Part 1: `SPL ZK-Tokens` Protocol Overview

Solana Labs

Last Updated: December 17, 2021

1 Privacy on Solana

Blockchains and smart contract platforms have gained a considerable amount of traction in the last few years, transforming the way we reason about trust in computation. As these technologies are gaining mainstream adoption, blockchains are starting to store an increasing amount of sensitive user data. A number of advanced cryptographic techniques such as zero-knowledge proof systems have been proposed to enhance privacy on public blockchains. Unfortunatley, most of these technologies require heavy on-chain computation, making them slow and expensive to run on many existing blockchains. Solana, however, is well-positioned to adopt and process many of these privacy-enhancing technologies with industry-leading speed, capacity, and cheap costs.

As an L1 blockchain, Solana can theoretically serve as the compute layer for any existing cryptographic protocol. Currently, the Solana runtime constraints do make it difficult to process certain cryptographic operations on chain. As our software matures, we hope to incrementally ease many of these constraints, enabling a wide variety of cryptographic operations possible to compute on chain. In parallel, we are currently building privacy features into Solana's `SPL` tokens themselves,[1] facilitating the development of privacy-enhancing applications on Solana. In this document, we provide the technical details on the privacy protocol that we are currently building to incorporate into the `SPL` token API.

1.1 Confidential Payments

By a private payment system, one generally refers to a payment system with two main properties:

- **Confidentiality**: This property refers to the privacy of the *amount* associated with a transaction. Suppose that Alice initiates a confidential transfer of 10 tokens from her account to Bob's account. Such a transfer may reveal that some transfer did take place between Alice and Bob, but it does not reveal that 10 tokens were transfered between the two parties.

- **Anonymity**: This property refers to the privacy of the *identities* of the sender and receiver of a transaction. Using the same example as above, an anonymous transfer of 10 tokens between Alice and Bob may reveal that a transfer of 10 tokens have taken place, but it does not reveal that Alice and Bob were the two parties involved in such a transfer.

[1] `SPL` tokens are Solana's analogue of Ethereum's `ERC20` tokens.

Our current focus at Solana Labs is enabling *confidential* transactions of `SPL` tokens. For this, we are developing the `ZK-Token` program that extends the regular `SPL Token` program to enable confidential transfer of funds.

SPL ZK-Tokens. The `ZK-Token` program is based on a protocol that was developed in-house at Solana and was designed to interoperate with the existing `SPL Token` program. `ZK-Tokens` can be enabled for any existing `Token` mint. Once enabled, a `ZK-Token` account can be created from any `Token` account: the `ZK-Token` account address is derived as a PDA from the linked `Token` account and mint. Once the accounts are created, funds can be transferred between regular `Token` accounts and `ZK-Token` accounts. Between `ZK-Token` accounts, funds can be transferred confidentially.[2]

Regarding the underlying cryptography, Solana Labs designed the `ZK-Token` protocol to serve as a thin layer of encryption on top of `SPL` tokens. It relies on two lightweight cryptographic objects: an additively-homomorphic encryption scheme and a corresponding rangeproof system. For encryption, we use a slight variant of the standard *ElGamal* encryption scheme. For rangeproofs, we use an adaptation of the popular Bulletproofs system [3] to work with our variant of ElGamal encryption. Both of these objects are lightweight in comparison to more advanced cryptographic objects such as general-purpose SNARKs, and they can be implemented on a pairing-free elliptic curve such as Curve25519 [2]. As Solana already uses Curve25519 for cryptographic signatures with highly optimized signature verification, we plan to similarly optimize for fast, parallel proof verification.

Reader guide. This document is divided into two parts. Part 1 is geared towards developers that wish to understand the underlying cryptographic protocol for the `ZK-Token` program. We do assume that readers have some basic familiarity with cryptographic concepts such as cryptographic keys, encryption, and signatures.

Part 2 of the document is written for cryptographers and academics. For these readers, we still recommend skimming through the overview in Section 2. Some of our treatment of the protocol may appear quite verbose, but it summarizes the main features of the `ZK-Token` protocol. Then, the readers can immediately jump into Part 2 of the document for a more formal treatment of the protocol and security proofs.

2 Protocol Overview

In this section, we provide the main intuition behind the underlying cryptography that is used in the `ZK-Token` program. We start with a minimal description of the `Token` API in Section 2.1. Then, in each of the subsequent subsections, we incrementally add cryptograhpic features to the `Token` program towards the `ZK-Token` program.

2.1 Starting Point: Basic SPL Tokens

To set the stage for the `ZK-Token` program, we provide a minimal description of the standard `Token` API. For the full details on the `Token` program, we refer to the token program documentation [1] for the full specifications.

The main data structures that are used in the `Token` program are Mint and Account. The Mint data structure is used to store the global information for a particular class of tokens:

[2]The precise way the `ZK-Token` program will work with the regular `Token` program is still in the works and may change from what is specified here.

Mint:

> Address: The address of the mint information.
> Authority: The address of the "owner" of the mint.
> Supply: The total supply of tokens.

The Account data structure is used to store the token balance of a user:

Account:

> Address: The address of the account information.
> Mint: The address of the mint that is associated with this account.
> Owner: The address of the owner account that owns this account.
> Balance: The balance of tokens that this account holds.

A user can initialize an account data structure by submitting an InitializeAccount instruction. The InitializeAccount instruction specifies an account address for the account data, an address for the associated mint, and an owner address.

With these two data structures, users can interact with the token program on the Solana blockchain with the following set of instructions:

- $\mathsf{InitializeMint}(\mathsf{addr}, \mathsf{auth})$: Creates a new token mint by initializing a Mint data structure at the address addr and with a specified mint authority auth.

- $\mathsf{InitializeAccount}(\mathsf{addr}, \mathsf{mint})$: Creates a new token account by initializing an Account data structure associated with the specified mint at the address addr.

- $\mathsf{MintTo}(\mathsf{addr}, \mathsf{amt}, \sigma_{\mathsf{mint\text{-}auth}})$: Mints the specified amount amt of tokens to the account at the address addr.

  The token program processes this instruction only if the specified signature $\sigma_{\mathsf{mint\text{-}auth}}$ verifies under the token mint-authority's public key.

- $\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$: Transfers the amount amt of tokens from the specified source account to the specified destination account.

  The token program processes this instruction only if the specified signature σ_{sender} verifies under the source account's owner public key.[3]

- $\mathsf{CloseAccount}(\mathsf{addr}, \sigma_{\mathsf{owner}})$: Closes the account specified at the address addr.

  The token program processes this instruction only if the specified signature σ_{owner} verifies under the account owner's public key.

2.2 Basic Intuition: SPL Tokens with Encryption and Proofs

Encryption. Since the Account data structure is stored on chain, any user can look up the balance associated with other user accounts. In the `ZK-Token` program, we use the most basic way to hide the balances: keep them in encrypted form. Consider the following example of the Account data structure:

[3]Throughout this section, we use *sender* and *receiver* to refer to the source account owner and destination account owner respectively in a transfer instruction.

Account:

Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
Balance: 50

To hide the account balance, we can encrypt the balance under the account owner's public key before storing it on chain:

Account:

Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
Balance: $\mathsf{Encrypt}(\mathsf{pk}_{\mathsf{owner}}, 50)$

We can similarly use encryption to hide transfer amounts in a transaction. Consider the following example of a transfer instruction:

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
amt: 10

To hide the transaction amount, we can encrypt the amount under the sender's public key before submitting it to the chain:

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
amt: $\mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, 10)$

By simply encrypting account balances and transfer amounts, we can add confidentiality to the `Token` program.

Linear homomorphism. One problem with this simple approach is that the token program cannot deduct or add transaction amounts to accounts as they are all in encrypted form. One way to resolve this issue is to use a class of encryption schemes that are *linearly homomorphic* such as the *ElGamal* encryption scheme.[4] An encryption scheme is linearly homomorphic if for any two numbers $x_0, x_1 \in \mathbb{Z}_p$ and their encryptions $\mathsf{ct}_0 = \mathsf{Encrypt}(\mathsf{pk}, x_0)$, $\mathsf{ct}_1 = \mathsf{Encrypt}(\mathsf{pk}, x_1)$ under the same public key, there exist ciphertext-specific add and subtract operations such that

$$\mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}_1 + \mathsf{ct}_2) = x_0 + x_1,$$
$$\mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}_1 - \mathsf{ct}_2) = x_0 - x_1,$$

[4]The most traditional version of the ElGamal encryption scheme where the message is encoded as a group element is not linearly homomorphic. However, if the message is encoded as an "exponenet", the ElGamal encryption scheme is linearly homomorphic. This variant is sometimes referred to as the *exponential* ElGamal.

In other words, a linearly homomorphic encryption scheme allows numbers to be added and subtracted in encrypted form. The sum and the difference of the individual encryptions of x_0, x_1 result in a ciphertext that is equivalent to an encryption of the sum and the difference of the numbers x_0, x_1.

By using a linearly homomorphic encryption scheme to encrypt balances and transfer amounts, we can allow the token program to process balances and transfer amounts in encrypted form. As linear homomorphism holds only when ciphertexts are encrypted under the same public keys, we require that a transfer amount to be encrypted under both the sender and receiver public keys:

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

> source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
> amt: $\mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, 10)$, $\mathsf{Encrypt}(\mathsf{pk}_{\mathsf{receiver}}, 10)$

Then, upon receiving a transfer instruction of this form, the token program can subtract and add ciphertexts to the source and destination accounts accordingly:

$\mathsf{Account}_{\mathsf{source}}$:

> Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
> Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
> Balance: $\mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, 50) - \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, 10) \approx \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, 40)$

$\mathsf{Account}_{\mathsf{destination}}$:

> Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
> Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
> Balance: $\mathsf{Encrypt}(\mathsf{pk}_{\mathsf{receiver}}, 50) + \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{receiver}}, 10) \approx \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{receiver}}, 60)$

Zero-knowledge proofs. Another problem with encrypting account balances and transfer amounts is that the token program cannot check the validity of a transfer amount. For instance, a user with an account balance of 50 tokens should not be able to transfer 70 tokens to another account. For regular `SPL` tokens, the token program can easily detect that there are not enough funds in the user's account. If account balances and transfer amounts are encrypted, then these values are hidden to the token program itself, preventing it from verifying the validity of a transaction.

Therefore, in the `ZK-Token` program, we require that transfer instructions include zero-knowledge proofs that validate the correctness of the transfer.[5] There are two key components to the proofs that are required for a transfer instruction.[6]

- *Range Proof*: Range proofs are special types of zero-knowledge proof systems that allow users to generate a proof π_{range} that a ciphertext ct encrypts a value $x \in \mathbb{Z}$ that falls in a specified range $\ell, u \in \mathbb{Z}$:

$$\mathsf{Verify}_{\ell,u}(\mathsf{ct}, \pi_{\mathsf{range}}) = 1 \quad \iff \quad \mathsf{ct} = \mathsf{Encrypt}(\mathsf{pk}, x) \text{ and } \ell \leq x < u.$$

[5]Technically speaking, all the cryptographic proofs that are used in the `ZK-Token` program are cryptographic *arguments* as opposed to *proofs*. To keep the presentation as simple as possible, we still refer to these cryptographic objects as "proofs" throughout the text.

[6]We later remove the need for equality proofs in transfer instructions (see Section 2.4.)

The zero-knowledge property guarantees that π_{range} does not reveal the actual value of x, but only the fact that $\ell \le x < u$.

In the `ZK-Tokens` program, we require that a transfer instruction contains a range proof that certify the following:

- The proof should certify that there are enough funds in the source account. Specifically, let $\mathsf{ct}_{\mathsf{source}}$ be the encrypted balance of a source account and $\mathsf{ct}_{\mathsf{trans\text{-}amt}}$ be the encrypted amount of a transfer. Then we require that the proof certifies that

  $$\mathsf{ct}_{\mathsf{source}} - \mathsf{ct}_{\mathsf{trans\text{-}amt}} = \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, x) \text{ such that } 0 \le x < 2^{64}.$$

  By verifying the proof, the token program can check that there are enough funds in the source account.
- The proof should certify that the transfer amount itself is a positive 64-bit number. Let $\mathsf{ct}_{\mathsf{trans\text{-}amt}}$ be the encrypted amount of a transfer. Then we require that the proof certifies that

  $$\mathsf{ct}_{\mathsf{trans\text{-}amt}} = \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, x) \text{ such that } 0 \le x < 2^{64}.$$

  By verifying the proof, the SPL program can check that the transfer amount is a positive number and therefore, cannot create a negative transfer.

• *Equality Proof*: Recall that a transfer instruction contains two ciphertexts of the transfer value x: a ciphertext under the sender public key $\mathsf{ct}_{\mathsf{sender}} = \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, x)$ and one under the receiver public key $\mathsf{ct}_{\mathsf{receiver}} = \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{receiver}}, x)$. However, a malicious user can encrypt two different values for $\mathsf{ct}_{\mathsf{sender}}$ and $\mathsf{ct}_{\mathsf{receiver}}$. Therefore, we require that a transfer instruction contains an equality proof π_{eq} certifying that the two ciphertexts encrypt the same value:

$$\mathsf{ct}_{\mathsf{sender}} = \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{sender}}, x_0), \mathsf{ct}_{\mathsf{receiver}} = \mathsf{Encrypt}(\mathsf{pk}_{\mathsf{receiver}}, x_1) \text{ such that } x_0 = x_1.$$

The zero-knowledge property guarantees that π_{eq} does not reveal the actual values of x_0, x_1, but only the fact that $x_0 = x_1$.

We formally specifiy the proof generation and verification algorithms in Part 2 of the document.

2.3 Usability Features

Encryption key. In our overview of the `ZK-Token` program in the previous subsection, we used the public key (signing key) of the account owner to encrypt the balance of an account. In our actual implementation of the `ZK-Tokens` program, we use a separate account-specific encryption key to encrypt the account balances.

Account:

Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
EncryptionKey: ek
Balance: $\mathsf{Encrypt}(\mathsf{ek}, 50)$

The ElGamal public key ek can be set by the owner of the account by including it as part of an $\mathsf{InitializeAccount}(\mathsf{ek})$ instruction. A corresponding decryption key dk can be stored privately on a client-side wallet.

In general, the use of a single public-secret key pair for both signatures and encryption is discouraged for potential security vulnerabilities and therefore, it makes sense to use separate dedicated key pairs for just encryption. More importantly, the `ZK-Tokens` API is designed to be as general as possible for application developers. Separate dedicated keys for signing transactions and decrypting transaction amounts allow for a more flexible API.

In a potential application, the decryption key dk can be shared among multiple users (i.e. regulators) that should have access to an account balance. Although these users can decrypt account balances, only the owner of the account that have access to the signing key sk can sign a transaction that initiates a transfer of tokens. The `ZK-Tokens` protocol also allows the owner of an account to submit a new encryption key ek' via the $\mathsf{UpdateAccountPk}$ instruction to replace the associated encryption key. This instruction can, for instance, be used to revoke another user's access to an account balance.

Global Auditor. As separate decryption keys are associated with each user accounts, users can provide read access to *specific* account balances to potential auditors. The `ZK-Token` program also supports *global* auditor feature that can be optionally enabled for mints. Specifically, for each `SPL-Token` mint, the `ZK-Token` program maintains a corresponding $\mathsf{TransferAuditor}$ account that maintains a global auditor encryption key. This auditor encryption key can be specified when the `ZK-Token` program is first enabled for a specific `SPL-Token` mint and can be subsequently updated via the $\mathsf{UpdateTransferAuditor}$ instruction. If the transfer auditor feature is enabled, then any transfer instruction must additionally contain an encryption of the transfer amount under the auditor's encryption key.

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
amt: $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, 10)$, $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, 10)$, $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, 10)$

This allows any entity with a corresponding auditor decryption key to be able to decrypt any transfer amounts for a particular mint.

Similarly to how a sender can encrypt inconsistent transfer amounts under the source and destination encryption keys, it can encrypt inconsistent transfer amounts for the auditor ciphertext. Therefore, if the global auditor feature is enabled, the `ZK-Token` program requires that a transfer instruction contains an additional zero-knowledge proof that the transfer amount is encrypted properly under the auditor's public key.

Pending and available balance. One way an attacker can compromise the usability of a `ZK-Token` account is by taking advantage of the "front-running" problem. Zero-knowledge proofs are verified with respect to the encrypted balance of an account. Suppose that a user Alice generates a proof with respect to her current encrypted account balance. If another user Bob transfers some tokens to Alice, and Bob's transaction is processed first, then Alice's transaction will be rejected by the token program as the proof will not verify with respect to the newly updated ciphertext.

Under normal circumstances, upon a rejection by the program, Alice can simply look up the newly updated ciphertext and submit a new transaction. However, if a malicious attacker continuously floods the network with a transfer to Alice's account, then the account may theoretically become unusable.[7] To prevent this type of attack, we modify the account data structure such that the encrypted balance of an account is divided into two separate components: the *pending* balance and the *available* balance.

> Account:
>
> > Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> > Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
> > Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
> > EncryptionKey: ek
> > PendingBalance: $\mathsf{Encrypt}(\mathsf{ek}, 10)$
> > AvailableBalance: $\mathsf{Encrypt}(\mathsf{ek}, 50)$

Any outgoing funds from an account is subtracted from its available balance. Any incoming funds to an account is added to its pending balance.

As an example, consider a transfer instruction that moves 10 tokens from a sender's account to a receiver's account.

> $\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:
>
> > source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> > destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
> > amt: $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, 10)$, $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, 10)$, $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, 10)$

Upon receiving this a transaction and after verifying its validity, the token program subtracts the encrypted amount from the sender's available balance and adds the encrypted amount to the receiver's pending balance.

> $\mathsf{Account}_{\mathsf{source}}$:
>
> > Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> > Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
> > Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
> > EncryptionKey: $\mathsf{ek}_{\mathsf{sender}}$
> > PendingBalance: $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, 10)$
> > AvailableBalance: $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, 50) - \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, 10) \approx \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, 40)$

> $\mathsf{Account}_{\mathsf{destination}}$:
>
> > Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> > Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
> > Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
> > EncryptionKey: $\mathsf{ek}_{\mathsf{receiver}}$
> > PendingBalance: $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, 10) + \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, 10) \approx \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, 20)$
> > AvailableBalance: $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, 10)$

[7] We do believe that such an attack on the Solana network is unlikely to actually occur in practice. With current transaction throughput and latency, it will be extremely costly for an attack to forcibly disable an account for any extended period of time due to transaction costs.

This modification removes the sender's ability to change the receiver's available balance of a source account. As range proofs are generated and verified with respect to the available balance, this prevents a user's transaction from being invalidated due to a transaction that is generated by another user.

An account's pending balance can be merged into its available balance via the ApplyPendingBalance instruction, which only the owner of the account can authorize. Upon receiving this instruction and after verifying that the owner of the account signed the transaction, the token program adds the pending balance into the available balance.

Account:

> Address: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> Mint: `Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB`
> Owner: `5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6`
> EncryptionKey: ek
> PendingBalance: $\mathsf{Encrypt}(\mathsf{ek}, 0)$
> AvailableBalance: $\mathsf{Encrypt}(\mathsf{ek},\ 50_{\mathsf{available}}) + \mathsf{Encrypt}(\mathsf{ek},\ 10_{\mathsf{pending}}) \approx \mathsf{Encrypt}(\mathsf{ek},\ 60)$

2.4 Cryptographic Optimizations

Dealing with discrete log. A well-known limitation of using a linearly-homomorphic ElGamal encryption is the inefficiency of decryption. The linearly-homomorphic ElGamal encryption scheme is defined over a cyclic group $\mathbb{G}$ of prime order $p \in \mathbb{N}$. Let ct_x be an ElGamal encryption of a value $x \in \mathbb{Z}$. Then, in an ElGamal encryption scheme, a successful decryption of ct_x result in the group element $x \cdot G$ where G is a fixed generator for $\mathbb{G}$. To completely recover the originally encrypted value x, one must recover x from $x \cdot G$, which is a hard computational problem called the *discrete log problem*, which requires an exponential time to compute.

To deal with this problem of expensive message recovery, cryptographic protocols generally resrict the messages to be encrypted to small numbers. However, as the `Token` program supports 64-bit transfer amounts, the `ZK-Token` program must ideally support 64-bit numbers as well. Even with modern hardware, fully decrypting 64-bit numbers from an ElGamal decryption is intractable. This means that given a transfer of any large amount, neither the receiver nor the auditor can decrypt the transfer amount efficiently:

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

> source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
> destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
> amt:
>> $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{tx\text{-}amt}})$,
>> $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, x_{\mathsf{tx\text{-}amt}})$,
>> $\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, x_{\mathsf{tx\text{-}amt}})$

To fix this problem in a transfer instruction, we require that a 64-bit transfer amount $x_{\mathsf{tx\text{-}amt}}$ be divided into two 32-bit numbers x_{lo}, x_{hi} such that $x_{\mathsf{tx\text{-}amt}} = x_{\mathsf{lo}} + 2^{32} \cdot x_{\mathsf{hi}}$ and then encrypted separately:

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
amt:

$\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{lo}}),\ \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{hi}})$
$\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, x_{\mathsf{lo}}),\ \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, x_{\mathsf{hi}})$
$\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, x_{\mathsf{lo}}),\ \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, x_{\mathsf{hi}})$

Since the encrypted values of ElGamal ciphertexts are all 32-bit numbers, they can all be decrypted and recovered by the receiver and the auditor (See Section 3.3). Naturally, instead of requiring that the range proof associated with a transfer instruction certify that ciphertexts encrypt 64-bit transfer amounts, we require that it certifies multiple 32-bit transfer amounts.

Even if the transfer amount $x_{\mathsf{tx\text{-}amt}}$ is encrypted as two different ciphertexts, they can be homomorphically merged into a single encryption of $x_{\mathsf{tx\text{-}amt}}$:

$$\begin{aligned}\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{lo}}) + 2^{32} \cdot \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{hi}}) &= \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{lo}} + 2^{32} \cdot x_{\mathsf{hi}}) \\ &= \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{tx\text{-}amt}}).\end{aligned}$$

Upon receiving a transfer instruction, the token program merges the corresponding ciphertexts, and then subtracts and adds them to the source and destination account balances respectively.

Twisted ElGamal encryption A key challenge in designing any private payment system is minimizing the size of a transaction. In the `ZK-Tokens` program, we make a number of optimizations that reduces transaction size. Among these optimizations, a significant amount of savings stem from the use of the *twisted* ElGamal encryption, which was formulated in the work of [4] as a simple variant of the standard ElGamal encryption scheme. In the twisted ElGamal encryption, a ciphertext is divided into two components:

- A *Pedersen commitment* of the encrypted message, which is independent of the public key.
- A *decryption handle* that encodes the encryption randomness with respect to a specific public key, and is independent of the encrypted message.

We provide the formal details of the twisted ElGamal encryption in Section 3.

The use of twisted ElGamal encryption in place of the standard ElGamal encryption scheme provides two main savings. First, range proofs such as Bulletproofs [3] can be used almost directly on twisted ElGamal encryption. These systems are designed specifically for Pedersen commitments and are not compatible with the standard ElGamal encryption scheme without additional proof components that serves as a conceptual "bridge" between Pedersen commitments and ElGamal ciphertexts. In a twisted ElGamal encryption scheme, messages are already encoded as Pedersen commitments and therefore, there is no need for these extra components.[8]

Next, the use of twisted ElGamal encryption removes the need for equality proofs. Twisted ElGamal ciphertexts are divided into two components: Pedersen commitments and decryption handles. Since Pedersen commitments are independent of the public keys for which the ciphertexts are encrypted under, these components can be shared among multiple ciphertexts that encrypt same messages. For instance, consider the following transfer instruction from above:

[8]An encryptor still need to prove the *well-formedness* of twisted ElGamal cipehrtexts. With simple tweaks on the Bulletproof setup protocol, this can be done with minimal overhead (+2 groups elements).

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
amt:

$\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{lo}}),\ \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{sender}}, x_{\mathsf{hi}})$
$\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, x_{\mathsf{lo}}),\ \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{receiver}}, x_{\mathsf{hi}})$
$\mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, x_{\mathsf{lo}}),\ \mathsf{Encrypt}(\mathsf{ek}_{\mathsf{auditor}}, x_{\mathsf{hi}})$

Instead of including six ciphertexts (64 bytes each) on a single instruction, one can generate a single commitment for each $x_{\mathsf{lo}}, x_{\mathsf{hi}}$ and then include decryption handles for each of these two commitments:

$\mathsf{Transfer}(\mathsf{source}, \mathsf{destination}, \mathsf{amt}, \sigma_{\mathsf{sender}})$:

source: `EzMCP3PVVkZD4YFXhJNKhgwMihuxxeFqH4F6uPxFxFAe`
destination: `0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7`
amt:

Commitments : $\mathsf{PedComm}(x_{\mathsf{lo}}),\ \mathsf{PedComm}(x_{\mathsf{hi}})$
Handles:

$\mathsf{DecHandle}_{x_{\mathsf{lo}}}(\mathsf{ek}_{\mathsf{sender}}),\ \mathsf{DecHandle}_{x_{\mathsf{hi}}}(\mathsf{ek}_{\mathsf{sender}}),$
$\mathsf{DecHandle}_{x_{\mathsf{lo}}}(\mathsf{ek}_{\mathsf{receiver}}),\ \mathsf{DecHandle}_{x_{\mathsf{hi}}}(\mathsf{ek}_{\mathsf{sender}}),$
$\mathsf{DecHandle}_{x_{\mathsf{lo}}}(\mathsf{ek}_{\mathsf{auditor}}),\ \mathsf{DecHandle}_{x_{\mathsf{hi}}}(\mathsf{ek}_{\mathsf{sender}}),$

Each decryption handle can be combined with the corresponding Pedersen commitments to form a complete ElGamal ciphertext. For example, the auditor can use its decryption key $\mathsf{dk}_{\mathsf{auditor}}$ to decrypt the pairs

$$\mathsf{Decrypt}(\mathsf{dk}_{\mathsf{auditor}}, \mathsf{PedComm}(x_{\mathsf{lo}}), \mathsf{DecHandle}_{x_{\mathsf{lo}}}(\mathsf{ek}_{\mathsf{auditor}})) = x_{\mathsf{lo}}$$
$$\mathsf{Decrypt}(\mathsf{dk}_{\mathsf{auditor}}, \mathsf{PedComm}(x_{\mathsf{hi}}), \mathsf{DecHandle}_{x_{\mathsf{hi}}}(\mathsf{ek}_{\mathsf{auditor}})) = x_{\mathsf{hi}}$$

As Pedersen commitments and decryption handles are 32 bytes each, this way of encrypting transaction amounts result in savings in the total ciphertext size associated with a transfer instruction. More importantly, since Pedersen commitments are shared among multiple ciphertexts, there is no need for equality proofs to be included in a transfer instruction. The instruction still requires proofs that each decryption handles are generated correctly, but these proofs are much smaller in size compared to equality proofs. We provide the details on the proof requirements for transfer instructions in Part 2 of the document.

3 Encryption in the ZK-Token Program

In this section, we describe the twisted ElGamal encryption scheme [4]. We begin by defining basic cryptographic objects called *Pedersen commitments* in Section 3.1. We then specify the algorithms for the twisted ElGamal encryption scheme in Section 3.2. Finally, we discuss the discrete log problem that makes up a crucial component of the ElGamal decryption algorithm in Section 3.3.

Throughout this section, we work over an abstract cyclic group $(\mathbb{G}, +)$ of prime order $p \in \mathbb{N}$. In our actual implementation, we use the Ristretto group implementation over Curve25519 [5]. Following the implementation convention, we refer to elements in $\mathbb{Z}_p$ as *scalars* and elements in

$\mathbb{G}$ as *points*. Also, we use the standard pk and sk to denote the public encryption key and secret decryption key respectively. This is in contrast to Section 2.3 where we used ek and dk for the encryption and decryption keys to distinguish them from a user's public and signing key.

3.1 Pedersen Commitments

In cryptography, a commitment scheme is a message encoding algorithm Commit that has the following syntax:

- $\mathsf{Commit}(x, r) \to C$: On input a scalar message x and a randomly generated "opening" r, the algorithm returns an encoding C that is generally referred to as a *commitment*.

A commitment scheme is secure if it satisfies the following two properties:

- *Binding*: Given a commitment C, it is hard to find two message-opening pairs (x_0, r_0), (x_1, r_1) such that
$$\mathsf{Commit}(x_0, r_0) = \mathsf{Commit}(x_1, r_1).$$
- *Hiding*: As long as an opening r is generated from a proper randomness source, a commitment C does not reveal any information about the committed message x.

The simplest example of a cryptographic commitment scheme is a scheme that uses a cryptographic hash function H:
$$\mathsf{Commit}(x, r) = H(x \| r),$$
where the message x and opening r are arbitrary bit-strings. A cryptographic hash function H is collision-resistant and therefore, it is hard to find two message-opening pairs (x_0, r_0) and (x_1, r_1) such that $H(x_0 \| r_0) = H(x_1 \| r_1)$. Furthermore, if the opening r is a sufficiently long, then the output of the hash does not reveal information about x.

In the `ZK-Token` program, we use a specific commitment scheme called the Pedersen commitment scheme. Pedersen commitments are defined with respect to two fixed group elements $G, H \in \mathbb{G}$. The commitment algorithm is defined as follows:

- $\mathsf{Commit}(x, r) \to C$: On input a scalar message $x \in \mathbb{Z}_p$ and a randomly generated opening $r \in \mathbb{Z}_p$, the algorithm returns
$$C = x \cdot G + r \cdot H \quad \in \mathbb{G}.$$

It is well-known that the Pedersen commitment scheme over any cryptographically robust group $\mathbb{G}$ satisfies both the security properties of binding and hiding.

3.2 Twisted ElGamal Encryption

Pedersen commitments are incoporated into many cryptographic protocols as their algebraic structure makes it convenient to design zero-knowledge proof systems around. In the setting of the `ZK-Token` program, we require zero-knowledge proofs to work over ciphertexts, not commitments. However, if we use a specific encryption scheme called *twisted ElGamal encryption*, then we can almost directly use proof systems that are designed specifically for Pedersen commitments for ciphertexts.

In the twisted ElGamal encryption scheme, a ciphertext can be viewed as a wrapper around Pedersen commitment. As in the setting of Pedersen commitments, the twisted ElGamal encryption is defined with respect to two fixed group elements $G, H \in \mathbb{G}$ and are specified as follows:

- $\mathsf{KeyGen}() \to (\mathsf{pk}, \mathsf{sk})$: The key generation algorithm generates a random scalar $s \leftarrow_{\mathsf{R}} \mathbb{Z}_p$. It computes $P = s^{-1} \cdot H$ and sets
$$\mathsf{pk} = P, \quad \mathsf{sk} = s.$$
- $\mathsf{Encrypt}(\mathsf{pk}, x) \to \mathsf{ct}$: The encryption algorithm takes in a public key $\mathsf{pk} = P \in \mathbb{G}$ and a message (either 32-bit or 64-bit number for the `ZK-Token` program) represented as a scalar $x \in \mathbb{Z}_p$ to be encrypted. It samples a random scalar $r \leftarrow_{\mathsf{R}} \mathbb{Z}_p$ and then computes the following components:
  1. *Pedersen commitment*: $C = r \cdot H + x \cdot G$,
  2. *Decryption handle*: $D = r \cdot P$.

  It returns $\mathsf{ct} = (C, D)$.
- $\mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}) \to x$: The decryption algorithm takes in a secret key $\mathsf{sk} = s$ and a ciphertext $\mathsf{ct} = (C, D)$ as input. It computes
$$C = C - s \cdot D \quad \in \mathbb{G},$$
  and then solves the discrete log problem to recover $x \in \mathbb{Z}_p$ for which $x \cdot G = P$.

Correctness. The correctness of the decryption algorithm can be easily verified. Let $\mathsf{pk} = P$, $\mathsf{sk} = s$ be a public-secret key pair and let $\mathsf{ct} = (C = r \cdot H + x \cdot G, D = r \cdot P)$ be a properly encrypted ciphertext of a scalar message $x \in \mathbb{Z}_p$. Then, a proper decryption of the ciphertext produces:

$$\begin{aligned} C - s \cdot D &= (rH + xG) - s \cdot (rP) \\ &= rH + xG - r \cdot (sP) \\ &= rH + xG - rH \\ &= xG \end{aligned}$$

Assuming that the discrete log is solved correctly, the decryption algorithm recovers the original message $x \in \mathbb{Z}_p$.

3.3 Computing Discrete Log.

Let $\mathbb{G}$ be a cyclic group of prime order $p \in \mathbb{N}$, and let $G \in \mathbb{G}$ be any fixed point in $\mathbb{G}$ generally referred to as the "generator" for $\mathbb{G}$. Then, any point $C \in \mathbb{G}$ can be represented as a multiple of G:

$$x \cdot G = C.$$

The discrete log problem is the computational task of recovering x given a generator G and a target C:

- **Input**: generator $G \in \mathbb{G}$ and target $C \in \mathbb{G}$
- **Output**: $x \in \mathbb{Z}_p$ such that $x \cdot G = C$

If C is an arbitrary random element in the group G, then this problem is well-known to be hard to compute. However, if there is a guarantee that the unique scalar $x \in \mathbb{Z}_p$ for which $C = x \cdot G$ is a small number (e.g. $0 \leq x < 2^{32}$), then the problem can be solved efficiently.

Suppose that x is a 32-bit number. The simplest way to solve the discrete log problem is to enumerate through all possible values $0 \leq x < 2^{32}$ and test for the relation $C = x \cdot G$. Although this brute-force way of computing discrete log is in the realm of feasibility with modern hardware, we can solve the problem much more efficiently by relying on the space-time tradeoff. Let $x_{\mathsf{lo}}, x_{\mathsf{hi}}$ be two numbers represented by the least and most significant 16-bits of x such that $x = x_{\mathsf{lo}} + 2^{16} \cdot x_{\mathsf{hi}}$. Then the discrete log relation can be represented as

$$\begin{aligned} C &= x \cdot G \\ &= (x_{\mathsf{lo}} + 2^{16} \cdot x_{\mathsf{hi}}) \cdot G \\ &= \underbrace{x_{\mathsf{lo}} \cdot G}_{\text{online}} + \underbrace{2^{16} \cdot x_{\mathsf{hi}} \cdot G}_{\text{offline}} \end{aligned}$$

Using this relation, one can solve the discrete log problem by first computing all possible values of $2^{16} \cdot x_{\mathsf{hi}} \cdot G$ for $x_{\mathsf{hi}} = 0, 1, \ldots, 2^{16} - 1$ and then solving for x_{lo}:

$\mathsf{ComputeDiscreteLog}(G, C)$:

1. Instantiate a look-up table (e.g. HashMap). For $x_{\mathsf{hi}} = 0, \ldots 2^{16} - 1$, store the following key-value pairs
   - `key`: $2^{16} \cdot x_{\mathsf{hi}} \cdot G$
   - `value`: x_{hi}
2. For $x_{\mathsf{lo}} = 0, \ldots, 2^{16} - 1$, check whether a key $C - x_{\mathsf{lo}} \cdot G$ exists in the look-up table. If it exists, then take the table entry x_{hi} and return $x = x_{\mathsf{lo}} + 2^{16} \cdot x_{\mathsf{hi}}$.

We note that the look-up table in step 1 above is independent of the discrete log target $C \in \mathbb{G}$. Therefore, the look-up table can be pre-computed once offline and can be re-used for multiple instances of discrete log with a fixed generator $G \in \mathbb{G}$. The look-up table stores 2^{16} entries of 16-bit numbers, which translates to around 130 kilobytes. With this amount of pre-computation, any discrete log challenge can be solved in a matter of seconds.

We note that the algorithm above divides the 32-bit number x specifically into two 16-bit numbers x_{lo} and x_{hi}. This choice of numbers is purely for simplicity; the numbers x_{lo} and x_{hi} can be chosen to be arbitrary ℓ_{lo} and ℓ_{hi}-bit numbers for which $\ell_{\mathsf{lo}} + \ell_{\mathsf{hi}} = 32$. The numbers ℓ_{lo} and ℓ_{hi} define a space-time trade-off in computing the discrete log problem. For instance, by setting ℓ_{lo} smaller and ℓ_{hi} larger, the discrete log problem can be solved in less than a second with larger pre-computed data.

4 Instructions

Coming soon!

References

[1] Solana Program Library Docs. https://spl.solana.com. Accessed: 2021-08-18.

[2] Bernstein, D. J. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings* (2006), M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958 of *Lecture Notes in Computer Science*, Springer, pp. 207–228.

[3] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., and Maxwell, G. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 315–334.

[4] Chen, Y., Ma, X., Tang, C., and Au, M. H. Pgc: Decentralized confidential payment system with auditability. In *European Symposium on Research in Computer Security* (2020), Springer, pp. 591–610.

[5] Lovecruft, I. A., and de Valence, H. `curve25519-dalek`. https://github.com/dalek-cryptography/curve25519-dalek, 2021.