POLYCRYPT

Private Account Balances in Safe

Jan Bormet
jan@perun.network

Andreas Erwig
andreas.erwig@tu-darmstadt.de

Sebastian Faust
sebastian.faust@gmail.com

Philipp-Florens Lehwalder
philipp@perun.network

1 Introduction

In the realm of blockchain technology, privacy remains a paramount concern, particularly as the adoption of cryptocurrencies and decentralized applications continues to surge. While Safe accounts provide a number of modular features that allow individual users and organizations to control and manage their funds and onchain identity in a secure manner, there is a significant concern regarding the full transparency of onchain balances and assets associated with an account, which are directly visible to all participants within the blockchain network. As account balances are publicly visible, high-balance accounts of individuals and organizations may become subject to unwanted attention from attackers who attempt to steal funds or take over their account through hacking, phishing, or other methods. Also, privacy in and of itself is a desirable property, as people generally dislike publicly disclosing the amount of money they own.

There are existing solutions to this problem in cryptocurrencies, such as BIP32 wallets [Wik12], where one can derive new public keys from a master public key mpk in an unlinkable manner. Unlinkability ensures that public keys derived from the same master public key cannot be linked to each other (i.e. they are indistinguishable from randomly generated keys). Crucially, the holder of the corresponding master secret key msk can similarly derive the corresponding secret keys for the public keys. In practice, this can be used to generate a new public key for each received payment. Let's say that Alice wants to receive a payment from Bob. To this end, Alice generates a new public key pk from mpk and computes the corresponding address A_{pk}. Alice sends A_{pk} to Bob, who then sends the payment to A_{pk}. When Alice wants to spend the received payment, she can derive the corresponding secret key sk from msk to sign the transaction.

While this solution works well for Externally Owned Accounts (EOAs) in Ethereum, i.e., user accounts, it does not work for Safe accounts, which are smart contracts. While we could create a new Safe account for every received payment that can be controlled by the derived key through a BIP32 wallet, this would be impractical for several reasons. Most importantly, you still want to make use if the features and advantages of Safe accounts over traditional wallets, such as the ability to have multiple owners, spending limits, recovery mechanisms, and more. Hence, you want your funds to

be hidden and unlinkable to your Safe Smart Account or Master Safe, but still entirely managed and secured through your Safe. Naively, one could just add all these policies to the derived Safe account for every received payment, but this would run into several issues: First, it would publicly link all derived Safe accounts to their respective Master Safe, which would defeat the purpose of unlinkability. Second, it would be impractical to dynamically impose changes to the account policy, e.g., a change of threshold, to all Safe Smart Accounts that belong to you due to high fees.

Our solution is to collect all "hidden" funds in a single smart contract, which we call AssetHolder. This AssetHolder acts as a shared private vault of assets. The funds are linked to their respective Safe Smart Account through a commitment to its respective address. The hiding property of this commitment ensures unlinkability, while the binding property ensures that the funds are only accessible by the owner of the Safe Smart Account. Whenever one wants to spend some hidden funds, they call a withdraw function on the AssetHolder, providing an opening to the commitment as well as some authentication data. The withdraw function of the AssetHolder then checks the opening and invokes a callback to the Safe Smart Account that was committed to, passing on the authentication data. The Safe contract verifies that the authentication data fulfills the policy of the account, e.g., a threshold of owners authorized the payment through signatures. We note that the downside of our approach is that received payments are linkable to the recipient's own Safe Smart Account upon spending, but we argue that this is a reasonable trade-off for the benefits of our approach.

2 Technical Description

In the following, we describe the proposed protocol and technical details behind our solution for "hiding' or obfuscating the balances of Safe Smart Accounts, which is based on a cryptographic commitment scheme.

2.1 Commitment Schemes

Our main building block is commitment scheme, which is a tuple of algorithms $\mathcal{C} = (\mathsf{Commit}, \mathsf{Verify})$:

$\mathsf{Commit}(m; r) \to (c, op)$**.** The Commit algorithms takes as input a message m and randomness r. It outputs a commitment c as well as an opening op.

$\mathsf{Verify}(c, op, m) \to 1/0$**.** The Verify algorithm takes as input a commitment c as well as a message m and an opening op and outputs either 1, indicating that op validly opens c to m or 0, indicating that it is an invalid opening.

A commitment scheme must achieve the three following properties:

Correctness. Correctness states that every valid commitment must verify successfully, i.e., for all choices of m and r it must hold that $\mathsf{Verify}(\mathsf{Commit}(m; r), m) = 1$.

Hiding. The hiding property ensures that the commitment c reveals nothing about the message m that was committed to. It is most commonly defined through a security game, where an adversary can pick two messages m_0 and m_1 at random. The game then randomly commits either to m_0 or m_1 and sends the resulting commitment c to the adversary. The adversary now has to guess, whether c belongs to m_0 or m_1. A commitment scheme is considered to achieve the hiding property if any adversary's probability of guessing correctly is essentially $\frac{1}{2}$.

Binding. The binding property ensures that once someone has committed to a message m with the respective commitment c, then it is no longer possible to change the message that was

committed to (i.e. find an opening op' and a *different* message m such that $\mathsf{Verify}(c, op', m') = 1$). This property is similarly defined through a security game, where an adversary wins by producing a commitment c and to *different* messages along with openings so that both openings are valid.

In practice, commitment schemes are used to lock in a choice or message by committing to it and publishing the commitment c. The hiding property protects the committer, as nobody can learn anything about the message that was committed to before the committer publishes the opening. Conversely, the binding property protects everybody else as the committer can not change the message, once c is published.

2.2 Hidden Account Balances from Commitment Schemes

The main new component is the so-called AssetHolder smart contract. At its core, the AssetHolder holds all assets of the participating parties, i.e., Smart Account owners who want to hide their balance. It consists of two functions Deposit and Withdraw as well as a mapping *assets* that maps commitments to balances (see Figure 1). Account owners or other EOAs can deposit assets into the contract calling the payable AssetHolder.Deposit function while providing a *commitment*. Given the commitment within a deposit call, the AssetHolder registers the commitment in a mapping to the deposited balance. In order to withdraw or sending funds to other addresses using the AssetHolder.Withdraw one has to provide a valid *opening*. When requesting the withdrawal of some deposited funds (to the sender's or a different address), the AssetHolder first validates the opening with a registered commitment and then invokes the validation function of the corresponding Safe Smart Account and only permits the withdrawal if it succeeds. We note that it is possible to deposit multiple times to the same commitment, which will just result in adding up the deposited balance, and could even be done by parties without knowledge of the corresponding opening. Withdrawing, however, is only possible using the valid opening, which is ensured due to the hiding and biding property of the commitment scheme.

We instantiate the commitment scheme using a hash function, particularly Keccak-256 (SHA-3), which is natively supported in the Ethereum Virtual Machine (EVM), and can therefore be efficiently used. The owner of a Smart account with address addr generates a commitment c locally by choosing a random value $r \leftarrow_{\$} \mathbb{Z}_q$ and computing the commitment $c \leftarrow \mathsf{H}(\text{addr}||r)$. The opening of this commitment then equals to $op = (addr, r)$, which is verified by computing the hash of the opening and comparing it to the stored commitments.

We provide the pseudocode of a basic construction of the AssetHolder in Figure 1. We discuss possible improvements and extensions in the following section. The Deposit function inputs a commitment c and a balance bal to be deposited (in practice, this would correspond to the balance sent to the payable Smart Contract function) and then stores the balance in the mapping *assets*. While it is possible to deposit multiple times to the same commitment, which will just result in adding up the deposited balance, see Line 4, one should rather deposit to different commitments each time. Note that the Deposit does not have to be necessarily called by the owner of the Safe Smart Account, but any other account to which the owner has provided the commitment as a recipient method.

The Withdraw function takes as input the owner's address $\text{addr}_{\text{owner}}$, a random value r, which correspond to the opening of the commitment, as well as the receiver's address $\text{addr}_{\text{recv}}$, along with the balance to be withdrawn bal to this address, and some authentication data auth, e.g., signatures for the Safe Smart Account logic. In Line 2 of the Withdraw function, the commitment c is implicitly verified by computing the commitment from the owner's address $\text{addr}_{\text{owner}}$ and the random value r and checking whether it has been registered in the *assets* mapping and whether the balance is sufficient, see Line -. After the commitment is checked, in Line 8, the Withdraw function calls the

validation function of the Safe Smart Account related to the owner's address $\text{addr}_{\text{owner}}$, passing on the opening r, the receiver's address $\text{addr}_{\text{recv}}$, the balance bal, and the authentication data $auth$. Here, we assume the validation method simply reverts if the transaction does not meet the policies of the Safe Smart Account. Finally, the Withdraw function reduces the balance of the commitment by the withdrawn amount and transfers the balance to the receiver's address $\text{addr}_{\text{recv}}$, see Line 9-10.

Deposit(c, bal)

```
1 : if assets[c] = ⊥ then
2 :     assets[c] ← bal
3 : else
4 :     assets[c] ← assets[c] + bal
5 : fi
```

Withdraw$(\text{addr}_{\text{owner}}, r, \text{addr}_{\text{recv}}, \text{bal}, \text{auth})$

```
1 : Compute c ← H(addr_owner, r)
2 : if assets[c] = ⊥ then
3 :     return ⊥
4 : fi
5 : if assets[c] < bal then
6 :     return ⊥
7 : fi
8 : SafeAcc(addr_owner).validate(r, addr_recv, bal, auth)
9 : assets[c] ← assets[c] − bal
10 : transfer(addr_recv, bal)
11 : return 1
```

Figure 1: The basic AssetHolder algorithms.

3 Privacy Analysis

Crucially, this scheme hardens the identification of individual balances from Safe Smart Accounts as more participants are involved and interact with this system. The deposit of assets into the AssetHolder contract can be made by any account without any further relationship to the Safe Smart Account owner(s). Prior to the actual deposit, the assets to be deposited may come from different sources and, ideally, are already obfuscated by mixing services to prevent any correlation to the account from being hidden. We, therefore, assume in the future that the assets do not leak any correlation to the owners when being deposited using a commitment.

However, this privacy guarantee only holds until a withdrawal is made with the correct opening, where the owner address of the Safe Smart Account is inevitably revealed. This is due to the fact that the withdrawal, or in general any transaction, has to be validated using the rules defined in the owner's Safe Smart Account. In the case that not all the assets are withdrawn at once, the remaining balance would thus also be linked to the owner's address. This negative effect can be again obfuscated by transferring the "change" to one or, ideally, multiple new commitments, which makes it harder to trace all assets back to the owner.

Even better, instead of directly withdrawing the assets to a receiver address, one could instead transfer the assets to a commitment given by the receiver (provided offline or on a public board), and again, the change to fresh commitments. This way, although the Safe Account's address is still revealed in this process, as long as the assets are only transferred inside the AssetHolder contract, i.e., from commitments to commitment, it remains oblivious to which receiver address the assets are actually sent to. As soon as a withdrawal is made from a commitment to a Safe Smart Account or

an EOA, one could be able to link parts of the involved transactions, which is getting more difficult as more transactions with more participants have taken place in between.

So far, we have only given informal arguments about the security and privacy of this scheme. We expect that formally proving the security of this scheme is straightforward, i.e., that only the commitment creator, i.e., the Safe Smart Account owner, can withdraw the assets from the AssetHolder contract. However, formally modeling and proving the privacy properties of this scheme is more challenging, but we expect to provide reasonable heuristics and believe in improving privacy guarantees with further extensions.

3.1 Protecting against Replay Attacks and Frontrunning

One important attack vector to consider is replay attacks and frontrunning. In replay attacks, an attacker may try to replay a withdrawal transaction (or part of it) in order to withdraw funds again, possibly with a different recipient address. Frontrunning attacks are related, as attackers would see a withdrawal transaction online before it is included in a block and then try to publish it with some modifications, such that the balance is withdrawn to a different recipient address. It is important for the protocol to protect against both attack scenarios, as either could lead to a loss of funds.

Security against replay attacks can be achieved with various techniques. One simple approach is to allow only a single withdrawal per commitment (e.g., the entire balance of the commitment is spent). After a successful withdrawal, the respective commitment would just be removed from the *assets* mapping (or alternatively, the balance in the mapping would be set to 0). This approach trivially prevents replay attacks, as any replayed withdrawal would fail due to the commitment not being present in the mapping anymore. It is not a real restriction to disallow multiple withdrawals from a commitment, as the commitment is linked with the owning Smart Account anyway upon withdrawal, which breaks any anonymity after of the first withdrawal. Somewhat related, one could withdraw only parts of the funds to some address and transfer the change to a new commitment created by the owner, or even better, by another receiver of the funds, as discussed above. However, these solutions are not yet satisfactory as they do not practically address frontrunning. Attackers may still see a transaction and just publish it with a different recipient address before the original transaction is processed.

To prevent this, one needs to make sure that the withdrawal is not malleable. Non-malleability ensures that an attacker can not modify anything about the withdrawal without invalidating it. A common technique to achieve this is using a signature, which would be included in the *auth* parameter of the withdrawal function. The signature would be generated over everything that is relevant to the withdrawal (i.e., the opening r, the recipient address, the balance, and possibly the rest of the *auth* data). The Safe Account would then need to verify this signature in the "validate" callback according to its policy. This solution prevents (effective) frontrunning attacks, as an attacker would not be able to modify anything meaningful, such as the recipient address or the balance of the withdrawal, without invalidating this signature, which would cause the transaction to fail. The downside is that this solution only makes sense for policies that somehow involve signatures and public keys. A different approach would be making use of the replay and frontrunning prevention mechanisms of the underlying blockchain and setting a whitelist of allowed "msg.sender" or "tx.origin" for withdrawals in the Safe Smart Account.

4 Conclusion and Future Work

In this report, we introduced our approach to hide balances related to Safe Smart Accounts and obfuscate which account they belong to while maintaining the ability to enforce the Safe Smart

Account's policies. We introduced the AssetHolder contract, which acts as a mixer and collects all hidden funds from participating Smart Accounts. It ensures that one can not link a balance to the Safe Smart Account that owns this balance until it is withdrawn while enforcing that withdrawals can only be made by the Safe Account that owns the balance. The more participants are involved in this system by depositing assets, the harder it becomes to trace the individual balances and their connection to the Safe Smart Account owner.

4.1 Future Work

This report only provides a basic overview of our ideas for improving the privacy of Safe. We believe that there is a lot of possible future work to be done in this area. We list some concrete directions here.

Formal Analysis and Model. This report contains informal descriptions and security arguments conveying the basic ideas. One important piece of future work is developing a security model (especially for privacy) together with a formal analysis of our protocol in such a model. This is necessary to ensure security as well as give an exact definition of the privacy guarantees that are achieved.

Improving Privacy upon Withdrawal. Our current idea necessarily links a hidden balance with the Safe Smart Account that owns this balance upon withdrawal, thus compromising privacy. The key problem here is that we want to apply the policy of the Safe Account, which owns a balance upon withdrawal. For this, any variant of our idea needs to at least check the *current* policy of the Safe that owns this balance (the construction we provide here implicitly does this through the "validate" callback). We believe that there are solutions involving zero-knowledge proofs or fully homomorphic encryption that can avoid this callback and thus provide privacy throughout withdrawals. These techniques are impractical, though, and need further research to be applied in practice. We note here that in our discussions on current research topics in Safe, we were informed about ongoing research about cross-chain Safe. The problem there looks pretty similar, as one essentially wants to efficiently prove the current state of a Safe on a (different) blockchain. Zero-knowledge proofs will be useful to cross-chain Safe research because of their succinctness and efficiency. Our idea would also make use of the zero-knowledge property, as we essentially would need to prove that in the current state of the blockchain, there is this Safe account with this policy that owns this balance and the given $auth$ satisfies the policy, without revealing which Safe Account is concerned.

Compliance. Whenever a new privacy technology is introduced in the field of cryptocurrencies, it is important to consider the legal and regulatory requirements that apply to it, in particular, anti-money laundering (AML) and know your customer (KYC). While the exact analysis of our scheme with regard to current regulation is not in the scope of this report, but will be considered in future work, we still want to give a brief overview to some relevant points. First, we believe that due to the linking upon withdrawal, it is always possible to prove the origin (and destination) of payments made to you once you withdraw them. However, this problem will become relevant when using techniques to avoid this linking. One approach to maintain a provable origin would be zero-knowledge proofs (e.g., prove that the funds do not originate from some blacklisted addresses). One other interesting feature would be allowing owners to prove that they control certain funds. With the existing scheme, it is trivial to prove that you control at least a certain amount of funds for an institution by simply revealing the openings of the respective commitments. This trivial approach is not very useful, though, because the target institution could publish your proof and thus convince the public that you control funds, which defeats the purpose of hiding them. One can fix

this problem by using so-called *designated verifier zero-knowledge proofs* [CC17], where the proof can only be verified by a dedicated party, i.e, some regulatory body. However, this party is not able of convincing third parties that the owner indeed controls the funds, which is a desirable property. Proving more complicated statements, such as that you control *at most* a certain amount of funds, is more challenging and requires more future research.

Practical Implementation. We have not yet implemented our idea in practice, but we believe that the general functionality is feasible and can be implemented on Ethereum. The validate callback, which is required to enforce the Safe Smart Account's policies, is not directly supported by Safe; we believe this could be implemented with a plugin. Regarding future work, the potential zero-knowledge proof statements could be more complex, which may also affect the gas costs for the transactions.

4.2 Next Steps

For the next steps, we plan to invest further research in ensuring stronger privacy guarantees while formalizing the security and privacy properties of our scheme. A natural first step would be a prototype implementation of our idea, which could be done in a follow-up grant. Simultaneously, fulfilling applicable compliance requirements using new techniques is definitely a key challenge. We appreciate any feedback and collaboration from others who work on related privacy-preserving technologies for the Safe ecosystem.

References

[Wik12] Bitcoin Wiki. *BIP 0032*. 2012. URL: https://en.bitcoin.it/wiki/BIP_0032 (visited on 03/20/2024).

[CC17] Pyrros Chaidos and Geoffroy Couteau. *Efficient Designated-Verifier Non-Interactive Zero-Knowledge Proofs of Knowledge*. Cryptology ePrint Archive, Paper 2017/1029. https://eprint.iacr.org/2017/1029. 2017. URL: https://eprint.iacr.org/2017/1029.