anoncreds

Samuel Schlesinger

April 24, 2025

1 Introduction

Many APIs expose metered access through credits, where you buy or otherwise receive a number of credits and use them to access the API. Often, a user of an API will identify themselves using an API key, a unique value which identifies the account. Then, when they use an endpoint which costs c credits, the database can check whether they have enough and subsequently dock them c credits. This approach is flexible for API providers and is useful as a budgeting tool for API users.

On the downside, this approach introduces an unnecessary tracking identifier: the API key itself. In this work, we attempt to design anonymous methods of tracking credits, and build an Anonymous Credit System (ACS).

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*

An Anonymous Credit System is ***fiscally sound*** *if, for any efficient adversary A, the probability that $flag$ is sent to 1 in the following experiment is negligible.*

1. *Generate keys (w, x) and provide w to A.*
2. *A is given access to an oracle which runs $Issue$ between A and issuer. Let m be a running sum of all total amounts ever issued. Return the credit token to A.*
3. *A is given access to an oracle which runs $(b, resp) \leftarrow Refund(proof, c)$ for an arbitrary proof and c provided by A. If tag has been reused or $b = 0$, return. Otherwise, add c to m', a running sum of all amounts ever successfully spent using this oracle. If $m' > m$, set $flag = 1$. resp, tag is returned to A.*

Definition 4. *Anonymity*

An Anonymous Credit System is ***anonymous*** *if the success probability, for any efficient adversary A, the probability that $flag$ is set to 1 in the following experiment will not be greater than $1/k$, where k is the number of unspent transactions at the end of the experiment.*

1. *Generate keys (w, x) and provide w, x to A. Set integer $i = 0$.*

2. *A is given access to an oracle $Issue(n)$ which runs the $Issue$ protocol between an honest client and A. Each time, $token_i$ is set to the result of $Issue(n)$, n_i is set to n, and i is incremented.*

3. *A is given access to an oracle $(b, token, tag) \leftarrow Spend(i, c)$ which runs the $Spend(token_i, c)$ protocol between an honest client and A. If $token_i$ has already been spent, or $c > n_i$, then $flag$ is set to 0 and the program halts. The tag and $proof$ are returned to A. $token_i$ is set to token, and i is incremented.*

4. *Whenever A chooses, they can choose to complete the experiment. To complete it, there must be at least two unspent transactions. A chooses a value c which is less than or equal to the remaining balance of all unspent tokens. Then, a random j is selected from the set of all remaining unspent tokens, and $Spend(token_j, c)$ is run between an honest client and A. A receives the token output from Spend, as well as the tag, and is given an opportunity to guess k. If $k = j$, then $flag$ is set to 1, otherwise $flag$ is sent to 0. The program halts.*

2 Engineering Considerations

Token Database

When operating an ACS, one must have a database of all the k values they've already seen. This is likely a much larger database than the one used in the API key approach, mapping users to their balances. One natural modification to the scheme is to rotate keys regularly, requesting all clients to get a new Credit Token with the new private key, without revealing their balances. This way, we can eventually throw away old k values. Doing this actually means that, at most, we'll have a k for the total number of spends which take place in each epoch and the previous epoch. In fact, this could potentially be a smaller amount of total storage than simply holding the balance. However, this has the pernicious effect of your tokens expiring if you don't exchange them. This means that you must active manage your coin balances, which is a pretty significant cost.

Token Distribution

Further, one must have a method of determining who gets Credit Tokens issued to them. This might be costly to maintain, and depending on your services, you might have to do a KYC type of flow at this stage.

Key Compromise

A serious consideration for real life deployments is emergency key rotation. If a private key is leaked, then all tokens cannot be trusted, and all holders of currency will be very naturally upset. For this reason, token users who hold a significant amount of credits might want to keep records of their issuance and spending, such that they can redeem their credits if a key is leaked. However, this violates their privacy and is a very bad situation for everyone. Because your Credit Tokens contain random values you've never revealed, you will at least not have your credits stolen from you if you keep a record for this case.

References

[1] Dan Boneh and Victor Shoup (2023) A Graduate Course in Cryptography, Version 0.6

[2] Stefano Tessaro and Chenzhi Zhu (2023) Revisiting BBS Signatures