Poster: Azeroth: Auditable Zero-knowledge Transaction in Smart Contracts

Gweonho Jeong  
*Hanyang University*  
Seoul, Republic of Korea  
kwonhojeong@hanyang.ac.kr

Nuri Lee  
*Kookmin University*  
Seoul, Republic of Korea  
nuri@kookmin.ac.kr

Jihye Kim  
*Kookmin University*  
Seoul, Republic of Korea  
jihyek@kookmin.ac.kr

Hyunok Oh  
*Hanyang University*  
Seoul, Republic of Korea  
ho@hanyang.ac.kr

I. Introduction

In this paper, we propose an auditable zero-knowledge transaction framework called Azeroth based on zk-SNARK [1]. The Azeroth framework provides privacy, verifiability, and auditability for personal digital assets while maintaining its efficiency of transactions. Azeroth preserves the original functionality of the account-based blockchain as well as providing the additional zero-knowledge feature to meet the standard privacy requirements. Azeroth is devised using encryption for two-recipients (i.e., the recipient and the auditor) so that the auditor can audit all transactions. While the auditor can audit a transaction, it cannot manipulate any transaction. Azeroth enhances the privacy of the transaction by performing multiple functions such as deposit, withdrawal, and transfer in one transaction. For the real-world use, we adopt a SNARK-friendly hash algorithm [2], [3] to instantiate encryption to have an efficient proving time and execute experiments in various platforms.

II. Data Structure

Account. There are two types of accounts in Azeroth: an externally owned account denoted as EOA, and an encrypted account denoted as ENA. Unlike traditional account type (EOA), ENA keeps a ciphertext indicating plain amount of account. The smart contract of Azeroth controls the ENA’s lifecycle.

Auditor key. An auditor generates a pair of private/public keys (ask, apk) used in the public key system; apk is used when a user generates an encrypted transaction, while ask is used when an auditor needs to audit the ciphertext.

User key. Each user generates a pair of private/public keys (usk = (kENA, skown, shenc), upk = (addr,(pkown, pkenc)).

- kENA: It indicates a secret key for encrypted account of ENA in a symmetric-key encryption system.
- (shenc, pkenc): These keys are used in a public-key encryption system; shenc is used to encrypt ciphertexts taken from transactions while pkenc is to encrypt transactions.
- addr: It is a user address and computed by hashing pkenc.

III. Scheme

A. Overview

We construct Azeroth by integrating deposit/withdrawal transactions and public/private transfer transactions to a single transaction zkTransfer. Figure 1 illustrates the zkTransfer. In zkTransfer, vpub and vpub are publicly visible values, vENA indicates the decrypted value of sct. The updated vnew priv is encrypted and stored as sct in ENA. The amount (vnew priv) included in a commitment can be used as input if a user has its opening key; the opening key is delivered in a ciphertext pct so that only the destined user can correctly decrypt it. To prevent from double spending, for each spent commitment a nullifier is generated by hashing the commitment and the private key skown, and appended to the blockchain. Finally, zkTransfer proves that all of the above procedures are correctly performed by generating a zk-SNARK proof [1]. Auditability is achieved by utilizing a public key encryption with two recipients; all pct ciphertexts can be decrypted by an auditor as well as a receiver so that the auditor can monitor all the transactions.

B. Algorithms

Azeroth consists of three components: Client, Smart Contract, and Relation.

[Azeroth Client]

1. SetupClient(1λ, Rzkt) → pp: It takes a security parameter and a relation as input, and returns the public parameter pp.
2. KeyGenAuditClient(pp) → (ask, apk), TxKG: This algorithm takes a public parameter pp and outputs an auditor pair (ask, apk), and a transaction TxKG to register the auditor public key.
3. KeyGenUserClient(pp) → (usk, upk), TXKG: This algorithm takes a public parameter pp and outputs a user pair (usk, upk) = ((kENA, skown, shenc), (addr, pkown, pkenc)), and a transaction TXKG to register the user public key.
4. zkTransferClient(note, apk, usk, upk, vpub, vpub, vpub, vpub, vpub, vpub, vpub, vpub, vpub, EOAvpub) → Txzk: On inputs of a note, an auditor key, a sender’s key pair, a receiver’s key pair, a sender’s public key, public/private value amounts, and the receiver account EOAvpub for vpub, this algorithm collects values from a sender’s EOA, ENA, and a note, then sends...
them to a receiver's EOA, and a new commitment. The remaining balance is stored back to the sender's ENA. The internal procedures are described as follows:

1) Consuming note(cm, o, v): It proves the knowledge of v using the opening o and the membership of a commitment cm, and derives a nullifier nf.
2) Generating cm\_new: By executing COM(v^\text{out}, addr\_recv; o_{\text{new}}), a new commitment and an opening are obtained. Then it encrypts (o_{\text{new}}, v^\text{out}; addr\_recv) and outputs pct.
3) Processing currency: The sender's ENA balance is updated as v^\text{new} = v^\text{old} + v^\text{in} - v^\text{out}.

With prepared witnesses and statements, the algorithm generates a zk-SNARK proof and finally outputs a zkTransfer transaction Tx\_ZKT.

[Azeroth Smart Contract]

In the smart contract, zkTransfer checks the validity of the transaction such as verifying a proof. If the transaction is valid, the smart contract processes the transaction such as updating the MT with cm\_new and appending the nullifier nf to List\_nf.

[Azeroth Relation]

The statement and witness of Relation R\_ZKT are as follows:

\[ \vec{x} = (\text{apk}, rt, nf, \text{upk}_{\text{send}}, cm_{\text{new}}, \text{sct}_{\text{old}}, \text{sct}_{\text{new}}, v^\text{pub}, v^\text{out}, v^\text{in}, \text{pct}_{\text{new}}) \]
\[ \vec{w} = (\text{usk}_{\text{send}}, cm_{\text{old}}, o_{\text{old}}, v^\text{priv}, \text{upk}_{\text{recv}}, o_{\text{new}}, v^\text{priv}, v^\text{out}, \text{aux}_{\text{new}}, \text{Path}) \]

We say that a witness \( \vec{w} \) is valid for a statement \( \vec{x} \), if and only if the following holds:

1) If \( v^\text{priv} > 0 \), then cm\_old exists in MT with given rt and Path.
2) \( pk_{\text{send}} = \text{CRH}(x_{\text{sk}_{\text{send}}}) \).
3) The user address addr\_send and addr\_recv are well-formed.
4) cm\_old and cm\_new are valid.
5) nf is derived from cm\_old and sct_{\text{old}}.
6) \text{pct}_{\text{new}} is an encryption of cm\_new via aux_{\text{new}}.
7) \text{sct}_{\text{new}} is an encryption of updated ENA balance.
8) All amounts (e.g., \( v^\text{in}, v^\text{out}, \ldots \)) are not negative.

IV. EXPERIMENT

In our experiment, the term \( \text{cfg}_{\text{HashDepth}} \) denotes a configuration of Merkle hash tree depth and hash type in Azeroth. The experiment is executed on a Server\(^1\).

Overall performance. The execution time 4.04s of Setup is composed of the zk-SNARK key generation time 2.2s and the deployment time 1.84s of the Azeroth's smart contract to the blockchain. Setup consumes a considerable amount of gas due to the initialization of Merkle Tree. In zkTransfer, the executed time is 4.38s including both the Client part and the Smart Contract part. The gas is mainly consumed to verify the SNARK proof and update the Merkle hash tree.

| Table I | EXECUTION TIME AND GAS USAGE OF Azeroth with \( \text{cfg}_{\text{MMCC7,32}} \) [1] |
|---|---|---|---|---|---|
| | Setup | RegisterAuditor | RegisterUser | zk transfer | Audit |
| Time (s) | 4.04 | 0.02 | 0.017 | 4.38 | 0.03 |
| Gas | 5,790,800 | 63,179 | 45,543 | 1,535,937 | N/A |

REFERENCES


\(^1\)3.10GHz Intel Xeon Gold 6264R, 256GB, and Ubuntu 20.04