ZKP has been used in quite some places and is expected to be used even more widely. However for authentication scenarios, a systematic way to protect the verifier from replay attack or malleability is still much needed. In this thread, I'd like to review existing ad hoc practices, and propose a more systematic way to address such issues. Feedbacks are much appreciated.

Problem Statement

Consider a reputation system

, where some high rank users from certain social networks could gain some privileges. To protect privacy, these users rely on zkp technology to prove themselves: they generate some proof data showing that they meet certain criteria without disclosing more data, and present the proof to the system.

proof data = zkp (data, data_certifier, criteria)

Upon receiving such a proof data, however, the system has no way to tell if the proof is generated by the presenter, or if he or she just replays somebody else's data.

This is a typical authentication use. In such cases, the proof system must provide knowledge soundness

, and the verifier must authenticate the prover. Authentication scenario is intrinsically interactive.

However when a proof system is turned into non-interactive, often by means of Fiat-Shamir heuristic, we must pinpoint the prover. Otherwise the proof could be replayed even if the underlying proof system do provide knowledge soundness.

An Ad Hoc Solution

In id3's solution, an Authorization Claim contains a public key, and the user shall has the corresponding private key. Therefore, when Peggy presents the AuthClaim (containing some zkp data), she must sign the data with her private key.

The privacy concern here is, that the public key in the AuthClaim might be collected and tracked by malicious verifiers to link users' activities in various places and to infer users' identity.

Proposed Systematic Solution

Since authentication is intrinsically interactive, we can bring back interaction to the proof. That's, Victor could challenge Peggy, and Peggy must respond by adding the challenge to the circuit computation.

Suppose there is a session between Peggy and Victor. Then the challenge could be derived by hashing relevant context data:

c := Hash(domain separator, //application specific string proof scheme, //Groth16, Plonk, etc. public parameters, //such as curve, generators, etc. application level session identifier)

If session identifier is missing, or insufficient to prevent replay attack, then Victor could generate a nonce.

To add the challenge c

to the circuit computation, the circuit must take c

as input. To ensure c

is indeed part of the computation (otherwise Mallory could replace c

in an existing proof), a simple trick is to do squares. Below example is adapted fromTornado Cash

signal input challenge; //… challengeSquare <== challenge * challenge;

Then Victor could follow same procedure to derive c

, and use it as part of the public input to verify the proof.

This also protects verifier from malleable proof (such as Groth16). And that's all.