Confidential Security Contract (XSC) Standard

Toghrul Maharramov
Dusk Network
toghrul@dusk.network

Version 1.0

Contents

1 Introduction

The goal of the document is to outline the Confidential Security Contract (**XSC** in further context) standard which forms the basis of the security tokenization procedure on top of the Dusk Network protocol. Acting as an extension to DER-1 (Dusk Network Enhancement Recommendation 1), this document is split into three sections - **Key Notations**, **Abstract Standard** and **Concrete Standard**.

Key Notations section defines the keywords and mathematical symbols used to abstract the reader from the underlying protocol.

Abstract Standard section defines the non-technical standard.

Concrete Standard section defines the technical standard.

Wallet Implementation section defines the additional requirements for the wallet implementations.

2 Key Notations

2.1 User Definitions

The standard defines four distinct *roles* of users:

1. **Contract Owner** - A user registered as a contract owner. One is implicitly registered as a **Contract Owner** of an XSC through an addition of the *public key* into the compulsory `owner_address` field of the contract.
2. **Third Party** - Three types of bodies overseeing the contract:
   (a) **View Only** - An overseeing regulatory body in a relevant jurisdiction with an ability to access the encrypted data.
   (b) **Co-sign** - An overseeing regulatory body in a relevant jurisdiction with an ability to co-sign on the `supplyMint` and `transferForce` methods.
   (c) **Guard** - An overseeing regulatory body in a relevant jurisdiction with an ability to transfer the ownership of a contract to a new address.
3. **Contract User** - A user eligible to interact and change the state of the contract. The eligibility requirements are defined by the **Contract Owner**.
4. **Platform User** - A Dusk Network protocol user and the user eligible to access the state of the contract. Labeled as *user* when the context is clear.

2.2 Transaction Definitions

The standard defines two distinct transaction types:

1. **Invocation Transaction** - A transaction invoking a contract method which changes the state of the contract. An **Invocation Transaction** has to be recorded on the ledger and is exclusive to the **Contract Owner**, **Third Parties** (except **View Only**) and the **Contract Users**.
2. **Call Transaction** - A transaction calling a contract method which doesn't change the state of the contract. A **Call Transaction** DOES NOT have to be recorded on the ledger.

2.3 User Authentication and Communication Definitions

1. **Address** - A derivative of the master secret key `msk`, an *address* is a one-time public key included in the transaction output enabling the user in possession of the matching secret key to spend the output.
2. **Identity** - A derivative of the master secret key `msk`, an *identity* is included in the `whitelist`.

3. **View Key** - A *view key* enables the decryption of the transaction data without an ability of the "viewer" to spend the output. The transaction data is encrypted with *viewkey* $\texttt{vk} = r \cdot \texttt{identity}$ with $R = r \cdot G$ included in the transaction.
    (a) **Single User View Key** - A single user view key is an encryption/decryption key which can be derived through a single key pair.
    (b) **Multi User View Key** - A multi user view key is an encryption/decryption key which can be derived through multiple key pairs, for example: $\texttt{vsk} = r \cdot \texttt{identity} \cdot \texttt{contractOwner}$.

2.4 Method Definitions

Methods are highlighted in colour: red if exclusive to the **Contract Owner**, green if exclusive to the **Contract Owner** and **Third Parties** (**Co-sign**), violet if exclusive to the **Contract Owner** and **Third Parties** (**Guard**), blue if exclusive to the **Contract Owner** and **Contract Users** and black if available to all users.

2.5 Functions

$\Theta()$ - is an abstract transaction state function (in a concrete implementation, the values highlighted below are obfuscated and proven in zero-knowledge), which includes the following four values:

1. o - the originator of the transaction.
2. i_o - the identity of the originator of the transaction.
3. *input* - the input/s of the transaction.
4. b - the asset balance of the transaction input.
5. b_{DUSK} - the DUSK balance of the transaction input.

$\pi \leftarrow \mathcal{ZP}(x, y)$ - a zero-knowledge proof generation function. Takes public inputs x and witness w to produce proof π.
$\texttt{commitment} \leftarrow \mathcal{COM}^G(x, r)$ - a Pedersen Commitment generation function in group G and randomness r.

2.6 Zero-knowledge Proofs

2.6.1 Proof Generation Functions

$\mathcal{SI}_G(\texttt{set}, \texttt{element})$ - a set inclusion proof which takes `set` and `element` as private inputs and outputs proofs π.
$\mathcal{SK}_G(\texttt{sk})$ - a secret key knowledge proof which takes secret key `sk` as a private input and outputs proof π.
$\mathcal{E}_G(\texttt{plaintext})$ - an encryption correctness proof which takes `plaintext` as a

private input and outputs proof π.
$\mathcal{D}_G(\texttt{plaintext})$ - an encrypted value knowledge proof which takes `plaintext` as a private input and outputs proof π.
$\mathcal{ARP}_G(\texttt{value})$ - an arbitrary value range proof which takes `value` as a private input and outputs proof π.
$\mathcal{RP}_G(\texttt{value}, \pi)$ - a range proof verification which takes `value` as a private input and outputs proof π.

2.6.2 Proof Verification Functions

$\mathcal{SI}_V(\texttt{set}, \pi)$ - a set inclusion proof verification which takes `set` and proof π as public inputs and outputs 1 if the proof has been generated correctly.
$\mathcal{SK}_V(\texttt{pk}, \pi)$ - a secret key knowledge proof verification which takes public key `pk` and proof π as public inputs and outputs 1 if the proof has been generated correctly.
$\mathcal{E}_V(\texttt{encryptedvalue}, \pi)$ - an encryption correctness proof verification which takes `encryptedvalue` and proof π as public inputs and outputs 1 if the proof has been generated correctly.
$\mathcal{D}_V(\texttt{encryptedvalue}, \pi)$ - an encrypted value knowledge proof verification which takes `encryptedvalue` and proof π as public inputs and outputs 1 if the proof has been generated correctly.
$\mathcal{ARP}_V(\pi)$ - an arbitrary value range proof verification which takes proof π as a public input and outputs 1 if the proof has been generated correctly.
$\mathcal{RP}_V(\pi)$ - a range proof verification which takes proof π as a public input and outputs 1 if the proof has been generated correctly.

3 Abstract Standard

3.1 Compulsory Methods

3.1.1 Supply

The total supply of a particular contract is defined during the contract initiation (outlined in the upcoming **Dusk Network Protocol Specification** paper). The **Contract Owner** may choose to decrease and increase a supply through a buyback of the existing digital shares or the minting (though only with approval of the **Third Party** (**Co-sign**)) of new digital shares respectively. Please note that the standard will provide the **Contract Owner** with multiple buyback options, including set-price purchase of digital shares or Dutch Auction. More details will be provided in the upcoming iterations of the document.

3.1.2 Contract Ownership and Interaction

Contract Owner represents a single or multiple public keys with unique privileges (outlined in **Section 4** of the document). **Contract Owner** may choose to

transfer the ownership of the contract to a different public key or share the ownership with other entity/ies. The governance model enables the **Third Parties** (**Guard**) to transfer the ownership of the contract to the new *publickey* without the need for the **Contract Owner** approval in case of a loss of the *private key* corresponding to the existing `owner_address` of the contract respectively.

3.1.3 Whitelist

The **Contract Owner** has an ability to add a set identities into a whitelist. A **Platform User** can only become a **Contract User** iff his/her identity (the definition of identity is dependent on the regulatory standards of the relevant jurisdiction, with a default being the *public view key* [outlined in the upcoming **Dusk Network Protocol Specification** paper] of the **Contract User**) is a member of a whitelist. The user is required to prove inclusion of his/her identity in the whitelist in zero-knowledge in order to invoke the **Contract User**-exclusive methods.

3.1.4 Transfers

Multiple types of transfer methods are available in the XSC standard. One transfer method enables the transfer of digital shares from one **Contract User** to another. Additional transfer method is exclusive to the **Contract Owner** and **Third Parties** (**Co-sign**), enabling forced transfer of digital shares.

3.1.5 Dividends

The dividends in a particular contract are paid out by the **Contract Owner**. The aforementioned action is accomplished in zero-knowledge, meaning that neither information about the dividend amount payed out nor the receiver of the payment is leaked.

3.2 Optional Methods

3.2.1 Voting Procedures

Digital shares can often provide the user with voting rights, which periodically culminates in shareholder votes on important issues. The **Contract Users** may be eligible to vote in such events as long as the **Contract User** can prove an ownership of at least 1 digital share or, alternatively, be eligible to vote only if the **Contract User**'s balance exceeds a certain minimum threshold of points. The vote transactions are encrypted to either the **Contract Owner**'s or the **Third Parties**'s (**View**) (or both) addresses, which are decrypted by the aforementioned parties after the expiration of the voting period before the outcome of the vote is made public. The vote can be cast only once per share and the individual votes are required to remain anonymous.

3.2.2 Messages

The **Contract Owner**, **Third Parties** or the **Contract User** may choose to send private messages to other eligible parties. The functionality can be enabled through end-to-end encryption, enabling the users to communicate with each other.

3.2.3 Alerts

The **Contract Owner** or the **Third Parties** may have an option to issue encrypted alerts, either prior to the shareholder voting disclosing the information or propagate urgent information.

4 Concrete Standard

4.1 Compulsory Methods

4.1.1 `function totalSupply`

A method to provide *user* with information about the total supply of the digital shares of the contract. The total supply is a sum of the circulating supply and the treasure supply.

4.1.2 `function supplyBuyback`

A method to buyback a portion of the circulating digital shares from the **Contract Users**. **NOTE**: The details of the method will be specified in the future iterations of the document.

Function $\mathcal{F}_{\texttt{supplyBuyback}}$

$\mathcal{F}_{\texttt{supplyBuyback}}$.Create(int64 value, int64 $\texttt{r}_{begin}$, int64 $\texttt{r}_{end}$, type type):

1. Verify $(\Theta(o) == \texttt{conctractOwner}) \cap (\texttt{totalSupply} > 0) \cap (0 < \texttt{value} \leq \texttt{totalSupply}) \cap (\texttt{r}_{begin} \geq \mathcal{S}(r)) \cap (\texttt{r}_{end} > \texttt{r}_{begin}) \cap (\texttt{type} \in T_{AUCTION}) \cap (\texttt{auction} \neq (1, \cdot, \cdot, \cdot, \cdot))$.
2. If true, set auction tuple $(0, \cdot, \cdot, \cdot, \cdot)$ to (1,type,value,$\texttt{r}_{begin}$,$\texttt{r}_{end}$).

$\mathcal{F}_{\texttt{supplyBuyback}}$.Participate(bid bid, int64 amount):

1. Verify $(\Theta(o) \in \texttt{whitelist}) \cap (\texttt{auction} == (1, \cdot, \cdot, \cdot, \cdot)) \cap (\Theta(b) \geq \texttt{amount}) \cap (0 < \texttt{value} \leq \texttt{auction.amount}) \cap (\mathcal{S}(r) \geq \texttt{auction.r}_{begin}) \cap (\mathcal{S}(r) \leq \texttt{auction.r}_{end})$.
2. If true, add tuple (bid, amount, address) to the bid set $\mathcal{B}$.

$\mathcal{F}_{\texttt{supplyBuyback}}$.End():

1. Verify $(\Theta(o) == \texttt{conctractOwner}) \cap (\texttt{auction} == (1, \cdot, \cdot, \cdot, \cdot)) \cap (\mathcal{S}(r) \geq \texttt{r}_{end}) \cap (\Theta(b_{DUSK}) \geq \sum_{i=0}^{n} (\texttt{winner}_i(\mathcal{B}).\texttt{bid}) \cdot (\texttt{winner}_i(\mathcal{B}).\texttt{amount}))$.
2. If true, for $0 \leq i < n$, create an output tuple $(\mathcal{COM}^G(\texttt{winner}_i(\mathcal{B}).\texttt{bid} \cdot \texttt{winner}_i(\mathcal{B}).\texttt{amount}, \texttt{blinder}_i), \texttt{winner}_i(\mathcal{B}).\texttt{address})$.
3. Set $\texttt{totalSupply} = \texttt{totalSupply} - \sum_{i=0}^{n} \texttt{winner}_i(\mathcal{B}).\texttt{amount}$.
4. Set auction tuple to $(0, \cdot, \cdot, \cdot, \cdot)$.

4.1.3 function supplyMint

A method to mint new digital shares and increase the total supply.

Function $\mathcal{F}_{\texttt{supplyMint}}$

$\mathcal{F}_{\texttt{supplyMint}}(\texttt{int64 value}, \texttt{signature sig}_{\texttt{cosign}})$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cap (\texttt{sig}_{\texttt{cosign}} = TRUE) \cap (\texttt{value} > 0 \cap 0 < \texttt{totalSupply} + \texttt{value} < 2^{64})$.
2. $\texttt{totalSupply} = \texttt{totalSupply} + \texttt{value}$.

4.1.4 function contractOwner

A method to return the *address* of the **Contract Owner** `contractOwner`.

4.1.5 function contractCreate

A method to create an executable contract on the storage. The compiled contract is added to the state via a special contract creation transaction (outlined in the upcoming **Dusk Network Protocol Specification** paper), however, the contract CANNOT be interacted with until the `contractCreate` method is invoked. The `contractCreate` method can be invoked once per contract.

Function $\mathcal{F}_{\texttt{contractCreate}}$

$\mathcal{F}_{\texttt{contractCreate}}(\texttt{address contractAddress})$:

1. Verify $(\texttt{contractAddress} \in \texttt{state}) \cap (\Theta(o) == \texttt{contractOwner}) \cap (\texttt{contractStatus.created} == 0)$.
2. Set $\texttt{contractOwner} = \Theta(o)$, $\texttt{contractStatus.created} = 1$ and $\texttt{contractStatus.active} = 1$.

4.1.6 function contractResume

Function $\mathcal{F}_{\texttt{contractResume}}$

$\mathcal{F}_{\texttt{contractResume}}()$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cap (\texttt{contractStatus.active} == 0)$.
2. If true, set $\texttt{contractStatus.active} = 1$.

4.1.7 function contractPause

A method to pause the execution of the state-modifying calls (i.e. **Invocation Transactions**). The underlying protocol is still able to execute non-state-modifying calls (i.e. **Call Transactions**). `contractResume` method supersedes the discussed method.

Function $\mathcal{F}_{\texttt{contractPause}}$

$\mathcal{F}_{\texttt{contractPause}}()$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cap (\texttt{contractStatus.active} == 1)$.
2. If true, set $\texttt{contractStatus.active} = 0$.

4.1.8 function contractStatus

A method to provide *user* with information about the contract status. The method consists of two `bool` values - `created` and `active`.

4.1.9 function contractOwnershipTransfer

A method to transfer the ownership of the contract to a new *address*.

Function $\mathcal{F}_{\texttt{contractOwnershipTransfer}}$

$\mathcal{F}_{\texttt{ContractOwnershipTransfer}}(\texttt{address adrress})$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cup (\Theta(o) == \texttt{contractGuard})$.
2. If true, set $\texttt{contractOwner} = \texttt{address}$.

4.1.10 function addToWhitelist

A method to add an *identity* to the `whitelist`.

Function $\mathcal{F}_{\texttt{addToWhitelist}}$

$\mathcal{F}_{\texttt{addToWhitelist}}(\texttt{identity identity}_i)$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cap (\texttt{identity}_i \notin \texttt{whitelist})$.
2. If true, append $\texttt{identity}_i$ to the `whitelist`.

4.1.11 function removeFromToWhitelist

A method to remove an *identity* from the `whitelist`.

Function $\mathcal{F}_{\texttt{removeFromWhitelist}}$

$\mathcal{F}_{\texttt{removeFromWhitelist}}(\texttt{identity identity}_i)$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cap (\texttt{identity}_i \in \texttt{whitelist})$.
2. If true, remove $\texttt{identity}_i$ to the `whitelist`.

4.1.12 function whitelist

A method to provide *user* with information about the contents of the `whitelist`.

4.1.13 function transferTo

Function $\mathcal{F}_{\texttt{transferTo}}$

$\mathcal{F}_{\texttt{transferTo}}(\texttt{proof}\pi, \texttt{address address}, \texttt{commitment commitment})$:

1. Verify $(\pi = (\Theta(i_o) \in \texttt{whitelist}) \cap (\Theta(input).\texttt{address} == \texttt{sk} \cdot G) \cap (\texttt{commitment} == \mathcal{COM}^G(\texttt{value}, \texttt{blinder})) \cap (0 \leq \texttt{value} < 2^{64}) \cap (\texttt{balance}(\Theta(b), \texttt{value}))) == TRUE$, where secret key `sk` is corresponding to `address`.
2. If true, create an output tuple $\langle \texttt{commitment}, \texttt{address} \rangle$.

4.1.14 function transferForce

In case of an illegal transaction being added to the state, the **Contract Owner** with an authorization of the **Third Party (Co-sign)** is allowed to force a transaction to undo the aforementioned illegal transaction.

Function $\mathcal{F}_{\texttt{transferForce}}$

$\mathcal{F}_{\texttt{transferForce}}(\texttt{proof}\,\pi, \texttt{address address}_i, \texttt{address address}_j, \texttt{commitment commitment}, \texttt{signature sig}_{\texttt{co-sign}})$:

1. Verify $(\Theta(o) == \texttt{contractOwner}) \cap (\texttt{sig}_{\texttt{co-sign}} = TRUE) \cap \langle(\pi = (\texttt{address}_i = \mathcal{SA}(\texttt{identity}_i, \texttt{randomness}_i)) \cap (\texttt{identity}_i \in \texttt{whitelist}) \cap (\texttt{address}_j = \mathcal{SA}(\texttt{identity}_j, \texttt{randomness}_j)) \cap (\texttt{identity}_j \in \texttt{whitelist}) \cap (\texttt{commitment} = \mathcal{COM}(\texttt{value}, \texttt{blinder})) \cap (0 \leq \texttt{value} < 2^{64})) == TRUE\rangle$.
2. If true, create an output tuple $\langle \texttt{commitment}, \texttt{address}_j \rangle$.

4.1.15 function dividendPush

4.2 Optional Methods

4.2.1 function contractDelete

A method to remove a contract from the blockchain. **Remove** means that the **Platform Users** will lose the access to the aforementioned Contract and the contract is deleted from the state, however, a user could reconstruct the state of the contract up until the last state before the delete method had been called.

Function $\mathcal{F}_{\texttt{contractDelete}}$

$\mathcal{F}_{\texttt{contractDelete}}()$:

1. Verify $(\Theta(o) == \texttt{contractOwner})$.
2. If true, $\texttt{contractStatus.created} = 0$ and $\texttt{storage} = \varnothing$.

4.2.2 function voteSend

A method to vote throughout the shareholder voting sessions. The **Contract Users** are permitted to case one vote once per digital share. The votes are encrypted to the addresses of the **Contract Owner** and the **Third Party/ies**.

Function $\mathcal{F}_{\texttt{voteSend}}$

$\mathcal{F}_{\texttt{voteSend}}(\texttt{proof}\pi, \texttt{string vote})$:

1. Verify $(\pi = (\texttt{identity} \in \texttt{whitelist}) \cap (\Theta(input).\texttt{address} == \texttt{sk} \cdot G) \cap (\texttt{vote} = \texttt{enc}(\texttt{data}, \Theta(b)) == TRUE$.
2. If true, add to the vote list `votes`.

4.2.3 function voteResult

A method to publish the voting results. After the expiration of the voting period, the **Contract Owner** constructs a zero-knowledge proof of the calculation of the voting outcome and includes the proof alongside the outcome in the `voteResult` message.

4.2.4 function message

Function $\mathcal{F}_{\texttt{message}}$

$\mathcal{F}_{\texttt{message}}(\texttt{proof}\pi, \texttt{address}, \texttt{string message})$:

1. Verify $(\pi = \texttt{identity} \in \texttt{whitelist}) == TRUE \cup (\Theta(o) = \texttt{contractOwner})$.
2. If true, save to the state.

4.2.5 function alert

Function $\mathcal{F}_{\texttt{alert}}$

$\mathcal{F}_{\texttt{alert}}(\texttt{string alert})$:

1. Verify $\Theta(o) == \texttt{contractOwner}$.
2. If true, propagate the `alert`.

5 Wallet Implementation

Aside from the generic processes, such as transfer capability, key management, etc., the user expects the wallet to fulfill, an XSC-compatible wallet has to include an array functionality to specifically support the standard.

5.1 Proof Generation

Zero-knowledge proofs represent an integral part of the XSC standard and are vital to the seamless performance of the protocol. An XSC-compatible wallet implementation is expected to include zero-knowledge proof generation functionality $\pi \leftarrow \mathcal{ZP}(x, y)$. The aforementioned functionality has to support the following proof types: $\mathcal{SI}$ (could be both Merkle Tree-based set or a linear set), $\mathcal{SK}$, $\mathcal{E}$, $\mathcal{ARP}$ and $\mathcal{RP}$.

5.2 Adhering to the Rules Defined in a Contract

The wallet implementation has to be capable of interpreting the smart contract code and impose the rules defined in the contract on the user. For example, the user is allowed to interact with the specific methods of the contract depending on the role of the user defined in contract.