International Journal of Computational Science and Information Technology (IJCSITY)
Vol. 13, No. 1, February 2025

Atlantis Protocol Private Succinct Transfers On Public Networks

Oleksandr Kurbatov, Kyrylo Riabov, Mykhailo Velykodnyi

Abstract

This document proposes a combination of several techniques to construct anonymous and untraceable payment systems. The proposed system supports arbitrary transfer amounts and enables the simultaneous transfer of multiple assets.

1 Introduction

Public tokens and stablecoins are a problem for consumers. It could be quite risky if any external party can see the particular balance and trace the history of interactions and relations between users and organizations.

Several solutions solve privacy issues, but they have several technical limitations that we want to reduce. The first solution is the Mimblewimble protocol introduced in [Jed16]. This protocol allows the creation of compact cryptographic commitments for UTXOs and efficiently aggregates them by producing only one signature value and range proofs for outputs. The drawbacks of the protocol are 1 – the ability to track which commitments are being spent within the particular transaction and 2 – the ability to operate only with a single currency.

The second solution we refer to is Tornado Cash [PSS19]. This approach allows us to keep the deposit commitments within the tree and break the connections when withdrawing coins. Tornado Cash provides untraceability of payments, but at the same time, users need to operate only with the same denomination values (not allowing Sudoku analysis [Atl14]) and create new contract instances to operate with different assets.

We introduce the Atlantis protocol, which allows us to inherit and combine properties from the mentioned solutions and create the payment method with the next set of features:

1. **Anonymity**. No public keys or addresses represent users while transferring coins.
2. **Untraceability**. No connection between newly created commitments and spent commitments.
3. **Limitness**. No limitations on the amount of funds within the payment.
4. **Heterogeneity**. It's possible to support several assets in one commitment without increasing the commitment size.

2 Preliminaries

$\mathbb{G}$ a cyclic group of prime order p written additively, $G \in \mathbb{G}$ is the group generator. $a \in \mathbb{F}_p$ is a scalar value and $A \in \mathbb{G}$ is a group element. $\mathsf{hash}(m) \to h \in \mathbb{F}_p$ is the cryptographic hash function that takes as an input an arbitrary message m and returns the field element h. $\mathsf{NUMS}(i).\mathsf{toPoint} \to A \in \mathbb{G}$ is the "Nothing-up-my-sleeve" [BLS15] operation that converts the arbitrary input to the group element A the way nobody can find a such as $A = aG$.

We define the relation for the proof as $\mathcal{R} = \{(w, x) \in \mathcal{W} \times \mathcal{X} : \phi_1(w, x), \phi_2(w, x), \dots, \phi_m(w, x)\}$, where w is a witness data, x is a public data and $\phi_1(w, x), \phi_2(w, x), \dots, \phi_m(w, x)$ the set of relations must be proven simultaneously.

DOI : 10.5121/ijcsity.2025.13101

We are referring to the Schnorr signature algorithm [Sch89] within two functions: $\mathsf{sigGen}(sk, m) \to \mathsf{sig}$, where $\mathsf{sig} = \langle R, s\rangle$ is the signature value, s is the signer's private key and m is the signed message; and $\mathsf{sigVer}(\mathsf{sig}, P, m) \to \mathsf{bool}$ is a signature verification function that operates with the public key P and returns the result of signature verification.

The last component is a Merkle proof $\pi_t \in \mathbb{F}_p^{(n)}$ is the list of node values that leads to the root value t. The Merkle proof for the element x can be verified by $\mathsf{MPVer}(x, \pi_t, t) \to \mathsf{bool}$.

3 Commitment construction

The commitment $\mathsf{C} \in \mathbb{G}$ represents the user's balance and is constructed as follows [Ped91]:

$$\mathsf{C} = aH + skG,$$

where:

a – an amount of coins in the commitment

sk – user's secret key

$H, G \leftarrow \mathbb{G}$ – the set of generators

For simplification, let's say H is the asset generator, and G is the ownership generator. We can extend this commitment construction scheme to support several assets by:

$$\mathsf{C} = \sum_{i=0}^{n} a_i H_i + skG,$$

where i is the asset identifier (it can be the contract's address of a particular asset, for example), a_i is the number of coins with identifier i, $H_i \leftarrow \mathsf{NUMS}(i).\mathsf{toPoint}$ – the generator that represents a particular coin type i.

So, C is a single EC point representing several user balances with appropriate coin types. Then, the main idea is that we can operate with different assets in a homomorphic way, proving the conservation rule (the number of coins in input commitments equals the number of coins in output commitments).

All commitments are stored in the Sparse Merkle Tree [ide24], allowing their untraceability within transfer and withdrawal operations. At the same time, we will collect nullifiers connected to secret keys as $\mathsf{null} = \mathsf{hash}(sk)$, preventing commitment from being spent twice. We can store these nullifiers in the map on the contract or collect them in the additional tree (in these cases, users must generate exclusion proofs and add nullifiers into this tree after transfers and withdrawals).

4 Protocol description

We can divide the protocol into the list of operations it supports:

1. **Deposit**. Allows the coins to be converted into a cryptographic commitment stored in the tree.
2. **Transfer**. Spends the existing commitment(s) and creates a set of new unspent commitment(s).
3. **Withdrawal**. Allows converting the commitment(s) to the public number of coins and withdrawal to the user's account.

We can define the list of contract public parameters needed for all operations validation: G is the constant ownership generator; $\langle i, H_i\rangle^n, i \in [0, n]$ – the HashMap with contracts' identifiers and corresponding asset generators. This HashMap can be updated by adding the needed asset contract address and calculating $\langle i', H_i'\rangle \leftarrow \mathsf{NUMS}(i').\mathsf{toPoint}$ by the Atlantis contract (to be sure NUMS procedure is performed correctly).

4.1 Deposit

Deposit operations don't require proof; the contract can calculate the commitment based on deposit details and public values provided by the deposit initiator. At the same time, the contract must verify that the deposited asset is supported (there are no technical limitations for the list of supported contracts, but it affects the complexity of the range of proofs generated for transfer and withdrawal transactions).

Algorithm 1 Deposit

Inputs:

- *Private:*

  sk – user's secret key

- *Public:*

  $a_i, i \in [0, n]$ – amounts of deposited coins for different asset types

  $H_i, i \in [0, n]$ – coin types' generators

  G – ownership generator

  $P = skG$ – deposit public key

Depositing process:

1. The user makes a deposit and provides P value to the contract.
2. Contract extracts a_i and $H_i, \forall i \in [0, n]$ based on the deposit data.
3. Contract calculates the commitment as

$$\mathsf{C} = \sum_{i=0}^{n} a_i H_i + P$$

4. The contract stores the point C in the tree by the $\mathsf{index} = \mathsf{hash}(\mathsf{C})$

While depositing funds, the user can create not one but several commitments. For that they can define i, a_{i_j}, where $j \in [0, m]$, m – is the number of commitments. Each $\mathsf{C}_j = \sum_{i=0}^{n} a_{i_j} H_i + P_j$ is added into the separate tree leaf.

4.2 Transfer

A transfer operation spends the list of existing commitments and creates new ones. However, for transfer, a sender and a recipient should cooperate in constructing the payment transaction. We can describe this process in the following steps:

1. The sender calculates the list of nullifiers for commitments that need to be spent and provides it to the recipient.

2. The recipient forms the commitment based on transfer details (amount and assets' types) and their key and then produces the signature of nullifiers. Additionally, the recipient generates a range of proof for the amount in their commitment(s).

3. The sender produces their signatures, aggregates them with the recipient's, and generates the final proof (including commitment(s) for the change if required).

Algorithm 2 Transfer

Initiation:

1. The sender has the list of unspent commitments $\mathsf{C}_s^{(n)}$ with secret keys $sk_s^{(n)}$
2. They calculate the list of nullifiers as $\forall i \in [0, n], \mathsf{null}_{s_i} \leftarrow \mathsf{hash}(sk_i)$
3. Then, the sender transfers the list $\mathsf{null}_s^{(n)}$ to the recipient

Preparation:

1. The recipient generates $s_r, P_r = sk_r G$ and their output commitment as
$$\mathsf{C}_r = \sum_{i=0}^{n} a_i H_i + P_r.$$
2. The recipient generates the signature $\mathsf{sig}_r \leftarrow \mathsf{sigGen}(sk_r, \mathsf{null}_s^{(n)})$
3. The recipient generates the range proof π_{rp_r} for the relation:
$$\mathcal{R}_{rp} = \{a_i^{(n)}, sk_r; G, H_i^{(n)}, \mathsf{C}_r : \forall i \in [0, n], a_i \in [0, 2^{128}), \mathsf{C}_r = \sum_{i=0}^{n} a_i H_i + sk_r G\}$$
4. The recipient transfers sig_r and π_{rp_r} to the sender

Transfer execution:

1. The sender generates the list of signatures for all input commitments
$$\mathsf{sig}_{s_i} \leftarrow \mathsf{sigGen}(sk_{s_i}, \mathsf{null}_s^{(n)})$$
2. The sender generates $s_c, P_c = sk_c G$ and their commitment for the change as
$$\mathsf{C}_c = \sum_{i=0}^{n} a_i H_i + P_c$$
3. The sender generates the signature $\mathsf{sig}_c \leftarrow \mathsf{sigGen}(sk_c, \mathsf{null}_s^{(n)})$
4. The sender generates the change's range proof π_{rp_c} for the relation
$$\mathcal{R}_{rp} = \{a_i^{(n)}, sk_c; G, H_i^{(n)}, \mathsf{C}_c : \forall i \in [0, n], a_i \in [0, 2^{128}), \mathsf{C}_c = \sum_{i=0}^{n} a_i H_i + sk_c G\}$$
5. The sender calculates the aggregated signature for all input and output commitments as
$$\mathsf{sig}_{agg} = \sum_{i=0}^{n} \mathsf{sig}_{s_i} - \mathsf{sig}_r - \mathsf{sig}_c$$
6. The sender generates the final payment proof π_p for the following relation
$$\begin{gathered}\mathcal{R}_p = \{sk_s^{(n)}, \mathsf{sig}_{agg}, \mathsf{C}_s^{(n)}, \pi_t^{(n)}; \mathsf{null}_c^{(n)}, \mathsf{C}_r, \mathsf{C}_c, t : \\ \mathsf{sigVer}(\mathsf{sig}_{agg}, \sum_{i=0}^{n} \mathsf{C}_{s_i} - \mathsf{C}_r - \mathsf{C}_c, \mathsf{null}_s^{(n)}) \rightarrow \mathsf{true}, \\ \forall i \in [0, n], \mathsf{null}_{s_i} \leftarrow \mathsf{hash}(sk_{s_i}), \\ \forall i \in [0, n], \mathsf{MPVer}(\mathsf{C}_{s_i}, \pi_{t_i}, t) \rightarrow \mathsf{true}\}\end{gathered}$$
7. The sender provides π_{rp_r}, π_{rp_c} and π_p to the contract together with the public data. If all proofs are correct, the contract adds C_r and C_c in the tree and $\mathsf{null}_s^{(n)}$ into the list of spent commitments.

In the same way, the user can send payments to several recipients within one interaction – they need to generate the signatures and range proofs for all output commitments and provide it to the sender, which then aggregates them once.

4.3 Withdrawal

While withdrawing coins, the user proves there is an unspent commitment with the appropriate amount of coins with the particular types and publishes the nullifier connected to the secret key.

Algorithm 3 Withdrawal

Inputs:

- *Private:*

  sk – user's secret key

  C – user's commitment

  π_t – the Merkle proof for the commitment inclusion

- *Public:*

  $a_i, i \in [0, n]$ – amounts of coins for different asset types

  $H_i, i \in [0, n]$ – coin types' generators

  G – ownership generator

  $\mathsf{null} \leftarrow \mathsf{hash}(sk)$

Proving:

Generate proof π_w for relation:

$$\mathcal{R}_w = \{sk, \mathsf{C}, \pi_t; a_i^{(n)}, G, H_i^{(n)}, \mathsf{null} : \mathsf{null} \leftarrow \mathsf{hash}(sk), \mathsf{MPVer}(\mathsf{C}, \pi_t, t) \rightarrow \mathsf{true}, \mathsf{C} = \sum_{i=0}^{n} a_i H_i + skG\}$$

The proof π_w is verified by the contract. If it's correct and null isn't included in the list of spent nullifiers, the user withdraws coins to the public balance.

5 Security properties and regulation aspects

The described version of Atlantis Protocol provides the following properties:

- Despite providing the user's initial public key for the deposit, it doesn't appear in transfer and withdrawal operations.
- When users transfer or withdraw coins, they hide the commitment in the list of all existing ones.
- A commitment with the same nullifier can't be spent twice (if we want to provide an ability using the same public key, we can introduce an additional "nullifier" generator and extend Pedersen's commitment with the third component).
- The user can operate with any amount without limitations based on particular denominations.
- Potentially, an infinitive number of different assets can be supported within a single commitment without increasing proof verification complexity. However, the complexity of proof generation must be taken into account.
- When we recommended using proofs for $[0, 2^{128})$ range, we assumed operating with 254-bit amounts (it shouldn't be possible to overflow the output number of coins with a reasonable amount of output commitments). It can be different in the final implementation.

- The Sudoku analysis is possible if the user makes no transfer between deposit and withdrawal operations.

The privacy-preserving properties of such a solution may raise some regulatory concerns, so we encourage using additional constructions that will help achieve **eligible privacy** and protect users from the associated risks. One is the inclusion and exclusion lists proposed in [BSG23], allowing users to select the quorum of commitments to be associated/disassociated.

If an instance of such an application supports centralized or partially decentralized governance, its administrators can forcibly add suspicious commitments to an exclusion list, making them impossible to spend. A timelock value can be added to the deposit, up to which the commitment cannot be spent. Adding such a value slightly increases the cost of generating a proof but does not compromise the privacy of the commitment being spent (the user simply proves that the timelock in the value tree is greater than the current timestamp). If administrators don't put the commitment on the prohibited list, it can be spent after the defined locktime.

References

[Sch89] Claus-Peter Schnorr. *Efficient Signature Generation by Smart Cards*. https://link.springer.com/article/10.1007/BF00196725. Accessed: 2024-11-18. 1989.

[Ped91] Torben Pryds Pedersen. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*. https://link.springer.com/chapter/10.1007/3-540-46766-1_9. Accessed: 2024-11-18. 1991.

[Atl14] Kristov Atlas. *Weak Privacy Guarantees for SharedCoin Mixing Service*. https://www.coinjoinsudoku.com/advisory. Last accessed 18 November 2024. 2014.

[BLS15] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. *The BADA55 Digital Signature Scheme*. https://bada55.cr.yp.to/bada55-20150925.pdf. Accessed: 2024-11-18. 2015.

[Jed16] Tom Elvis Jedusor. *Mimblewimble*. https://scalingbitcoin.org/papers/mimblewimble.txt. Accessed: 2024-11-18. 2016.

[PSS19] Alexey Pertsev, Roman Semenov, and Roman Storm. *Tornado Cash: Non-Custodial, Decentralized Privacy Solution*. https://berkeley-defi.github.io/assets/material/TornadoCashWhitepaper.pdf. Last accessed 18 November 2024. 2019.

[BSG23] Vitalik Buterin, Ameen Soleimani, and Matt Garnett. *Blockchain Privacy and Regulatory Compliance: Towards a Practical Equilibrium*. https://vitalik.ca/general/2023/05/22/privacy.html. Accessed: 2024-11-18. 2023.

[ide24] iden3. *Sparse Merkle Tree*. https://docs.iden3.io/publications/pdfs/Merkle-Tree.pdf. Accessed: 2024-09-19. 2024.