Confidential Security Contract Standard

Toghrul Maharramov
Dusk Network
toghrul@dusk.network
v2.0.0

February 18, 2021

Abstract

Compliant security token issuance and lifycycle management standard is an important milestone on the way to bridging the gap between traditional and decentralized finance. In this paper, we define the second iteration of the Confidential Security Contract Standard, capable of facilitating a compliant security token issuance and lifecycle management framework through the use of novel cryptography. The standard enables the prospective issuer to exert granular control over the contract. The defined standard is to be natively supported in the Dusk Network protocol, a blockchain-based protocol built from the ground up to support compliant financial applications.

Contents

1 Introduction

Confidential Security Contract Standard (to be referred to as *XSC* from hereon) is a standard created to facilitate the requirements for compliant security token issuance and lifecycle management on the Dusk Network protocol. The standard ensures the preservation of transactional confidentiality while simultaneously guaranteeing auditability and compliance through the use of advanced cryptographic techniques. Specifically, XSC heavily relies on zero-knowledge proofs to attest to the correctness of the computations and to obfuscate the transactional data as well as provable encryption to guarantee auditability and transparency to the relevant parties involved. The document describes the generalized configuration of the contract standard.

1.1 Business Summary

XSC consists of a few dozen parameters that can be configured to capture the requirements of almost any financial instrument. There is room for custom configurations for anyone to use as they see fit. In addition, based on the popularity of certain asset classes, standardized templates can be selected as a starting point for further customization of said asset classes. This document outlines the relevant technical parameters that could be toggled on or off, and what the standard preset for a baseline security looks like. At an abstract level, the standard is configured as follows:

Theme	Configuration	Section
Smart Contract Management[1]	ON	**4.1**
Roles, Access Control and Whitelist Management	ON	**2.3, 4.2-4.3**
Confidential Settlement[2]	ON	**2.2, 3.1, 4.4**
Voting	ON	**3.2, 4.5**
Dividend/Coupon Claims[3]	ON	**3.3, 4.6**
Supply Management	OFF	**4.9**

NOTE: This preset configuration forms the baseline for security issuance and lifecycle management, and the preset configuration can be changed to allow for customized implementations for specific requirements of a said security.

[1] The owner of the smart contract has access to all contract management functionality.

[2] Securities are a regulated financial instruments and onlooker confidentiality is a legal requirement.

[3] Users should be able to receive their dividends at a fixed frequency.

2 Key Terminology and Notations

2.1 Notations

2.1.1 General Notations

A textttleftarrow symbol denotes an assignment function, e.g. $A \leftarrow B$ stands for the assignment of B to A. A = symbol denotes the equivalency function, e.g. $A = B$ stands for A is equivalent to B.

$\mathcal{F}$ denotes a public function of the contract (**NOTE:** functions always return `bool` after the termination of the execution). $\mathcal{F}$ with a subscipt upper case letter defines the accessibility of the given function, e.g. $\mathcal{F}_O$ stands for a function accessible to the owner set O of the contract. $\mathcal{F}$ with a superscript capitalized LaTeX `mathsf` font lettering defines name of the given function, e.g. $\mathcal{F}^{\mathsf{FUNCTION}}$ stands for a function named **Function**. $\mathcal{Q}$ denotes a public query to the state of the contract (**NOTE:** queries can return arbitrarily-typed and sized data). $\mathcal{Q}$ with a superscript capitalized LaTeX `mathsf` font lettering defines name of the given query, e.g. $\mathcal{Q}^{\mathsf{QUERY}}$ stands for a query named **Query**.

A LaTeX `mathsf` font lettering with the word "Tree" appended to it denotes a Merkle Tree structure, e.g. $\mathsf{merkleTree}$. A $\mathsf{merkleTree}$ with a superscript capital R defines the root of the Merkle Tree, e.g. $\mathsf{merkleTree}^R$.

Greek letter σ denotes a signature. σ with subscript lower case LaTeX `mathsf` font lettering defines a signature scheme utilized to generate said signature, e.g. $\sigma_{\mathsf{schnorr}}$, is a signature generated with Schnorr signature scheme. Greek letter ν denotes a nullifier. ν with subscript capitalized letter defines a nullifier corresponding to the given structure, e.g. ν_I, is the nullifier of identity I. Greek letter π denotes a zero-knowledge proof. π with a subscript lower case LaTeX `mathsf` lettering defines name of the given zero-knowledge proof, e.g. π_{plonk} stands for a zero-knowledge proof named **PLONK**.

2.1.2 Public Key Infrastructure

Name	Notation	Definition
Secret Spend Key	$ssk \leftarrow (a, b)$	A key pair enabling the computation of sk
Public Spend Key	$psk \leftarrow (A, B)$	A key pair enabling the computation of pk
View Key	$vk \leftarrow (a, B)$	A key pair enabling the location of pk
Secret Key	sk	A key enabling the spending of transaction with pk
One-time Key	pk	A key identifying the recipient of the transaction
Identity	$I \leftarrow (pk, R)$	A key pair identifying the whitelisted user

2.2 Primitives

2.2.1 Hash Function

$\mathcal{H}()$ is a zero-knowledge proof-friendly hash function which takes message m of an arbitrary size as an input and produces constant-size output x:

$$x \leftarrow \mathcal{H}(m).$$

To be considered cryptographically secure, hash functions are required to comply with the following requirements:

1. **Pre-image resistance**. The probability of a Probabilistic Polynomial-Time (PPT) Adversary $\mathcal{A}$ finding m given x (i.e. $m \leftarrow \mathcal{H}^{-1}(x)$) is negligible.
2. **Second preimage resistance**. The probability of $\mathcal{A}$ finding m_2 given m_1, where $\mathcal{H}(m_1) = \mathcal{H}(m_2)$ and $m_1 \neq m_2$, is negligible.
3. **Collision resistance**. The probability of $\mathcal{A}$ finding m_1 and m_2, where $\mathcal{H}(m_1) = \mathcal{H}(m_2)$ and $m_1 \neq m_2$, is negligible.

For the hash function utilized in the Dusk Network protocol there exist no attacks violating the aforementioned properties taking less than 2^{128} calls, i.e. the hash function has a security level of 128 bits against all of the aforementioned attacks.

Specifically, XSC is instantiated with Poseidon [Gra+19] hash function as $\mathcal{H}()$.

2.2.2 Merkle Tree

Merkle Tree is a tree-like cryptographic structure which is constructed through recursive hashing of the child nodes beginning with leaf nodes and ending with a single root node. In order to prove the inclusion of a certain leaf node in a Merkle Tree $\mathsf{merkleTree}$, the Prover P has to provide the Verifier V with the Merkle Tree path opening the leaf node N, N^P, which includes the aforementioned leaf node as well as the neighboring node for every level of the tree.

2.2.3 Elliptic Curve

Elliptic curves are a common cryptographic primitive. The discrete logarithm problem in a group of curve points is presumed to be hard, and the 128-bit security is achieved for rather small fields (from 256 to 384 bits) compared to integer fields where much bigger (2048 bits and higher) modulus is required. The discrete logarithm problem is for given curve points G and H on the curve find integer x such that $s \cdot G = H$ where $\cdot$ is the scalar multiplication in the group $\mathcal{G}$.

Specifically, XSC is instantiated with JubJub [BHW] elliptic curve.

2.2.4 Stealth Address Scheme

Stealth address is a one-time public key generated via a scheme based on Diffie-Hellman Key Exchange (DHKE) [DH76], proposed in [Sab13]. The scheme conceives three key pair types:

1. **secret spend key**, $ssk \leftarrow (a, b)$; where a and b are randomly generated scalars and represent a pair of secret keys.

2. **public spend key**, $psk \leftarrow (A, B)$; where $A = a \cdot G$ (G is a generator of a group $\mathcal{G}$) and $B = b \cdot G$ are compact representations of points on elliptic curve and represent a pair of public keys.

3. **view key**, $vk \leftarrow (a, B)$; where a is a randomly generated scalar and $B = b \cdot G$ is a compact representation of a point on elliptic curve and represent a secret and public key respectively.

To generate a one-time key (i.e. stealth address), the receiver R is required to share his public spend key, psk, with the sender S, after which S is to proceed with following steps:

1. Generate a random scalar r.

2. Compute a one-time public key $pk \leftarrow \mathcal{H}_p(r \cdot A) \cdot G + B$.

3. Compute $R \leftarrow r \cdot G$ and propagate (pk, R) to receiver R.

To detect a message addressed to R, R is required to scan through the incoming messages using view key, vk, to check whether $pk = \mathcal{H}(R \cdot a) \cdot G + B$ holds true for one of the transactions.

To compute the spend key, sk, corresponding to the one-time public key, pk, R is required to complete the following computation using his secret spend key, ssk: $sk \leftarrow \mathcal{H}(R \cdot a) + b$.

2.2.5 Encryption Scheme

$\mathcal{E}()$ is a zero-knowledge proof friendly encryption function which takes plaintext m and encryption key k as an input and produces ciphertext e as an output:

$$c \leftarrow \mathcal{E}_k(m).$$

Specifically, XSC is instantiated with ElGamal asymmetric encryption scheme in conjunction with a permutation-based AEAD, concretely setup with Poseidon-SpongeWrap [Kho20] symmetric encryption scheme as $\mathcal{E}()$.

To encrypt a plaintext to multiple keys, the sender S is required to encrypt a symmetric key to multiple recipients using elliptic curve ElGamal function $\mathcal{E}^A()$ and then encrypt the plaintext to the aforementioned key:

1. Generate random scalar r.
2. Compute $R \leftarrow r \cdot G$.
3. For each recipient pk_i generate random scalar s_i and compute $W_i \leftarrow s_i \cdot pk_i + R$ and $V_i \leftarrow s_i \cdot G$.
4. Encrypt plaintext m to P, $W \leftarrow \mathcal{E}_P^S(m)$ where $\mathcal{E}^S$ is Poseidon-SpongeWrap with the initial state being initialized to $0||R$ and propagate the full ciphertext $e \leftarrow (V_1, W_1, V_2, W_2, \ldots, V_t, W_t, W)$ to recievers $R_1, R_2, ..., R_t$.

To decrypt the ciphertext, the reciever R_i with a corresponding sk_i such that $pk_i \leftarrow sk_i \cdot G$ computes:

$$R = W_i - pk_i \cdot V_i$$

and decrypts

$$C \leftarrow \mathcal{E}_R^{-1}(c).$$

2.2.6 Commitment Scheme

$\mathcal{C}()$ is a commitment scheme which takes value v and a random blinder b as an input and produces commitment c as an output:

$$c \leftarrow \mathcal{C}(v, b),$$

where

$$\mathcal{C}(v, b) = v \cdot G + b \cdot H$$

Commitment scheme enables the prover P to commit to a value privately while having a capability to reveal the value P has committed to at a later point. Formally, a secure commitment scheme must be hiding (the probability of $\mathcal{A}$ extracting value v from the commitment is negligible) and binding (the probability of $\mathcal{A}$ finding another value v and blinder b capable of opening commitment c is negligible).

Specifically, XSC is instantiated with Pedersen commitment scheme [Ped91] as $\mathcal{C}()$.

2.2.7 Signature Scheme

$\Sigma()$ is a a signature scheme which takes message m and secret key as an input and produces signature σ as an output:

$$\sigma \leftarrow \Sigma^S(m, sk),$$

where the signature can be verified with:

$$\Sigma^V(\sigma, m, pk) = \texttt{true},$$

where $pk = sk \cdot G$.

Signature scheme enables prover P to authenticate a message by binding his secret key to the message in a verifiable way. To process a long message, a cryptographic hash function is deployed. To be considered cryptographically secure, signature schemes are to adhere to the following requirements:

1. **Unforgeability**. The probability of $\mathcal{A}$ being able to reproduce signature σ given message m is negligible.
2. **Message binding**. The probability of $\mathcal{A}$ being to find message m_2 given message m_1 to produce $\sigma_1 = \sigma_2$ where $\sigma_1 \leftarrow \Sigma_{sk}^{S}(m_1)$ and $\sigma_2 \leftarrow \Sigma_{sk}^{S}(m_2)$ is negligible.
3. **Non-malleability**. The signature value can not be modified to another value valid for the same message.

Specifically, XSC is instantiated with Schnorr signature scheme [**Schnorr**] as $\Sigma()$.

2.2.8 Zero-Knowledge Proof Scheme

$\Pi()$ is a zero-knowledge proof scheme which takes public values p, private values w and prover key pk as an input and produces proof π as an output:

$$\pi \leftarrow \Pi^{P}(p, w, \mathsf{pk}),$$

where π can be verified with:

$$\Pi^{V}(p, \pi, \mathsf{vk}) = \mathtt{true},$$

where vk is a verifier key.

To be considered cryptographically secure, zero knowledge proof of knowledge protocol has to adhere to the following requirements:

1. **Completeness**. An honest prover P succeeds in convincing the verifier V of the statement.
2. **Soundness**. The probability of $\mathcal{A}$ proving an invalid statement to verifier V is negligible.
3. **Zero-knowledgeness**. The proof reveals no information other than the fact that the statement is true.

Specifically, XSC is instantiated with $\mathcal{P}\mathfrak{lonK}$ [GWC19] zero-knowledge proof scheme as $\Pi()$.

2.2.9 Transaction Scheme

Dusk Network protocol utilizes a novel transaction scheme called Zedger for the Confidential Security Contract Standard. Zedger is a hybrid transaction model designed to facilitate the extensive privacy and functional requirements of security tokenization and lifecycle management. Zedger utilizes a data structure called *Sparse Merkle-Segment Trie* forming the basis for a concept of *private memory* tasked with the preservation of the privacy of the individual accounts while simultaneously guaranteeing the incorruptability of the data represented in the aforementioned structure. In Zedger, an account is comprised of a *Balance Trie* responsible for storing the account-related balance data for each logged segment. Each account can be associated with only one valid Balance Trie at time t.

Zedger requires the explicit approval of the receiver before the transfer is considered to be settled, meaning that the settlement is completed in two distinct transactions. When transferring assets to another account, the sender is required to mint a Coin C which the receiver is to consume within a limited period of time. If C is not consumed within the aforementioned limited period of time, the sender can consume the C, annulling changes made to the Balance Trie during the original initiation of a transfer.

The minted Coins are stored in Merkle Tree structure called *Coin Tree* $\mathsf{coinTree}$. The use of a Merkle Tree enables the protocol to facilitate confidential transfers of securities. As a double-spending attack prevention, each Coin C has a unique corresponding nullifier ν_C, only known to the sender S and the receiver R of the given transfer. Once a Coin C is spent, the corresponding nullifier is added to the set of existing nullifiers, preventing anybody else from spending C again.

For every Balance Trie update, the root of the Balance Trie (called *Memory Slot* M) is added to the to the a Merkle Tree structure called *Memory Tree* $\mathsf{memoryTree}$. As with Coins, each Memory Slot M has a corresponding nullifier ν_M preventing anybody from updating the same Balance Trie twice.

Zedger is comprised of 7 state mutation functions M outlined below:

Create

Transaction $\mathsf{M}^{\mathsf{CREATE}}()$ is responsible for the registration of a user account for a whitelisted user (**NOTE:** a whitelisted user is permitted to register 1 account). $\mathsf{M}^{\mathsf{CREATE}}()$ creates an empty Memory Slot M corresponding to the whitelisted identity.

Send

Transaction $\mathsf{M}^{\mathsf{SEND}}()$ is responsible for the initiation of the transfer from sender S to receiver R. $\mathsf{M}^{\mathsf{SEND}}()$ mints a new Coin C corresponding to the amount

being transferred as well as deducting the equivalent amount from the sender's balance.

Accept

Transaction $\mathsf{M}^{\mathsf{ACCEPT}}()$ is responsible for the settlement of the transfer on the side of the receiver R. $\mathsf{M}^{\mathsf{ACCEPT}}()$ consumes a previously minted Coin C (only if the predefined validity period of the said Coin has not elapsed) via the revelation of the corresponding nullifier as well as the addition of the equivalent amount to the amount included in the consumed Coin C to the receiver's balance.

Settle

Transaction $\mathsf{M}^{\mathsf{SETTLE}}()$ is responsible for the settlement of the transfer on the side of the sender S. Once the Coin C is consumed by the receiver R, S is required to finalize the transfer on the side of S through the update to his balance via $\mathsf{M}^{\mathsf{SETTLE}}()$.

Claim

Transaction $\mathsf{M}^{\mathsf{CLAIM}}()$ is responsible for the invalidation of the initial transfer in case the receiver R has not claimed Coin C within a predefined validity period. $\mathsf{M}^{\mathsf{CLAIM}}()$ reverts the changes to the balance of S enacted upon the initialization of the transfer.

Vote

Transaction $\mathsf{M}^{\mathsf{VOTE}}()$ is responsible for voting with an eligible amount of voting right-giving security at the specified voting height.

Push Dividend

Transaction $\mathsf{M}^{\mathsf{PUSH\_DIVIDEND}}()$ is responsible for pushing the dividend on behalf of a user U for a predefined period.

2.3 Roles

2.3.1 Owner

Owner set O represents a set of public keys $\{pk_1, ..., pk_n\}$ either defined prior to the deployment of the contract or updated during the operation of the contract. O has a capability to customize the governance of the contract, permitting O to set the number m of signatures corresponding to the public keys of the set O required to enact any changes to the contract, where $1 \leq m \leq n$.

2.3.2 Auditor

Auditor A represents a public key pk. The aforementioned public key is utilized to encrypt relevant transactional data, enabling auditability and compliance. pk can either correspond to a single secret key or a m-of-n secret keys, generated through the use of threshold cryptography. The latter approach can provide access to the encrypted data to multiple independent parties, if required, or act as a fail-safe in case of a loss of a secret key.

2.3.3 User

User U represents an identity I eligible to interact and mutate the state of the contract. To become a U, one is required to be added to the whitelist of the corresponding contract by O. To access the functionality of the contract, U is obliged to prove his/her identity I inclusion in the whitelist in zero-knowledge.

3 Abstract Standard

3.1 Transfers

Securities are permitted to be transferred between the whitelisted Us. The receiver R is required to publish an explicit agreement to the transfer within a predefined validity time, settling the transfer. The relevant transaction data, such as the sender identity I and the amount being transferred, is encrypted to the auditor A's public key, enabling trivial reconstruction of capitalization tables and ensuring that only eligible investors as defined by the Owners compliance criteria have been accepted. In case the transfer is not settled within the aforementioned viability time, the sender S is required to rollback any changes to his/her account, effectively negating the transfer.

3.2 Voting

Securities can provide the U with voting rights, which periodically culminates in shareholder votes on important issues. U may be eligible to vote in such events as long as U can prove the ownership of a security or, alternatively, be eligible to vote only if U's balance exceeds a predefined threshold. The vote transactions are encrypted to the auditor A's public key along with a commitment to the balance being voted with. The voting results are revealed after the expiration of the voting period through the decryption of the openings of the vote commitments and the publication of the openings to the commitment comprised of the sum of all individual vote commitments.

3.3 Dividend Claims

Securities can provide the U with rights to periodically receive dividends. U may be eligible to receive dividends as long as U can prove the ownership of a security. To claim a dividend, O is to publish a transaction corresponding to U, distributing the dividend to U.

4 Concrete Standard

The XSC contract is comprised of multiple obligatory and optional functions, accessible to O, A or U depending on the function.

4.1 Operations

The section defines the functionality overseeing the operations of the contract.

4.1.1 Pause Contract

The function enabling O to pause the contract operations until the functionality is explicitly resumed is:

$$\mathcal{F}_O^{\mathsf{PAUSE}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.1.2 Resume Contract

The function enabling O to resume the contract operations is:

$$\mathcal{F}_O^{\mathsf{RESUME}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{nonce}),$$

where $[\sigma_{pk_1}, ..., \sigma_{pk_m}]$ is a set of signatures corresponding to at least m public keys part of O and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.1.3 Delete Contract

The function enabling O to delete the contract is:

$$\mathcal{F}_O^{\mathsf{DELETE}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

NOTE: Deleting the contract DOES NOT remove the transaction history related to the contract. The function removes the contract from the global state and prevents anybody from accessing the contract following its deletion.

4.2 Access Control

The section defines the functionality overseeing the access control of the contract. Any changes to the access control of the contract can be enacted instantly or after a predefined cooldown period, with the latter acting as a fail-safe against contract takeover attempts in case of a theft of a secret key/s corresponding to the public keys in O.

4.2.1 Add Owner

The function enabling O to add new owner public key pk_a to O is:

$$\mathcal{F}_O^{\mathsf{ADD\_OWNER}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], pk_a, \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, pk_a is the public key of the owner to be added to O and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.2.2 Remove Owner

The function enabling O to remove existing owner public key pk_r from O is:

$$\mathcal{F}_O^{\mathsf{REMOVE\_OWNER}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], pk_r, \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, pk_r is the public key of the owner to be removed from O and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

NOTE: O is required to contain at least one public key at any given time. If the owner is trying to revoke access of the remaining public key/s and substitute the aforementioned public key/s with other public key/s, he/she is required to first add the latter public key/s to O before removing the former.

4.2.3 Set Governance

The function enabling O to set governance threshold for O is:

$$\mathcal{F}_O^{\mathsf{SET\_GOVERNANCE}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], g, \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, g is the governance threshold equivalent to the number of public keys in O with the corresponding signatures σ required to enact changes to the contract and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

NOTE: $1 \leq g \leq n$, where is the cardinality of O (i.e. $n \leftarrow |O|$).

4.2.4 Set Auditor

The function enabling O to set the public key pk_s of the auditor A:

$$\mathcal{F}_O^{\mathsf{SET\_AUDITOR}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], pk_s, \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, pk_s is the public key of the auditor A to be set and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.2.5 Set Cooldown

The function enabling O to set a cooldown period for access control functionality of the contract is:

$$\mathcal{F}_O^{\mathsf{SET\_COOLDOWN}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], t_{cooldown}, \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, $t_{cooldown}$ is a cooldown period and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

NOTE: $0 \leq t_{cooldown} \leq t_{cooldownmax}$, where $t_{cooldownmax}$ is the maximum permitted cooldown period.

4.3 Whitelist

The section defines the functionality overseeing the whitelist management of the contract. A whitelist can be individually managed per contract, or alternatively, act as a centralized repository of whitelisted identities I to which multiple contracts have access.

4.3.1 Add Whitelist

The function enabling O to add new identities I to the whitelist $\mathsf{whitelistTree}$ is:

$$\mathcal{F}_O^{\mathsf{ADD\_WHITELIST}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], [I_1, ..., I_n], \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, $[I_1, ..., I_n]$ is a set of identities to be added from the whitelist $\mathsf{whitelistTree}$ and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.3.2 Remove Whitelist

The function enabling O to remove existing identities I to the whitelist $\mathsf{whitelistTree}$ is:

$$\mathcal{F}_O^{\mathsf{REMOVE\_WHITELIST}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], [I_1, ..., I_n], \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, $[I_1, ..., I_n]$ is a set of identities to be removed from the whitelist $\mathsf{whitelistTree}$ and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.4 Transactions

The section defines the functionality overseeing the transfers of the asset represented by the contract.

4.4.1 Register

The function enabling U to register an account corresponding to identity I is:

$$\mathcal{F}_U^{\mathsf{REGISTER}}(\mathsf{whitelistTree}^R, \nu_I, M_{new}, \pi_{\mathsf{plonk}}^{\mathsf{create}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, ν_I is the nullifier of the identity I, M_{new} is the created Memory Slot and $\pi_{\mathsf{plonk}}^{\mathsf{create}}$ is the proof of the consistency of the relevant computation.

NOTE:

4.4.2 Send

The function enabling U to initiate a transfer of a security to receiver's one-time key pk is:

$$\mathcal{F}_U^{\mathsf{SEND}}(\mathsf{whitelistTree}^R, \mathsf{memorySlotTree}^R, \nu_{M_{old}}, M_{new}, C, \pi_{\mathsf{plonk}}^{\mathsf{send}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, M_{new} is the updated Memory Slot and $\pi_{\mathsf{plonk}}^{\mathsf{SEND}}$ is the proof of the consistency of the relevant computation.

4.4.3 Accept

The function enabling U to accept a transfer of b bonds is:

$$\mathcal{F}_U^{\mathsf{ACCEPT}}(\mathsf{whitelistTree}^R, \mathsf{memorySlotTree}^R, \mathsf{coinTree}^R, \nu_{M_{old}}, \nu_C, M_{new}, \pi_{\mathsf{plonk}}^{\mathsf{accept}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\mathsf{coinTree}^R$ is the root of the Coin Tree $\mathsf{coinTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, ν_C is the nullifier of the Coin C, M_{new} is the updated Memory Slot and $\pi_{\mathsf{plonk}}^{\mathsf{accept}}$ is the proof of the consistency of the relevant computation.

4.4.4 Settle

The function enabling U to settle a transfer of b bonds is:

$$\mathcal{F}_U^{\mathsf{SETTLE}}(\mathsf{whitelistTree}^R, \mathsf{memorySlotTree}^R, \mathsf{coinTree}^R, \nu_{M_{old}}, \nu^C, M_{new}, \pi_{\mathsf{plonk}}^{\mathsf{settle}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\mathsf{coinTree}^R$ is the root of the Coin Tree $\mathsf{coinTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, ν_C is the nullifier of the Coin C, M_{new} is the updated Memory Slot and $\pi_{\mathsf{plonk}}^{\mathsf{settle}}$ is the proof of the consistency of the relevant computation.

4.4.5 Redeem

The function enabling U to redeem a transfer of b bonds is:

$$\mathcal{F}_U^{\mathsf{REDEEM}}(\mathsf{whitelistTree}^R, \mathsf{memorySlotTree}^R, \mathsf{coinTree}^R, \nu_{M_{old}}, \nu_C, M_{new}, \pi_{\mathsf{plonk}}^{\mathsf{claim}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\mathsf{coinTree}^R$ is the root of the Coin Tree $\mathsf{coinTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, ν_C is the nullifier of the Coin C, M_{new} is the updated Memory Slot and $\pi_{\mathsf{plonk}}^{\mathsf{claim}}$ is the proof of the consistency of the relevant computation.

4.4.6 Force Transfer

The function enabling O to force transfers from one account to another is:

$$\mathcal{F}_O^{\mathsf{FORCE\_TRANSFER}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{memorySlotTree}^R, \mathsf{coinTree}^R, \nu_{M_{old,S}}, \nu_{M_{old,R}}\nu_C, M_{new,S}, M_{new,R}, \pi_{\mathsf{plonk}}^{\mathsf{forceTransfer}}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\mathsf{coinTree}^R$ is the root of the Coin Tree $\mathsf{coinTree}$, $\nu_{M_{old,S}}$ is the nullifier of the previous Memory Slot of the sender S $M_{old,S}$, $\nu_{M_{old,R}}$ is the nullifier of the previous Memory Slot of the receiver R $M_{old,R}$, ν_C is the nullifier of the Coin C, $M_{new,S}$ is the updated Memory Slot of the sender S, $M_{new,R}$ is the updated Memory Slot of the receiver R and $\pi_{\mathsf{plonk}}^{\mathsf{forceTransfer}}$ is the proof of the consistency of the relevant computation.

NOTE: In case of a forced transfer, the settlement of a transfer is instantenious and does not require an explicit approval from the receiver of the transfer.

4.4.7 Update Account

The function enabling O to update accounts in case of a loss of the corresponding secret key or else is:

$$\mathcal{F}_O^{\mathsf{UPDATE\_ACCOUNT}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{memorySlotTree}^R, \nu_{M_{old}}, M_{new}, \pi_{\mathsf{plonk}}^{\mathsf{updateAccount}}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, M_{new} is the updated Memory Slot and $\pi_{\mathsf{plonk}}^{\mathsf{updateAccount}}$ is the proof of the consistency of the relevant computation.

4.5 Voting

The section defines the functionality overseeing the voting procedures of the contract.

4.5.1 Initiate Voting

The function enabling O to initiate voting is:

$$\mathcal{F}_O^{\mathsf{INITIATE\_VOTING}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], [\mathsf{subject}_1, ..., \mathsf{subject}_n], h_{votestart}, h_{voteend}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, $[\mathsf{subject}_1, ..., \mathsf{subject}_n]$ is the set of subjects to voted upon, $h_{votestart}$ is the block height indicating the start of the voting and $h_{voteend}$ is the block height indicating the end of the voting.

4.5.2 Vote

The function enabling U to case a vote for the corresponding subjects is:

$$\mathcal{F}_U^{\mathsf{VOTE}}(\mathsf{whitelistTree}^R, \mathsf{memorySlotTree}^R, \nu_{M_{old}}, M_{new}, [\mathsf{vote}_1, ..., \mathsf{vote}_n], \pi_{\mathsf{plonk}}^{\mathsf{vote}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, M_{new} is the updated Memory Slot, $[\mathsf{vote}_1, ..., \mathsf{vote}_n]$ is a set of votes corresponding to subjects $[\mathsf{subject}_1, ..., \mathsf{subject}_n]$ and $\pi_{\mathsf{plonk}}^{\mathsf{vote}}$ is the proof of the consistency of the relevant computation.

4.5.3 Voting Result

The function enabling O to publish the results of the vote is:

$$\mathcal{F}_O^{\mathsf{VOTING\_RESULT}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], [\mathsf{voteResult}_1, ..., \mathsf{voteResult}_n]),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O and $[\mathsf{voteResult}_1, ..., \mathsf{voteResult}_n]$ is a set of voting results corresponding to subjects $[\mathsf{subject}_1, ..., \mathsf{subject}_n]$.

4.6 Dividend Claims

The section defines the functionality overseeing the dividend distribution procedures of the contract.

4.6.1 Initiate Dividend Payout

The function enabling O to initiate dividend payout is:

$$\mathcal{F}_O^{\mathsf{INITIATE\_PAYOUT}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{payout}, h_{payoutstart}, \mathsf{nonce}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O, *payout* is the ratio to be payed per one unit of security, $h_{payoutstart}$ is the block height indicating the start of the payout period, and nonce is an integer corresponding to the in-contract counter protecting against replay attacks.

4.6.2 Distribute Dividend

The function enabling O to distribute dividends to the eligible receivers is:

$$\mathcal{F}_O^{\mathsf{DISTRIBUTE}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{memorySlotTree}^R, \nu_{M_{old}}, M_{new}, \mathsf{payoutData}, \pi_{\mathsf{plonk}}^{\mathsf{pushDividend}}),$$

where $[\sigma_{pk_1}, ..., \sigma_{pk_m}]$ is a set of signatures corresponding to at least m public keys part of O, $\mathsf{memorySlotTree}^R$ is the root of the Memory Slot Tree $\mathsf{memorySlotTree}$, $\nu_{M_{old}}$ is the nullifier of the previous Memory Slot M_{old}, M_{new} is the updated Memory Slot, $\mathsf{payoutData}$ is the relevant data corresponding to the dividend payout and $\pi_{\mathsf{plonk}}^{\mathsf{pushDividend}}$ is the proof of the consistency of the relevant computation.

NOTE: The dividend can be either distributed through supply dilution or with other currencies/tokens depending on the distribution parameters defined by O.

4.7 Supply Management

The section defines the functionality overseeing the optional procedures of the contract.

4.7.1 Dilute Supply

The function enabling O to dilute the supply of the securities is:

$$\mathcal{F}_O^{\mathsf{DILUTE\_SUPPLY}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{dilutionAmount}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O and $\mathsf{dilutionAmount}$ is the amount supply is to be diluted by.

NOTE: The owner O has an option to sell the supply difference through an on-chain auction.

4.7.2 Buyback

The function enabling O to buyback a portion of the circulating supply is:

$$\mathcal{F}_O^{\mathsf{BUYBACK}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{buybackTerms}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O and $\mathsf{buybackTerms}$ is the set of values defining the buyback terms.

NOTE: The owner O has an option to buy a portion of the circulating supply through an on-chain auction.

4.8 Communication

The section defines the functionality overseeing the communication procedures of the contract.

4.8.1 Announce

The function enabling O to publish an announcement is:

$$\mathcal{F}_O^{\mathsf{ANNOUNCE}}([\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}], \mathsf{announcement}),$$

where $[\sigma_{\mathsf{schnorr},1}, ..., \sigma_{\mathsf{schnorr},m}]$ is a set of signatures corresponding to at least m public keys part of O and $\mathsf{announcement}$ is an announcement.

NOTE: The **Announce** function does not communicate the announcement contents, instead notifying users U of the announcement by publishing the unique identifier of the announcement corresponding to the contents of the announcement accessible in the browser.

4.8.2 Message

The function enabling U to communicate to other Us is:

$$\mathcal{F}_U^{\mathsf{MESSAGE}}(\mathsf{whitelistTree}^R, \mathsf{message}, \pi_{\mathsf{plonk}}^{\mathsf{message}}),$$

where $\mathsf{whitelistTree}^R$ is the root of the Whitelist Tree $\mathsf{whitelistTree}$, $\mathsf{message}$ is the message being sent and $\pi_{\mathsf{plonk}}^{\mathsf{message}}$ is the proof of the consistency of the relevant computation.

NOTE: The **Message** function does not communicate the message contents, instead notifying user U of the message by publishing the unique identifier of the message corresponding to the contents of the message and the encypted shared decryption key making the message accessible in the browser.

4.9 Queries

The section defines the functionality overseeing the queries of the contract accessible to the protocol participants.

4.9.1 Get Contract Status

The function enabling one to enquire about the contract status is:

$$\texttt{bool} \leftarrow \mathcal{Q}^{\mathsf{CONTRACT\_STATUS}}(),$$

where `bool` is the indication that the contract is either active (i.e. `1`) or paused (i.e. `0`).

4.9.2 Get Supply

The function enabling one to enquire about the supply is:

$$\mathsf{supply} \leftarrow \mathcal{Q}^{\mathsf{SUPPLY}}(),$$

where supply is the circulating supply of the the security.

4.9.3 Get Owner

The function enabling one to enquire about the set of public keys corresponding to O is:

$$[pk_1, ..., pk_n] \leftarrow \mathcal{Q}^{\mathsf{OWNER}}(),$$

where $[pk_1, ..., pk_n]$ is the set of public keys corresponding to O.

4.9.4 Get Auditor

The function enabling one to enquire about the public key of the auditor A is:

$$pk_a \leftarrow \mathcal{Q}^{\mathsf{AUDITOR}}(),$$

where pk_a is the public key corresponding to auditor A.

4.9.5 Get Cooldown Period

The function enabling one to enquire about the cooldown period $\mathsf{cooldown}$ is:

$$\mathsf{cooldown} \leftarrow \mathcal{Q}^{\mathsf{COOLDOWN}}(),$$

where $\mathsf{cooldown}$ is the current cooldown period.

4.9.6 Get Whitelist

The function enabling one to enquire about the identities in whitelist $\mathsf{whitelistTree}$ is:

$$[I_1, ..., I_n] \leftarrow \mathcal{Q}^{\mathsf{WHITELIST}}(),$$

where $[I_1, ..., I_n]$ is a set of identities in the whitelist $\mathsf{whitelistTree}$.

4.9.7 Get Identity

The function enabling one to enquire about the inclusion of the identity I to the whitelist $\mathsf{whitelistTree}$ is:

$$\texttt{bool} \leftarrow \mathcal{Q}^{\mathsf{IDENTITY}}(I),$$

where `bool` is the indication that the identity I either belongs to whitelist $\mathsf{whitelistTree}$ (i.e. `1`) or not (i.e. `0`).

4.9.8 Get Voting Subjects

The function enabling one to enquire about the voting subjects is:

$$[\mathsf{subject}_1, ..., \mathsf{subject}_n] \leftarrow \mathcal{Q}^{\mathsf{VOTING\_SUBJECTS}}(),$$

where $[\mathsf{subject}_1, ..., \mathsf{subject}_n]$ is a set of subjects being voted on.

4.9.9 Get Voting Result

The function enabling one to enquire about the voting result to the subjects $[\mathsf{subject}_1, ..., \mathsf{subject}_n]$ is:

$$[\mathsf{voteresult}_1, ..., \mathsf{voteresult}_n] \leftarrow \mathcal{Q}^{\mathsf{VOTING\_RESULT}}([\mathsf{subject}_1, ..., \mathsf{subject}_n]),$$

where $[\mathsf{voteresult}_1, ..., \mathsf{voteresult}_n]$ is the voting results to the subjects $[\mathsf{subject}_1, ..., \mathsf{subject}_n]$.

5 Contract Upgradeability

While contracts deployed on the Dusk Network protocol and the transactions interacting with the contract contents are immutable, there exist multiple approaches enabling limited upgradeability of the contracts.

A proxy contract implies that the deployed XSC-compliant contract consists of two distinct contracts:

1. Storage Contract,
2. Logic Contract.

Both approaches imply that in case of an upgrade, the contract storage does not have to be migrated and will remain accessible to the new Logic Contracts.

5.1 Storage Proxy Contract

In case of the Storage Contract acting as proxy, the Storage Contract acts as an entry point to the XSC-compliant contract for the users interacting with the aforementioned contract while the Logic Contract is accessed through the inter-contract calls. The use the Storage Contract as a proxy means that in case the contract logic is to be modified, the contract owner only has to update the path to the new Logic Contract abstracting the users from the upgrade.

5.2 Logic Proxy Contract

In case of the Logic Contract acting as proxy, the Logic Contract acts as an entry point to the XSC-compliant contract for the users interacting with the aforementioned contract while the Storage Contract is accessed through the inter-contract calls. The use the Logic Contract as a proxy means that in case the contract logic is to be modified, the users would have to call the updated contract, rendering the previous contract access point inaccessible.

6 Upcoming Features

6.1 Inter-Contract Transfers

In the upcoming version of the XSC standard, we will present additional functionality capable of facilitating inter-contract transfers of securities. Specifically, the features will permit the O to whitelist contract addresses permitted to receive and send the security corresponding to the XSC in question. Such feature will permit the creation of Layer 2 (L2) solutions, such as an L2 Decentralized Exchange (DEX).

L2 protocols are a scalability solution for blockchain-based protocols, creating a separate data and settlement layer to the Layer 1 (L1) protocol (e.g. Dusk Network) while leveraging the security of the underlying L1.

DEX is an non-custodian exchange protocol, utilizing an on-chain orderbook with an on-/off-chain matching engine, or alternatively a constant product formula (such as Uniswap [AZR20]) to facilitate the exchange of assets.

References

[DH76] Whitfield Diffie and Martin E. Hellman. "New directions in cryptography". In: *IEEE Trans. Information Theory* 22 (1976), pp. 644–654.

[Ped91] Torben P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *CRYPTO*. 1991.

[Sab13] Nicolas van Saberhagen. "CryptoNote v 2.0". In: (2013).

[GWC19] A. Gabizon, Zachary J. Williamson, and Oana Ciobotaru. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 953.

[Gra+19] L. Grassi et al. "Starkad and Poseidon: New Hash Functions for Zero Knowledge Proof Systems". In: *IACR Cryptol. ePrint Arch.* 2019 (2019), p. 458.

[AZR20] Hayden Adams, Noah Zinsmeister, and Dan Robinson. *Uniswap v2 Core*. Available at `https://uniswap.org/whitepaper.pdf`. 2020.

[Kho20] Dmitry Khovratovich. *Encryption with Poseidon*. Available at `https://drive.google.com/file/d/1EVrP3DzoGbmzkRmYnyEDcIQcXVU7GlOd/view`. 2020.

[BHW] Hopwood Sean Bowe, T. George Hornby, and Nathan Wilcox. *Zcash Protocol Specication*. Available at `https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf`.