anoncreds

Samuel Schlesinger

April 15, 2025

Definition 1. *An Anonymous Credit Scheme (ACS) consists of several probabilistic polynomial time algorithms, which compose several protocols.*

The algorithms are:

1. $(x, w) \leftarrow GenerateKeyPair$

   Run by the issuer before any Credit Tokens can be issued and spent, this algorithm generates a keypair for the Issuer.

   - *w: public key*
   - *x: private key*

2. $(p, req) \leftarrow RequestIssuance$

   Run by the client to generate a request for a Credit Token, this algorithm generates client secrets which will be used to construct the Credit Token, as well as a request for a Credit Token.

   - *p: client secrets*
   - *req: issuance request*

3. $resp \leftarrow Mint(x, req, n)$

   Run by the issuer in response to a request for a Credit Token, this algorithm verifies the issuance request and issues an issuance response, while remaining blind to the nullifier which is associated with this Credit Token. This algorithm fails if the issuance request is incorrect.

   - *x: the issuer's private key*
   - *req: issuance request*
   - *n: the number of credits which will be issued in the Credit Token constructed from this response*
   - *resp: a response to an issuance request*

4. $token \leftarrow IssueCredits(p, w, req, resp)$

   Run by the client upon receiving an issuance response from the issuer, this algorithm verifies the issuance response and forms the new Credit Token from the client secrets and issuance response.

- *p: client secrets*
- *w: issuer public key*
- *req: issuance request*
- *resp: issuance response*
- *token: a Credit Token*

5. $(p, sp) \leftarrow ProveSpend(token, charge)$

   Run by the client to generate a request to spend a Credit Token, this algorithm generates client secrets which will be used to construct the Credit Token, as well as a request for a Credit Token.

   - *token: the Credit Token we are requesting to spend*
   - *charge: the number of credits we're requesting to spend*
   - *p: client secrets*
   - *sp: proof of correct spend*

6. $k \leftarrow GetNullifier(sp)$

   Run by the issuer to extract the nullifier from the proof of correct spend. The nullifier is the random value associated with the underlying Credit Token which is revealed upon spending, preventing the double spending a Credit Token.

   - *sp: proof of correct spend*
   - *k: the nullifier*

7. $refund \leftarrow Refund(x, sp, c)$

   Run by the issuer to respond to a request to spend a Credit Token, this algorithm verifies the proof of correct spend of the given number of credits c and issues a refund for the remaining credits. This algorithm fails if the spend proof is incorrect.

   - *x: the issuer's private key*
   - *sp: proof of correct spend*
   - *c: the number of credits the client wishes to spend*
   - *refund: a refund*

8. $token \leftarrow RefundCredits(p, sp, refund, w)$

   Run by the client upon receiving a refund from the issuer, this algorithm verifies the refund and forms the new Credit Token from the client secrets and the refund.

   - *p: client secrets*

- *sp: proof of correct spend*
- *refund: a refund*
- *w: issuer's public key*
- *token: a Credit Token*

The protocols are run between the client and the server, initiated by the client:

1. *Issue*

   The issuer knows their private key x, their public key w, and the number of credits they wish to issue, n.

   The client knows the public key of the issuer, w.

   ```
   $(p, req) \leftarrow RequestIssuance$ // run by client
   // client sends req to issuer
   $resp \leftarrow Mint(x, req, n)$ // run by issuer
   // issuer sends resp to client
   $token \leftarrow IssueCredits(p, w, req, resp)$ // run by client
   return token
   ```

2. *Spend*

   The issuer knows their private key x, their public key w, and the number of credits c the client wishes to spend. They maintain a database db which contains all previously seen nullifiers k.

   The client knows their token, the issuer's public key w, and the number of credits c they wish to spend.

   ```
   $(p, sp) \leftarrow ProveSpend(token, charge)$ // run by the client
   // client sends sp, charge to the issuer
   $refund \leftarrow Refund(x, sp, c)$ // run by issuer
   $k \leftarrow GetNullifier(sp$
   if db.lookup$(k)$, fail
   db.insert$(k)$
   // issuer sends refund to client
   $token \leftarrow RefundCredits(p, w, req, resp)$ // run by client
   return token
   ```

Definition 2. *An Anonymous Credit System is* ***correct*** *if, for any efficient adversary A, the probability that $flag$ is set to 1 in the following experiment is negligible.*

1. *Generate keys* (w, x) *and provide* w *to* A.
2. A *can interact with the following oracles, with* i *being initialized as* 0 *and* db *being initialized emptily:*
   - $Issue(n)$ *runs the* $Issue$ *protocol between an honest issuer and client with input* n *and lets token be the resulting credits token. If* $Issue$ *fails, set* $flag = 1$. *Otherwise,* $token_i = token$ *and increment* i.
   - $Spend(j, c)$, *which can be called only once for each* j *and only when the balance* n *of* $token_j$ *satisfies* $c \leq n$. *Run the Spend protocol using db, c, and* $token_j$, *returning token. Send sp, c, k to A, and set* $flag = 1$ *if the protocol fails. Set* $token_i = token$ *and increment* i.

Definition 3. *Fiscal Soundness*
(High level) an issuer should be assured that the total number of credits spent is less than or equal to the total number of credits issued.

Definition 4. *Anonymity*
(High level) clients should be confident that their spends cannot be correlated to their issuances or previous spends with probability better than guessing. In particular, assuming there are multiple unspent credit tokens with greater than or equal to c credits, an issuer should not have any advantage over random chance in guessing which of these tokens were spent.