Libsnark Tutorial: Fundamental Relations and Gadgets for FinTech

Chan Nam Ngo
channam.ngo@unitn.it
University of Trento, Trento, Italy

November 21, 2018

1 Common Scenario

Let us consider the following scenario. A FinTech system often comprises N players. Each player P_i owns a secret inventory I_i. Suppose the secret inventory $I_i = (c_i, a_i)$ where c_i is the current cash balance and a_i is the current goods amount of P_i. As a simple example allow a player P_i post an offer to trade (buy or sell) a goods amount v at price p. For simplicity the system will halt until another player P_j accepts the offer and make the transaction. Security requirements of such system is simple: (1) the transactions are correct, (2) I_i and I_j stay secret (the traded amount v can be public) and (3) players P_i and P_j stay anonymous.

2 Crypto Implementation

Let us start slowly with an implementation that ignores player anonymity for now assuming that there is a commitment scheme $[x] = \mathsf{com}(x)$ (the corresponding randomness $r(x)$ is omitted for the sake of description, we will mention the randomness when necessary).

2.1 Initialization

A simple solution would be for all players to initially commit to his/her own secret inventory.

For each player P_i in $\mathbb{P}$

1. Commits to $[c_i] = \mathsf{com}(c_i)$
2. Commits to $[a_i] = \mathsf{com}(a_i)$

After this all players would store a list of tuples $\{([c_i], [a_i])\}_{i=1}^{N}$.

2.2 Making an offer

To make an offer of amount v at price p. Let us agree on the convention that to buy $v > 0$ and to sell $v < 0$. A player P_i making an offer can do the following.

Player P_i now owns a tuple $([c_i], [a_i])$ proceeds as follows.

1. Locally updates $c'_i = c_i - v \cdot p$
2. Locally updates $a'_i = a_i + v$
3. Commits to $[c'_i] = \mathsf{com}(c'_i)$
4. Commits to $[a'_i] = \mathsf{com}(a'_i)$

2.3 Accepting an offer

A player P_j accepting an offer can do the following.

Player P_j now owns a tuple $([c_j], [a_j])$ proceeds as follows.

1. Locally updates $c'_j = c_j + v \cdot p$
2. Locally updates $a'_j = a_j - v$
3. Commits to $[c'_j] = \mathsf{com}(c'_j)$
4. Commits to $[a'_j] = \mathsf{com}(a'_j)$

2.4 Maintaining Integrity

Zero-Knowledge Proof can be used for maintaining integrity in such systems. Randomnesses for commitments are omitted for the sake of description.

Table 1: Maintaining Interity with Relations

Purpose	**Statement**	**Witness**	**Conditions**
Can make a buy offer	$[c_i], v, p$	c_i	$c_i \geq v \cdot p$
Can make a sell offer	$[a_i], v$	a_i	$a_i \geq v$
P_i updates correctly	$[c_i], [a_i], [c'_i], [a'_i], v, p$	c_i, a_i, c'_i, a'_i	$c'_i = c_i - v \cdot p$ and $a'_i = a_i + v$
Can accept a sell offer	$[c_j], v, p$	c_j	$c_j \geq v \cdot p$
Can accept a buy offer	$[a_j], v$	a_j	$a_j \geq v$
P_j updates correctly	$[c_j], [a_j], [c'_j], [a'_j], v, p$	c_j, a_j, c'_j, a'_j	$c'_j = c_j + v \cdot p$ and $a'_j = a_j - v$

Table 2: Merkle Tree's supported operations

Definition	**Description**
$\rho = \mathsf{Add}(T, [x])$	Adds a new leaf (the hash of $[x]$) to the tree and generates a new root *root*.
$path = path(T, [x])$	Returns the authentication path from $[x]$ to ρ.
$\{0,1\} \leftarrow \mathsf{Auth}(\rho, path, [x])$	Authenticates $[x]$ in T w.r.t. the authentication path *path* (where output 1 means the authentication succeeded).

2.5 Maintaining Anonymity

The overall state can be captured by a token τ_i that is a commitment of all values in the inventory (with fresh randomness $r(\tau)$); initially, such value is only known to the player itself. Each player keeps the token secret and broadcasts a commitment to it in order to commit to a new inventory; such an inventory is considered as *unspent*. At a later point, a player can reveal the token and retrieve a previously committed inventory, in which case we say the inventory is *spent*, as the corresponding token cannot be used anymore.

The anonymity of the inventory is guaranteed by relying on Merkle trees in conjunction with the zero-knowledge proofs. Throughout the execution of the protocol, a Merkle tree T based on a collision-resistant hash function (the same one that), where the leafs are commitments, is maintained and updated. ρ denotes the root of the tree, and *path* denotes the authentication path from a leaf $[v]$ to the root ρ. The number of leafs is not fixed a-priori, one can efficiently update a Merkle tree by appending a new leaf, resulting in a new tree; this can be done in time/space proportional to tree depth. Table 2 summarizes the supported ops Add, *path* and Auth of a Merkle tree T: adding a node to the tree (Add), returning an authentication path from the root to a value (*path*), and a verification function that return 1 if the a path is authentic and 0 otherwise (Auth). (see Table 2.)

Preserving Players' Anonymity. The commitment (the retrieval) of trader inventories to the Merkle Tree T is obtained by running a sub-protocols Π_{com} (resp. Π_{ret}) as follows:

- In Π_{com}, the player broadcasts a commitment to the token corresponding to the current inventory $\tau_i = \mathsf{com}(c_i || a_i)$. Thus, the player proves that the token is correctly constructed (similarly to proving consistency of a commitment) and appended into the Merkle tree (with operation Add), before broadcasting the new root of the tree. The other players will check that the new root is correctly computed before accepting it.

- In Π_{ret}, a player can retrieve a previously committed inventory, and *spend* it, by revealing the secret *unspent* token τ_i and proving that the newly committed values are consistent updates of the values previously committed; this is done by proving: (1) $[\tau_i]$ is a leaf of the current tree and the new values are correctly updated (see the maintaining integrity part). Every time an inventory is retrieved, two sets of commitments are generated corresponding to the inventory values before and after the update. The token τ_i is now marked as *spent* and will not be usable for retrieving any inventory.