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
hoh@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 $(\mathsf{ask}, \mathsf{apk})$ 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 $(\mathsf{usk} = (k_{\mathsf{ENA}}, sk_{\mathsf{own}}, sk_{\mathsf{enc}}), \mathsf{upk} = (\mathsf{addr}, pk_{\mathsf{own}}, pk_{\mathsf{enc}}))$.

- k_{ENA} : It indicates a secret key for encrypted account of ENA in a symmetric-key encryption system.
- $(sk_{\mathsf{own}}, pk_{\mathsf{own}})$: pk_{own} is computed by hashing sk_{own}. The key pair is used to prove the ownership of an account in a transaction. Note that sk_{own} is additionally used to generate a nullifier, which prevents double-spending.
- $(sk_{\mathsf{enc}}, pk_{\mathsf{enc}})$: These keys are used in a public-key encryption system; sk_{enc} is used to decrypt ciphertexts taken from transactions while pk_{enc} is to encrypt transactions.
- addr : It is a user address and computed by hashing pk_{own} and pk_{enc}.

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, $\mathsf{v}_{\mathsf{in}}^{\mathsf{pub}}$ and $\mathsf{v}_{\mathsf{out}}^{\mathsf{pub}}$ are publicly visible values. $\mathsf{v}^{\mathsf{ENA}}$ indicates the decrypted value of sct. The updated $\mathsf{v}_{\mathsf{new}}^{\mathsf{ENA}}$ is encrypted and stored as sct^* in ENA. The amount $(\mathsf{v}_{\mathsf{in}}^{\mathsf{priv}})$ 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 sk_{own}, 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**]

- $\mathsf{Setup}_{\mathsf{Client}}(1^\lambda, \mathcal{R}_{\mathsf{ZKT}}) \rightarrow \mathsf{pp}$: It takes a security parameter and a relation as input, and returns the public parameter pp.
- $\mathsf{KeyGenAudit}_{\mathsf{Client}}(\mathsf{pp}) \rightarrow (\mathsf{ask}, \mathsf{apk}), \mathsf{Tx}_{\mathsf{KGA}}$: This algorithm takes a public parameter pp and outputs an auditor pair $(\mathsf{ask}, \mathsf{apk})$, and a transaction $\mathsf{Tx}_{\mathsf{KGA}}$ to register the auditor public key.
- $\mathsf{KeyGenUser}_{\mathsf{Client}}(\mathsf{pp}) \rightarrow (\mathsf{usk}, \mathsf{upk}), \mathsf{Tx}_{\mathsf{KGU}}$: This algorithm takes a public parameter pp and outputs a user key pair $(\mathsf{usk}, \mathsf{upk}) = ((k_{\mathsf{ENA}}, sk_{\mathsf{own}}, sk_{\mathsf{enc}}), (\mathsf{addr}, pk_{\mathsf{own}}, pk_{\mathsf{enc}}))$, and a transaction $\mathsf{Tx}_{\mathsf{KGU}}$ to register the user public key.
- $\mathsf{zkTransfer}_{\mathsf{Client}}(\mathsf{note}, \mathsf{apk}, \mathsf{usk}^{\mathsf{send}}, \mathsf{upk}^{\mathsf{send}}, \mathsf{upk}^{\mathsf{recv}}, \mathsf{v}_{\mathsf{out}}^{\mathsf{priv}}, \mathsf{v}_{\mathsf{in}}^{\mathsf{pub}}, \mathsf{v}_{\mathsf{out}}^{\mathsf{pub}}, \mathsf{EOA}^{\mathsf{recv}}) \rightarrow \mathsf{Tx}_{\mathsf{ZKT}}$: On inputs of a note, an auditor public key apk, a sender's key pair, a receiver's public key, public/private value amounts, and the receiver account $\mathsf{EOA}^{\mathsf{recv}}$ for $\mathsf{v}_{\mathsf{out}}^{\mathsf{pub}}$, this algorithm collects values from a sender's EOA, ENA, and a note, then sends

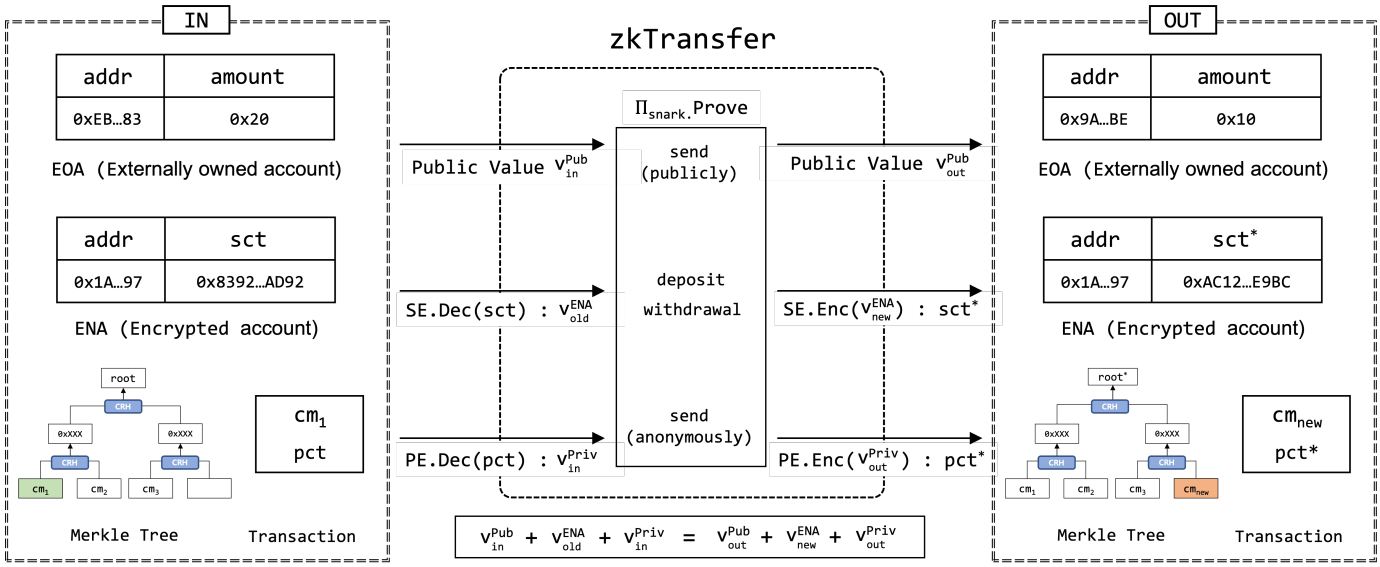

Fig. 1. Overview of zkTransfer

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

With prepared witnesses and statements, the algorithm generates a zk-SNARK proof and finally outputs a zkTransfer transaction $\mathsf{Tx}_{\mathsf{ZKT}}$.

[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 $\mathsf{cm}_{\mathsf{new}}$ and appending the nullifier nf to $\mathsf{List}_{\mathsf{nf}}$.

[Azeroth **Relation**]

The statement and witness of Relation $\mathcal{R}_{\mathsf{ZKT}}$ are as follows:

$$\vec{x} = (\mathsf{apk}, \mathsf{rt}, \mathsf{nf}, \mathsf{upk}^{\mathsf{send}}, \mathsf{cm}_{\mathsf{new}}, \mathsf{sct}_{\mathsf{old}}, \mathsf{sct}_{\mathsf{new}}, \mathsf{v}_{\mathsf{in}}^{\mathsf{pub}}, \mathsf{v}_{\mathsf{out}}^{\mathsf{pub}}, \mathsf{pct}_{\mathsf{new}})$$

$$\vec{w} = (\mathsf{usk}^{\mathsf{send}}, \mathsf{cm}_{\mathsf{old}}, \mathsf{o}_{\mathsf{old}}, \mathsf{v}_{\mathsf{in}}^{\mathsf{priv}}, \mathsf{upk}^{\mathsf{recv}}, \mathsf{o}_{\mathsf{new}}, \mathsf{v}_{\mathsf{out}}^{\mathsf{priv}}, \mathsf{aux}_{\mathsf{new}}, \mathsf{Path})$$

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

1) If $\mathsf{v}_{\mathsf{in}}^{\mathsf{priv}} > 0$, then $\mathsf{cm}_{\mathsf{old}}$ exists in MT with given rt and Path.
2) $pk_{\mathsf{own}}^{\mathsf{send}} = \mathsf{CRH}(sk_{\mathsf{own}}^{\mathsf{send}})$.
3) The user address $\mathsf{addr}^{\mathsf{send}}$ and $\mathsf{addr}^{\mathsf{recv}}$ are well-formed.
4) $\mathsf{cm}_{\mathsf{old}}$ and $\mathsf{cm}_{\mathsf{new}}$ are valid.
5) nf is derived from $\mathsf{cm}_{\mathsf{old}}$ and $sk_{\mathsf{own}}^{\mathsf{send}}$.
6) $\mathsf{pct}_{\mathsf{new}}$ is an encryption of $\mathsf{cm}_{\mathsf{new}}$ via $\mathsf{aux}_{\mathsf{new}}$.
7) $\mathsf{sct}_{\mathsf{new}}$ is an encryption of updated ENA balance.
8) All amounts (e.g., $\mathsf{v}_{\mathsf{in}}^{\mathsf{priv}}, \mathsf{v}_{\mathsf{in}}^{\mathsf{pub}}, \ldots$) are not negative.

IV. Experiment

In our experiment, the term $\mathsf{cfg}_{\mathsf{Hash},\mathsf{Depth}}$ 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 $\mathsf{cfg}_{\mathsf{MiMC7},32}$ [1]

	Azeroth				
	Setup	RegisterAuditor	RegisterUser	zkTransfer	Audit
Time (s)	4.04	0.02	0.017	4.38	0.03
Gas	5,790,800	63,179	45,543	1,555,957	N/A

References

[1] Groth, Jens, "On the Size of Pairing-Based Non-interactive Arguments," EUROCRYPT, pp. 305-326, 2016.
[2] Martin R. Albrecht and Lorenzo Grassi and Christian Rechberger and Arnab Roy and Tyge Tiessen, "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity," ASIACRYPT (1), pp. 191-219, 2016.
[3] Lorenzo Grassi and Dmitry Khovratovich and Christian Rechberger and Arnab Roy and Markus Schofnegger, "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems," USENIX Security 21, pp. 519-535, 2021.

[1] 3.10GHz Intel Xeon Gold 6264R, 256GB, and Ubuntu 20.04