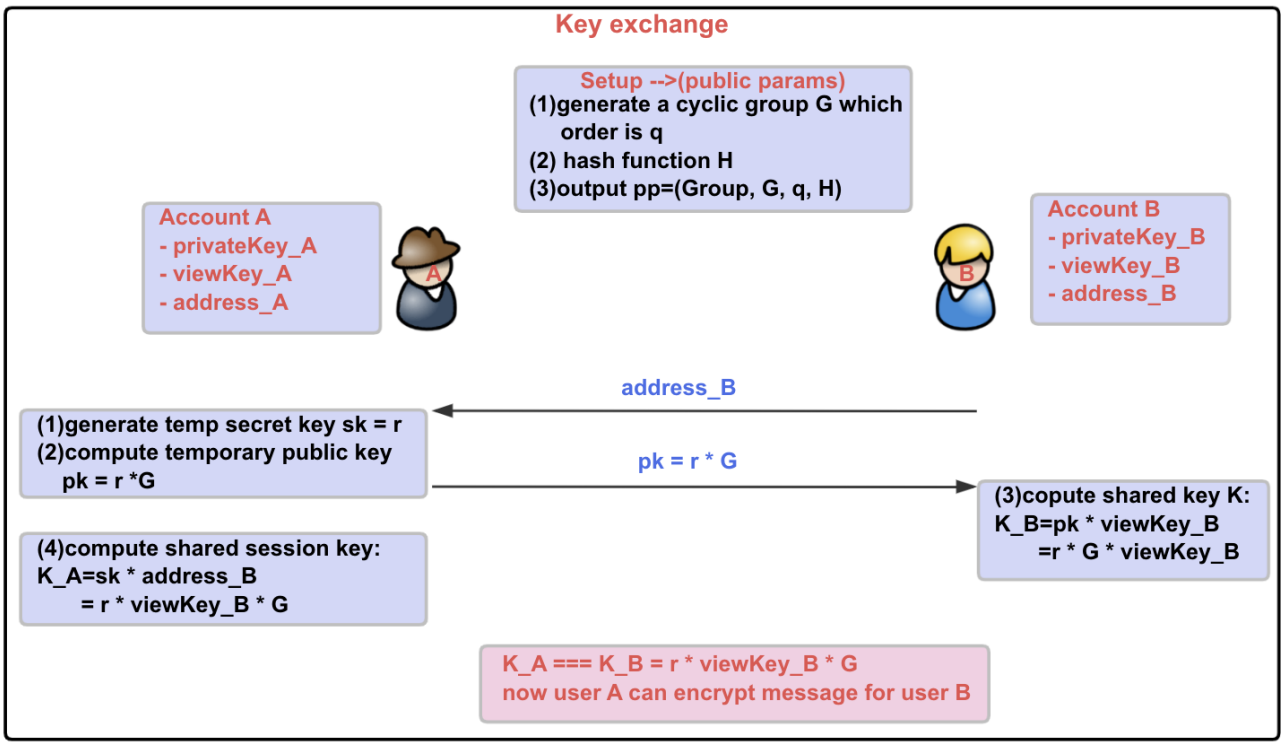
In Aleo, records are consumed and created through state functions, and a transaction contains multiple state transition functions. By default, records are private, meaning all fields within the record are encrypted. Before introducing the encryption algorithm for records, understanding the key-sharing protocol is essential.

3.2.1 Key-Sharing Protocol

The following describes how User A generates information that only User B can decrypt:

- User A obtains the public address of message recipient B (publicly available).
- User A generates temporary private and public keys and shares the public key with User B.
- User B can calculate a shared key, K_B.
- User A can also calculate a shared key, K_A.

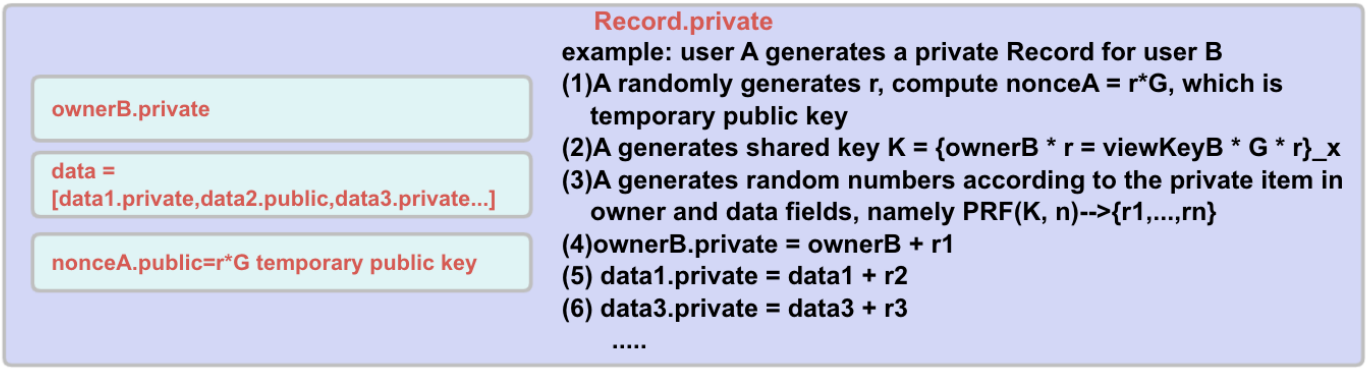
Calculation reveals that K_A is equal to K_B. Thus, User A and User B do not disclose any privacy information during their interaction but achieve a shared session key. This is crucial for record encryption.



3.2.2 Record Encryption

Assuming User A wants to create an encrypted Record for User B, using the key-sharing protocol mentioned earlier:

- User A generates a temporary public key for User B, which is used as a nonce and made public.
- User A generates a shared key, K, and uses it as an input for a pseudorandom function (PRF) to generate multiple random values.
- The random values generated by the PRF are used to randomize/encrypt the private fields in the Record.

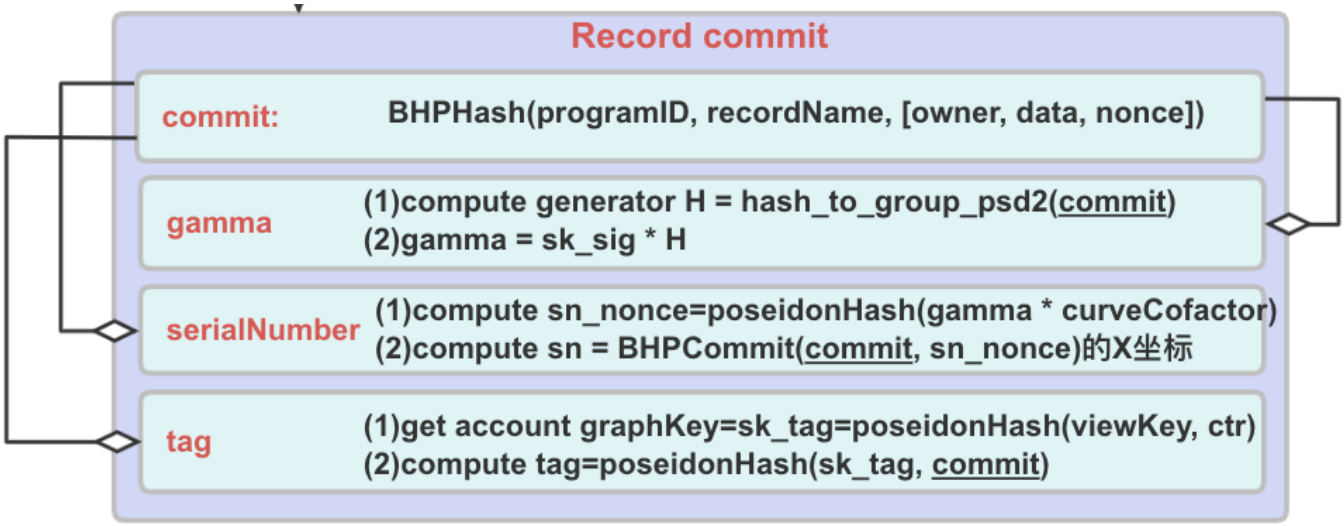


Once this Record is signed and added to the blockchain as an output of a transaction, it remains in ciphertext form. Only the owner, User B, can decrypt and view the data.

- User B uses his viewKey and the nonce value from the Record to calculate the shared key, K.
- By using PRF(K), User B can generate multiple random values and decrypt the original plaintext data of the Record.

3.3 Commitment of Record

During transaction signing, all inputs and outputs, categorized as constant, public, private, or Record types, undergo processing as the message for the signature. For Record types, both inputs and outputs go through commit processing, resulting in the generation of commit, gamma, serialNumber, and tag.



4. Aleo Transactions

This section provides a brief introduction to several transaction types in Aleo's credits.aleo. Before delving into transactions, we'll briefly introduce some Aleo Programs.

4.1 Program

In Aleo, zero-knowledge programs can be written in two languages:

- Leo: Similar to Ethereum's Solidity, Leo is a high-level language for developing various privacy applications in Aleo. Programs are then compiled into Aleo.VM-readable instructions (Aleo instructions/bytecode) using a compiler.
- Aleo Instructions: Aleo Instructions is a statically typed language used to write private applications. Through zero-knowledge proof technology, Aleo achieves off-chain computation and on-chain verification of computation consistency.

```
// Leo language
program helloworld.aleo {
  transition hello(public a: u32, b: u32) -> u32 {
      let c: u32 = a + b;
      return c;
  }
}

// The Leo code above compiles to the following Aleo instructions
program helloworld.aleo;
function hello:
    input r0 as u32.public;
    input r1 as u32.private;
    add r0 r1 into r2;
    output r2 as u32.private;
```

Programs offer:

- Extensibility: Users can define their programs freely.
- Isolation: Malicious programs won't affect honest programs.
- Inter-process communication: Program functions can interact.

4.2 Overview of Aleo's ZK Protocols

In recent years, zero-knowledge proofs have been widely applied to blockchains, becoming a crucial tool for privacy and scalability. They find applications in verifiable outsourced computation, anonymous credentials, range proofs, privacy-preserving cryptocurrencies, and extensive use of Layer 2 solutions. As client transactions are frequent, deploying efficient and practical zero-knowledge proof protocols requires proofs that are sufficiently small and verification that is highly efficient (small proof size and fast verification).

Aleo achieves privacy features through the use of zero-knowledge proof technology, primarily utilizing the Marlin protocol, a specific instance of the zk-SNARKs protocol. Based on whether there is a trusted setup, zk-SNARKs constructions can be briefly categorized into the following types:

- Trusted-Setup: Represented by Groth16, it uses the KZG polynomial commitment scheme. This scheme introduces a trusted CRS that needs to be regenerated for different application scenarios. It is more suitable for a single immutable application scenario. Zcash, for example, uses this technology to implement privacy transaction proofs based on the UTXO model.
- Updatable SRS: Represented by Plonk, this protocol is currently the most widely used. Essentially, it also requires a trusted public reference string setup. However, the global public parameters only need to be generated once, and they can be updated for different application scenarios. Therefore, it can be applied to various scenarios.
- No Trusted Setup: The typical protocol represented by Bulletproofs does not require a trusted parameter setup. However, its proof generation and verification are relatively slower than the two mentioned above.

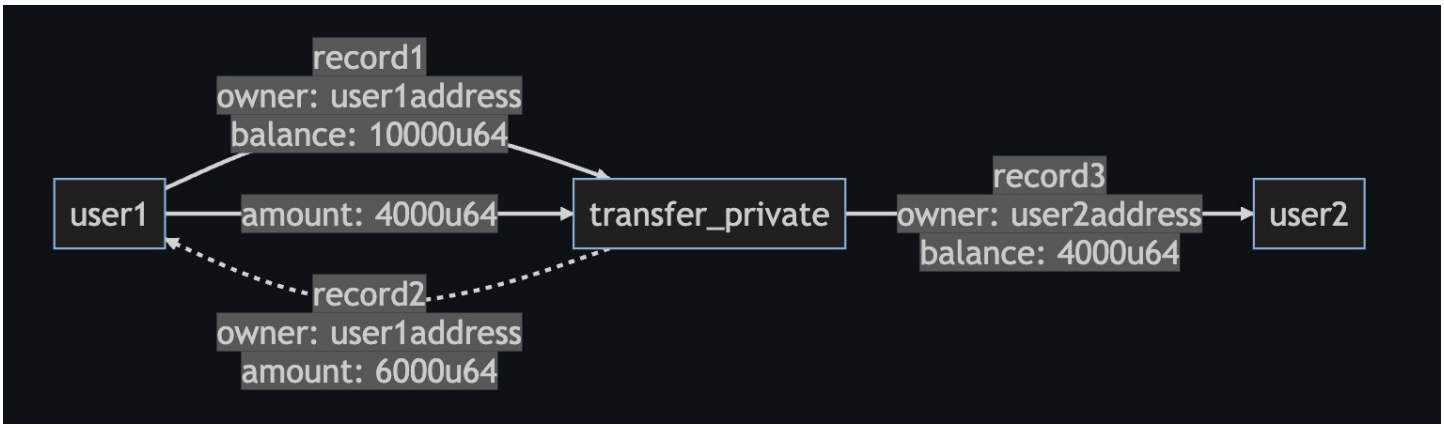
	Protocols	Assumption	Proof Size	Prover	Verifier	Practical
—	∑-Protocol	Discrete-Log	Long	Linear	Linear	Yes
CRS	Groth16	Discrete-Log	Short	FFT	Efficient	Yes
No Setup	STARK	OWF	Shortish	FFT	Efficient	Not Quite
	Bulletproof	Discrete-Log	Short	Linear	Linear	Yes
SRS	Plonk	Discrete-Log	Short	FFT	Linear	Not Quite

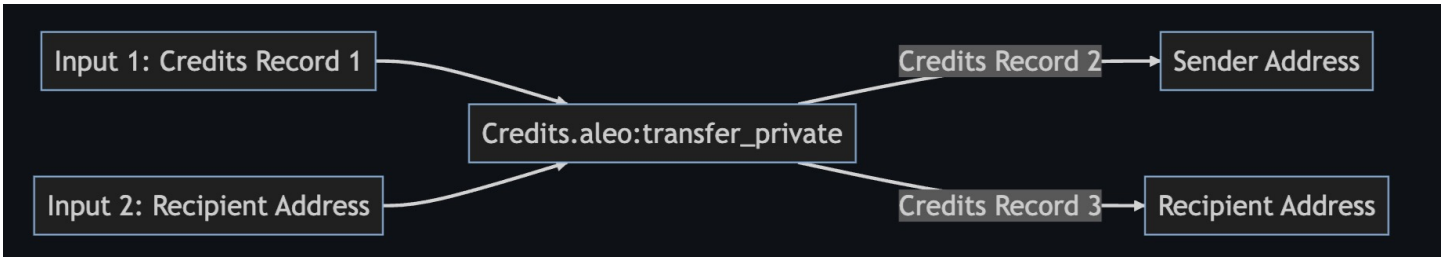
Aleo uses the Marlin protocol, and its difference from Plonk lies in Marlin's arithmetic language being R1CS, while Plonk is plonkish (custom gate + lookup). Additionally, Aleo implements a layer of recursive zero-knowledge proofs for detailed protocols, to be elaborated later.

4.3 Aleo Credit Transactions

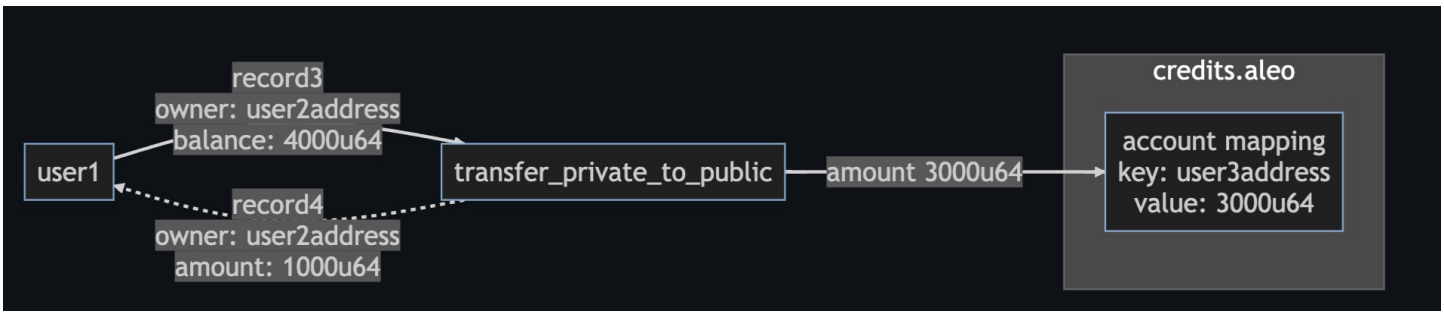
Aleo deployed the native token contract credits.aleo. This contract implements privacy transfer functionality. The contract mainly defines two data structures: record credits and mapping accounts. Record credits are used to record private states, while mapping accounts are used to record public states. The contract supports the following types of transactions:

- transfer_private: Implements a completely private transfer transaction between the sender and receiver. This involves mutual transfers using private records. For example, User 1 consumes two records, record 1 and record 2 (similar to UTXO), and generates a new record 3 for User 2. The encryption method for records is detailed in the previous Section 3.

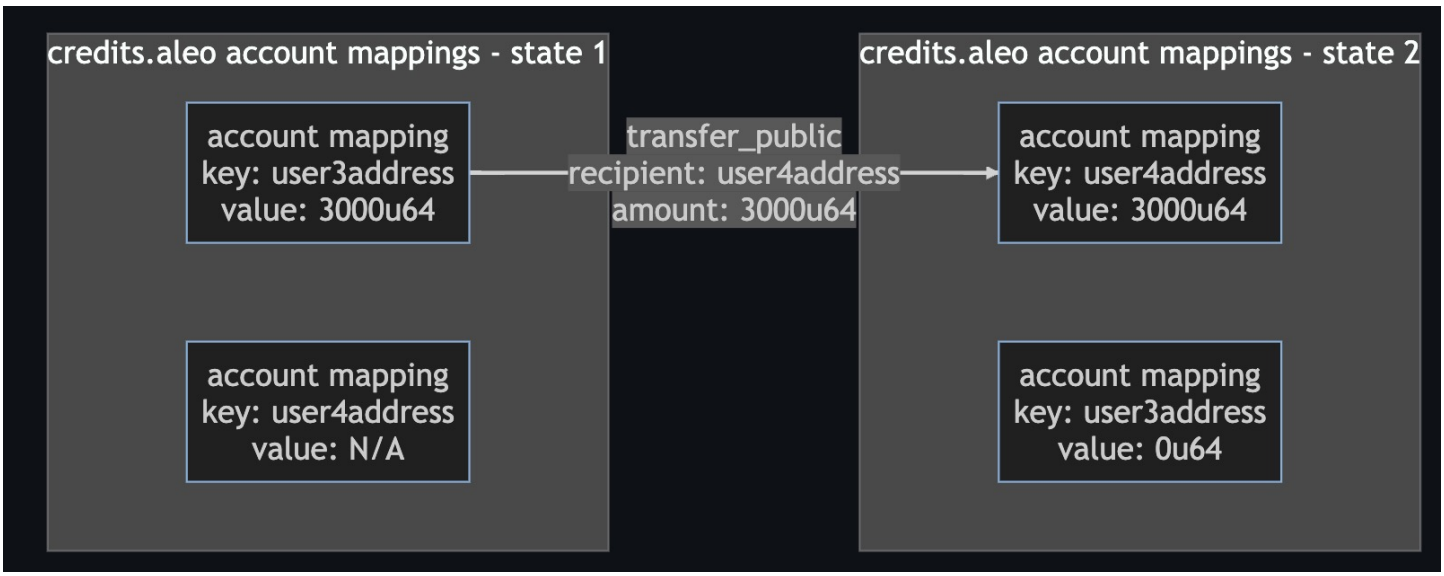




- transfer_private_to_public: Implements the functionality of a private sender transferring funds to a public receiver. For example, User1 consumes two records, record3, and record4, and initiates a transfer to the public user3. Since user3 is public, the transaction will record the state of user3 using a mapping account.



- transfer_public: Represents a completely open and transparent transaction.



- transfer_public_to_private: Implements the functionality of a public sender transferring funds to a private receiver.


credits.aleo account mappings - state 1
account mapping
key: user5address
value: 3000u64
recipient: user6address
amount: 3000u64
transfer_public_to_private
record5
owner: user6address
amount: 3000u64
user6
credits.aleo account mappings - state 2
account mapping
key: user5address
value: 0u64