IEEE ICBC'24 1570978817

Privacy-Preserving Account-Abstraction for Teams on EVM chains

***Abstract*—This paper introduces ZK-Team, a solution addressing the challenges faced by organizations in enabling shared ownership of accounts among team members. The conventional method involves smart contracts, but usability issues and privacy concerns hinder practicality. ZK-Team leverages Ethereum's Account Abstraction (ERC-4337) and zero-knowledge proofs to allow shared accounts without dedicated EOAs (Externally Owned Accounts) for gas fees, ensuring privacy. Our solution includes an innovative mechanism for transparently managing zero-knowledge inputs, minimizing on-chain storage without relying on an off-chain storage. The paper discusses both the theoretical aspects of the solution and its implementation available as an open-source project.**

***Index Terms*—Blockchain, Smart Contracts, Decentralized Applications, Privacy, Account Abstraction, Zero-Knowledge Proofs**

I. Introduction

Let us consider the challenge faced by organizations seeking to facilitate shared ownership of an account by enabling team members to transact seamlessly as a unified entity. A typical solution involves locking funds in a smart contract and establishing rules to govern authorized individuals and their spending limits. That contract would automatically block transactions for users who have exhausted their allowances with these limits recorded in a simple data structure mapping user addresses to allowance amounts. Authorized users utilize their individual Externally Owned Account (EOA) to interact with the contract, ensuring compliance with their allowance before executing transactions.

While the development of such a smart contract is straightforward, two significant drawbacks hinder its practicality. First, there is a usability concern, as team members are burdened with maintaining dedicated EOAs provisioned with ETH to cover transaction gas fees. Second, privacy is compromised by the on-chain recording of each team member's address and their corresponding allowance values, a potential issue for teams, particularly small businesses, seeking to avoid public exposure of such information.

Addressing these challenges, our paper introduces ZK-Team, a novel solution empowering organizations to manage team members while preserving individual privacy. ZK-Team leverages two key technologies: 1) the newest Ethereum standard Account Abstraction (ERC-4337), enabling team members to use a shared account without requiring a dedicated EOA for gas fee payments, and 2) zero-knowledge proofs, ensuring confidential transactions by concealing individual addresses and respective allowances. Emphasizing usability, our solution incorporates an innovative mechanism to manage zero-knowledge inputs in a seamlessly transparent way for the administrators and users. This mechanism minimizes on-chain information storage without relying on an off-chain storage, enhancing overall security and efficiency.

The paper is structured into three parts. In section II, we details our zero-knowledge Account Abstraction code model, explaining how confidential transactions are enabled by concealing user identities and allowances through the submission of a zero-knowledge proof with each transaction. In section III, we delve into the management of zero-knowledge inputs using Hierarchical Deterministic Wallets. Additionally, we propose a method to prevent attacks on our wallet by detecting and neutralizing misuse or improper configurations that may compromise wallet funds. Section IV provides further insight into our implementation and deployment, now available as an open-source project. We elaborate on additional security measures, including protection against spoofing attacks, and practical considerations embedded in the architecture to fortify users privacy.

II. Zero-knowledge Account Abstraction

In this section, we lay out our zero-knowledge account abstraction. First, we demonstrate how account abstraction solves our usability problem by enabling users to send gasless transactions as a unified entity. Then, we illustrate how we integrate zero-knowledge proofs into our account abstraction to preserve users' privacy.

A. Using Account Abstraction

In a nutshell, Account Abstraction is a newer Ethereum standard (ERC-4337) [1], [2] that facilitates the creation of smart contracts without the need for individual Externally Owned Accounts (EOAs) to cover transactional gas fees. Instead of sending transactions signed by an EOA private key, users transmit pseudo-transaction objects known as "user operations". These operations are verified and executed by a specialized singleton smart contract referred to as an "EntryPoint" that will executed the actual transaction on behalf of an account abstraction instance. A user operation encapsulates crucial parameters, including the address of the account abstraction contract (`sender`), deployment code (`initCode`) for the contract, a nonce to prevent replay

attacks (`nonce`), the signature for validation (`signature`), and additional parameters for computing verification and transaction fees—details beyond the scope of this discussion.

Our focus centers on the `signature` field, acting as a means to authenticate users authorized to use a given instance of an account abstraction. In the ERC-4337 reference implementation, users sign the user operation with their EOA private key. While users are not obligated to have funds to cover gas fees, this practice exposes users' identity, as their address is embedded in the signature. Notably, the standard does not mandate the `signature` field to adhere strictly to an ECDSA signature and allows for custom authentication mechanisms.

In our ZK-Team solution, we propose adopting a zero-knowledge proof as a substitute for the ECDSA signature to authenticate user operations. The underlying concept entails users proving their status as authorized users without disclosing any identifying information. Beyond safeguarding user identity, our solution extends to protecting identity-related information, such as the user's allowance. Consequently, when users submit a transaction to the account abstraction, they must provide a proof validating the sender's identity and ensuring that the transaction value does not exceed the specified allowance. This zero-knowledge proof will be verified on-chain by the account abstraction contract based on public values that do not reveal any information about user identities and their allowances.

B. A first approach using zero-knowledge

Zero-knowledge proofs are commonly employed in numerous blockchain applications for confidential transactions (e.g., *Zcash* [3]) and scalability (e.g. zk-rollups in *Polygon* layer-2 [4]). In this proposal, we set aside the scalability use case to instead focus on leveraging zero-knowledge proofs for enabling confidential transactions.

With a zero-knowledge proof, you can demonstrate knowledge of a secret value w (referred to as the *witness*) such that $C(x, w) = \text{true}$ for a given function C (known as the *circuit*) and a public value x, all without revealing w. More specifically, a zero-knowledge proof comprises three algorithms, denoted as G, P, and V defined as follows:

- The key generator G takes a function C and generates two publicly available keys: a proving key p_k and a verification key v_k.
- The prover P takes as input the proving key p_k, a public input x, and a private witness w. The algorithm generates a proof $p = P(p_k, x, w)$, indicating that the prover knows a witness w satisfying the program.
- The verifier V computes $V(v_k, p, x)$, returning $true$ if the proof is correct, and $false$ otherwise. Thus, this function returns $true$ if the prover knows a witness w satisfying $C(x, w) = true$.

In this paper, we assume that G, P, and V are already provided by the underlying non-interactive zero-knowledge framework that we use (see section IV). Our contribution is to define:

- the private inputs w that will be used to generate the proof. Since the proof generation is done off-chain, these values should be known by the user only.
- the public inputs x that will be used to generate and verify the proof. Since the verification will be done on-chain, these values will be stored partly in the transaction and in the account abstraction contract.
- the circuit C that will be used to generate the prover code (embedded in the client code) and the verifier code (embedded in the smart contract code).

As a first approach, let's say that, to use their account, a user must prove that they know a secret value s (linked to their identity) and their allowance amount a. Instead of storing these values directly in the contract, we store a commitment hash $k = H(s, a)$ that ties these values together without revealing them. Thus, a user can generate a proof that they know s and a, and the contract can verify that these values are correct given k. Once the proof is verified, the contract is supposed to execute the transaction and update the commitment hash by decreasing the allowance. However, the contract cannot compute the new commitment hash without knowing the secret and the allowance amount in the first place. This means that the contract cannot create the next commitment hash, and it must be provided by the user itself necessarily.

As a second approach, let's make the user include in the transaction the new commitment hash that binds the updated allowance with a newer secret (to prevent a credential reuse attack [5]). Now it means that the user must also prove that this new commitment hash is somehow related to the previous one. Indeed, the user must prove that the "new" allowance (denoted a_n) for the new commitment hash is exactly equal to the "old" allowance (denoted a_o) value minus the transaction amount requested. The user sends the proof and the newer commitment hash to the contract. Then the contract will verify the proof given the old commitment hash k_o and the new one k_n. If the proof is valid, it means that the commitment hash is correct and the contract finally can execute the transaction before updating the stored commitment hash with its newest value.

Putting it all together, we obtain the following zero-knowledge proof system. The private inputs w are:

- a_o as the old allowance before the transaction
- s_o as the old secret that the user must know to use the allowance
- a_n as the new allowance after the transaction has been executed

- s_n as the new secret that the user will need to know for the next transaction

The public values x are:

- v as the amount spent in the transaction. It should correspond to the difference between the old allowance a_o and the new allowance a_n.
- k_o as the old commitment hash calculated as such $H(s_o, a_o)$
- k_n as the new commitment hash calculated as such $H(s_n, a_n)$

The zero-knowledge circuit C is defined as follows:

$$\begin{aligned} & v >= 0 \\ \wedge \quad & a_o >= v \\ \wedge \quad & a_n == a_o - v \\ \wedge \quad & k_o == H(s_o, a_o) \\ \wedge \quad & k_n == H(s_n, a_n) \end{aligned}$$

When the user wants to send a user operation to the account abstraction, they generate a proof $p = P(p_k, \{v, k_o, k_n\}, \{a_o, s_o, a_n, s_n\})$ and uses it as the user operation's signature. After sending that user operation to the *EntryPoint*, the account abstraction verifies that proof $V(v_k, p, v, k_o, k_n)$. If the proof is valid, the contract updates the allowance state by storing the new commitment hash k_n in lieu of the old one.

In this first approach of our protocol, an attacker cannot infer the user's identity nor the allowance by simply looking at individual transactions since the secrets and the allowances cannot be recovered from the hashes or the proofs. However, there is still a significant privacy issue considering that the attacker can look at all transactions together. Using the value, the old commitment hash and the new one, the attacker can link all of these transactions together and infer the total amount that each particular user has spent. Hence, we are facing a *linkability* problem.

C. Solving the linkability problem using proof of membership

We propose to solve the linkability problem identified in the previous section by using a technique called *proof of membership*. This technique is, for instance, used in cryptocurrency mixers such as *Tornado Cash* [6] to prevent attackers from linking deposit and withdraw transactions together. The idea behind proof of membership is that, rather than providing both the old commitment and the new commitment together, users will instead provide a proof showing that they know a secret and an allowance for one of the commitment hashes among all the ones previously stored but without specifying which one explicitly. This can be accomplished by using a Merkle-Tree [7] and embedding a Merkle-Tree proof inside the zero-knowledge proof [8].

However, using proof of membership raises yet another issue. If the user does not specify which old commitment hash is being used, he/she can reuse the same allowance since the contract cannot distinguish which commitment hash has been used already. To solve this problem, we are going to use a nullifier scheme by adding yet another secret value, called a *nullifier* n, to the commitment hash: $k = H(s, n, a)$. When using an allowance, the user must provide the hash of the previous nullifier $q = H(n)$, and the zero-knowledge proof should now check that the nullifier hash (given as a public value) is the hash of the same nullifier value used for the commitment hash. When the contract validates the proof, the contract stores the nullifier hash so that it cannot be reused. Using this scheme, an attacker is no longer capable of linking the nullifier hash with the corresponding commitment hash.

Using the proof of membership and the nullifier scheme, here is the new zero-knowledge proof system. The private inputs w are:

- a_o as the old allowance before the transaction
- s_o as the old secret that the user must know to use the allowance
- n_o as the old nullifier that the user must know to use its allowance once
- m_o as the old Merkle-Tree proof showing that the old commitment hash is in the old Merkle tree
- a_n as the new allowance after the transaction has been executed
- s_n as the new secret that the user will need to know for the next transaction
- n_n as the new nullifier that the user will need for the next transaction
- m_n as the new Merkle-Tree proof showing that the new commitment hash is in the new Merkle tree

The public value x are:

- v as the amount spent in the transaction. It should correspond to the difference between the old allowance a_o and the new allowance a_n.
- q_o as the old nullifier hash calculated as $H(n_o)$
- r_o as the old Merkle-Tree root
- k_n as the new commitment hash calculated as $H(s_n, n_n, a_n)$
- r_n of the new Merkle Tree once the k_n has been inserted in the tree

The new zero-knowledge circuit C goes as follows:

$$\begin{aligned} & v >= 0 \\ \wedge \quad & a_o >= v \\ \wedge \quad & a_n == a_o - v \\ \wedge \quad & q_o == H(n_o) \\ \wedge \quad & k_o == H(s_o, n_o, a_o) \\ \wedge \quad & \mathsf{validateMerkleProof}(m_o, r_o, k_o) \\ \wedge \quad & k_n == H(s_n, n_n, a_n) \\ \wedge \quad & \mathsf{validateMerkleProof}(m_n, r_n, k_n) \end{aligned}$$

The function $\mathsf{validateMerkleProof}(m, r, k)$ is an auxiliary circuit that checks that the Merkle-Tree proof (m) is valid given the root (r) and the leaf (k). Such a circuit has been

proposed in [9].

By using proof of membership, users can prove that they know the secret, the nullifier, and the allowance for one of the commitment hashes without explicitly specifying which one. This approach solves the linkability problem identified before, as an attacker can no longer link the old commitment hash with the new one. In addition, the nullifier hash prevents a user from using its allowance twice.

To implement this approach, our account abstraction contract should store several public values. First, it should maintain a Merkle-Tree with all commitment values to provide the two rightful tree root values when verifying the proof. Secondly, it should store all nullifier hashes after they have been used in a transaction. Beyond validating the proof, the contract must also check that the nullifier value given as public input has not been recorded already; otherwise, the transaction must be rejected.

III. Managing users secrets

In our solution, we assume that each ZK-Team account has an administrator capable of setting user allowances. Based on the zero-knowledge model detailed in the previous transaction, when the administrator wants to set a user's allowance, he needs to generate a triplet $(secret, nullifier, allowance)$, calculate the commitment hash $k = H(secret, nullifier, allowance)$, push this hash into the account contract's Merkle-Tree, and share this triplet with the user. Thus, when the user wants to use its newly created allowance to send a transaction, he should use the given triplet and generate a new one with a newer secret, nullifier, and the updated allowance based on the transaction value. That new triplet will be reused as input for the next transaction and so on and so forth.

However, if the admin wants to track users' allowances, he must be able to identify the origin of each user's transactions. Indeed, this is impossible with our model that has been built to protect users' privacy. An external observer cannot attribute ownership of each transaction. This is only possible if each user shares all of their secret and nullifier values with the administrator confidentially. Sharing secrets in a decentralized application can be done on-chain (encrypted) or off-chain (trusted storage). Sharing that many secrets and nullifiers on-chain would be expensive and prone to brute-force attacks. Similarly, ensuring the reliability and security of a trusted storage is complex.

In this section, we propose to solve this usability problem by avoiding storing these secret and nullifier values but having a deterministic way to generate them on demand based on a shared seed between the administrator and a given user. However, instead of reinventing the wheel, we have decided to use a Hierarchical Deterministic (HD) Wallet for that purpose.

A. Managing users' secrets and nullifiers using an HD Wallet

A Hierarchical Deterministic (HD) Wallet is a specialized type of cryptocurrency wallet that employs a deterministic wallet structure to generate an extensive hierarchy of cryptographic key pairs [10]. Unlike traditional wallets that generate keys randomly, HD wallets use a master seed, typically represented as a mnemonic phrase, from which an entire tree of keys can be derived in a predictable and reproducible manner. This hierarchical structure enhances security and convenience, allowing users to generate an infinite number of public and private key pairs from a single seed.

In our setting, we are not seeking to generate public and key pairs per se. But considering that those keys are 2^{236} unforgeable numbers, our idea is to use them as secret and nullifier values respectively. Moreover, the hierarchical structure of the HD wallet enables the administrator to manage multiple transactions over multiple users over multiple accounts by using the following path to derive the hierarchical keys:

$$\mathtt{m}/\mathsf{account}'/\mathsf{user}'/\mathsf{transaction}$$

- the account index allows the administrator to manage multiple ZK-Team accounts using the same mnemonic seed phrase.
- the user index allows the administrator to manage multiple users for a given ZK-Team account index
- the transaction index gives the secret (as the private key) and the nullifier (as the public key)

It is important to mention that the account index and the user index keys are "hardened" (denoted $'$ in the path). A hardened key is a type of cryptographic key that is derived from a parent key using a one-way function, making it computationally infeasible to deduce the parent key from the hardened child key. This additional layer of security ensures that compromising one key in the hierarchy does not compromise the entire wallet. In our context, we want to prevent any compromised account key or user key from putting other accounts and users at risk.

Now, let's see how to use this wallet in practice. Let's consider a scenario in which the administrator creates his first ZK-Team account (account index 0). First, the administrator creates an HD wallet from a seed that can be generated from a BIP-39 mnemonic phrase [11] (generated randomly the first time). Thus, to set the allowance for the first user (user index 0), the administrator needs to 1) generate a secret and a nullifier for the first transaction (transaction index 0) using the HD path $\mathtt{m/0'/0'/0}$ and push the commitment hash to the contract. Finally, the administrator shares the key $\mathtt{m/0'/0'}$ with the user that can then derive the same secret and nullifier values (child key 0), the next one (child key 1) and so on and so forth. By having access to the same series

of secrets and nullifiers, the administrator can de-anonymize user's transactions and track the changes in their allowances.

B. Detecting and correcting account misuses

When using our HD wallet solution, users are expected to always derive their secrets and nullifiers using their private key and following the sequence of transaction indexes. Thus, the administrator can generate the same sequence to de-anonymize transactions and track allowances. While this solution significantly improves the usability of our approach, it introduces a security flaw that allows users to emit transactions untraceable by the administrator. While crafting their n^{th} transaction, the user could generate an arbitrary secret and nullifier pair instead of using the one from the HD wallet at the transaction index $n + 1$. Once the transaction is executed, the administrator would still be able to de-anonymize this n^{th} transaction using the HD wallet at transaction index n. However, the user can now craft the $(n + 1)^{\text{th}}$ transaction using his rogue secret and nullifier values as inputs. Once that new transaction is executed, the administrator is incapable of de-anonymizing it using the HD wallet.

To mitigate this security flaw, we propose to give the administrator the capability of detecting and neutralizing fraudulent commitment hashes that can be used to execute untraceable transactions. Going back to our example, the transaction n^{th} is not illegitimate since the administrator can trace it. However, that transaction contains the next commitment hash that will be added to the Merkle-Tree by the account abstraction contract once the transaction is executed. This commitment hash has been built on the rogue secret and nullifier values and enables the user to make the next transaction untraceable.

Our idea is to give the capability to the administrator to monitor those new commitment hashes, detect the rogue ones, and discard them. By doing so, the user will not be able to use that fraudulent commitment hash, and the untraceable transaction will not be accepted by the contract. Detecting fraudulent commitment hashes cannot be done on-chain without revealing some of the private inputs. However, doing the detection off-chain is a straightforward process. When a new transaction is submitted, the administrator can check whether the value of the commitment hash corresponds to the one derived from the HD wallet. If these hashes mismatch, the administrator should discard it by removing the fraudulent commitment hash from the Merkle-Tree in the contract. By doing so, the user can no longer use it to execute an untraceable transaction and the user's account can no longer being used until the administrator resets the allowance.

IV. Implementation and further security considerations

Note to the reviewer: *All the code associated with this research will be shared with the open-source community. In adherence to the principles of the double-blind review process, explicit references to the public code repository are omitted. However, in the interest of ensuring reproducibility, the essential components of the code have been provided as figures.*

A. Implementation

The implementation of our solution consists of two major components: 1) the client, responsible for constructing zero-knowledge proofs and transmitting user operations, and 2) the account abstraction contract, which verifies proofs and executes transactions. It is crucial to note that our application is a fully decentralized application [12], as it operates without the need for any backend or trusted storage.

Our zero-knowledge proof system is implemented based on the succinct non-interactive argument of knowledge (ZK-SNARKs) [13], [14], specifically using the *Groth16* protocol [15]. The zero-knowledge circuit is specified in CIRCOM, a language for implementing zero-knowledge proof systems that has been introduced by Bellés-Muñoz et al. in [16]. Figure 1 illustrates the generation of the commitment hash and the Merkle-Tree. The function `InputHasher` takes the secret, nullifier, allowance, and an existing Merkle-Tree (via the variables `treeSiblings` and `treePathIndices`) as inputs and returns the commitment hash and the updated Merkle-Tree. We utilize the *Poseidon* hashing library [17] and adopt the Merkle-Tree definition from *the Zk-Kit* library [18]. Figure 2 showcases the implementation of our zero-knowledge circuit, which encapsulates our core model detailed in Section II-C. Ultimately, we employ the *SnarkJs* library [19] to compile our circuit into two components: the client code (written in WebAssembly), used by users to generate proofs, and the smart contract code (written in Solidity), employed by our account abstraction contract to verify proofs.

Our account abstraction contract extends the ERC-4337 reference implementation [20]. As depicted in Figure 3, our contract incorporates additional variables for storing the Merkle-Tree (using the *Zk-Kit* library again) and the utilized nullifier hashes. The `ValidateSignature` function initially checks whether the `signature` field of the user operation constitutes a valid ECDSA signature or a zero-knowledge proof. In the case of a signature, indicating a user operation for setting a user allowance, it must originate from the administrator (in this case, the `owner`) necessarily (lines 13-16).

If the signature is a zero-knowledge proof, the contract extracts its arguments (lines 18-21) and verifies the proof (lines 40-45) using the verifier contract (generated from the CIRCOM circuit and deployed at the address stored in the variable `verifier`). However, mere proof verification is insufficient. The contract must ensure that: 1) the public signals, such as the transaction value, commitment hash, and nullifier hash, correspond to those in the user operation

Fig. 1. Building Commitment Hash and the Merkle-Tree

```
template InputHasher(levels){
    signal input secret;
    signal input nullifier;
    signal input allowance;

    signal output commitmentHash;
    component commitmentHasher = Poseidon(3);
    commitmentHasher.inputs <== [nullifier, secret, allowance];
    commitmentHash <== commitmentHasher.out;

    signal input treeSiblings[levels];
    signal input treePathIndices[levels];
    signal output root;

    component tree = MerkleTree(levels);
    tree.leaf <== commitmentHasher.out;
    for (var i = 0; i < levels; i++) {
        tree.siblings[i] <== treeSiblings[i];
        tree.pathIndices[i] <== treePathIndices[i];
    }

    root <== tree.root;
}
```

`callData` (lines 22-30); 2) the old root embedded in the proof matches the current Merkle-Root (lines 33-34); and 3) the new root corresponds to the latest root after simulating the insertion of the commitment hash (lines 35-36). Finally, the contract checks that the nullifier hash has not been used previously (lines 31-32).

B. Preventing spoofing attacks

In cases where the user operation's `signature` parameter represents a legitimate ECDSA signature, any attempt by an attacker to modify other user operation parameters would render that signature invalid. Subsequently, signature verification would fail, leading to the transaction being reverted. However, caution is warranted when the `signature` parameter is a zero-knowledge proof. In this scenario, an attacker could execute a front-running attack by crafting a new `calldata` parameter that shares the same value, commitment hash, and nullifier hash but designates a different recipient. Since the zero-knowledge proof is unrelated to the recipient in the `calldata` parameter, the proof would still be valid. Such a manipulated user operation could succeed, enabling the attacker to misappropriate funds from the account.

To thwart such spoofing attacks, it is imperative to establish a connection between the recipient but more generally the `calldata` value and the zero-knowledge proof, ensuring that any modification to the transaction `callData` renders the zero-knowledge proof invalid. This rationale underscores the inclusion of a public signal named `callDataHash` in our CIRCOM circuit, representing the hash of the user operation's `callData` parameter. Additionally, we introduced an ancillary public signal, `callDataHashSquared` (calculated as the square of `callDataHash` on lines 54-56), to circumvent potential circuit optimization by the CIRCOM compiler, which might remove `callDataHash` from the proof if deemed unused. Ultimately, the contract must verify that the user operation's `callData` remains unaltered compared to the data presented in the proof (refer to Figure 3, lines 37-39).

C. Better privacy when using ZK-Team accounts

Our solution safeguards user privacy through two key mechanisms: 1) preventing user operations from divulging any personally identifiable information, and 2) preventing the linkage of transactions with one another. However, it is crucial to recognize potential threats in a production setting, where an attacker may operate clandestinely behind the bundler or network node utilized by users. In such scenarios, the attacker might seek to correlate users' IP addresses with their corresponding user operations or network requests, introducing a potential privacy vulnerability that we ought to take into account.

To enhance privacy defenses, we suggest two strategies: organizations can either employ a trusted proxy to safeguard their users; and/or users can leverage privacy tools, such as the Tor network [21]. Implementing either of these solutions serves as an intermediary between users and the network, effectively obscuring the origin of network requests and thwarting attempts to link IP addresses with specific operations.

V. Conclusion

ZK-Team presents a comprehensive solution for organizations seeking efficient team management while mitigating the usability and privacy challenges associated with shared account ownership. Leveraging the novel Ethereum standard,

"Account Abstraction" (ERC-4337), it facilitates the creation of smart contracts without requiring individual Externally Owned Accounts (EOAs) to cover transactional gas fees. Our solution uses zero-knowledge proofs to ensure that user operations remain devoid of personally identifiable information, and transactions retain unlinkability. Another contribution is to use an HD wallet for user management as a way to provide a user-friendly solution to create all zero-knowledge inputs deterministically but yet confidentially. This approach enables administrators to easily monitor user allowances and rectify any potential misuse. In the end, we ambition our solution to be an appropriate turnkey approach to team and asset management.

REFERENCES

[1] V. Buterin *et al.*, "Ethereum: a next generation smart contract and decentralized application platform (2013)," *URL {http://ethereum. org/ethereum. html}*, 2017.

[2] Ethereum Foundation, "Erc-4337: Account abstraction using alt mempool," https://eips.ethereum.org/EIPS/eip-4337, 2021, ethereum Improvement Proposal, ERC-4337.

[3] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox *et al.*, "Zcash protocol specification," *GitHub: San Francisco, CA, USA*, vol. 4, no. 220, p. 32, 2016.

[4] L. T. Thibault, T. Sarry, and A. S. Hafid, "Blockchain scaling using rollups: A comprehensive survey," *IEEE Access*, 2022.

[5] M. Albrecht, K. Paterson, and G. Watson, "Dancing on the lip of the volcano: Chosen ciphertext attacks on apple imessage," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[6] A. Pertsev, R. Semenov, and R. Storm, "Tornado cash privacy solution version 1.4," *Tornado cash privacy solution version*, vol. 1, 2019.

[7] R. C. Merkle, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 20, no. 7, pp. 396–404, 1987.

[8] X. Yang and W. Li, "A zero-knowledge-proof-based digital identity management scheme in blockchain," *Computers & Security*, vol. 99, p. 102050, 2020.

[9] S. Micali, M. Rabin, and J. Kilian, "Zero-knowledge sets," in *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 2003, pp. 80–91.

[10] P. Wuille and M. Hearn, "Bip32: Hierarchical deterministic wallets," Bitcoin Improvement Proposal, 2012, https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

[11] e. a. Narayanan, S., "Bip-39: Mnemonic code for generating deterministic keys," Bitcoin Improvement Proposal, 2013, https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[12] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. Leung, "Decentralized applications: The blockchain-empowered software system," *IEEE Access*, vol. 6, pp. 53 019–53 033, 2018.

[13] J. Groth, "Short non-interactive zero-knowledge proofs," in *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 2010, pp. 341–358.

[14] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Communications of the ACM*, vol. 59, no. 2, pp. 103–112, 2016.

[15] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 2016, pp. 305–326.

[16] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[17] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for {Zero-Knowledge} proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 519–535.

[18] P. . S. Explorations, "ZK-kit," https://github.com/privacy-scaling-explorations/zk-kit, 2022.

[19] iden3, "SnarkJS," https://github.com/iden3/snarkjs, 2022.

[20] E. Infinitism, "Account Abstraction," https://github.com/eth-infinitism/account-abstraction, 2023.

[21] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999.

Fig. 2. Building the Transaction Proof

```
template Transact(levels) {

      // public input values
      signal input value;
      signal input callDataHash;

      // public output values
      signal output oldNullifierHash;
      signal output oldRoot;
      signal output newCommitmentHash;
      signal output newRoot;

      // private old input values
      signal input oldAllowance;
      signal input oldNullifier;
      signal input oldSecret;
      signal input oldTreeSiblings[levels];
      signal input oldTreePathIndices[levels];

      component oldInputHasher = InputHasher(levels);
      oldInputHasher.nullifier <== oldNullifier;
      oldInputHasher.secret <== oldSecret;
      oldInputHasher.allowance <== oldAllowance;
      oldInputHasher.treeSiblings <== oldTreeSiblings;
      oldInputHasher.treePathIndices <== oldTreePathIndices;

      component nullifierHasher = Poseidon(1);
      nullifierHasher.inputs <== [oldNullifier];
      oldNullifierHash <== nullifierHasher.out;
      oldRoot <== oldInputHasher.root;

      // private new input values
      signal input newAllowance;
      signal input newNullifier;
      signal input newSecret;
      signal input newTreeSiblings[levels];
      signal input newTreePathIndices[levels];

      component newInputHasher = InputHasher(levels);
      newInputHasher.nullifier <== newNullifier;
      newInputHasher.secret <== newSecret;
      newInputHasher.allowance <== newAllowance;
      newInputHasher.treeSiblings <== newTreeSiblings;
      newInputHasher.treePathIndices <== newTreePathIndices;

      newCommitmentHash <== newInputHasher.commitmentHash;
      newRoot <== newInputHasher.root;

      // check value and allowances
      assert(value >= 0);
      assert(oldAllowance >= value);
      assert(newAllowance == oldAllowance - value);

      // hidden signals to prevent spoofing
      signal callDataHashSquared;
      callDataHashSquared <== callDataHash * callDataHash;
}

component main {public [value, callDataHash]} = Transact(20);
```

Fig. 3. ZK-Team Account Abstraction Contract

```
contract ZkTeamAccount is BaseAccount, TokenCallbackHandler, UUPSUpgradeable, Initializable {

    MerkleTreeData public tree;
    mapping(uint256 => bytes32) public nullifierHashes;
    Groth16Verifier public immutable verifier;

    ...

    function validateSignature(UserOperation calldata userOp, bytes32 userOpHash)
    public view returns (uint256 validationData) {
        // check if the signature is an ECDSA signature or else zk-proof
        if (userOp.signature.length == 65){
            // check owner signature
            bytes32 hash = userOpHash.toEthSignedMessageHash();
            if (owner != hash.recover(userOp.signature)) return SIG_VALIDATION_FAILED;
            else return 0;
        } else {
            // extract public signals from proof
            ( uint256[2] memory pi_a, uint256[2][2] memory pi_b,
              uint256[2] memory pi_c, uint256[6] memory signals )
              = abi.decode(userOp.signature, (uint256[2], uint256[2][2], uint256[2], uint256[6]));
            // extract data from callData
            ( uint256 nullifierHash, uint256 commitmentHash, uint256 value )
              = abi.decode(userOp.callData[4:100], (uint256, uint256, uint256));
            // check value matches calldata
            if (value != signals[4]) return SIG_VALIDATION_FAILED;
            // check commitmentHash matches the callData
            if (commitmentHash != signals[2]) return SIG_VALIDATION_FAILED;
            // check nullifierHash matches the callData
            if (nullifierHash != signals[0]) return SIG_VALIDATION_FAILED;
            // check nullifierHash has not been used already
            if (nullifierHashes[nullifierHash] != bytes32(0))  return SIG_VALIDATION_FAILED;
            // check old root
            if (tree.root == signals[1]) return SIG_VALIDATION_FAILED;
            // check new root
            if (signals[3] != tree.simulatedInsert(commitmentHash)) return SIG_VALIDATION_FAILED;
            // check callData hash matches the hash of calldata
            uint hash = PoseidonT2.hash([uint(keccak256(userOp.callData))]);
            if (hash != signals[5])  return SIG_VALIDATION_FAILED;
            // check the proof
            (bool valid, ) = address(verifier).staticcall(
                                abi.encodeWithSelector(Groth16Verifier.verifyProof.selector,
                                pi_a, pi_b, pi_c, signals
                            ));
            if (!valid) return SIG_VALIDATION_FAILED;
            // finally
            return 0;
        }
    }
    ...
}
```