Decentralized Identity Authentication With Auditability and Privacy

Bithin Alangot, Pawel Szalachowski, Tien Tuan Anh Dinh, Souhail Meftah, Jeff Ivanos Gana, Khin Mi Mi Aung, Zengpeng Li
Singapore University of Technology and Design, Singapore
Institute of Infocomm Research (I2R), A*Star

Abstract

Decentralized Identity (DID) systems aim to give users full control over their identities by using blockchains as the identity providers. However, when user credentials are compromised, it is impossible in existing DID systems for the users to detect credential misuse. In this paper, we propose new DID authentication protocols with two properties: *auditability* and *privacy*. The former enables the detection of malicious authentication events, while the latter prevents an adversary from linking an authentication event to the corresponding user and service provider. We present two protocols that achieve auditability with varying privacy and performance guarantees. The first protocol has high performance, but it reveals information about the user. The second protocol achieves full privacy, but it incurs higher performance overhead. We present a formal security analysis of our privacy-preserving protocol using Tamarin prover. We implement and evaluate their performance with a permissioned blockchain deployed over the Amazon AWS and local cloud infrastructure. The results demonstrate that the first protocol is able to support realistic authentication workloads while the second is nearly-practical.

1 Introduction

Decentralized Identity (DID) [2] is an identity system in which users have full control over their identities. DID systems (or DIDs) today leverage blockchains to manage the mapping between identifiers and other data such as cryptographic credentials. Examples include Evernym [4], Nuggets [6], Hyperledger Indy [5], Blockstack [1] and CanDID [32]. The security of these systems depend on that of the underlying blockchains. In particular, identities are tamper evident and highly available, without relying on centralized trusted parties. However, existing DIDs cannot detect misuse of compromised identities. More specifically, it is impossible to detect that a user's credential has been stolen and used by the adversary to authenticate to a service provider. This is a limitation compared to centralized identity systems that log all authentication events. For example, Google's last login feature [8] provides users with the details of all the login events by their accounts.

Our goal is to design DID authentication protocols that enable users to detect credential misuse. We define two security properties for such protocols: *auditability* which allows users to detect unauthorized authentication events that use their credentials, and *privacy* which prevents the adversary from linking an authentication to the corresponding user and service provider. The challenge here is to design protocols that have both properties and reasonable performance. One naive solution is to store the full content of all authentication events to the blockchain. This achieves auditability because the user can see the complete history of his credential usage, but reveals both the user and the service provider of every event, thus achieving no privacy.

We propose two protocols that achieve auditability with varying privacy and performance guarantees. The first protocol has high performance but limited privacy. It works as follows. For each authentication, the user increments a counter, signs it with the key associated with his identity. This signed counter is sent to the service provider, which tries to log it to the blockchain. If successful, the user is authenticated. The blockchain ensures that it only accepts consecutive counter values from the correct user. By keeping track of the latest counter value, the user can detect malicious authentication by checking if a future counter value was logged on the blockchain. However, because counter values are signed, each authentication event reveals who the user is.

The second protocol achieves full privacy but incurs high performance overhead. The key challenge in designing this protocol is how to hide user information from the authentication event. To illustrate this, we describe a protocol using Verifiable Random Functions (VRFs), and shows that it only works under strong security assumption. We address this challenge by using the Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zk-SNARK) primitive [23]. More specifically, the client generates a token tkn based on a counter value cnt, and a zero-knowledge proof with which the blockchain can verify that some token tkn' derived from the value $cnt - 1$ is already in the blockchain, but without revealing this token or the counter value. Our evaluation shows 3 order of magnitude lower performance compared to the first protocol. This overhead is due to the cost of verification which is done by the blockchain.

In summary, we make the following contributions: 1) We define two security properties for detecting credential misuse in DID authentication, namely auditability and privacy, 2) We propose two protocols that achieve auditability with varying privacy and performance guarantees. The first has high performance but limited privacy, the second has full privacy but low performance, 3) We provide a formal security analysis of our privacy-preserving protocol using Tamarin Prover and 4) We implement and evaluate the two protocols using Hyperledger Fabric v0.6[1]. The results confirm that the overhead for privacy is substantial: the protocol can only handle 1 authentication event every 3s. However, by lowering the privacy guarantee, our protocol can perform over 380 events per second, which means it can handle the average workload of today's centralized authentication systems.

[1] Hyperledger Fabric v0.6 has higher performance and security guarantees than v1.4[20]

The rest of the paper is organized as follows. Section 2 provides relevant background on blockchain, DIDs, and cryptographic primitives. Section 3 describes the system model, threat model, and security properties. Section 4 and Section 5 present the two protocols in detail. Section 6 gives a formal security analysis using Tamarin Prover. Section 7 discusses the implementation and performance of the two protocols. Section 8 reviews related works, before Section 9 concludes.

2 Background

2.1 Blockchain

A blockchain system consists of distrusting nodes that together implement a replicated, tamper-evident, append-only log data structure called a ledger. The ledger comprises of blocks, each block contains transactions that modify some global states, and the blocks are linked through cryptographic hash pointers. The nodes keep the global chain of blocks consistent through a consensus protocol. Most blockchains support smart contracts which are user-defined functions executed by all nodes in the network.

Existing blockchains can be broadly classified into *permissionless* and *permissioned* based on their consensus protocol [20]. Examples of the former include Bitcoin [36] and Ethereum [43], in which any nodes can join and leave. Examples of the latter include Hyperledger Fabric [12] and Quorum [15], in which membership is strictly enforced. Permissionless blockchains use variants of Proof-of-Work (PoW) consensus protocol, whereas permissioned ones use variants of Byzantine Fault Tolerant (BFT) protocols [25].

A blockchain executes and stores user transactions in the ledger. The user can ask for proof that her transaction is included in the ledger without downloading and re-executing the entire ledger. In a permissioned blockchain, this proof consists of a statement (that the transaction is included) signed by $f + 1$ nodes (f is the maximum number of Byzantine nodes that can be tolerated). In a permissionless blockchain, all transactions are represented by a Merkle root hash included in the block, thus the proof consists of the valid Merkle path to the root hash of the latest block.

2.2 Decentralized Identity Authentication

Decentralized Identity (DID) [2] is an identity system in which the identity information is owned by the entity which created it. This is as opposed to the centralized identity management system, like Microsoft's Active Directory, in which a third party stores and modifies identities. Most DIDs today use blockchains to store a mapping between a unique identifier (representing an entity), called *did*, and other meta data, such as cryptographic keys.

An entity registers a new *did* with the corresponding metadata, *meta*, by executing a smart contract on a blockchain that stores (*did, meta*) on the blockchain. To retrieve the corresponding metadata of a given *did*, i.e., to resolve the *did*, the user provides the string (<*did*>:<*method-specific identifier*>:<*method*>) as input to the DID (which points to a resolver). The system then triggers the specified method on the specified blockchain that returns the (*did, meta*) tuple.

The benefits of DIDs are derived directly from those of blockchains: the (*did, meta*) tuples are tamper-evident and highly available. In addition, DIDs can guarantee that only the original owner of the registered *did* can update the corresponding metadata, thereby giving the user full control of her identity.

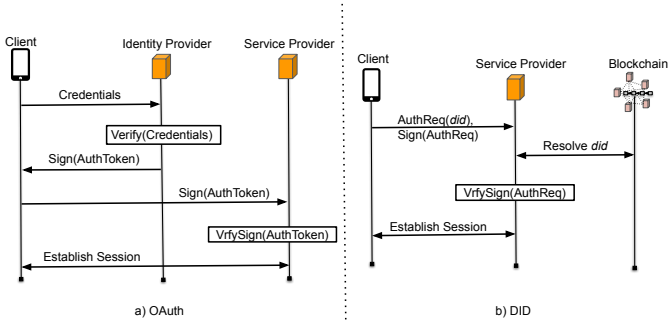

Figure 1: Comparison between OAuth and DID authentication.

DID based authentication The DID can replace centralized PKI systems in public-key based authentication [3]. Figure 1(a) shows how OAuth works with centralized identity management systems. In particular, to authenticate to a service provider, the user first present her credentials to her identity provider, which after verification issues a signed token. The user submits this token to the service provider, which verifies it using PKI before establishing a session with the user. Using DID, as shown in Figure 1(b), the user sends a signed, fresh authentication request using her private key (whose public key is stored in the metadata). The service provider resolves user's *did* to verify the request, then establishes a session with the user.

2.3 Cryptographic Primitives

Hash Function. A cryptographic hash function is defined as $H : \mathcal{X} \rightarrow \mathcal{Y}$ where $\mathcal{X}$ is called the domain, and $\mathcal{Y}$ is called the range. The function is collision-resistant, meaning that it is hard to find two different inputs producing the same output. Additionally, it is preimage resistant, i.e., given an element in the range, it should be computationally infeasible to find an input that maps to that element.

Unforgeable Signatures. A signature scheme is defined as the three following algorithms. (a) $KeyGen(1^\lambda)$ is a key generation algorithm that takes a security parameter λ as input and outputs the (signing) secret key sk and the verification key vk, respectively. (b) $Sig_{sk}(msg)$ is a signing algorithm that takes the secret key sk and a message msg as inputs, and outputs a signature σ. (c) $vrfySig_{vk}(msg, \sigma)$ is a signature verification algorithm that takes a verification key vk, a message msg and a signature σ as input, and outputs True if the signature matches the key and the message, or False otherwise. We assume that the digital scheme used is existentially unforgeable under chosen message attacks, i.e., EUF-CMA secure.

Verifiable Random Function (VRF). A VRF [35] is the following tuple of algorithms. (a) $KeyGen(1^\lambda)$ is a key generation algorithm that takes a security parameter λ as input and outputs a keypair, i.e., the secret key and the verification key (sk and vk, respectively). (b) $VRF_{sk}(X)$ is an algorithm that takes the secret key sk and X as inputs and outputs a pseudorandom hash h and a proof γ (the value h is unique for the inputs sk and X). (c) $vrfyVRF_{vk}(X, h, \gamma)$ is

a verification algorithm that takes verification key vk with values X, h, γ; and returns True if γ proves that h was created for X with the corresponding key sk, False otherwise. In practice, VRFs can be implemented with unique signatures.

zk-SNARK. zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) [22] is a non-interactive zero-knowledge proving systems allowing to prove and verify the integrity of an arbitrary computation (represented as an arithmetic circuit). zk-SNARK consists of the following algorithms. (a) $zkGen(C, \lambda)$ is a key generation algorithm that takes an arbitrary circuit C and secret parameter λ as inputs, and generates a proving key z_{pk} and a verification key z_{vk}. Generation of these keys constitutes a setup phase and is trusted. (b) $zkProve_{z_{pk}}(in, w)$ a function that takes public inputs in and private witness w as inputs, and outputs a succinct integrity proof of computation π. (c) $zkVrfy_{z_{vk}}(\pi, in)$ a function that takes as input a verification key z_{vk}, a proof π, and public inputs in, and returns True if the proof is valid, False otherwise. Although, we describe one of our protocols using zk-SNARKs, it can be combined with any proving system with the same capabilities. In particular, with systems that offer better performance or do not require trusted setup [21].

3 Overview

In this section, we first give an overview of our protocols. We then present the system model, the desired security and system properties, and the threat model.

3.1 Overview

With centralized identity providers, for example, Google or Facebook, users rely on the provider to manage their identity and log all the authentication events associated with it. The provider can ensure that one user cannot see another user's authentication log. There are two problems associated with this setting. First, the provider has to be trusted to log all authentication events, including malicious ones. Second, the provider can see all the logs, thus there is no user privacy against the provider.

Current DID based authentication systems [3] do not solve these two problems. Although they do not rely on trusted, centralized parties, they do not enforce the logging of all authentication events. As a result, it is impossible for users to detect misuse of their credentials. Furthermore, simply moving the logs from a centralized system to the blockchain violate user privacy, as data in the blockchain is publicly visible.

We address these two problems in the paper. We define two properties: *auditability*, which allows user to see the complete log of all authentication events associated with her credential, and *privacy* which prevents linking an authentication event to the corresponding user and service provider. We propose two DID authentication protocols that achieve auditability and varying privacy guarantee. The first protocol, i.e., the baseline, stores authentication tokens to the blockchain for each authentication event. It is simple and efficient but reveals links between authentication events and identities. The second protocol addresses the challenge in hiding user information in the authentication event, by using a combination of VRF and zk-SNARK primitives. We highlight why both are needed by showing an intermediate protocol that only uses VRF and demonstrating that it fails to achieve auditability within our model. The final protocol ensures privacy but at the cost of high performance overhead.

Table 1: Comparison of our protocols against existing DID authentication protocols.

	Auditability	Privacy	Performance
Existing protocols	✘	✘	✔
Baseline	✔	✘	✔
Privacy-preserving	✔	✔	✘

3.2 System and Threat Model

Our system consists of three types of entities: blockchain, service provider, and client.

- **Blockchain** is a blockchain system, such as Hyperledger Fabric or Ethereum, where users can register their *did*'s and retrieve a valid proof that a transaction is added to the blockchain.
- **Client** represents the user. It stores secret data such as private keys associated with the user's *did*, and runs the DID protocol on behalf of the user.
- **Service provider (or service)** provides resources that user wants to access. It runs a DID authentication protocol to authenticate user before giving her access to the resource.

We define our threat model with respect to each entity in the system as follows. The blockchain is trusted to protect integrity of the stored data. It is also highly available, i.e., not subject to denial of service attacks. The client is trusted to follow the protocol, but the adversary can steal its credentials (corresponding to the client's identity). The service provider can be compromised. In this case, it can collude with the adversary that has stolen an user's credential to prevent the user from detecting the credential misuse. When not compromised, the service provider is semi-honest, that is, it follows the authentication protocol correctly, but it wants to learn information other than what the authentication protocol reveals. In particular, it may want to learn which client authenticated to which (other) service providers, since knowledge of clients behavior may give a service provider an important competitive advantage.

3.3 Properties

Our goal is to design and implement DID authentication protocols that achieve a good trade-off among three desirable properties.

- **Auditability:** the user can detect unauthorized authentication events that uses her credential. More specifically, whenever an adversary uses the stolen credentials with an honest server, this fact can be later on detected by the compromised client. Our protocols support "passive" detection, when the client detects the misuse while authenticating to an honest server, or "active" when the client queries the blockchain to find the misuse.
- **Privacy:** given an authentication event logged in the blockchain, the adversary cannot link it a) to the identity being the subject of the authentication event, b) to the service provider involved, and c) to any previous authentication event triggered by the identity. We assume that in privacy attacks, the adversary can constantly observe the blockchain content, but we leave network-level de-anonymization attacks out of the scope of this submission (i.e., the adversary cannot eavesdrop network connections of clients and services).

- **Performance:** the protocol achieves sufficient throughput to handle realistic authentication workload.

Table 1 compares our protocols with existing DID authentication protocols, using the three properties. We note that the first two security properties are in addition to what provided by existing DID protocols.

4 Baseline Protocol

The baseline protocol is simple and achieves most, but not all, of the desired system properties with minimal overheads. It bases on a naive idea, where for each authentication event its corresponding counter is logged on the blockchain and verified that it is generated by the owner of a *did* and the counter is consistent with the previous one. We discuss the protocol in two phases, 1) registration, in which a user publishes his identity in the blockchain and 2) authentication, where we show a protocol for logging authentication counters.

4.1 Registration

In the registration phase, the client first generates its keypair $\langle vk, sk\rangle \leftarrow KeyGen(1^\lambda)$, then derives its identity $did \leftarrow H(vk)$, and initializes its authentication counter $cnt \leftarrow 0$. With these values, the client creates a registration request

$$regReq \leftarrow \langle vk, info, Sig_{sk}(vk \| info)\rangle$$

including $info$ with other human readable information (e.g., a name or a comment) and the signature computed over the content of the request. The request is sent to the blockchain where the *registerDID* smart contract method handles it. The method implements basic sanity checks including signature validation as well as insertion of the verification key along with other information into the *identity map* (*didMap*) which is a mapping between *did* (key) and a $\langle vk, info, cnt\rangle$ tuple (value), whereupon the registration *cnt* is set to 0.

Upon successful execution of the code, blockchain nodes create an inclusion proof β^{did}, stating that the identity was successfully registered and is part of the blockchain. The registration process is complete and the client is ready to use its *did* for authentication.

4.2 Extended Authentication

Our protocol extends the standard authentication process. In the authentication phase, the client first derives a new counter *cnt* by incrementing it by 1, then he generates an ephemeral keypair $\langle vk_e, sk_e\rangle \leftarrow KeyGen(1^\lambda)$ (a new ephemeral keypair is created per session and we describe its purpose later in this section), and creates the following requests

$$cntReq \leftarrow \langle did, cnt, Sig_{sk}(did \| cnt), Sig_{sk_e}(did \| cnt), vk_e\rangle$$

$$authReq \leftarrow \langle \beta^{did}, cntReq\rangle$$

with its *did*, proof of inclusion of the *did*, a new (consecutive) counter, and the signatures over *did* and *cnt*. The *authReq* request is sent to the service which upon receiving the request verifies the proof of inclusion and the signatures.

At this point, the client and service run any authentication protocol they wish to use. We do not mandate any specific scheme, since we foresee our protocol to be integrated within existing protocols like DIDAuth or a TLS, which may require additional message flows or may provide additional security properties (like service authentication). However, we require that during the authentication process, the service ensures that the client possesses the secret keys corresponding to its "long-term" *did* key (*vk*) and its ephemeral key vk_e. (We also envision, that the ephemeral keypair can be used for providing other session security features, like perfect forward secrecy.) A way of realizing such authentication, similar to some popular TLS authentication modes, would be to require the client a) to sign vk_e with *sk*, send the corresponding signature to the service, and, b) use sk_e to sign the authentication context which would contain all exchanged messages as well as a nonce provided by the service. This "standard" authentication phase, ensuring possession of *vk* and vk_e, is also part of other protocols we propose later in this paper (i.e., the protocols from Section 5.1 and Section 5.2).

Algorithm 1: Logging authentication events (the baseline protocol).

```
incrementCounter(did, cnt, σ)
1  assert did ∈ didMap
2  assert cnt == didMap[did].cnt + 1
3  vk ← didMap[did].vk
4  assert vrfySig_vk(did‖cnt, σ)
5  didMap[did].cnt ← cnt
```

After the successful authentication, the service extracts and forwards *cntReq* to the blockchain platform, where *incrementCounter* smart contract method handles the request as shown in Algorithm 1. The method verifies 1) the signature and ensures that it belongs to the right *did*, and 2) the current counter mapped to *did* has the value $cnt - 1$ which confirms no occurrence of credential misuse. Once validated, the counter value in the *identity map* is incremented. Upon successful execution of the code, blockchain nodes create an inclusion proof β^{cnt} (i.e., a statement signed by at least $f + 1$ nodes), stating that the request *cntReq* was successfully processed by the platform. The service validates the inclusion proof as shown in Algorithm 2 and forwards it to the client who verifies that the inclusion proof is correct and matches the intended *cntReq* message. After the process is finished, the service proceeds to establish a session with the client.

Signing *cntReq* by an ephemeral key prevents replay attacks, guarantying the client that his request was processed by the blockchain. Since an adversary can control the service and the client's long-term credentials (i.e., *sk*), without signing the request by an ephemeral key, the adversary could create exactly the same *cntReq* (even if it was accompanied by a random nonce), use it to authenticate with an honest service, and replay the blockchain inclusion proof to the client. With an honest service verifying the possession of ephemeral sk_e such a replay attack is impossible.

The baseline protocol satisfies the auditability property since for each authentication event using *did*, the identity map's entry corresponding to *did* is updated with the consecutive counter value. The client can easily audit the consistency between its local counter value with the counter value stored in the identity map to check for any credential misuse. Moreover, detection can be also passive, since for any authentication after a misuse happens, the blockchain would not process *cntReq* successfully what would be noticed by the client. We focus on the basic functionality of our protocol, however, the protocol can be extended. For instance, to identify against which service the malicious authentication was performed,

Algorithm 2: Object obj's inclusion proof verification.

```
vrfyInclusion (obj, β^obj)
    assert len(β^obj.KeySig) == f + 1
    for (k, σ) ∈ β^obj.KeySig do
        assert k ∈ TrustedNodes
        assert k is unique
        assert VrfySig_k(obj, σ)
    end
```

additional metadata encrypted with the client's public key could also be logged along with the counter.

4.3 Limitation

Counter requests do not reveal to which services the given identity authenticated. However, since the authentication counters are publicly logged and service providers submit identity information such as a *did* and signature along with the counter to the blockchain, anyone who can see the transactions on the blockchain can notice how often a user authenticates using his identity. A privacy cautious user may not want to share this information with the public which may be challenging as by design the information on the blockchain is public. Although, there were proposed privacy preserving blockchain platforms, such as Solidus [26] and zkLedger [37], which allow storing encrypted content on-chain, they require trusted third-party auditors. However, such an assumption in the DID ecosystem would be against its principles of self-sovereign identity (i.e., an owner has full control of its identity). Therefore, in the next protocol, we explain how to provide auditability with a privacy guarantee over an open blockchain.

5 Privacy-Preserving Protocol

In this section, we try to address the limitations described in Section 4.3. First, in Section 5.1, we sketch an intermediate protocol that deploying VRFs can provide the desired properties but requires additional security assumptions. Then we extend this idea to the final protocol, which in addition to VRFs deploys a computational zero-knowledge proving system.

5.1 Intermediate Protocol

While investigating the drawbacks of the baseline scheme, it becomes clear that the privacy issues are caused by the linkage between sent counters and the identity that has issued them. One idea to break such a linkage is to allow a client to prove that the counters are consecutive and authentic (i.e., created by the client), but without leaving public evidence of that fact. We present a protocol that accomplishes that, requiring services to verify the linkage (thus, they become trusted in that scope). In our construction, we use a VRF that for a given counter, outputs a unique and deterministic hash with a proof guaranteeing its authenticity. Hash-proof pairs can ensure services that the hashes are authentic, while hashes recorded in the blockchain ensure privacy (since without their proofs cannot be attributed to identities) and give clients the ability to audit which of their counter values were used. Basing of those insights, in the following sections we give more details of such a protocol.[2]

[2]Alternative designs, to the presented in this section, could include a system where authentication counters are encrypted under the identity's public key and the corresponding ciphertexts are submitted to the blockchain. Unfortunately, such a solution a) would require clients to constantly keep validating all new ciphertexts (in order to audit them), and b) would allow anyone to use the public key to generate and submit a valid ciphertext that would be interpreted by the client as a credential misuse.

5.1.1 Registration The registration process is similar to the baseline protocol. First, a client generates its keypair $\langle vk, sk\rangle \leftarrow KeyGen(1^\lambda)$, derives its identity $did \leftarrow H(vk)$, sets its local counter $cnt \leftarrow 0$, computes $\langle tkn, \perp\rangle \leftarrow VRF_{sk}(cnt)$ where VRF's hash tkn is called an *authentication token*, and sends to the blockchain platform the following registration request[3]

$$regReq \leftarrow \langle vk, info, tkn, Sig_{sk}(vk\|info\|tkn)\rangle.$$

The blockchain smart contract a) verifies the signature, b) adds did to the identity map (in contrast to the previous protocol, counter values are not associated with identities), c) checks that tkn is not in the *token set*, and if so d) adds the token to this set. The token set is the set of all tokens that the blockchain platform has accepted. If all these checks pass, the blockchain nodes return the did and token inclusion proofs (β^{did} and β^{tkn} respectively), proving that the identity and the token are part of the blockchain. After receiving and validating the proofs, the client is ready to use its identity for authentication.

5.1.2 Extended Authentication Prior to any authentication, the client increments its counter cnt and generates an ephemeral keypair $\langle vk_e, sk_e\rangle \leftarrow KeyGen(1^\lambda)$ (which as in the previous protocol is used for replay protection). Then, the clients initiate the process whose goal is to convince the service that a) the client's identity is registered with the blockchain, and b) the authentication process is *consistent* with the previous authentication of this client. The client prepares the following messages

$$\langle tkn_{-1}, \gamma_{-1}\rangle \leftarrow VRF_{sk}(cnt-1)$$
$$\langle tkn, \gamma\rangle \leftarrow VRF_{sk}(cnt)$$
$$tknReq \leftarrow \langle tkn, Sig_{sk_e}(tkn), vk_e\rangle$$
$$authReq \leftarrow \langle did, \beta^{did}, \beta^{tkn_{-1}}, cnt, tkn_{-1}, \gamma_{-1}, \gamma, tknReq\rangle$$

and sends $authReq$ to the service, which runs the standard authentication of vk and vk_e (as in the baseline protocol – Section 4.2) and validates the messages, 1) verifying that the tokens tkn_{-1} and tkn are authentically generated by the client for the inputs $cnt-1$ and cnt, correspondingly, i.e., $vrfyVRF_{vk}(cnt, tkn, \gamma)$ and $vrfyVRF_{vk}(cnt-1, tkn_{-1}, \gamma_{-1})$ both output True, 2) the identity inclusion proof β^{did} is correct and corresponds to vk, 3) the token inclusion proof $\beta^{tkn_{-1}}$ corresponds to the authentication token tkn_{-1}), 3) $tknReq$ is signed correctly, and 4) the authentication token tkn is not registered in the blockchain.

For the last check, the service needs to contact the blockchain platform and obtain a proof that tkn was not appended to the blockchain's tokens set (it could be realized efficiently with authenticated dictionaries [28]). Therefore the token request $tknReq$ is submitted[4] to the blockchain, where nodes only check if the token is not in the token set.[5] If so, tkn gets inserted into the tokens set and the blockchain platform returns an inclusion proof β^{tkn} (claiming that the $tknReq$ was processed correctly and tkn became part of the

[3]For a simple description, we assume that the clients use the same keypair in VRFs and digital signatures. In practice, different cryptographic primitives may require different keypairs.

[5]The nodes cannot check tkn's authenticity, since it is not accompanied with its corresponding proof γ.

blockchain) to the service which forwards it to the client (the client will use the proof for the next authentication). With these checks, the service and the client get confident that the previous and the current token are consistent with each other and no credential misuse occurred. At any time, the client can audit its authentications by simply checking whether the next token value tkn_{+1}, computed from $\langle tkn_{+1}, \perp \rangle \leftarrow VRF_{sk}(cnt + 1))$, is in the blockchain (if it is, then it implies that its credentials were misused).

5.1.3 Discussion The use of a VRF, guarantees that the service can verify that the tokens were computed by the given identity holder and they are consecutive, while submitted tokens do not reveal corresponding identities or previous tokens to the public, since VRF hashes (i.e., tokens) cannot be verified without their corresponding proofs (we also note that the token set acts as the anonymity set in this case). Moreover, tokens are deterministic, thus clients can easily check for potential credential misuses by blockchain lookups querying their next token values.

Unfortunately, the protocol requires the assumption that the participating services are honest as they verify the continuity between tokens (the blockchain platform checks only for the token uniqueness in the token set). To illustrate it, let us assume an adversary compromising client's secret key sk and a service controlled by the adversary (in fact, nothing prevents the adversary to act as a service too). In such a case, the adversary would produce and insert to the blockchain's token set a token tkn' computed from $\langle tkn', \perp \rangle \leftarrow VRF_{sk}(k)$ where k is significantly larger than the client's counter cnt. With this malicious token inserted, the adversary can keep authenticating with honest services using consecutive tokens (starting from $k + 1$). Since the client would be auditing blockchain for cnt's consecutive tokens, he would not be able to detect the malicious token (at least for a long time period).

Below, we show how this intermediate protocol can be turned into the final protocol, where the linkage between tokens and identities is verified by the blockchain (i.e., services do not have to be trusted in that scope) while not revealed the linkage to the public, preventing the described attack.

5.2 Final Protocol

The insecurity of the previous approach is caused by the fact that the verification is implemented only at a service, since verifying the token authenticity by the blockchain nodes would violate the privacy. In our final protocol, we employ a computational integrity proofing system, like zk-SNARK, to "move" the verification from the service to the blockchain nodes, however, preserving the privacy. Therefore, in addition to ensuring consecutive counters, a client needs to prove non-interactively (as the proof is to be verified by the blockchain nodes) and in zero knowledge that tokens were created under the same private key; and the previous token is in the blockchain. This requires a composition of non-interactive zero-knowledge VRF and signature verifications, and although a system like zk-SNARK can be seen as a "heavyweight" tool, it seems unavoidable to realize this. We see the optimization of this construction as interesting future work.

Algorithm 3: Circuit for proving that the previous token of the client is in the blockchain. The boxed arguments are secret and known only to the client (i.e., prover).

$Token(\boxed{vk}, \boxed{cnt}, tkn, \boxed{\gamma}, \boxed{tkn_{-1}}, \boxed{\gamma_{-1}}, \boxed{\beta^{tkn_{-1}}})$

1 **assert** $vrfyVRF_{vk}(cnt - 1, tkn_{-1}, \gamma_{-1})$
2 **assert** $vrfyVRF_{vk}(cnt, tkn, \gamma)$
3 $vrfyInclusion(tkn_{-1}, \beta^{tkn_{-1}})$ (see Alg. 2)

The registration process in the final protocol is exactly the same as in the intermediate protocol (see Section 5.1.1). Also the blockchain platform in this protocol maintains the same data structures (i.e., the identity map and the token set). Prior the protocol deployment, the proofing system requires a setup phase where proving and verification keys are generated for a circuit, i.e., $\langle z_{pk}, z_{vk} \rangle \leftarrow zkGen(Token, \lambda)$ in our case, and the blockchain platform is preloaded with the verification key. In some systems, the setup phase is assumed to be run by a trusted party. zk-SNARK shares this limitation and, in practice, this assumption can be relaxed by a multi-party computation ceremony [16].

5.2.1 New Token and its Consistency To generate a new token tkn, the client increments the counter and computes the token and its proof as in the previous protocol, i.e., $\langle tkn, \gamma \rangle \leftarrow VRF_{sk}(cnt)$ We emphasize, that due to the properties of VRFs, the value of each token is unique and deterministic. After a new token is computed, the client has to create a proof that this token is consecutive to its previous token tkn_{-1} which is already logged in the blockchain's token set, but without revealing any information about this token (in particular, that can be a proof of the initial token). To accomplish it, we use a non-interactive zero-knowledge computation integrity proofing system (item 2.3) together with the $Token$ circuit used to conduct the relevant computations (see Algorithm 3). The circuit takes as an input the client's verification key, the current counter, VRF outputs corresponding to the new and previous tokens, and the inclusion proof of the previous token. Note, that only the new token is passed as a public argument, thus the computation integrity proof will not reveal any other information, like the client's counter or its identity. In the first two lines, the circuit ensures that the tokens are created by an entity possessing the same secret key sk and for the consecutive values of the counter cnt. The third line validates the inclusion proof $\beta^{tkn_{-1}}$, ensuring that the previous token tkn_{-1} is indeed in the blockchain.

For a new token tkn and a list of secret arguments (as shown in Algorithm 3), the client generates a computation integrity proof $\pi \leftarrow zkProve_{z_{pk}}(tkn, w)$, where w is a witness. The proof will ensure a verifier (i.e., the blockchain platform) that the two tokens are created using the same secret key and the consecutive counter values, and that the previous token is part of the blockchain. Once the proof is generated the client is ready to contact the service and conduct the authentication process. It is important to note, that although generating the proof is relatively time consuming (i.e., seconds), it does not have to be conducted just before the authentication, but can be computed at any point in time between two consecutive authentications.

[5] Since we assumed network-level de-anonymization to be outside our adversary model (see Section 3.3), the request's origin (i.e., the service) cannot be de-anonymized. (It also applies to the requests sent in the final protocol from Section 5.2.) To relax this assumption, the service could use an anonymity network [42] while interacting with the blockchain platform.

Algorithm 4: Smart contract method for logging authentication events (the final protocol).

```
TokenVrfy(π, tkn)
1   assert tkn ∉ tokenSet
2   assert zkVrfy_{z_vk}(π, tkn)
3   tokenSet.add(tkn)
```

5.2.2 Extended Authentication After deriving a new token and proving its computations as described above, the client generates an ephemeral keypair $\langle vk_e, sk_e \rangle \leftarrow KeyGen(1^\lambda)$ (as previously, used against replay attacks)

$$tknReq \leftarrow \langle \pi, tkn, Sig_{sk_e}(\pi \| tkn), vk_e \rangle$$
$$authReq \leftarrow \langle did, \beta^{did}, cnt, \gamma, tknReq \rangle$$

and initiates the authentication with the service by sending $authReq$.

The service authenticates client's vk and vk_e keys by a standard authentication protocol (as in the previous protocols – Section 4.2). In addition, the service a) validates the inclusion proof β^{did} and checks whether it corresponds to the claimed did, and b) ensures that the VRF output is correct and authentic, i.e., $vrfyVRF_{vk}(cnt, tkn, \gamma)$ outputs True (vk associated with did for VRF verification is obtained from β^{did}). By these checks, the service makes sure that the new token corresponds to the did. The service does not have to validate the computation integrity proof since it will be validated by the blockchain platform in the next step. The service passes $tknReq$ to the blockchain where the request is handled by the *TokenVrfy* smart contract method (see Algorithm 4). It verifies that a) the integrity computation proof is correct, calling $zkVrfy_{z_{vk}}(\pi, tkn)$, and that b) tkn is not in the token set. These checks ensure the blockchain platform that the new token is not yet added and it was created with the same secret key and as a consecutive counter of some other (unrevealed) token, which already is logged. After a successful validation, the new token is added to the token set and the blockchain nodes create an inclusion proof β^{tkn}, stating that the $tknReq$ request was processed correctly and the new token is added to the blockchain. The service receives and validates the proof, sends it to the client (who validates it too), and proceeds with session establishment.

The protocol provides auditability since the client can query the token set from the blockchain for its consecutive token or just notice a blockchain's error response (while processing $tknReq$) during an authentication. In contrast to the previous protocol, a token added to the token set (except for the registration) has to be accompanied with a proof that its preceding token is already in the set (which proves in zero-knowledge tokens' continuity).

5.2.3 Discussion The protocol hides token-identity mappings as well as login frequency from an adversary observing the blockchain (i.e., the adversary sees new tokens, but is unable to link them to their originators or other tokens). However, if a client uses its identity with a service multiple times, then the service learns the corresponding counters. Any pair of counters allows the service to deduct how many authentications, to other services, the identity has conducted between these two counters. Although we do not believe it is a major privacy issue, our protocol can be extended by the following modifications to mitigate it: a) Prior the registration, the client generates its counter cnt as a large secret number. b) For every authentication, a new token and its proof are computed as $\langle tkn, \gamma \rangle \leftarrow VRF_{sk}(H(cnt))$ (the only difference is that the counter is internally hashed). c) The service receives $H(cnt)$, and not cnt as previously, and verifies the token authenticity by calling $vrfyVRF_{vk}(H(cnt), tkn, \gamma)$. d) The circuit verifies two tokens analogically, guaranteeing that the hashes are for the two consecutive values.

These changes would hide counter values from services, as they would learn only counters' hashes. It would eliminate the mentioned privacy leak, however, it would require a more complicated circuit.

6 Security Proofs with Tamarin

We use Tamarin Prover [33] to prove the security properties of DID authentication protocol that provide both auditability and privacy. This is the first to the best of our knowledge formal security modeling of a DID authentication protocol. All the Tamarin code can be found at [17].

6.1 The Tamarin Prover

Tamarin prover [33] is a powerful automated tool for modeling and analysis of security protocols. In Tamarin, protocols and adversaries are described by using multiset rewriting rules, and trace properties such as authentication and secrecy are specified in first-order logic statements. These rules are represented by states or facts where the symbolic representation of the adversary's knowledge, network messages, and freshly generated information are stored. The protocols and adversaries interact by updating and generating network messages based on the rules. The security properties are modeled as trace properties, which specify a sequence of rules' action.

Tamarin work in Dolev-Yao (DY) model, which is a symbolic model in which messages are represented as terms. In DY, the attacker controls the network and he can see all the messages and can manipulate, block or re-order them. However, the attacker is not capable of breaking cryptographic primitives such as forging the signature.

In Tamarin protocol execution is represented by a labeled state transition system. A *state* is a finite multiset of facts that includes arguments of the state and their current snapshot of protocol execution. The facts include the local state of all participants, messages sent over the network, and messages an attacker can construct from these messages. The action performed by the protocol participants and the attacker is specified by *rules* which rewrites one state to another. The *rules* are of the form, $a - [b] - - > c$ where a is a set of premises, b is a set of actions (optional), and c is a set of conclusions. Executing the rules need all premises to currently present in the state, satisfy requirements of all actions, and result in adding conclusions to the state while the premises are deleted. The following are some of the important prefixes use in Tamarin,

(1) x denotes fresh variable x
(2) $\$x$ denotes public variable x
(3) $\#i$ denotes timestamp i
(4) $!Fact$ denotes persistent $Fact$

Here, the only difference between persistent facts (with ! prefix) and linear facts is that persistent facts are never removed from the state, while linear facts can be removed after executing some rules.

Specifically, if there is a linear fact as a premise, but it does not appear as a conclusion, then it will be removed (consumed by the corresponding rule). To handle this, persistent facts are sometimes utilized.

Builtins, Functions, Equations In Tamarin a Function is an expression involving variables that maps some arguments to other arguments. We can define our functions or use builtins functions. Whereas an Equation is a statement used to model properties of a function. Equations can also be user-defined or extracted from built-in message theories.

In our DID authentication Tamarin model, we use various Builtins such as,

- **signing:** This theory models signature scheme. The functions symbols are $sign/2, verify/3, pk/1, true$ which satisfy the equation $verify(sign(m, sk), m, pk(sk)) = true$.
- **hashing:** This theory models a hash function. The function symbol is $h/1$ and no equations.
- **multiset:** This theory introduces the associative-commutative operator "+" which is usually used to model multisets.

The custom functions with their respective equations are as follows,

- $prove/3, vrfy/3$ with $vrfy(prove(inp, w, sk), inp, pk(sk)) = true$ helps to do obtain the zero knowledge proof and verify the proof.
- $token/2, gamma/2, vrfyVRF/4$ with $vrfyVRF(x, token(x, sk), gamma(x, sk), pk(sk)) = true$ help to obtain and verify VRF output which the token and token proof (gamma).
- $t1/1[private], t2/1[private]$ with $t1(token(a, b)) = a, t2(token(a, b)) = b$
- $g1/1[private], g2/1[private]$ with $g1(gamma(a, b)) = a$, $g2(gamma(a, b)) = b$
  Both t1 and g1 are getter functions. For example, given a pair of tokens, t1 takes the first element of the token.

6.2 Privacy-preserving DID authentication model in Tamarin

We model DID Authentication in a standard way and explain the set of rules that we define to verify the security properties.

We start with rules that help the clients to initialize their new identifier and authentication token corresponding to it,

```
rule Register_pk:
[ Fr(~ltkA) ] --$ > [!Ltk($A, ~ltkA),
                                  !Pk($A, pk(~ltkA))]

rule Register_did:
[!Pk($A, pkA)] --> [!Did($A, h(pkA))]

rule Register_tkn:
[!Ltk($A, ltkA)]
-->
[!Token($A, token('0', ltkA)),
                    !Gamma($A, gamma('0', ltkA))]
```

The *Register_tkn* rule consists of $Token$ Fact which ensures that token *tkn* and token VRF proof *gamma* belong to a specific client.

Next, we have a registration request rule created by the client using which it can register the initialized values to the blockchain,

```
rule Registration_request:
let p = <pkA, ~info, token('0', ltkA)>
in
[ Fr(~info), !Token($A, token('0', ltkA)),
!Ltk($A, ltkA), !Pk($A, pkA) ]
--[ Send_RR($A) ]->
[ Out(<p, sign(p, ltkA)>)]
```

Here, we take a new freshly generated information as well as token fact, and both secret key and public key as premises and output a public statement with its respective signature. This action will be called Send_RR as per stated in the rule.

In the next rule, as stated we demonstrate how the blockchain verifies the signature on the public inputs (included in p) from the client and check whether the token it received from the client is not registered with the blockchain using the received action facts,

```
rule Blockchain_receive:
[!Pk($A, pkA), In(<p, sig>), !Token($A, tkn),
!Did($A, did)]
--[Recv_RR($A)],
Eq(verify(sig, p, pkA), true), Unique(token) ]->
[!Did($A, did), !Betadid(did), !Betatkn(tkn)]
```

Here, Eq and $Unique$ are called restrictions such that to execute the premises, all the respective variables should satisfy the specified restrictions. Eq is an equality restriction denote by $Eq(x, y)$ means $x = y$, while $Unique$ is a unique restriction denote by $Unique(x)$ means x can only be executed once. Once, the signature verification is successful and the token is unique then it outputs *did* which belongs to the client and corresponding inclusion proofs as facts.

After registration the step, we define rules for the client to authenticate to a specific service which involves generating a new token and its corresponding computational integrity proof (which is the zkSNARK proof). We take in the previous token, gamma (token proof), betatoken (token inclusion proof), fresh variable w as witness and setup phase proving key as premises. Then, we check whether both token and gamma share the same counter and secret key as well as whether the betatoken corresponds to the token. Finally, we return a new token and gamma with its counter added by 1 and the computational integrity proof denoted as *Phi*.

```
rule Generate_tkn:
[ !Token($A, tkn), !Gamma($A, gam),
!Betatkn(betatoken), Fr(~w), !Ltk($Z, ltkz) ]
--[ Eq(g1(gam), t1(tkn)),
Eq(g2(gam), t2(tkn)),
Eq(betatoken, tkn) ]->
[ !Token($A, token(t1(tkn)+'1', t2(tkn))),
!Gamma($A, gamma(g1(gam)+'1', g2(gam))),
Phi($A, prove(token(t1(tkn)+'1', t2(tkn)),
~w, ltkz)) ]
```

Now, we move towards defining rules for privacy-preserving verification of authentication token at the blockchain. As the first step, we generate an ephemeral keypair which is done in a slightly different way from the standard since it needs to be signed with the client's secret key,

```
rule Generate_pke:
[ Fr(~ltkB), !Ltk($A, ltkA) ]
-->
[ Ltk2($A, ~ltkB), Pk2($A, sign(pk(~ltkB), ltkA)) ]
```

The ephemeral key helps to protect the client against a reply-attack, hence plays an important role. Once we have the ephemeral key, we define the next rule *Auth_req* to model the authentication request that is initiated by the client and then send to the service.

```
rule Auth_req:
let tknreq = <<proof, tkn>, sign(<proof,tkn>, ltkB), pkB>
in [Phi($A, proof), !Token($A, tkn), Ltk2($A, ltkB),
Pk2($A, pkB), !Did($A, did), !Betadid(betadid),
!gamma($A, gam) ] --[ Send_to_service($A), Eq(t1(tkn),
g1(gam))] -> [ Authreq(<did, betadid, t1(tkn), gam,
tknreq>) ]
```

Here, the new token *tkn* and token integrity proof *proof* is signed with the freshly generated ephemeral secret key *ltkB* to generate the *tknreq* which is embedded into the *Authreq* and send to the service. Next, we model the rule *Service_received* that defines how the service handles the authentication request from a client.

```
rule Service_receive:
let tknreq = <<proof, tkn>, sig, pkB>
in [ Authreq(<did, betadid, cnt, gamma, tknreq>,
!Pk($A, pkA))]--[ Recv_AR($A)],
Eq(vrfyVRF(cnt, tkn, gamma, pkA), true),
Eq(did, betadid) ]->
[ Tokenreq(tknreq) ]
```

This $Tokenreq(tknreq)$ is sent to the blockchain, after which we model the rule *Blockchain_receive* to receive the token from the service and output with an inclusion proof *betatoken*.

```
rule Blockchain_receive:
let tkereq = <<proof, tkn>, sig, pkB>
in [ Tokenreq(tknreq), !Pk($Z, pkz)
--[ Recv_TR(),
Eq(vrfy(proof, tkn, pkz), true), Unique(tkn) ]-->
[ !Betatkn(tkn) ]
```

The *betatoken* is defined with a persistent fact !*Betatkn*. In the final step, we model *Return_to_service* rule which models how the service verifies the inclusion proof received from the blockchain and establish a session with the client.

```
rule return_to_service:
[ !Token($A, tkn), !Betatkn(betatoken) ]
--[ Eq(betatoken, tkn), Finish() ]->
[ St_Session($A) ]
```

The rule takes a token and betatoken as premises, checks whether the betatoken corresponds to the token, and then initiates the session.

6.3 Security property

An attacker cannot login to an honest service using the identity on the blockchain without being detected. The client will be able to detect the occurrence of its credentials being misused at an honest service through active surveillance of the blockchain or its subsequent authentication events. An honest service or blockchain is an entity that is not controlled by the attacker. We formalize it as follows.

Definition 6.1. *A DID authentication protocol can achieve auditability and privacy with honest client C, honest service S, and blockchain B if C can register an identity on B and use it to authenticate with S and successfully log the event onto B anonymously then C hold a valid identity on B and it has not been used by an adversary to authenticate to an honest S.*

For DID authentication protocol, we use the following formalization.

```
lemma executable:
    exists-trace
    "Ex #i. Finish() @i
    & (All x y #i #j. (Send_RR(x)@i & Send_RR(y)@j)
                                ==> x = y & #i = #j)
    & (All x y #i #j. (Recv_RR(x)@i & Recv_RR(y)@j)
                                ==> x = y & #i = #j)
    & (All x y #i #j. (Send_to_service(x)@i &
                       Send_to_service(y)@j) ==> x = y)
    & (All x y #i #j. (Recv_AR(x)@i & Recv_AR(y)@j)
                                    ==> x = y & #i = #j)
    & (All #i #j. (Recv_TR()@i & Recv_TR()@j)
                                        ==> #i = #j)"
```

The above lemma checks whether there is a trace from client registration to start session passing all rules associated with,

(1) Client sends a registration request to the blockchain.
(2) Blockchain receives the registration request, verifies the signature, ensures the token received is unique, and returns the did and token inclusion proof.
(3) Client generates an authentication request to be sent to the service after deriving a new token and proving its computation.
(4) Service receives the authentication request, verifies the token is created with the respective counter and secret key, verifies whether the did correspond to its did inclusion proof, and sends a token request to the blockchain.
(5) Blockchain receives the token request, verifies it, after which the service establishes a session with the client.

We provide the Tamarin model which has 128 lines of code. The model was computed automatically on a desktop machine with 8 CPUs of type Intel(R) Core (TM) i7-8550U @1.80GHz, 16 GB for RAM, running Ubuntu 20.04.2 LTS. The runtime for our model is 3.716s and code is available at [17]. In Section 9 we also provide a UC modeling of our DID authentication protocol.

6.4 Privacy with Token Unlinkability

Our protocol achieves auditability while providing the privacy property defined as *token unlinkability* between (*did*, *token*) and (*service provider*, *token*). To formulate token unlinkability, we should depend on a typical indistinguishability game that an adversary plays with a challenger. However, to facilitate our token unlinkability analysis, we analyze it through the following three informal anonymity sets. *Anonymity client set* $\mathbb{S}_C$. A client C_0 with a registered did_0 is anonymous with respect to logging a token tkn_0 on to the blockchain if

and only if C_0 is anonymous within a set of clients $\{C_1, C_2, \cdots, C_n\}$ defined as $\mathbb{S}_C$ anonymity set.

Anonymity service set $\mathbb{S}_S$. A service provider S_0 is anonymous with respect to logging token to the blockchain on behalf of the clients in the set $\mathbb{S}_C$ if and only if S_0 is anonymous within a set of service providers $\{S_1, S_2, \cdots, S_n\}$ defined as $\mathbb{S}_S$ service anonymity set.

Anonymity token set $\mathbb{S}_{BC}$. A token tkn_0 on the blockchain is anonymous if and only if tkn_0 is indistinguishable from other tokens $\{tkn_1, tkn_2, \cdots, tkn_n\}$ within a set defined as $\mathbb{S}_{BC}$ token anonymity set.

The privacy guarantee of our protocol depends on the size of the anonymity set. If there is only a single client $C_0 \in \mathbb{S}_C$ and one token $tkn_0 \in \mathbb{S}_{BC}$, then a union set $\mathbb{S}_C \cup \mathbb{S}_{BC}$ with the set size at most n^2 compromises privacy with the probability $1/n^2 + negl(\lambda)$. Similarly, in the case of service providers.

7 Evaluation

7.1 Performance Evaluation

We implement our baseline and privacy-preserving protocols on Hyperledger Fabric.[6] This is a permissioned blockchain that uses PBFT consensus, and supports Turing-complete smart contracts called *chaincodes*. It executes smart contracts inside Docker containers.

Our implementation consists of smart contract functions written in Golang which provides rich support for complex data structures and cryptographic libraries. Signatures are implemented using *crypto/ecdsa* library with ed25519 curve. The *TokenVrf* function as in Algorithm 3, which contains the most complex logic for verification, is implemented using *go-snark* [10].

The client and service provider are written in Golang. They interact with the blockchain via REST APIs. The client uses *circom* compiler [7] to compile the *Token* circuit from Algorithm 3. To make the circuit more efficient, we use EdDSA with the Babyjub [9] elliptic curve to implement VRF, and MiMC7 [14] as the hash function. Computation integrity proofs are generated using websnark.[7]

7.1.1 Methodology We evaluate the performance of our protocols using the following metrics, 1) Throughput: the number of authentication events completed per second, 2) Latency: the end-to-end latency at the client and 3) Cost breakdown: the computation cost at the client, service provider, and the blockchain node. We pre-generate the proofs and use up to 5 clients to generate increasing loads to the blockchain. Throughput is computed as the number of transactions included in the blockchain over the measurement period, which is set to 10 minutes. End-to-end latency is computed as the time from when the client submits the transaction to when the transaction is included in the blockchain.

We vary the size of the blockchain by varying its fault tolerance threshold f. The number of nodes is $3f + 1$. We vary f from 0 to 6. This is to measure the impact of consensus protocol on the overall performance. At $f = 0$, there is no consensus. At $f > 0$, the cost of consensus increases with f.

Our experiments are run on both a local cluster and on AWS. Each cluster node has 8GB RAM, 2.10 GHz CPU, 1Gbps Ethernet, and runs Ubuntu 18.04 OS. For AWS, we use `t1.xlarge` instances (4 vCPUs, 16GB RAM) across two regions (Singapore and Hong Kong). Unless stated otherwise, the results presented below are for local cluster, averaged over 5 runs.

[6]We use an old version of Hyperledger Fabric, i.e. v0.6, instead of the newer versions, because it has better security and performance.
[7]https://github.com/iden3/websnark

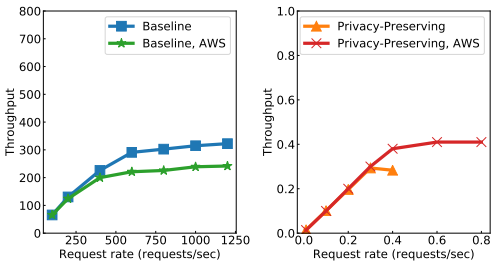

Figure 2: Throughput with varying client request rates ($f = 4$).

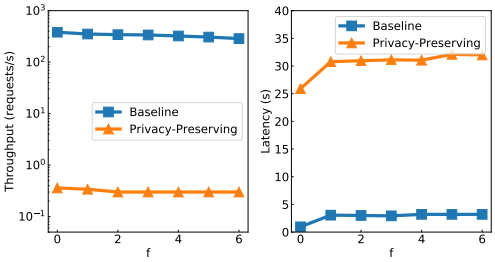

Figure 3: Peak throughput and latency.

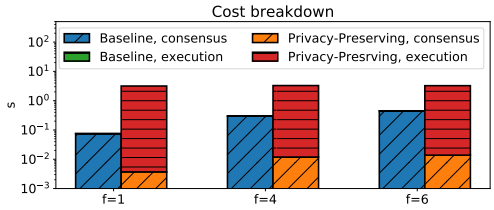

Figure 4: Cost breakdown at blockchain nodes.

7.1.2 Results We first measure throughput by varying client request rates. The results for $f = 4$ to (or 13 nodes) are shown in Figure 2 (results for other values of f are similar).

The throughput of the baseline protocol increases up to 380, whereas that of the privacy-preserving protocol peaks at 0.35 authentication events per second (or roughly 1 authentication per every 3 seconds). The low throughput of the latter is due to the overhead of zkSNARK primitives. Each proof is only 784 bytes in size, but verifying it takes 3.18*s* on average. The baseline performance is lower than, but consistent with the results in [20], because of the overhead in signature verification.

Figure 3 shows the peak throughputs for varying f, and the latency at these throughputs. It can be seen that increasing the number of blockchain nodes has no effect on the privacy-preserving protocol. In particular, the throughput remains at 0.35 authentication events per second. In contrast, the baseline protocol suffers a drop of 30% from 380 to 280 events per second. The latency remains constant with increasing f: 3s on average for the baseline, and 30s on average for the privacy-preserving protocol. We observe that the throughputs when running on AWS are slightly lower for the baseline and higher for the other protocol. It is because network

bandwidth across AWS regions are lower than that of the local cluster, and that the AWS instances have more powerful CPUs than the local cluster's VMs.

To better understand the performance difference, we examine the cost breakdown at the blockchain node. In particular, we measure the cost of consensus and transaction execution at the blockchain. Figure 4 compares these costs for the two protocols, with different values of f. Consensus is the major cost in the baseline protocol, whereas transaction execution contributes most to the cost in the privacy-preserving protocol. In particular, for $f = 1$, the consensus and execution latency in the baseline protocol are 0.074s and 0.0023s respectively (0.0052s and 0.0036s for AWS). In the privacy-preserving protocol, they are 0.0037s and 3.16s respectively (0.0012s and 2.3s for AWS). As the result, the baseline protocol is sensitive the consensus cost which increases with f, whereas the other protocol is only affected by the execution cost.

So far, we have ignored the cost of generating proofs at the client. Before generating proofs, the system has to first perform the one-time setup phase (i.e., execute *zkGen*), after which clients can generate proofs (i.e., calling *zkProve*). The cost of the setup phase, and of proof generation is shown in Table 2. It can be seen that the setup phase is expensive, and its cost increases when more signatures are included in the proof (since the circuit size increases). For example, with $f = 5$ (or 16 nodes), the setup takes almost 2 hours to complete. We note, however, that this cost is incurred only once, and can be amortized over time.

7.2 Discussion

We remark that even though the throughput of the privacy-preserving protocol is low, the baseline protocol achieves high enough throughput that makes it practical. In particular, we note that the authentication workload at Google is reported to be 20 requests per second over the dataset of $677,000$ users [31]. Extrapolating from this, the baseline protocol can support workloads of up to $13M$ users. Importantly, this protocol can scale horizontally with a sharded blockchain, because different identities can be partitioned and processed independently in multiple shards. It means the protocol can scale out to support billions of users.

7.3 Integration

Our privacy-preserving protocol can be integrated with the latest verifiable credential scheme such as Coconut [41] and EL Passo [44] to provide logging of authentication events that can help the users to detect credential misuse. We briefly explain how our protocol can be integrated.

Coconut is a selective disclosure credential scheme where a user first sends a coconut issue request to a set of authorities (identity providers) with public and private attributes using which each authority issue a partial credential. The user aggregates a threshold number of shares to obtain a consolidated credential and then invoke a show protocol to selectively disclose attributes or make statements about her to a service (doing authentication). This credential can be used across multiple services without connecting with the authorities during every show protocol execution. Our approach is to associate a token with each credential issue request (during which a token is initialized and registered with the blockchain managed by the authorities), and before each show protocol, the user needs to generate a new token consistent with previous token and token integrity proof which is then sent to a service who redirect it to the blockchain. The service will be able to alert the users (or user can audit) of a misuse if the verification of integrity proof fails at the blockchain.

Table 2: The cost of setup and generating proofs at the client.

# sigs in π_{in}^{tkn}	Setup (mins)	Generate Proof (secs)
2	60.53	14.44
3	73.41	15.35
4	97.81	16.79
5	109.60	17.88
6	118.86	18.09

EL Passo uses a verifiable credential to overcome the privacy issues of the traditional Single-Sign-On (SSO) systems. The user can obtain a verifiable credential from an identity provider and use it across multiple relying parties (service) without going through the identity provider thus preventing the identity provider from tracking user activities. However, the drawback is that this also prevents the users' ability to detect any possible credential misuse. Since the user cannot rely on the identity provider for logging the authentication events, we propose to deploy an additional blockchain infrastructure where the user can log the authentication event (or choose its own blockchain and then deploy smart contact to log the event, and during the authentication, the user need to specify to the relying party on which blockchain the token need to be logged). So during the setup phase, along with generating a user secret the user also initializes a token and obtains a credential from the identity provider associate with both these values. It then needs to register this token onto the blockchain before using the credential to authenticate with any relying party. The usage of the credential and subsequent detection of misuse during the sign-on phase is similar to Coconut.

8 Related Work

Our work is the first to propose DID authentication protocols that achieve auditability and privacy. In the following, we discuss related works with similar properties.

Blockstack [18] introduces a distributed namespace where a binding of a name to a key can be published on a public blockchain. Similarly, CONIKS [34] supports name to key binding, and provides publicly verifiable proof that the binding was added to a public (centralized) log. The integrity of this log depends on a number of auditors that frequently check the log. This design is implemented in Google's Key Transparency (KT) [13]. These systems provide similar guarantees to those of a DID system, that is, full user control of the name to key binding, and transparency of the data. However, like existing DIDs, they do not have the auditability property as defined in this paper. Chu et. al. [27] introduce Ticket Transparency which is an accountable single sign-on system with a privacy-preserving public log. The system allows users to detect if an identity provider has signed a fraudulent ticket and the ticket has been used to sign on to a service provider. The problem addressed by Ticket Transparency is similar to ours, but it is specific to existing single sign-on systems with centralized identity providers. Our work focuses on DIDs in which users manage their own identities.

Privacy Pass [29] is an anonymous user-authentication scheme that provides users with the ability to create and sign cryptographically "blinded" tokens for services that require PoW authentication. Extending this Facebook build a privacy-preserving logging scheme known as DIT [30] which addresses the constraints of implementing anonymized logging infrastructure that involves mobile clients and additional requirements of lower latency. Though both address token anonymity, however, achieving it while checking for token consistency could lead to similar overhead as our privacy-preserving approach.

Coconut [41] is a privacy-preserving verifiable credential scheme with selective disclosure property with support for threshold credential issuances. The users can obtain a credential from the blockchain and authenticate with multiple services while being unlinkable across different usage. EL Passo [44] provides a privacy-preserving Single-Sign-On (SSO) system based on verifiable credential that provide the same security guarantee of traditional OAuth protocol without compromising on usability. The credential usage is similar to coconut and both use PS signature [39] to implement a verifiable credential scheme. Unlike our scheme, EL Passo nor Coconut does not aim to detect credential misuse. It can be seen as orthogonal to our work and combining it with our solution could be a possible future work.

WAVE [19] and Droplet [40] are decentralized authorization systems that aims to replace centralized solutions such as OAuth. WAVE proposes novel cryptographic protocols for delegation of access permissions without relying on trusted parties. It uses a secure log based on Google's Trillian [11] data store. The log can be efficiently audited to detect any forks in its history, similarly to CONIKS. WAVE supports user privacy by using a Reverse Discoverable Encryption scheme to encrypt authorization policies so that only the users with appropriate permissions can see them. Droplet is very specific to managing access to time-series data in an IoT environment. WAVE and Droplet is complementary to our work since we focus on authentication instead of authorization.

Going beyond authentication, SAMPL [38] proposes a blockchain-based system for auditing the user surveillance process. The auditability property targeted by this system is different from our definition, as it refers to the detection of non-compliance to the surveillance process, for example, when a user is being over-surveilled. SAMPL requires the authority and relevant entities to log surveillance requests and responses to the blockchain, and ensure the privacy of such data when auditing.

9 Conclusions

In this paper, we proposed two DID authentication protocols that allow users to detect credential misuse. The first protocol achieves high performance, but with limited privacy. The second protocol achieves full privacy, but with high performance overhead. We gave a formal security analysis of the final protocol using Tamarin Prover. We implemented these protocols using Hyperledger Fabric, and evaluated their performance on a local cluster. The results showed that the first protocol achieves throughput that is sufficient to support realistic workloads. The second protocol incurs significant overhead, and suffers three orders of magnitude lower throughput. In future work, we plan to improve the two protocols by using more efficient blockchains with sharding and parallel execution.

References

[1] Blockstack. https://blockstack.org/, 2018.
[2] Decentralized identity foundation. https://identity.foundation/, 2018.
[3] Didauth. https://tinyurl.com/y89tahad, 2018.
[4] Evernym. https://www.evernym.com, 2018.
[5] Hyperledger indy. https://tinyurl.com/yycca4ek, 2018.
[6] Nuggets. https://nuggets.life/, 2018.
[7] Circom compiler. https://github.com/iden3/circom, 2019.
[8] Google last login. https://tinyurl.com/lqlg2xz, 2019.
[9] Babyjub. https://tinyurl.com/yc7kmcsj, 2020.
[10] go-snark. https://github.com/arnaucube/go-snark, 2020.
[11] Google trillian. https://github.com/google/trillian, 2020.
[12] Hyperledger fabric. https://tinyurl.com/ydaswf3j, 2020.
[13] Key transparency. https://tinyurl.com/ybhedmfs, 2020.
[14] Mimc7. https://tinyurl.com/y99c2khj, 2020.
[15] Quorum. https://www.goquorum.com, 2020.
[16] Zcash: Parameter generation. https://z.cash/technology/paramgen/, 2020.
[17] Did authentication tamarin model. https://github.com/bithinalangot/DIDAuthTamarin, 2021.
[18] Ali et al. Blockstack: A global naming and storage system secured by blockchains. In *USENIX ATC*, 2016.
[19] Andersen et al. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.
[20] Anh Dinh et al. Untangling blockchain: a data processing view of blockchain systems. *TKDE*, 2018.
[21] Ben-Sasson et al. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptology ePrint Archive*, 2018.
[22] Bitansky et al. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.
[23] Bowe et al. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017.
[24] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE FOCS*. IEEE, 2001.
[25] Castro et al. Practical byzantine fault tolerance. In *USENIX OSDI*, 1999.
[26] Cecchetti et. al. Solidus: Confidential distributed ledger transactions via pvorm. In *ACM CCS*, 2017.
[27] Chu et al. Ticket transparency: Accountable single sign-on with privacy-preserving public logs. In *SecureComm*. Springer, 2019.
[28] S. A. Crosby and D. S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM TISSEC*, 2011.
[29] Davidson et al. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018.
[30] S. Huang, S. I. S. Jeyaraman, S. Kushwah, C.-K. Lee, Z. Luo, P. M. A. Raghunathan, S. Shaikh, Y.-C. Sung, and A. Zhang. Dit: De-identified authenticated telemetry at scale.
[31] Kurt et al. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*, 2019.
[32] D. Maram, H. Malvai, F. Zhang, N. Jean-Louis, A. Frolov, T. Kell, T. Lobban, C. Moy, A. Juels, and A. Miller. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. *IACR Cryptol. ePrint Arch.*, 2020:934, 2020.
[33] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013.
[34] Melara et al. Coniks: Bringing key transparency to end users. In *USENIX Security*, 2015.
[35] Micali et. al. Verifiable random functions. In *IEEE FOCS*, 1999.
[36] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2019.
[37] Narula et al. zkledger: Privacy-preserving auditing for distributed ledgers. In *USENIX NSDI*, 2018.
[38] Panwar et al. Sampl: Scalable auditability of monitoring processes using public ledgers. In *ACM CCS*, 2019.
[39] D. Pointcheval and O. Sanders. Short randomizable signatures. In *Cryptographers' Track at the RSA Conference*, pages 111–126. Springer, 2016.
[40] Shafagh et al. Droplet: Decentralized authorization and access control for encrypted data streams. In *USENIX Security 2020*.
[41] Sonnino et al. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. *NDSS 2019*, 2018.
[42] Syverson et al. Tor: The second-generation onion router. In *USENIX Security*, 2004.
[43] Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.
[44] Z. Zhang, M. Król, A. Sonnino, L. Zhang, and E. Rivière. El passo: Privacy-preserving, asynchronous single sign-on. *arXiv preprint arXiv:2002.10289*, 2020.

Appendix

Here, we formally model the final protocol (Section 5.2) under the UC framework [24] using the ideal-world/real-world paradigm, in which parties are modeled as probabilistic polynomial-time interactive Turing machines.

A protocol π_{DID} is secure if there exists no environment $\mathcal{Z}$ that can distinguish whether it is interacting with adversary $\mathcal{A}$ and parties running protocol π_{DID} or with the ideal process for carrying out the desired task, where the ideal adversary $\mathcal{S}$ and dummy parties interact with the ideal functionality $\mathcal{F}_{\text{DID}}$. Every functionality and every protocol invocation should be instantiated with a unique session-ID that distinguishes it from other instantiations. More formally, we say that the protocol π_{DID} emulates the ideal process if, for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for all environments $\mathcal{Z}$, and the ensembles $\text{IDEAL}_{\mathcal{F}_{\text{DID}},\mathcal{S},\mathcal{Z}}$ and $\text{REAL}_{\pi_{\text{DID}},\mathcal{A},\mathcal{Z}}$ are computationally indistinguishable, i.e., $\text{IDEAL}_{\mathcal{F}_{\text{DID}},\mathcal{S},\mathcal{Z}} \approx \text{REAL}_{\pi_{\text{DID}},\mathcal{A},\mathcal{Z}}$. We outline a security proof in the UC framework to show the registration and authentication subprotocols that match $\mathcal{F}_{\text{Reg}}$ and $\mathcal{F}_{\text{Auth}}$ under a concurrent composition.

.1 UC Modelling Final Protocol

Functionality $\mathcal{F}_{\text{DID}}$: We briefly re-formalize our privacy-perserving protocol under UC model which consists of three phases 1) *initialization*, 2) *registration*, and 3) *authentication*. The functionality $\mathcal{F}_{\text{DID}}$ is parameterized with security parameter λ and algorithms of cryptographic primitives. In the initialization phase, $\mathcal{F}_{\text{DID}}$ receives command (Start, sid) from client $C_i \in \mathcal{C}$ and forwards a notification (Start, sid, C_i) to the simulator $\mathcal{S}$ (and stores this session). During registration, it receives a command (Register, sid, C_i, $regReq_i$) from the client $C_i \in \mathcal{C}$. $\mathcal{F}_{\text{DID}}$ checks for possible entry, if not makes a new entry $(sid, C_i, regReq_i, 1)$ and set $b_i = 1$, otherwise, ignore the command and return (Unregistered, sid, C_i, Error) to C_i and halt. After that, forwards command (Register, sid, C_i, 1) to adversary $\mathcal{S}$. Next, it receives (Corrupt, sid, C_i) from $\mathcal{S}$ to which it respond with an (Leak, $sid, C_i, regReq_i$) and records $(sid, C_i, regReq_i^*)$. In addition, $\mathcal{F}_{\text{DID}}$ passes the message to the C_i along with the did and token inclusion proofs (β^{did} and β^{tkn} respectively), and records (Registered, $sid, C_i, regReq_i$) (The client C_i is registered). Finally, in the authentication phase it receives a command, (Login, $sid, C_i, authReq_i$) from $C_i \in \mathcal{C}$, where $(authReq_i)_{i=1}^{N}$. The ideal functionality $\mathcal{F}_{\text{DID}}$ checks for possible entry $(authReq_i, 1)$ and then sends a notification (Login, $sid, C_i, authReq_i$) to $\mathcal{S}$ and SP. Otherwise, outputs $\perp$ and aborts. Now, SP input (Token-match, sid, SP, $authReq_i$) received from $\mathcal{F}_{\text{DID}}$, after which $\mathcal{F}_{\text{DID}}$ checks for an entry (sid, SP, $tknReg_i$, 1). Once confirmed, it sends a notification (Token-match, sid, SP, $authReq_i$) to $\mathcal{S}$, otherwise, ignore the command. Lastly, it receives (Token-matched, sid, SP, $Inclusion$) from SP, and validate that no entry (sid, SP, $Inclusion$, 0) is recorded before. In case of validation failure, $\mathcal{F}_{\text{DID}}$ returns (Error, sid) to SP and halts otherwise returns (Retrive-matched, sid, SP, $Inclusion$) to SP, and records the entry (sid, SP, $Inclusion$, 1).

Functionality $\mathcal{F}_{\text{ZK}}$: $\mathcal{F}_{\text{ZK}}$ is parameterized with a relation R and communicate with a prover P, a verifier V and an adversary $\mathcal{S}$. It receives a command (ZKProver, sid, (x, w)) from P for a statement x associated with a witness w, if $R(x, w) = 1$, then send $(zkproof, sid, x)$ to V and $\mathcal{S}$, and exit. Otherwise exit.

Functionality $\mathcal{G}_{\text{Ledger}}$: The ideal blockchain functionality $\mathcal{G}_{\text{Ledger}}$ is running with a set of clients $C \in \mathcal{C}$, a service provider SP, and an ideal adversary $\mathcal{S}$. $\mathcal{G}_{\text{Ledger}}$ holds a database indexed on integers and initializes a counter $idx = 0$. First, it receives a command (Write, sid, C_i, $regReq_i$) from C_i in the registration phase, after which $\mathcal{F}_{\text{DID}}$ checks whether (C, sid) is stored in the database under the index idx that is corresponding to cnt from the client. It sends a notification (Write,sid, C_i) to $\mathcal{S}$ and updates the counter $idx = idx + 1$. Next, it receives (Token-match, SP, sid, $authReq_i$) from $\mathcal{F}_{\text{Auth}}$ (i.e., the service provider SP) in the authentication phase, if there is an entry (sid, SP, $authReg_i$, 1), then check for a tuple $(C_i, tknReq_i)$ stored under the index idx. If yes, then pass (Token-matched, SP, $authReq_i$) to SP.

Functionality $\mathcal{F}_{\text{Auth}}$: This funcationality is parameterized with participants and $\mathcal{G}_{\text{Ledger}}$ for authentication. It receives (Login, sid, C_i, $authReq_i$) from $C_i \in \mathcal{C}$. If there is a record $(C_i, authReq_i, 1)$, then forward it to $\mathcal{S}$, and then return (Login-verified, sid, C_i, $authReq_i$) to C_i. After this, it interact with $\mathcal{G}_{\text{Ledger}}$ by sending a command (Token-match, sid, SP, $authReq_i$) to which it respond with (Token-matched,sid, SP, $authReq_i$). $\mathcal{F}_{\text{Auth}}$ records $(C_i, authReq_i)$ and forwards (Login-verified, sid, C_i,$authReq_i$) to C_i.

.2 Formalized Final Protocol

Protocol π_{DID}: Our construction operates in the ($\mathcal{F}_{\text{ZK}}$, $\mathcal{G}_{\text{Ledger}}$, $\mathcal{F}_{\text{Auth}}$)-hybrid model and in phases as explained in previous section. In the initialization phase, it receives a command (Start, sid) from the environment $\mathcal{Z}$, then the client C_i initializes requires keys.After this step, it sends command (Write, $sid, C_i, vk, vk_e, vk_{orf}$) to $\mathcal{G}_{\text{Ledger}}$ and SP. The smart-contract is initialized and created, it keeps z_{vk} privately and sends the z_{pk} to the client by invoking $(z_{vk}, z_{pk}) \leftarrow zkGen(\lambda, C)$ for a circuit C. In the registration phase, C_i generate the keys, drive did from public key and then interact with $\mathcal{G}_{\text{Ledger}}$ to obtains the inclusion proofs $\beta_{in}^{did}, \beta_{in}^{tkn_{cnt}}$, which it stores locally. In the authentication phase, the protocol receives command (Login, $sid, C_i, authReq_i$) from the environment $\mathcal{Z}$ that helps a client C_i to authenticate to a service provider. First, C_i derive a new token tkn and associate zero-knowledge proof of computation as explained in Section 5.2.1. Once proof is derived it sends a command (ZKProver, sid) to $\mathcal{F}_{\text{ZK}}$, and obtains $(zkproof, sid)$ as response. The legitimacy of the new token and its consistency is verified as shown in Algorithm 3.

Once validation is complete, (Login, $sid, C_i, authReq_i$) is send to $\mathcal{F}_{\text{Auth}}$. After receiving, it sends (Token-match, sid, SP, $authReq_i$) to $\mathcal{G}_{\text{Ledger}}$. At the ledger, the smart contract verifies the new token request $tknReq_i$ from a service provider as shown in Algorithm 4 and returns the notification (Token-matched, sid, SP, $Inclusion$) from $\mathcal{F}_{\text{Auth}}$. $\mathcal{F}_{\text{Auth}}$ then fetches $Inclusion$ from $\mathcal{G}_{\text{Ledger}}$ and validates the legitimacy, i.e., $\mathcal{F}_{\text{Auth}}$ returns (Login-verified, $sid, C_i, authReq_i$) to client C_i.

Theorem .1. *Let $\mathcal{F}_{DID}$ and $\mathcal{A}$ be an ideal functionality and a probabilistic polynomial-time (PPT) adversary for our proposed scheme, respectively, and let $\mathcal{S}$ be an ideal-world PPT simulator for $\mathcal{F}_{DID}$. Then the proposed scheme UC-realizes the ideal functionality $\mathcal{F}_{DID}$ for any*

PPT distinguishing environment $\mathcal{Z}$ in the ($\mathcal{F}_{ZK}$, $\mathcal{G}_{LEDGER}$, $\mathcal{F}_{AUTH}$)-hybrid model.

Construction of $\mathcal{S}$. In a nutshell, if a message is sent by an honest party to $\mathcal{F}_{\text{DID}}$, $\mathcal{S}$ emulates appropriate real world communications among participants for $\mathcal{Z}$ with information obtained from $\mathcal{F}_{\text{DID}}$. If a message is sent to $\mathcal{F}_{\text{DID}}$ by a corrupted party, $\mathcal{S}$ extracts the input and interacts with the corrupted party with the help of $\mathcal{F}_{\text{DID}}$.
Hybrid H.1 proceeds as in the real world protocol, except that $\mathcal{S}$ emulates $\mathcal{G}_{\text{LEDGER}}$. In particular, $\mathcal{S}$ invokes the signature scheme and generates a keypair (sk_C, vk_C), then keeps the sk_C and publishes vk_C. Whenever $\mathcal{A}$ wants to communicate with $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{S}$ records $\mathcal{A}$'s messages and faithfully emulates $\mathcal{G}_{\text{LEDGER}}$'s behavior. Similarly, $\mathcal{S}$ emulates $\mathcal{G}_{\text{LEDGER}}$ by storing items internally. As $\mathcal{A}$'s view in H.1 is perfectly simulated as in the real world, $\mathcal{Z}$ cannot distinguish between H.1 and H.0.
Hybrid H.2 proceeds as in H.1 except for modifying how to generate the signature. The indistinguishability between H.1 and H.2 can be shown by the following reduction to the EU-CMA property of the signature scheme. In more detail, if $\mathcal{A}$ sends forged attestations to $\mathcal{G}_{\text{LEDGER}}$, signature verification conducted by $\mathcal{G}_{\text{LEDGER}}$ will fail with all but a negligible probability. If $\mathcal{Z}$ can distinguish H.2 from H.1, $\mathcal{Z}$ and $\mathcal{A}$ can be used to win the game of signature forgery. As assumed, unforgeability is guaranteed by the signature scheme used.
Hybrid H.3 proceeds as in H.2 but has $\mathcal{S}$ emulate the authentication, which means the honest clients will send the command $(Login, sid)$ to $\mathcal{F}_{\text{AUTH}}$. $\mathcal{S}$ emulates messages from $\mathcal{G}_{\text{LEDGER}}$ as described above and $\mathcal{F}_{\text{ZK}}$. It is clear that $\mathcal{A}$'s view is distributed exactly as in H.2, as $\mathcal{S}$ can emulate $\mathcal{G}_{\text{LEDGER}}$ and $\mathcal{F}_{\text{ZK}}$ perfectly.

Hybrid H.4 proceeds as in H.3 except that, under the random oracle model, the adversary does not know the secret VRF key sk_{vrf}, but must distinguish between pairs (cnt, tkn) where tkn is the VRF hash output on counter input cnt, and pairs (cnt, r) where r is a random value. This adversary knows the public values pk_{vrf}, and it can easily compute $(tkn, \gamma) \leftarrow VRF_{sk_{vrf}}(cnt)$ if he knows the sk_{vrf}. However, sk_{vrf} is kept private. In addition, and $Sign_{sk_e}(tkn)$ looks random in the random oracles even knowing sk_{vrf}, thus, the pseudorandomness is guaranteed because the adversary cannot distinguish tkn from a randomness distribution and tkn is pseudorandom in the range of H. Further, suppose an adversary, given the secret key sk_{vrf}. The verification can satisfy both $vfyVRF_{pk_{vrf}}(cnt, \pi_{sk_{vrf}}(cnt)) = 1$ and $vfyVRF_{pk_{vrf}}(cnt, \pi'_{sk_{vrf}}(input)) = 1$ for cnt and cnt' and the uniqueness adversary cannot distinguish them unless $cnt = cnt'$ because of the uniqueness property of VRF. Hence, indistinguishability between H.4 and H.3 can be directly reduced to the uniqueness and the pseudorandomness properties of VRF.
Hybrid H.5 proceeds as in H.4, except for modifying how to generate the hash value. The indistinguishability between H.5 and H.4 can be shown by a reduction to the second pre-image resistance property of the hash function. In H.4, the output of VRF will be hashed via $H(\cdot)$, in H.5, sample a value from a uniform distribution at random and input it to $H(\cdot)$. If $\mathcal{Z}$ can distinguish H.5 from H.4, it implies that $\mathcal{A}$ can break the second pre-image resistance which contradicts with our assumptions.

It remains to observe that H.5 is identical to the ideal protocol. Then combining all together with this invariant ensures that H.5 precisely reflects the ideal execution of $\mathcal{F}_{\text{DID}}$ in the ($\mathcal{F}_{\text{ZK}}$, $\mathcal{G}_{\text{LEDGER}}$, $\mathcal{F}_{\text{AUTH}}$)-hybrid model.