Zero-Knowledge Credentials for Smart Contracts

Lucas Switzer, Tjaden Hess

May 13, 2018

Abstract

Public blockchains present unique opportunities for the implementation of autonomous and trustless systems, but suffer from trade-offs between privacy and expressivity. In this paper we present an implementation of a zkSNARK-based anonymous credential scheme for the Ethereum blockchain and give benchmarks for usage costs. We present as well an example application.

1 Introduction

While blockchains have found use cases in publicly accessible distributed systems, they pose a challenge in that due to their public nature it is currently impossible to attest to aspects of one's identity without some trusted credential issuer.

1.1 Prior Work

Identity systems for public blockchains have been proposed before in the form of serivces like Civic and Uport, wherein users can acquire attestations from trusted issuers about facets of their identity (name, age, country of origin, etc). These services promise "anonymous" attestation, but are *linkable*, i.e. multiple attestations by the same user are trivially identifiable, and *pseudonymous*, i.e. attestations are linked to a pseudonym that is known by the credential issuer. If a credential issuer leaks their records, then all past activity by a user is deanonymized.

A zero-knowledge credential scheme for blockchains was proposed by Garman et al. [3], using zkSNARKS. Our system adapts this scheme for the Ethereum[5] blockchain and implements more expressive forms of credentials that can be easily utilized for ad-hoc attestations in zero-knowledge.

1.2 Protocol Overview

1.2.1 Actors

The major actors in this protocol are *issuers*, *verifiers* and *users*. A verifier is a smart-contract that wants to ensure that users calling a function are authorized to do so. In order to authorize users, the contract designates an issuer, who is responsible for granting credentials to users and keeping a

Merkle tree of valid credentials. The verifier then requires that each user provide a zero-knowledge proof that the user possesses a credential in the merkle tree with attributes in the required range. Both credential issuance and usage is done in zero-knowledge and attestations are reusable and unlinkable.

1.3 Implementation

We implement zero-knowledge proofs using the elliptic curve pairing precompiled contracts in Ethereum, using the Pinnochio protocol [4]. We constructed the arithmetic circuits for the proofs using libsnark [1]. Commitment trees are stored using IPFS [2].

2 Properties and Features

Our zero-knowledge identity system allows for highly flexible ad-hoc issuance of credentials through *k-show* credentials and range-bounded attributes.

The public inputs are

$$\texttt{m\_root},\ \{l_1, \ldots, l_7\},\ \{u_1, \ldots, u_7\}, S, k$$

Each proof is an assertion that the prover holds a credential

$$c = (sk, \{attr_1, \ldots, attr_7\})$$

such that

1. The credential is a leaf of a merkle tree with root $\texttt{m\_root}$.
2. The credential contains some attributes $\{attr_1, \ldots, attr_7\}$ with $attr_i \in [l_i, u_i]$
3. The credential has been used fewer than k times with the salt S

3 Implementation

3.1 Credentials

Each credential is of the form

$$c = (\texttt{sk}, \{attr_1, \ldots, attr_n\})$$

where $\texttt{sk} \in \{0,1\}^\lambda$ and $attr_i \in \mathbb{Z}_{2^{32}}$.

3.2 Credential Requests

A credential request is a tuple of the form

$$r = (\texttt{merkle\_root}, \texttt{serial}, \{u_1, \ldots, u_n\}, \{\ell_1, \ldots, \ell_n\}, S, k)$$

where `merkle_root` is the root of a Merkle-tree of valid credential commitments, $u_i, \ell_i \in \mathbb{Z}_{2^{32}}$ are bounds on acceptable attribute values, S is a request-specific salt, and $k \in \mathbb{N}$ is the allowed number of uses of a single credential per salt S.

3.3 k-show Credentials

Every proof π that a user generates has as public input $\texttt{serial}_\pi$ which is defined as

$$H(\texttt{sk}_u || S_v || k)$$

and a public input `k_bound` such that $k \leq \texttt{k\_bound}$. For each proof that the verifier accepts, the verifier adds $\texttt{serial}_\pi$ to its nullifier set N. A verifier accepts π only if $\pi \notin N$. Thus, for a given salt S_v the user may produce up to `k_bound` distinct valid proofs, while remaining unlinkable. The verifier v may select S such that it changes over time, for instance such that it contains the hash of the most recent block with height a multiple of 1 million, which allows for rate-limited credentials.

3.4 Bounded-attributes

A proof for a given credential request r must assert that $attr_i \in [\ell_i, u_i]$ for all $1 \leq i \leq n$. This allows attestations such as "I possess over \$1 million", "I am between the ages of 21 and 45", or "I live in the USA", without revealing any additional information.

3.5 Credential Scheme

verifier

3.6 The ZKID Service

3.6.1 libsnark

3.6.2 JSON RPC

The ZKID client application acts as a JSON RPC service that can handle requests from client dApps. For the simplest implementation, the RPC service offered a single procedure: GenerateProofs. This procedure takes as input all the fields nessesary to construct the required set of public inputs for the proofs. If proof generation was successful, the service will respond with a set of generated proofs and relevant public inputs. The returned proofs can then be sent to the verifier contract.

3.7 Verifiers

It is expected, although not required, that credential-enabled contracts perform their own proof verification. For this reason, we have developed a Solidity library that enables contracts to verify proofs autonomously. The verifier library follows a scheme similar to that of Zokrates [insert citation here]. The library exposes a single method to be utlized by the higher-level contract: verifyTx. This method does precisely what the name implies and verifies as set of proofs encoded as bytes and returns true if all proofs where sucessfully verified and false otherwise.

It is important to note that all verification calculations are done on-chain and therefore can potentially require immense amounts of gas. For this reason, the number of proofs requested for verification should be kept to a minimum.

3.8 Issuers

Although there is no strict template for an issuer (asside from the credential contruction), we purpose a scheme that we believe will offers most of the elementary functions of an issuer. This basic scheme involves 2 main functions: credential-holder stake and distributed merkle tree storage.

3.8.1 Credential-Holder Stake

An obvious vulenerability of the credential system as described is the ability to share credentials. So, for example, an uncredible could may a credible party for a specific credential. We proprose that an issuer's require a credential-requester to submit a stake when applying for a credential. This stake can be redeemed by an user that can provide the issuer contract with a specified pre-image. Once redeemed, the credential because invalidated. Therefore, if a dishonest party were to give away their credential, the new holder of the credential could steal the original holder's stake. This requires issuers to set their stakes high enough such that no party would be willing to pay the price of the stake for the credential. A stake system could also lead to varying "tiers" of issuers where issuers that require higher stakes could be percieved as having higher credibility do the lower probability of users sharing said issuer's credential.

3.8.2 InterPlanetary File System (IPFS)

As designed, the ZKID service expects verifiers to store their issued-credential merkle trees using the distributed file system service, InterPlanetary File System (IPFS). By utilizing IPFS, the ZKID service promotes the distributed nature of its environment and further encourages issuers to take advantage of said distributed nature. All issuer contracts must supply a GetMerkleTreeAddress method that returns the IPFS address of their merkle root. Every issuer must also append the hashes of every issued credential to their merkle tree so that verifiers and clients can construct merkle tree paths and subsequently merkle tree membership proofs as nessesary.

3.9 dApp Framework and Credential-Enabled Contracts

Add some Introduction here

3.9.1 Credential Annotated ABI

The properly interface with the ZKID Service a dApp will have to provide an annotated ABI for its dependant contract(s). The required annotations describe the required credentials for a given method within the smart contract and will be used to propery construct public inputs and subsequently generate proofs for credentials. An annotated ABI function would appear as follow:

```
{"constant":false,
 "name":"Join",
 "inputs":[],
 "outputs":[],
 "payable":false,
 "type":"function","
  "credentials":[
    {"contract_salt":"0xdeadbeef",
    "high_bound":255,
    "low_bound":18,
    "k_bound":1,
    "description": "Confirms holder is over 18."
    }
  ]
}
```

Note that the description field is not nessesary for generation / verification of proofs, but is supplied soley for the use of the credential-acknowledgement framework described later.

3.9.2 ZKID web3 Framework

To a dApp's migration to credentail-enabled contracts as smooth as possible we have supplied a ZKID framework that interfaces with our prove-generation service and seemlessly collects and supplies verifcation arguments to credential-enabled methods. This allows dApp creators to integrate credential functionality without changing their already developed application code.

To utlize the framework a dApp simply replaces all existing contract calls with calls to their credential accepting counterparts. This can be accomplished very concisely by wrapping the credial-accepting method call with partially applied function. A full implementation of this exists in our example application and is depicted in [INSERT FIG NUMBER];

```
function OnJoinClick() {
  var credBlock = new CredentialBlock("Join", PartialAppJoin(), lottery_abi);
    credBlock.display();
}
```

```
function PartialAppJoin(/* Insert non-proof arguments here*/) {
  return function (proof_bytes) {
    contractInstance.methods["Join"](/* Non-Proof Arguments*/ ,proof_bytes).send();
  }
}
```

In the above example, "Join" allowed a users to join a lottery if they have the required credentials. Originally, took a number of non-proof arguments. The credential enabled join now takes the non-proof arguments as well as proof arguments. This partial application hides the non-proof arguments from the ZKID CredentialBlock and allows the CredentialBlock to supply the proof arguments post proof-generation. The dApp's code has now migrated its call to "Join" to a credential-enabled call with minimal additions.

4 Benchmarks

5 Example Application

To demonstrate the facilities provided by the our system we developed a simple lottery dApp that requries participants to be both over the age of 18 and have American Citizenship to participate. Along with the dApp, we created a credential-acknowledgement framework that would allow applications to prompt users for the required credentials before any proofs are generated.

6 Future Work

7 Conclusion

References

[1] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. URL https://eprint.iacr.org/2013/507.

[2] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. URL http://arxiv.org/abs/1407.3561.

[3] Christina Garman, Matthew Green, and Ian Miers. Decentralized Anonymous Credentials. URL http://eprint.iacr.org/2013/622.

[4] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. URL https://eprint.iacr.org/2013/279.

[5] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 151:1–32.