Full privacy in account-based cryptocurrencies v 0.12

Dmitry Khovratovich[1,2] and Mikhail Vladimirov[2]

1: Dusk Foundation
2: ABDK Consulting

July 2, 2019

1 Introduction

The first cryptocurrency Bitcoin defines transactions as a group of uniquely identified inputs and outputs, where the sums of inputs and outputs match, and where each input references a previously produced but yet unspent output. To spend an output one has to sign a spending transaction with a public key mentioned in the output. This model is called UTXO (unspent transaction outputs). In this model it is easy to reference a particular output in the past to be spent, but accounting is more difficult as one has to cluster all the outputs belonging to a particular public key. In Ethereum [W+14], Ripple, Stellar, Ethereum-based tokens, and some other cryptocurrencies a different approach is taken: each public key/address has its own balance, which is debited or credited explicitly in the protocol depending on the type of transaction (output or input, resp.) applied to the address. Such designs are called *account*-based cryptocurrencies in contrast to UTXO-based designs.

The distinction between those becomes crucial when enhancing a cryptocurrency with value privacy (keeping the transaction value hidden but provably consistent with the balance change) and user privacy (keeping both the sender and the recipient anonymous). Whereas the former is relatively easy to add by using homomorphic commitments or other mechanisms, the latter is more difficult. To keep the sender anonymous in the UTXO model, one can either hide him in a subset of other users (called *anonymity set* in Monero) or store all outputs in a Merkle tree (like in Zcash [zca18]) so that one can prove the knowledge of an output by providing a zero-knowledge proof of a Merkle opening using ZKSNARKs or other techniques. However, the accountability becomes a problem: it is difficult to assert that the user balance is equal to certain value because a user may own a number of unspent outputs, which are difficult or expensive to be provably accumulated in a single value. This limits adoption of the UTXO model to privacy-enhanced currencies with restrictions on balances such as security tokens with caps on ownership.

In turn, in account-based currencies, even if balances are hidden or encrypted, it is difficult to hide the balance change of a particular user as those whose balances do not change can be easily ruled out from potential spenders. If (almost) all balances are changed, it is again too expensive. If only a few balances are changed, the anonymity set is not full.

We thus face the following question:

Is it possible to have absolute sender privacy and efficient zero-knowledge proofs of balance in the same cryptocurrency?

We answer the question positively by merging the privacy approach of Zcash with the account model. Concretely, we introduce two new concepts, which have their own merit: (1) **private memory**, which enables fast proofs and provable updates for designated owner, and (2) **balance tree**, which is a data structure with efficient reads, writes, and zero-knowledge proof of balance for any time in the past. Providing an efficient implementation of these concepts with zero-knowledge friendly hash functions, we show how to build a token contract with full sender and value privacy, a whitelist of owners, and provably enforceable restrictions on the maximum ownable amount.

Related work

The literature on the anonymity in cryptocurrencies is vast and its survey is beyond the goals of this report. We mention only the designs that attempt to maximize both sender and value privacy.

Zcash [zca18] follows the UTXO model where it provides complete sender/recipient privacy and value secrecy, but no accountability proof.

Zether [BAZB19] encrypts user balances with ElGamal so that third parties can modify the ciphertexts while keeping the plaintext untouched. However, the amount of changes to the database grows linearly with the number of balances the sender hides in, so the privacy is effectively limited.

ZEXE [BCG+18] is a framework for verifiable decentralized computation, where users can possess and provably update their memory records. Our private memory concept shares some ideas with ZEXE's records but we use a single hash function to obtain the necessary features instead of different signature, commitment, and PRF primitives.

PGC [CM19] and Quisquis [FMMO18] operate similarly to Zether, but keeps the balances in commitments rather than in ciphertexts.

2 Techniques

2.1 Private memory

We first suggest a concept of private memory, where a user has full control on specific memory locations stored by the ledger. A private memory cell is, informally, a commitment to the user public key and data, so that a user can prove the ownership of the cell using the private key. Additionally, the user can prove statements about the data, e.g. the transitions between the data values that follow a certain logic.

We implement a private memory cell as a hash of user public key and data:

$$M = H(K, D),$$

where $K = H(k)$ is the hash of private key. Any zero-knowledge friendly hash function can be used, and we select Poseidon [GKK+19] to get fastest ZK proofs using[1] Bulletproofs [BBB+18] on the hash input.

Memory cells are stored in a Merkle tree and used as follows:

- To update D using function f, the user constructs a nullifier $N = H(k, p)$ where p is the position of M in the tree, and creates $M' = H(K, f(D))$. Then he broadcasts (M', N, π) where π is the proof of knowledge of k, p and that the same k is used in both M' and N. Optionally, the user can prove that M' is obtained using f. The ledger adds M' to the tree, records N and forbids its future use.

[1] SNARK [Gro16] and STARK [BBHR18] proofs are also possible.

- To prove a statement on D without updating it, the user shows N and proof π about N and D.

A user is supposed to insert a high-entropy value r into D to prevent others from learning D from M. Knowledge of r is required to prove the knowledge of D but does not allow to update M without knowing k.

Note that multiple reads without a write are linkable. To prevent that, the ledger can maintain a tree of used nullifiers so that a prover can prove the knowledge of a nullifier that has not been used yet. However, this can be done only with respect to some tree in the past, so a read will not be fresh.

2.2 Balance tree

In a user-centric bookkeeping we consider transactions $T = (V, t)$, where V is the amount and t is the timestamp. When V is positive (a debit entry) the user balance increases, whereas when it is negative (a credit entry), the balance decreases. Let $\mathcal{T}$ be a set of all transactions for the user, and let us define the *balance* at time t_0 as

$$B(t_0) = \sum_{(V_i,t_i)\in\mathcal{T}:\, t_i\le t_0} V_i.$$

Balance tree for $\mathcal{T}$ is a data structure, which efficiently calculates

- the balance at any given time;
- the maximum balance in any period $[t_1; t_2]$.

It can be implemented as a binary Merkle tree with transactions as leafs, ordered by timestamp. A tree node Q_N corresponds to the time segment $N = [t_1; t_2]$, is the parent to nodes Q_{N_1} and $Q_{N\setminus N_1}$, and contains

- The accumulated income from $\mathcal{T}_N$: $C_N = \sum_{T_i\in\mathcal{T}_N} V_i$
- The maximum accumulated income in the period: $M_N = \max_{t\in[t_1;t_2]} \sum_{T_i\in\mathcal{T}_{[t_1;t]}} V_i$.
- The hash of C_N, M_N, and child node hashes.

The hash of node Q_N, denoted by H_N, is computed as

$$H_N = H(C_N, M_N, H_{N_1}, H_{N\setminus N_1}).$$

The balance at any time t_0 is then the sum of all balances in nodes $\{Q_{N_j}\}$ such that $\cup N_j = [-\infty; t_0]$ and $N_i \cap N_{k\neq i} = \emptyset$no Q_{N_j}. There are at most $\log|\mathcal{T}|$ such segments and their nodes are located on the way from the leaf corresponding to t_0 to the root. It can be calculated efficiently and in zero knowledge. The same applies to the maximum balance.

It is also clear that one can insert a transaction to the history to the tree with updating only a logarithmic number of nodes, i.e. those on the way to the root.

3 Constructions

3.1 Capped smart contract

We implement a smart contract with the following logic:

- It is a token contract with fixed supply.
- Token owners are a predefined set of addresses $\mathcal{A}$.
- Each owner has an upper bound (cap) C on the amount of tokens he can hold.
- Each user A can send tokens to another user B at some moment t_0. User B can accept N tokens in time t_1 only if he has at most $(C - N)$ tokens in period $[t_0; t_1]$.

The currency flow is UTXO-based, like in Zcash, where output coins are added to the Merkle tree (we suggest the same hash function for it). Senders are also obliged to maintain their balance tree using private memory. Each user is allowed to add exactly one memory cell with data formatted as

$$D = (r, R),$$

where r is the randomness and R is the root of the balance tree. They can update their cell by producing nullifiers, so that at each moment of time only one cell is valid.

We implement two scenarios for cap enforcement, where the cap value is C. Let T be a transaction of V tokens from Alice to Bob, which is recorded by the ledger at time t_0:

1. In Scenario 1 V is deducted from Alice's balance at the time t_0, i.e. she adds $(-V, t_0)$ to her balance tree. The tokens can be spent by Bob at any time $t_1 > t_0$, but only if the maximum balance in $[t_0; t_1]$ is not bigger than $(C - V)$. If the tokens can not be accepted they are effectively lost (though one can devise a mechanism where Bob can forward them to the contract owner).

2. In Scenario 2 Alice sets (presumably by an agreement with Bob) the maximum redemption period D. If Bob does not claim tokens within this time frame, he loses this option and Alice can claim the tokens back at any time after $(t_0 + D)$. Then the Alice's balance does not change. If Bob does claim the tokens within the redemption period, he adds (V, t_0) to his balance tree. Alice can then add $(-V, t_0)$ to her tree at any time in the future by sending another transaction, which we denote SETTLE.

3.1.1 Scenario 1

Here Alice selects a timestamp t_0 which must be close to the current time. For each input coin with value V' and timestamp t' she adds a (V', t') to the balance tree. For each output coin with value V'' she adds an entry (V'', t_0) to the balance tree.

Each transaction contains:

- A list of output coins, each being a hash of value V, Bob address A, randomness r, and encryption of (V, r) on the encryption key of A. Each coin hash is added at position p and timestamp t to the Merkle tree $\mathcal{T}$ of coins.
- A list of nullifiers for input coins, each being a hash of the recipient private key k_A and the coin position p.
- A new value for the private memory cell M, being a hash of the sender address A_S, randomness r_M, and a root hash of Alice's balance tree.
- A nullifier for the value of cell M' being a hash of the cell position and Alice private key.
- A proof of correctness, that includes the knowledge of private keys, values, openings in the tree, and randomness of spent coins, the correctness and consistency between the old and a new balance tree, the correctness of the nullifier values.

- A policy proof: that the new maximum balance value does not exceed the cap, the inclusion of the recipient address in the list $\mathcal{A}$.

3.1.2 Scenario 2

There are two types of transactions: SPEND and SETTLE. In the SPEND transaction Alice selects a redemption period D for output coins. She updates the balance tree only for input coins and only if the input coin is spent before its redemption period passes:

- For each input coin with value V', timestamp t' and redemption period D' Alice adds (V', t') to the balance tree only if $t_0 < t' + D'$.

Each SPEND transaction contains

- A list of output coins, each being a hash of value V, recipient address A_R, sender address A_S, randomness r, and encryption of (V, r, D, A_S) on the encryption key of A_R. Each coin hash is added at position p and timestamp t to the add-only Merkle tree $\mathcal{T}$ of coins, which uses the Poseidon hash function.
- A list of nullifiers for input coins, each being a hash of the sender address A_S, the recipient address A_R, the randomness r, and the coin position p.
- A new value for the private memory cell M.
- A nullifier for the cell M' that belongs to Alice and contains the old value of her balance tree.
- A proof of correctness, that includes the knowledge of private keys, values, openings in the tree, and randomness of spent coins, the correctness and consistency of the balance tree, the correctness of the nullifier values.
- A policy proof: that the new maximum balance value does not exceed the cap, the inclusion of the recipient address in the list $\mathcal{A}$.
- The coin nullifier is the hash of both sender and recipient keys, its randomness r and its position p.

If Bob spends a coin of value V within the redemption period, Alice can then add $(-V, t_0)$ to her balance tree by sending a SETTLE transaction. The SETTLE transaction contains:

- A list of nullifiers that are settled.
- A new value for the private memory cell M.
- A nullifier for the cell M' that belongs to Alice and contains the old value of her balance tree.
- A proof of correctness of the nullifier values, and the consistency proof that M' has the old version of the balance tree of M.

4 Performance

Each balance tree correctness proof has size logarithmic in the number of transactions for the user, whereas each opening proof for spent coins has size logaritmic in the total number of transactions, which clearly dominates the former proof. We thus estimate that our construction would be at least as fast as ZCash being instantiated with Bulletproofs instead of SNARKs, but probably much faster as the Poseidon hash function yields more efficient proofs than Pedersen hash used in Zcash.

References

[BAZB19] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. *IACR Cryptology ePrint Archive*, 2019:191, 2019.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, pages 315–334. IEEE, 2018.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018:46, 2018.

[BCG+18] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. *IACR ePrint*, 962, 2018.

[CM19] Yu Chen and Xuecheng Ma. Pgc: Pretty good confidential transaction system with accountability. *IACR Cryptology ePrint Archive*, 2019:319, 2019.

[FMMO18] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. Technical report, Cryptology ePrint Archive, Report 2018/990, 2018, 2018.

[GKK+19] Lorenzo Grassi, Daniel Kales, Dmitry Khovratovich, Arnab Roy, Christian Rechberger, and Markus Schofnegger. Starkad and poseidon: New hash functions for zero knowledge proof systems. *IACR Cryptology ePrint Archive*, 2019:458, 2019.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[W+14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[zca18] ZCash protocol specification, 2018. `https://github.com/zcash/zips/blob/master/protocol/protocol.pdf`.