Suterusu architecture

Abstract. This work describes how a Mainchain user could exploit the Suterusu confidential transfer module to anonymize the mainchain transaction.

1 General framework

As shown in Fig. 1, a sender selects the mainchain coin from his wallet and runs a **Fund** smart contract that converts the mainchain coin into the second-layered coin Suter. The **Fund** contract will be run on the mainchain network. The sender initiates this **Fund** smart contract by running **CreateAddress** to create a Suter account and then run **CreateFundTx** algorithm to create a transaction that runs the **Fund** smart contract which converts the mainchain coin in his wallet to the Suter tokens and then adds these Suter tokens to the newly created Suter account.

After the **Fund** contract is confirmed, the Suter token will be anonymized by running the Suterusu confidential transfer contract. Here the Suter confidential transfer contract will guarantee the anonymity of the sender and receiver and also the transfer amount confidentiality. The sender initiates the confidential transfer transaction by running **CreateTransferTx** to create a transaction that triggers the Suter confidential transfer smart contract, i.e., **Transfer**.

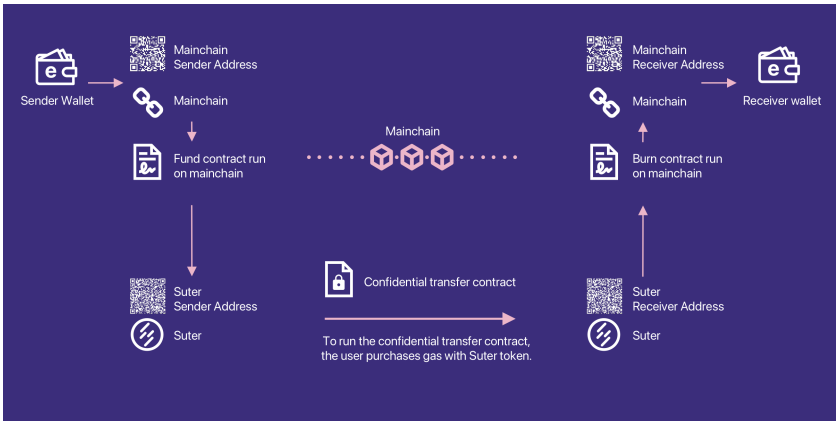

Fig. 1. Suter architecture

Finally, a burn contract will convert the anonymized Suter tokens back into the mainchain coin, which will be stored in the receiver wallet. The sender runs

CreateBurnTx algorithm to trigger the **Burn** smart contract to convert the Suter token in the anonymized Suter account into the mainchain coin in the receiver account.

The following describes how these contacts works and the respective user algorithms interact with these contracts.

Suter contracts

The Suter payment framework mainly runs the following three smart contracts to realize the confidential payment for smart contract platforms.

- **Fund**. Anybody can fund an account by specifying the public key y and transferring some mainchain coin. Fund converts the mainchain coin into Suter. The mainchain coin gets stored in the smart contract and the Suter are homomorphically added to ys balance. If the account does not exist yet, a new one is created. The transfer operation in this contract is also attached with a range proof ensuring the deposit amount does not exceed the upper limit of Suter token.
- **Burn**. Burn converts Suter back to the mainchain coin. It verifies the proof of burn against the statement of burn to ensure that the sender knows the right private key and is claiming the right amount. It also checks a signature on the transaction data and the current value of counter, which prevents replay attacks.
- **Transfer**. This operation enables the confidential transfer of Suter tokens. The proof of transfer makes sure that the ciphertext has the right form and the sender has enough money.

User algorithms

A user can run one of the following algorithms to interact with the smart contract. The output of these algorithms are raw transactions. They will be signed using the public key of the Mainchain account from which they are sent and destined to the Mainchain or Suter smart contract.

- **CreateAddress**$(1^\lambda)(sk, pk)$ provides a way for a user to uniquely identify itself to the smart contract. It takes the security parameter λ as input and outputs a secret key sk and the respective public key pk.
- **CreateFundTx**(pk, amt) adds funds to an account. It takes a public key pk and an amount amt as inputs.
- **CreateTransferTx**$(sk_{from}, pk_{to}, amt, st)$ transfers money from one account to another. It takes a secret key sk_{from}, a destination public key pk_{to}, an amount amt, and the state of the smart contract st at a certain block height h as inputs. It outputs tx_{trans}. (For anonymous transfers, this algorithm would also take a set AnonSet as input, which would contain the public keys of both the senders and receivers. AnonSet would be a part of the output too.)
- **CreateBurnTx**(sk, st) withdraws the entire balance from an account. It takes a secret key sk and a state st as inputs.

– **ReadBalance**(sk, st) reads the balance of an account. It takes a secret key sk and state st as inputs, and outputs an integer b.

For more details on these algorithms, the interested readers are referred to the Suterusu yellowpaper [2].

Note the sender is required to purchase the Suter token to pay for the fee of the anonymous transfer contract. The exact amount of gas cost of different contracts can be found in the gas cost document [1].

2 How to prevent front-running attack and replay attack

One missing piece in the above description is how our design can prevent the front-running attack. The front-running problem has to be taken into account when it comes to cross-chain functionality. The Zether paper has a detailed example of how the front-running attack can be launched: a ZK-proof in a transfer transaction needs to show that the remaining balance is positive. A user Alice generates this proof w.r.t. to her current account balance, stored in an encrypted form on the contract. However, if another user Bob transfers some ZTH to Alice, and Bobs transaction gets processed first, then Alices transaction will be rejected because the proof will not be valid anymore. Note that Bob may be a totally benign user yet Alice loses the fees she paid to process her transaction.

To address the front-running problem, the Suter framework keeps all the incoming transaction in a pending state. The system time is divided into epochs, where an epoch consists of k consecutive blocks. At the end of each epoch, pending transfers are rolled over into the corresponding accounts. Users are expected to publish their **transfer** or **burn** transactions at the beginning of an epoch. We then rollover an account in an epoch when the first message from this account is received; so, one message rolls over only one account. To achieve this, we define a separate (internal) method for rolling over, and the first thing every other method does is to call this method.

To prevent replay attacks, the Suter framework, like the Zether framework, also has an important feature: locking accounts to other smart contracts. The associate a nonce with every Suter account. The nonces are incremented as transactions are processed. A new transaction from an account must sign the latest value of the nonce associated with the account along with the transaction data, which includes any ZK-proof. This approach binds all components of a transaction together and ensures freshness. ZK-proofs cannot be imported into malicious transactions and valid transactions cannot be replayed.

References

1. Suterusu testchain gas cost, 2019.
2. Dr. Lin. Suterusu yellowpaper v 0.2, 2020.