IEEE ZKDAPPS'2024 1571004627

Portal: Time-bound and Replay-resistant Zero-knowledge Proofs for Single Sign-On

***Abstract*—Latest identity systems rely on public blockchains to enhance user autonomy and reduce tracking from conventional identity providers. At the same time, identity systems integrate novel technologies such as zero-knowledge proofs (ZKPs) to improve data privacy and data compliance. We show that a naive verification of ZKPs at smart contracts enables replay attacks: Attackers can replay ZKPs at arbitrary times without having access to the private inputs that are required for the computation of the ZKP. To solve this problem, we construct a transaction sequence which verifies time-bound and replay-resistant ZKPs at smart contracts. Our construction introduces an additional but constant fee of 0.14$ per verification of a ZKP on the public blockchain Ethereum. With our new construction, we propose Portal, a novel identity system for decentralized single sign-on.**

***Index Terms*—Zero-knowledge Proofs, Smart Contracts, Decentralized Resolution, Single Sign-On**

I. Introduction

Motivation: Almost every service of today's web manages *users* based on an identifiable session and requires a mechanism to authenticate *users* beforehand. The *user* authentication uniquely identifies every *user* of the system and guarantees that the session is unique to one *user*. To avoid each web service from implementing their own identity and authentication system, *OpenID Connect*, as the latest Single Sign-On (SSO) protocol, was standardized in 2014 [1]. The SSO paradigm delegates *user* authentication at a web service towards a third-party Identity Provider (IdP), which handles the unique identification of the *user* (cf. case *a* in Figure 1).

Even though delegated authorization and authentication via SSO is very convenient and cheap for *users*, IdPs can track every log-in and data access of a *user*. To solve the misaligned incentives between all parties, recent approaches such as Sign-In with Ethereum (SIWE) [2] or Polygon ID [3] replace the IdP with a public blockchain and provide *users* with new notions of autonomy [4]. Further, Polygon ID [3] employs Zero-knowledge Proof (ZKP) technology to enhance data privacy and data compliance of *users*. Modern identity systems rely on certification ecosystems, where issuers verify and attest to data claims made by *users* [3]. Similarly, recent works [5] rely on assumptions (e.g. existence of trustworthy issuers) which go beyond the requirements of SSO systems. Because, in the trust establishment phase of SSO systems, *users* agree to the IdPs's terms and conditions that require *users* to honestly create profiles without requesting specific credentials [1].

Challenge: In this work and according to the requirements found in SSO systems, we investigate the honest creation and management of data claims, which does not require any form of third-party attestation. In this scenario, we entirely

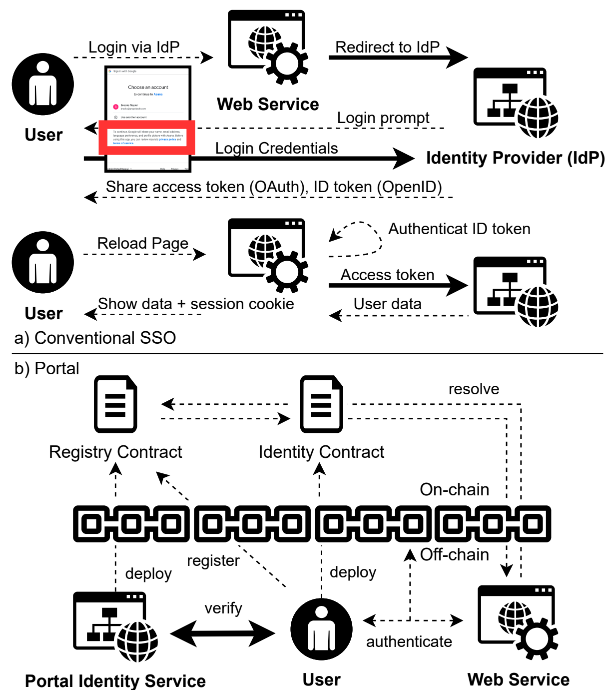

Fig. 1. a) Overview of the Single Sign-On (SSO) delegated authentication and authorization where the *user* agrees to a fixed policy (red box) of the Identity Provider (IdP). Bold arrows indicate *user*-to-IdP interactions which track *user* activities. b) Simplified view of the *Portal* identity system, where *users* manage data and authenticate towards web services with self custody.

rely on the interaction between *users* and smart contracts, where smart contracts verify the claims made by *users*. To create universal data profiles, *users* convince smart contracts of data compliance according to a public statement. If the smart contract successfully verifies the claim, then the smart contract accepts a mapping between the claim and the address of the *user*. If *users* prove claims on private data, then the smart contracts validate ZKPs asserting the claim. Based on accepted claims, *users* can authenticate to any third party.

Contribution: In this scenario, we find that replay attacks are a concern because blockchain transaction logs transparently expose transaction payloads to adversaries. Thus, any claim on public data can be replayed by re-executing the same contract functionality. However, blockchains, per default, bind transactions to specific timestamps such that *users* can be hold accountable for any claims that have been made in the past.

For claims on private data, we show that replay attacks can be prevented. To do so, we introduce a new transaction sequence which unequivocably binds the proof computation to a specific *user* and time (cf. Section IV-C). Instead of using a verifier-chosen nonce that binds a proof presentation to a verification session [5], [6], our transaction sequence relies on the blockchain Proof of Stake (PoS) randomness as the verifier-chosen nonce. Our transaction sequence achieves an efficient cost structure as it does not require additional contracts that prevent replay attacks (e.g. access control smart contracts [7]). Based on this contribution, we propose a novel identity system, called *Portal*, which supports on-chain and off-chain validations of ZKPs on *user* data during *user* authentication (cf. bottom part of Figure 1). In summary,

- We construct a new transaction sequence to secure the on-chain ZKP verification against replay attacks.
- We propose *Portal*, an alternative SSO solution with enhanced privacy and control.
- Concerning *Portal*, we open-source[1] a proof of concept, analyse the security (cf. Section IV-C), and evaluate the cost-efficiency (cf. Section VI).

In systems with strong know your customer (KYC) requirements, where users cannot be trusted to responsibly operate claims, we want to highlight that *Portal* should and can be used with third-party attestations.

[1]https://github.com/anonsubsub/portal

II. Preliminaries

A. Public Key Cryptography & Digital Signatures

Public Key Cryptography (PKC) systems provide users with complementing key pairs to enable applications such as digital signatures or asymmetric encryption. The key property of PKC is that the keying material at users consists of a public (*public key*) and private (*private key*) part, where the private part is never disclosed. Using PKC, we define a digital signature scheme on a message string m with the algorithms, where

- **pk.Setup**$(1^\lambda) \rightarrow (sk,\ pk)$ uses a security parameter to output a PKC private key sk and public key pk.
- **pk.Sign**$(sk,\ m) \rightarrow (\sigma)$ takes as input the secret key and a message string m, and outputs the signature σ.
- **pk.Verify**$(pk,\ m,\ \sigma) \rightarrow \{0, 1\}$ takes as input the public key, the message message string, and a signature, and outputs either a 1 if the signature verification succeeds. Otherwise, the output is a 0.

B. Zero-knowledge Proof System

A general-purpose ZKP system allows a *prover* to convince a *verifier* of knowing a secret witness w which satisfies a general statement expressed via a circuit $\mathcal{C}$. The *verifier* relies on an polynomial time algorithm to verify, according to a NP language, if w is a valid proof of the statement in $\mathcal{C}$. A ZKP system achieves the properties of (i) *completeness*, which ensures that an honest *prover* with a valid witness convinces an honest *verifier*, (ii) *soundness*, which ensures that a cheating *prover* without a valid witness cannot convince an honest *verifier*, and (iii) *zero-knowledge*, which ensures that a cheating *verifier* learns nothing beyond the validity of the proven statement. We use a ZKP system with the algorithms

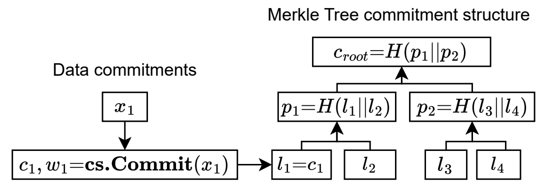

Fig. 2. Binary Merkle Tree commitment structure on a set of *data items* x_i, with $i \in \{0, \ldots, N\}$. The depicted Merkle Tree has a depth D=2, leafs $l_1, \ldots, l_{2^D}$, parents $p_1, p_{2*2^{D-1}-2}$, a root c_{root}, and depends on the hash function H. The root c_{root} represents the commitment string and the witness w consists of the internal witnesses w_i, with $i \in \{0, \ldots, N\}$ and a Merkle path $f_{path}(x_i)$ that depends on the committed *data items*. In this figure, the witness comprises the set of tuples w=$[(w_1, [l_2^2, p_2^2] = f_{path}(x_1))]$, where l_2^2 indicates that l_2 is the second concatenation when computing p_1.

- **zk.Setup**$(1^\lambda,\ ccs_{\mathcal{C}}) \rightarrow (pk,\ vk)$ takes as input a security parameter and a compiled constraint system expressing a circuit $\mathcal{C}$, and outputs the *prover* and *verifier* keys pk, vk.
- **zk.Prove**$(ccs_{\mathcal{C}},\ w,\ pk) \rightarrow \pi$ takes as input the compiled constraint system, a private witness, and the prover key pk and outputs a proof π.
- **zk.Verify**$(w_{pub},\ vk,\ \pi) \rightarrow \{0, 1\}$ takes as input a public witness w_{pub}, the verifier key vk, and the proof π and outputs a 1 if π combined with vk successfully verify against w_{pub}. Otherwise a 0 is returned.

C. Commitment Schemes

We define cryptographic commitment schemes with the following tuple of algorithms, where

- **cs.Commit**$(x) \rightarrow (c,\ w)$ takes as input the data x, generates a witness (e.g. randomness), and outputs a commitment string c and the witness w.
- **cs.Open**$(w,\ x,\ c) \rightarrow \{0, 1\}$ uses the witness to verify if the committed data matches the commitment string. In case of a match, the algorithm outputs 1, and 0 otherwise.

Commitment schemes are *hiding* if the commitment string c does not leak any information of x to an adversary with access to c. Commitment schemes are *binding* if there exists an unequivocal mapping between x, w, and c, such that an adversary cannot find a second valid opening yielding 1=**cs.Open**$(w',\ x',\ c)$, with $x' \neq x$, $w' \neq w$. A commitment opening maintains input privacy if a ZKP circuit $\mathcal{C}$ computes **cs.Open** while taking the witness w as a private input. We rely on different algorithms to construct commitments (e.g. via hash functions or a Merkle Tree (MT) [8], [9]). For example, computing an MT inclusion proof against a commitment c_{root} requires the ZKP circuit to derive c_{root}' based on the secrets x_1 and w, and check if c_{root}'=c_{root} (cf. Figure 2).

D. Secure Hash Functions

A secure hash function implements an algorithm, where

- **h.Hash**$(m) \rightarrow (h)$ takes as input a message string and outputs a constant size hash string h.

and guarantees three properties: *Preimage-resistance* ensures that given h, and attacker cannot find m if h = **h.Hash**(m).

Second preimage-resistance ensures that given m_1 an attacker cannot find m_2 such that **h.Hash**(m_1)=**h.Hash**(m_2) holds, with $m_1 \neq m_2$. *Collision-resistance* ensures that finding $m_1 \neq m_2$ with **h.Hash**(m_1)=**h.Hash**(m_2) is infeasible.

E. Blockchains & Smart Contracts

Public blockchains are open computer networks anyone can join, which run a *consensus* protocol to agree upon a common and correct *state* s_t at time t. The *state* maintains two types of accounts. The externally owned account (EOA) is controlled by a PKC key pair and is updated if a *user* owning the key pair sends signed transactions to the blockchain. The second type of an account is called *smart contract*, which is an executable program at an unique address that can be invoked by transactions. The execution of *smart contracts* is measured in *gas* and must be paid by a medium called cryptocurrency. If a new state update is proposed via a new block of transactions, then blockchain nodes apply new transactions and compare local *state* updates with the digests of the new block. If the verification succeeds, blockchain nodes locally apply the *state* update, and, with that, reach a new global state agreement.

Blockchains achieve mulitple properties, where *safety* provides *state* integrity according to past *states*. *Liveness* ensures that every transaction is eventually included in the *state*. *consistency* guarantees that every node eventually has the same view of the *state*. The redundant nodes of blockchains provide *fault-tolerance* and data immutability guarantees and transactions achieve *non-repudiation*, where the signature of every transaction unambiguously identifies a *user*.

III. System Model

A. Notations

Key pairs are the *public* and *private keys* of a PKC system.
Addresses are derived from a *user*'s *public key* and exist as 42-character hexadecimal strings appended with '0x'.
Wallets W generate and maintain *key pairs* and, with that, control the *address* W_{addr} corresponding to the *key pairs*.
Data items are key-value pairs, where the key string is a descriptor of the value instance that expresses the data.
Statements ϕ="key-op-$comp$" are strings that express relations between a value $comp$ and a data item with key=key. *Statements* use at least one key, one operator op (e.g. $>$,$<$,$\neq$,$\stackrel{?}{=}$,$\in$, etc.) and one comparison value $comp$ (e.g. threshold).
Claims exist as *public claims* claim^{pub}=$\{d, \phi$,t$\}$ and as *private claims* claim^{priv}=$\{d, \phi, L, e_{id}$,t$\}$. *Public claims* include the *data item* d, a *statement* ϕ, and a timestamp t. If the *data item* of claim^{pub} is stored externally, then d is set to a location identifier d=L. *Private claims* include a *data item* d, a statement ϕ, a location identifier L, an event identifier e_{id}, and a timestamp t. In claim^{priv}, the value instance of d is a commitment string c (e.g. d["age"] : c) and the location identifier points to a circuit storage address as L=L_{p_Π}.
Circuits are tuples p_Π=$\{\mathcal{C}, \phi, ccs_\mathcal{C}, w_{pub}, pk, vk, L_\mathcal{C}\}$, where the compiled constraint system $ccs_\mathcal{C}$ expresses a provable representation of a circuit $\mathcal{C}$ that implements the assertions expressed by the *statement* ϕ. To assert *statements*, the circuit $\mathcal{C}$ evaluates private inputs w to a representation which can be compared against public inputs w_{pub}. The prover and verifier keys pk,vk are created by running the setup algorithm **zk.Setup** of a proof system Π. If the verification call of the circuit $\mathcal{C}$ is deployed as a *smart contract*, then the locator $L_\mathcal{C}$ is set to the *address* of the *circuit contract*. Otherwise, $L_\mathcal{C}$=$null$.
Transactions are tuples tx=$\{\sigma, d_{pl}, t_{addr}, g_{used}\}$ with a signature σ from the transaction sender, a data payload d_{pl}, a *gas* value g_{used} and a destination *address* t_{addr}. *Transactions* are used to invoke and pay for *smart contract* calls at an address t_{addr} and provide non-repudiation of the transaction sender.
Circuit contracts $C^\mathcal{C}$ verify ZKPs on-chain and emit events e_{id} according to the outcome of a ZKP verification. *Circuit contracts* expose the *sample* and *verify* methods. If a *transaction* calls the *sample* method, then $C^\mathcal{C}$ associates a PoS randomness as a nonce with the wallet address of the *user* in a map $m[W_{addr}]nonce$. The randomness is used during the *verify* method which verifies a ZKP.

B. System Roles

Users hold *wallets*, deploy *identity contracts*, and register the *address* of the *identity contract* at the *registry contract* after passing an authenticity verification at the *identity service*. *Users* individually manage *claims* and *attestations*, and authenticate themselves at *third-party services* by linking or presenting data. *Users* count as *issuers* in the context of signing and sharing credentials towards other *users*.
Identity services deploy and maintain *registry* and *circuit contracts* and connect *users* to the *Portal* identity system. We envision non-profit organizations to take the role of the *identity service* and assume that *identity services* have the expertise to create secure ZKP circuits which evaluate *claims* of *users*.
Third-party services (e.g. web services) authenticate *users* based on the *Portal* identity system and trust *issuers*.
Blockchain networks provide decentralized and verifiable computation and storage through *smart contracts* and manage *registry*, *identity*, and *circuit contracts*.
Storage networks provide decentralized, fault-tolerant, and high-availability storage of data at locations L and are used to store larger data objects such as circuit parameters p_Π.

C. Threat Model

We assume that transactions sent to blockchain nodes are secured via Transport Layer Security (TLS) such that the TLS properties of message confidentiality, integrity, and authenticity hold. We assume that honest *users* are able to resolve the correct state s_t of the blockchain at time t. Additionally, we assume that collision resistant hash functions are used in the blockchain PoS protocol to determine the block randomness [10]. We assume active, adaptive, and probabilistic polynomial time (PPT) adversaries that are able to perform machine-in-the-middle (MITM) attacks and intercept communication traffic. However, adversaries are not able to block traffic indefinitely and cannot modify intercepted traffic. Adversaries can access transaction payloads by observing blockchain logs and replay transactions tx or ZKPs of a *user*.

$$\textbf{assertClaim}(d,\ \text{path}^{\text{MT}},\ \text{W}_{addr},\ \text{n};\ \textbf{root}^{\text{MT}},\ \textbf{W}_{addr},\ \textbf{n},\ \phi):$$

1. assert: $\mathbf{n} \stackrel{?}{=} n$; $\quad W_{addr} \stackrel{?}{=} \mathbf{W}_{addr}$; $\quad 1 \stackrel{?}{=} f_\phi(d)$
2. return: $1 \stackrel{?}{=}$ **cs.Open**(path$^{\text{MT}}$, d, **root**$^{\text{MT}}$)

Fig. 3. ZKP circuit to verify a *data item* d of a private claim against a MT commitment root$^{\text{MT}}$. The MT has a depth of 5 and a path$^{\text{MT}}$ as the private witness. The circuit has 9.29K constraints and evaluates d against ϕ="d[age]->-18" using the function f_ϕ. The semicolon **;** separates private inputs (left of **;**) from boldly formatted public inputs (right of **;**).

IV. Constructing Time-bound and Replay-resistant ZKPs

A. ZKP Verification at Smart Contracts

As the initial setup, we assume access to a circuit tuple p_Π, which has been instantiated by a trusted party p_0. The party p_0 derives the solidity verification code of $\mathcal{C}_1 \in p_\Pi$ for the creation and deployment of a circuit contract $\mathcal{C}^{\mathcal{C}_1}$ (cf. steps 1.4, 1.5 of Figure 4). Π uses a zero-knowledge proof system which compiles the circuit $\mathcal{C}_1$. The circuit $\mathcal{C}_1$ performs an address and nonce check, asserts a private *data item* against a statement ϕ, and checks if the *data item* computes to a public commitment string (cf. **assertClaim** logic of Figure 3). Now, a *user* as party p_1 is able to compile transactions with a payload that contains the bytes of a ZKP π, and call the deployed contract $\mathcal{C}^{\mathcal{C}_1}$ for an on-chain verification of π.

B. Binding ZKP Computations to the PoS Randomness

In the following we define a transaction sequence where a user p_1 compiles the transaction tx$_1$ to call the *sample* method of the contract $\mathcal{C}^{\mathcal{C}_1}$. Upon receiving tx$_1$, $\mathcal{C}^{\mathcal{C}_1}$ associates the latest PoS randomness r with the *user*'s wallet address by depositing both parameters into the map $m[W_{addr}]nonce$. Initially the randomness is concatenated with a state string to represent the nonce as $nonce$=$s_t.prevrandao||$"-0". After $\mathcal{C}^{\mathcal{C}_1}$ samples the nonce, *users* fetch and use the deposited nonce to compute a ZKP π using the circuit $\mathcal{C}_1$. To prevent replay attacks and ensure time-bound proofs (cf. Section IV-C), the ZKP circuit $\mathcal{C}_1$ takes in and compares both the *user*'s wallet address and the nonce as private inputs and public inputs. Notice that binding values (e.g. the nonce) to a ZKP computation via public inputs is secure [6]. In a transaction tx$_2$, p_1 calls the *verify* method of $\mathcal{C}^{\mathcal{C}_1}$, which upon a successful verification of π, sets the nonce to $m[W_{addr}]s_t.prevrandao||$"-1" and emits an event with an identifier e_{id}. If party p_1 presents e_{id} towards any *third-party service*, then the *third-party service* can use e_{id} to resolve and verify a successful on-chain ZKP verification via smart contract logs (cf. steps 2.1-2.8 in Figure 4).

C. Security Analysis

Theorem 1. *If a party p_1 with access to*

- *a smart contract $\mathcal{C}^{\mathcal{C}_1}$*
- *a secure proof system Π_π*
- *a secure signature scheme Π_σ*
- *a secure hash function Π_H*

performs the sequence of computations

1) *p_1 compiles and signs a transaction tx$_1$ with Π_σ.Sign*
2) *p_1 calls $\mathcal{C}^{\mathcal{C}_1}$.sample with tx$_1$ such that $\mathcal{C}^{\mathcal{C}_1}$ generates the prevrandao randomness r using Π_H.Hash and stores $m[W^{p_1}_{addr}]r||$"-0" at timestamp t_1*
3) *p_1 fetches r from $m[W^{p_1}_{addr}]$*
4) *p_1 computes π=Π_π.Prove($ccs_{\mathcal{C}_1}$,w,pk)*
5) *p_1 compiles and signs a transaction tx$_2$ with Π_σ.Sign, where $\pi \in$ tx$_2.d_{pl}$*
6) *p_1 calls $\mathcal{C}^{\mathcal{C}_1}$.verify with tx$_2$ and $\mathcal{C}^{\mathcal{C}_1}$ sets $m[W^{p_1}_{addr}]r||$"-1" at timestamp $t_2 > t_1$*

under the assumptions that

- *$\mathcal{C}^{\mathcal{C}_1}$ runs on a blockchain which guarantees liveness, consistency, safety, and fault-tolerance*

we say that the proof π is resistant against replay attacks performed by a malicious PPT adversary such that $\pi \in$ tx$_2$' is never accepted by $\mathcal{C}^{\mathcal{C}_1}$. Further, we say that the computing π is bound by the time t_1 and cannot be accepted after t_2.

Proof 1. With tx$_1$', the adversary $\mathcal{A}$ is capable of registering the same nonce as p_1 at $\mathcal{C}^{\mathcal{C}_1}$ at time t_1. However, $\mathcal{C}^{\mathcal{C}_1}$ maps the nonce of $\mathcal{A}$ at the address $m[W^{\mathcal{A}}_{addr}]$. After t_2, $\mathcal{A}$ uses the blockchain transaction logs to access tx$_2$ and, with that, π. If $\mathcal{A}$ replays π in a transaction tx$_2$' and calls $\mathcal{C}^{\mathcal{C}_1}$.verify, then the verification of circuit $\mathcal{C}_1$ fails because $\mathcal{C}^{\mathcal{C}_1}$ asserts the address $W^{\mathcal{A}}_{addr}$ as public input against the private input $W^{p_1}_{addr}$ which was used to compute π.

Due to the fact that the probability of a colliding r is negligible, $\mathcal{A}$ cannot register the same nonce twice. Thus, $\mathcal{A}$ cannot replay a previously accepted proof π which complies with the same nonce in the future at time $t_3 > t_2$. Even if a collision is found, $\mathcal{C}^{\mathcal{C}_1}$ prevents overwriting an existing entry with an incremented nonce value.

V. *Portal* Identity System

A. System Goals

Sybil resistance prevents an adversary to register an arbitrary amount of pseudonymous identities.
Decentralized resolutions allows a *third-party service* to resolve *user* data from a decentralized network.
On-chain & off-chain verification of private data allows *users* to (i) present data to a *third-party service*, where the data has been verified at smart contracts or (ii) interactively convince a *third-party service* of a data verification.
Decentralization guarantees that the storage and computation of user data remain publicly verifiable, trustless, and available.
Cost-efficiency optimizes operation costs for *third-party* and *identity services* and enables scalability of *Portal* with cheap maintenance costs for the *identity service*.

B. Architecture

The *Portal* architecture introduces two new contracts, where

- **Registry contracts** C^{reg} maintain a map $m[W_{addr}]C^{id}_{addr}$ linking registered *wallet addresses* and *addresses* of *identity contracts*. C^{reg} exposes a *register* method which requires the transaction payload to include an *identity service* signature on a new *identity contract address*.

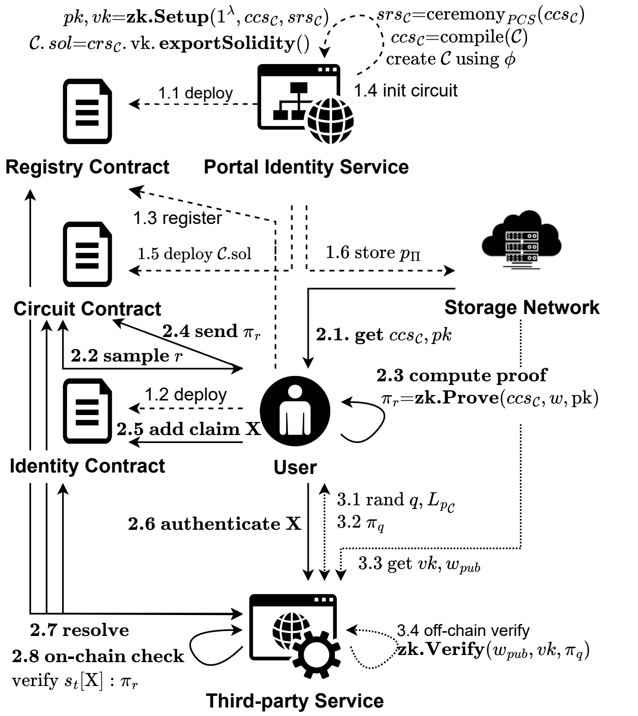

Fig. 4. *Portal* architecture in the context of managing a *private claim*. The system deployment, *user* registration, and the circuit pre-processing is indicated with dashed arrows (1.1-1.6). The on-chain verification of *private claims* at time t_1, and *private claim* presentation towards a *third-party service* is depicted with solid lines (2.1-2.8). The live verification at time $t > t_1$ of a *private claim* is indicated with dotted lines (3.1-3.4).

Further, for the identification of circuits, C^{reg} maintains a map $m[\text{name}^{\mathcal{C}}]L_{p_\Pi}$ which associates location identifiers of circuit parameters L_{p_Π} with circuit names name$^{\mathcal{C}}$.

- **Identity contracts** C^{id} maintain *claims*, *attestations*, and *revocations* with the maps $m[\text{name}^{\text{claim}}]$claim, $m[\text{name}^{\text{att}}]a$, and $m[a_{id}]$rev, where a_{id} is an attestation identifier. The unique strings name$^{\text{claim}}$, name$^{\text{att}}$ represent claim and attestation names.

The registration of a new *user* in the *Portal* system depends on two transactions. The first transaction deploys the *identity contract* C^{id} of the *user*. In the same way as the *registry contract*, the constructor of the *identity contract* sets the deploying party as the owner of the contract. Only the owner of C^{id} is able to call methods which modify the state of C^{id}. The compilation of the second transaction requires the *user* to obtain a signature $\sigma_{C^{id}_{addr}}$ from the *identity service* on the *identity contract address*. Before signing any C^{id}_{addr}, the *identity service* verifies and deduplicates *users*, such that *sybil resistance* holds in the *Portal* system. *Users* use the second transaction to invoke the *register* method at the *registry contract* C^{reg}, which checks the signature validity of $\sigma_{C^{id}_{addr}}$ before including the *user*'s wallet address and C^{id}_{addr} into the map $m[W_{addr}]C^{id}_{addr}$. If the *user* shares the wallet address W_{addr} with any *third-party service*, then the *third-party service* is able to resolve C^{id}_{addr} via the map $m[W_{addr}]C^{id}_{addr}$ such that *decentralized resolution* holds.

TABLE I
Portal DEPLOYMENT AND TRANSACTION COSTS.

Tx	Type	Cost (eth/$)	Time (ms)	Size (kB)
C^{reg}	deploy	4.15e-3/8.65	18	bc:6.5,tx:6.6
C^{id}	deploy	6.5e-3/13.56	10	bc:10.2,tx:10.7
$C^{\mathcal{C}_1}$	deploy	4.96e-3/10.29	**385**	bc:7.4,tx:12.4
register	call C^{reg}	7.4e-5/0.16	51	tx: 0.3
claim^{pub}	call C^{id}	6.4e-05/0.13	3	tx: 0.48
sample	call $C^{\mathcal{C}_1}$	6.6e-05/0.14	6	tx: 0.1
verify_π	call $C^{\mathcal{C}_1}$	8.4e-4/**1.76**	**252**	tx: 1.20
claim^{priv}	call C^{id}	3.9e-4/0.82	21	tx: 0.68

Once users are registered, *users* can create private claims by following the transaction sequence which prevents replay attacks (cf. Section IV). Further, *users* can decide to partake in a live verification of private data, where a proof system is deployed between the *user* and the *third-party service* (cf. steps 3.1-3.4 in Figure 4). The live verification ensures that private claims are not validated by smart contracts at timestamps in the past. The data verification modes of *Portal* ensure support for on-chain and off-chain verification of private data.

Third-party services resolve and verify *user* data through a *Portal* plugin, which performs a signature challenge before every data verification. In the same way as the SIWE sign-in challenge [2], our signature challenge demands the *user* to compute a signature on a randomly sampled nonce using the wallet *key pair*. In this case, the plugin samples the nonce.

VI. Evaluation

A. Implementation

The *Portal* proof of concept was conducted locally using the *Ganache*[2] test network (*v7.8.0*) as the public blockchain. We rely on the solidity compiler *solc v0.8.20* as the PoS block randomness *prevrandao* is available in all versions above *v0.8.18*. We develop a *Portal* Golang System Development Kit (SDK) to deploy and maintain *Portal* at every party and use the official Ethereum repository *go-ethereum*[3] including *abigen v1.10.16* to interact with smart contracts. We convert transaction costs into dollars based on the rate 2084.42$ per 1 *eth* (Nov. 2023) and select the gas price gas_{price}=28gwei according to the gas price of the Ethereum network[4]. We select the Golang *gnark* (*v0.9.1*) repository [11] as the ZKP system and configured (i) the *plonk* backend with a universal setup to verify ZKPs on-chain. To prove and store private *claims* efficiently, we benchmark the ZKP circuit $\mathcal{C}_1$ (cf. Figure 3), which evaluates *data items* of claims^{priv} as private input against a MT commitment as the public input. We use the MiMC hash function [12] to compress the MT data. We open-source the *Portal* code with the *smart contracts* and simulation scenarios in the repository[5].

[2]https://github.com/trufflesuite/ganache
[3]https://github.com/ethereum/go-ethereum
[4]https://etherscan.io/gastracker#chart_gasprice
[5]https://github.com/anonsubsub/portal

B. Costs Analysis

The evaluation uses a MacBook Pro with the Apple M1 Pro chip and 32 GB of Random Access Memory (RAM). The benchmarks average ten executions of the same experiment.

Table II shows the *Portal* cost analysis, where transaction costs are computed according to $\text{tx}_{\text{cost}}{=}gas_{\text{used}} \cdot gas_{\text{price}}$. We explain the execution times in the range of milliseconds with the local deployment of *Portal*. By deploying *Portal* on the Sepolia[6] testnet, we measured transaction resolution times taking around 150ms and transaction calls taking between 1.3s ($C^{\mathcal{C}_1}$ deploy) and 9.4s (sample+verify_π+claim^{priv}). We explain higher execution times of the transactions that deploy $\mathcal{C}_1$ and verify a proof of $\mathcal{C}_1$ with the corresponding higher transaction sizes. Compared to other contracts, which initialize empty maps, the byte code of $C^{\mathcal{C}_1}$ stores large cryptographic parameters which increase the transaction size of $C^{\mathcal{C}_1}$. Except transactions of the type deployment and the transaction to verify a ZKP on-chain, the cost per transaction remains below 1\$. Thus, as claims are verified once and shown multiple times, we consider *Portal* as cost-efficient.

VII. Discussion

A. Related Works

The work DecentID [13] introduces a *smart contract* identity system which resolves user data via four different contract types. In contrast to DecentID, *Portal* supports enhanced data privacy through on-chain and off-chain ZKP computations.

The work *zk-creds* [5] proposes the first construction of anonymous zero-knowledge Succinct Non-Interactive Arguments of Knowledge (zkSNARK) credentials. With a verifier-chosen nonce, *zk-creds* prevents credential replays towards the verifier in the off-chain context. By contrast, *Portal* works on chain and applies the PoS randomness to prove zkSNARK claims in unique verification sessions.

The work Zebra [7] introduces a zkSNARK credential scheme with an on-chain ZKP verification at an access control contract. Before a *user* authenticates at an application smart contract with a wallet address W_{addr}, the *user* posts a ZKP to the access control contract to provide access privileges to W_{addr}. Instead of relying on additional smart contracts, *Portal* improves the cost-efficiency by solving ZKP replay attacks via a cheap transaction sequence.

B. Limitations & Future Work

Portal runs on the native blockchain network called layer 1. To optimize transaction costs, we envision deploying *Portal* via scalable second layer networks (e.g. zk-rollups [14]). Concerning decentralizing the *identity service*, we either (i) register *users* based on a multi-party signature issued by multiple *identity services*, or (ii) maintain a list of public keys in the *registry contract*, such that public keys authorize *identity services*. To align *Portal* with standardization efforts, we see the *OpenID Connect*, W3C Decentralized Identity (DID) and Verifiable Credential (VC) compliance as appealing goals.

[6] https://www.alchemy.com/overviews/sepolia-testnet

TABLE II
Comparison with Related Works.

Paper	Dec. Resolution	On/Off-chain Verify	No Extra Contract
DecID	✓	✗ / ✗	✗
zk-creds	✗	✗ / ✓	✓
Zebra	✓	✓ / ✗	✗
Portal	✓	✓ / ✓	✓

VIII. Conclusion

In this work, we construct a replay-resistant ZKP verification at smart contracts. On top our our construction, we present *Portal*, a modern identity system with enhanced privacy and control. *Portal* provides *third-party services* with a plugin to resolve and verify private or public data claims of *users*. As such, *Portal* serves as the first SSO alternative with conventional usability that gives *users* a choice to pick enhanced control and privacy at small costs.

References

[1] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich, "Sok: single sign-on security—an evaluation of openid connect," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 251–266.

[2] W. Chang, G. Rocco, B. Millegan, N. Johnson, and O. Terbu, "Erc-4361: Sign-in with ethereum [draft]," https://eips.ethereum.org/EIPS/eip-4361, October 2021, [Online serial] Accessed: 2023-11-15.

[3] P. Labs, "Polygonid: A blockchain-native identity system," https://polygonid.com/, accessed: 2023-03-04.

[4] J. Ernstberger, J. Lauinger, F. Elsheimy, L. Zhou, S. Steinhorst, R. Canetti, A. Miller, A. Gervais, and D. Song, "Sok: Data sovereignty," *Cryptology ePrint Archive*, 2023.

[5] M. Rosenberg, J. White, C. Garman, and I. Miers, "zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 790–808.

[6] K. Baghery, M. Kohlweiss, J. Siim, and M. Volkhov, "Another look at extraction and randomization of groth's zk-snark," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*. Springer, 2021, pp. 457–475.

[7] D. Rathee, G. V. Policharla, T. Xie, R. Cottone, and D. Song, "Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi," *Cryptology ePrint Archive*, 2022.

[8] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse merkle trees: Caching strategies and secure (non-) membership proofs," in *Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21*. Springer, 2016, pp. 199–215.

[9] H. Ritzdorf, K. Wüst, A. Gervais, G. Felley, and S. Capkun, "Tls-n: Non-repudiation over tls enabling-ubiquitous content signing for disintermediation," *Cryptology ePrint Archive*, 2017.

[10] B. Edgington, "Upgrading ethereum: 2.9.2 randomness." https://eth2book.info/capella/part2/building_blocks/randomness/, 2023.

[11] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie, "Consensys/gnark: v0.9.0," Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.5819104

[12] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, and T. Tiessen, "Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 191–219.

[13] S. Friebe, I. Sobik, and M. Zitterbart, "Decentid: Decentralized and privacy-preserving identity storage system using smart contracts," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications*. IEEE, 2018, pp. 37–42.

[14] S. Motepalli, L. Freitas, and B. Livshits, "Sok: Decentralized sequencers for rollups," *arXiv preprint arXiv:2310.03616*, 2023.