Multiple Secret Leaders

Authors here

July 2, 2023

1 Some Notation

1. We will have n parties
2. We will have k leaders elected
3. We will have a "bid" published by a user $i \in [n]$ be denoted as b_i
4. We will denote a commitment as $\texttt{comm}_i$
5. We will denote some generic CRHF as h
6. We will say $\texttt{Enc}_{\text{TFHE}}$ and $\texttt{Dec}_{\text{TFHE}}$ for TFHE encoding and decoding respectively

2 Sketching Stuff Out

Say we want to elect k leaders (maybe with or without repetition) secretly out of n parties, we can run SSLE k times but that'd be painful. Let's not do that.

2.1 Simple sorting

A simple approach with runtime (assuming $k \leq n$) $O(k + n \log n)$ is to use sorting. At a high level, the idea is to somehow randomly sort the n parties then elect the top k parties in the sort. To do this we can use TFHE in a very similar way to the SSLE paper.

Algorithm 1 Simple Sorting Evaluation

Input: List of bids, $b_1, b_2, ..., b_n$
  $\alpha \leftarrow \texttt{ROM}(b_1, b_2, ...)$
  for $i \in [n]$ **do**
    $b'_i \leftarrow (\texttt{Enc}_{\text{TFHE}}(r_i) - \alpha, \texttt{Enc}_{\text{TFHE}}(\texttt{comm}_i))$
  $\texttt{sorted} = [b'_{j_1}, b'_{j_2}, ...] \leftarrow \texttt{Sort}_{\text{TFHE}}([b'_1, b'_2, ...])$
  return first k elements of $\texttt{sorted}$

1. **Setup**: Same exact setup with TFHE in SSLE. Setup TFHE and distribute shares. Each party also has some secret s_i and commitment $\texttt{comm}_i = \texttt{comm}(s_i)$.
2. **Publish Bid**: Each party samples $r_i \leftarrow U$ and publish $b_i = (\texttt{Enc}_{\text{TFHE}}(r_i), \texttt{Enc}_{\text{TFHE}}(\texttt{comm}_i))$.

3. **Evaluation**: This is done by one party whom may even be an external party. See algorithm 1 for details.

4. **Decoding**: At least t parties publish their threshold decryptions of the return from the evaluation.

5. **Proving**: For a party i to prove that they are the selected leader for the jth slot, they prove that they know the secret to produce $\texttt{comm}_i$ where $\texttt{comm}_i$ is the decrypted commitment for the ith slot.

2.2 Distributed/ Streaming Simple Sorting

Assume that n, k are a power of 2. We will keep a lot of the same from sorting but now look at selection as a sort of tournament. As a note, security is slightly reduced here as a participating party can know that they did not win before the final outcome of the election. I do not know if this matters.

For completeness, we will write out each step again

Algorithm 2 `PlayGame`. Homomorphically plays a game to decide which incoming bid "wins"

Input: Point $\alpha \in \mathbb{F}_q$,
two bids, $b_1 = (\texttt{Enc}_{\text{TFHE}}(r_1), \texttt{Enc}_{\text{TFHE}}(\texttt{comm}_1)), (\texttt{Enc}_{\text{TFHE}}(r_2), \texttt{Enc}_{\text{TFHE}}(\texttt{comm}_2))$
Output: Winning bid. The winning bid should be indistinguishable from a random bid to the non-players
$\texttt{closer} \leftarrow \texttt{Enc}_{\text{TFHE}}(r_1 - \alpha > r_2 - \alpha)$
return $\texttt{closer} \cdot b_1 + (1 - \texttt{closer}) \cdot b_2$

Algorithm 3 `EvalStream`. Evaluates a stream of bids as they come in.

Input: L, set of $(\texttt{bid}, \texttt{level})$ pairs. New bid and level, (b, ℓ). Also, stop level ℓ_{stop}
Output: New list L with remaining evaluations
if there is some for some $b', (b', \ell) \in L$ **then**
$\alpha \leftarrow \texttt{ROM}(b, b', \ell)$
$\bar{b} \leftarrow \texttt{PlayGame}(b, b')$
$L' \leftarrow L \setminus \{ (b', \ell) \}$
if **then**$\ell_{\text{stop}} = \ell - 1$
return $L \bigcup \{ (\bar{b}, \ell - 1) \}$
else
return $\texttt{EvalStream}(L', (\bar{b}, \ell - 1))$
else
$L' \leftarrow L \bigcup \{ (b, \ell) \}$
return L'

1. **Setup**: Same exact setup with TFHE in SSLE. Setup TFHE and distribute shares. Each party also has some secret s_i and commitment $\texttt{comm}_i = \texttt{comm}(s_i)$.

2. **Publish Bid**: Each party samples $r_i \leftarrow U$ and publish $b_i = (\texttt{Enc}_{\text{TFHE}}(r_i), \texttt{Enc}_{\text{TFHE}}(\texttt{comm}_i))$ each tagged with level $\log_2 n + 1$.

3. **Evaluation**: This step is different than that in the simple sorting version (section 2.1). Now, we can evaluate in a streaming fashion and even distribute evaluation (though we'll not go into details about the distributed implementation). See algorithm 3 for the algorithm. The idea is that we keep a list of bids and their levels; when a new bid comes in, we greedily remove any bids from the list which we can. We can run algorithm 3 until there is only one bid left in the list or until all bids up to a level have been processed. So, say that we have $k = 4$, we can run the algorithm until we are two levels away (in a tournament tree) from the final level. More formally, when we have processed all bids up to level $\ell_{\text{stop}} = \log_2 k + 2$ (and we only have bids at level $\log_2 k + 1$ in the list) we return the list and stop processing. When there are k elements in this list, all at level $\log_2 k + 1$, we are done.

   LS: I may have an off by one error here though I do not think so.

4. **Decoding**: At least t parties publish their threshold decryptions of the return from the evaluation

5. **Proving**: For a party i to prove that they are the selected leader for the jth slot, they prove that they know the secret to produce $\texttt{comm}_i$ where $\texttt{comm}_i$ is the decrypted commitment for the ith slot

3 Data Independent Streaming Sampler

Our construction makes heavy use of a data independent streaming sampler which we will define below. The streaming sampler relies on a data independent priority queue and in turn makes use of LS: Cite Shi protocol for an oblivious priority queue.

Encrypted Data Independent Queue

In this work, we will data independent queues as studied in [Tof11, MZ14, MDPB23]. Data independent data structures are unique as their control flow and memory access do not depend on input data ([MZ14]).

They are especially useful as they allow for efficient computation within FHE as control flow is not dependent on underlying ciphertexts. We use a data independent queue as outlined in [MDPB23] which allows for

- `PQ.Insert`: Inserts a tag and value, (p, x) into `PQ` according to the tag's priority.
- `PQ.ExtractFront`: Removes and returns the (p, y) with highest tag priority.
- `PQ.Front`: Returns the (p, y) with highest tag priority without removing the element.

Moreover, we note that the order is stable. I.e. the first inserted among equal tagged elements has a higher priority.

We will use the data independent queue within public key threshold FHE such that we now have an encrypted priority queue, `EncPQ` with functionality

- `EncPQ.Insert`: Inserts a tag and value, $\texttt{Enc}_{\text{TFHE}}(p, x)$ into `PQ` according to the tag's priority.
- `EncPQ.ExtractFront`: Removes and returns the $\texttt{Enc}_{\text{TFHE}}(p, y)$ with highest tag priority.
- `EncPQ.Front`: Returns the $\texttt{Enc}_{\text{TFHE}}(p, y)$ with highest tag priority without removing the element.

We further require that for $\texttt{Enc}_{\text{TFHE}}(p, y) \leftarrow \texttt{EncPQ.ExtractFront}$ (and $\texttt{EncPQ.Front}$) and $x_1, ..., x_n$ drawn from a random distribution,

$$\Big|\mathbf{Pr}[\mathcal{A}(x_1, \dots x_n, \texttt{Enc}_{\text{TFHE}}(p_1, x_1), ..., \texttt{Enc}_{\text{TFHE}}(p_n, x_n), \texttt{Enc}_{\text{TFHE}}(p, y), i) = 1] - \mathbf{Pr}[\texttt{Dec}_{\text{TFHE}}(\texttt{Enc}_{\text{TFHE}}(p, y)) = (p_i, x_i)]\Big| \leq \texttt{negl}(\lambda) \tag{1}$$

where (p_i, x_i) is the ith submitted message to the priority queue.

Lemma 3.1. *Assuming the indistinguishablity of TFHE cipher texts, $\texttt{Enc}_{\text{TFHE}}(m_1), \texttt{Enc}_{\text{TFHE}}(m_2)$ where $m_1 \neq m_2$ and are known to the adversary, eq. (1) holds.*

Proof. Assume towards contradiction that there exists an adversary $\mathcal{A}$ which can distinguish between a random message to the priority queue and the output of $\texttt{EncPQ.ExtractFront}$ (resp. $\texttt{EncPQ.Front}$). Then, we can construct an adversary $\mathcal{A}'$ which can distinguish TFHE cipher text, $\texttt{ct}_1 = \texttt{Enc}_{\text{TFHE}}(m_1), \texttt{ct}_2 = \texttt{Enc}_{\text{TFHE}}(m_2)$, as follows:

1. $\mathcal{A}'$ simulates an encrypted priority queue, $\texttt{EncPQ}$.
2. $\mathcal{A}'$ inserts $x' = \texttt{Enc}_{\text{TFHE}}(\texttt{ct}_1, r_1)$ into $\texttt{EncPQ}$ and then $x = \texttt{Enc}_{\text{TFHE}}(\texttt{ct}_2, r_2)$ into $\texttt{EncPQ}$ where r_1, r_2 are random.
3. The adversary then calls $\texttt{Enc}_{\text{TFHE}}(p, y) \leftarrow \texttt{EncPQ.ExtractFront}$ (resp. $\texttt{EncPQ.Front}$) and uses $\mathcal{A}$ to distinguish between x and x'.
4. If $\texttt{Enc}_{\text{TFHE}}(p, y) = x'$ output m_1, otherwise output m_2.

Clearly, if the adversary can distinguish the priority queue outcomes, then using $\mathcal{A}$, he can with non-negligible probability distinguish encryptions of m_1 and m_2 as, WLOG, $m_1 > m_2$ remains constant. An identical approach can then be taken to distinguish between encryptions of $m_1, ..., m_n$ except where the adversary inserts $m_1, ..., m_n$ into the queue and dequeues all items. □

Encrypted Streaming Sampler

A streaming sampler should output a random subset of size k from a stream of n elements with a random ordering. We can build a streaming sampler from a priority queue as follows:

Algorithm 4 Encrypted Gated Insert

Input: $\texttt{Enc}_{\text{TFHE}}(\text{coin}), \texttt{EncPQ}, \texttt{ct}_a$
Output: EncPQ'
$\texttt{ct}_b \leftarrow \texttt{EncPQ.ExtractFront}(\texttt{PQ})$
$\texttt{ct}_{\text{new}} = \texttt{Enc}_{\text{TFHE}}(\text{coin}) \cdot \texttt{ct}_a + (\texttt{Enc}_{\text{TFHE}}(\text{coin}) - 1) \cdot \texttt{ct}_b$
$\texttt{EncPQ.Insert}(ct_{\text{new}})$
return EncPQ

We will use a slightly modified version of a streaming sampler where randomness is submitted alongside the message. We can then implement the algorithm as follows

- $\texttt{Insert}(\texttt{ct}_i, \texttt{Enc}_{\text{TFHE}}(e_i))$ where $\texttt{ct}_i$ is a cipher text and $\texttt{Enc}_{\text{TFHE}}(e_i)$ is encrypted randomness
  - If the item is the m-th item and $m \leq k$, insert the item into the queue along with e_i as its tag via $\texttt{EncPQ.Insert}(\texttt{Enc}_{\text{TFHE}}(e_i), \texttt{ct})$.

– If $m > k$, we will evaluate the PRF (in FHE) to get an encoded random coin which is 1 with probability $1/m$ and 0 otherwise. Then, run algorithm 4 which will
  * replace the smallest labeled item in the queue with the new item if the coin is 1.
  * not replace the smallest item in the queue if the coin is 0.

- At the end of the stream, we output all the items in the priority queue, $\mathtt{ct}_{a_1} ... \mathtt{ct}_{a_k}$ in order by repeatably calling `EncPQ.ExtractFront`.

We can formalize soundness as a game as follows, for all $\ell \in [k], i \in [n]$,

$$\Big|\mathbf{Pr}[\mathcal{A}(\mathtt{ct}_1, \ldots, \mathtt{ct}_n, e_1, \ldots, e_n, \mathtt{ct}_{a_1}, \ldots, \mathtt{ct}_{a_k}, \ell, i) = 1] - \mathbf{Pr}[\mathtt{Dec}_{\mathrm{TFHE}}(\mathtt{ct}_{a_\ell}) = \mathtt{Dec}_{\mathrm{TFHE}}(\mathtt{ct}_i)]\Big| \leq \mathtt{negl}(\lambda) \tag{2}$$

In words, the adversary should not be able to guess with any advantage which of the ℓ-th outputted cipher text is associated with which inputted cipher text.

Lemma 3.2. *Our construction of a streaming sampler is sound.*

Proof. TODO: this should not be too bad □

4 MSLE Protocol

We use a similar notion of ideal functionality for a multi-secret leader election from the ideal functionality of single secret leader election of LS: CITE.

The MSLE functionality $\mathcal{F}_{\mathrm{MSLE}}$: Initialize $E, R \leftarrow \emptyset, \leftarrow 0$. Fix some $k \in \mathbb{N}$ to denote the number of rounds. Upon receiving,

- `register` from party P_i, set $R \leftarrow R \cup \{(i, n\}$, broadcast $(\mathtt{register}, i)$ to all parties and set $n \leftarrow n + 1$
- `elect`(eid) k leaders from all honest parties. If $|R| \geq k$ and eid was not requested before, randomly sample $W^{eid} \subseteq R$ where $|W^{eid}| = k$. Then, assign a random ordering to W^{eid} to get ordered set E^{eid}. Next, send $(\mathtt{outcome}, eid, a)$ to P_j for all $E^{eid}_a = (j, \cdot)$ and $(\mathtt{outcome}, eid, \perp)$ to P_i if $(i, \cdot) \notin E^{eid}$. Store $E \leftarrow E \bigcup \{E^{eid}\}$.
- `reveal`(eid, ℓ) from P_i: for $E_{eid} \in E$, if $i = E^{eid}_a$, broadcast $(\mathtt{result}, eid, \ell, i)$. Otherwise, broadcast $(\mathtt{rejected}, eid, \ell, i)$.

Figure 1: Description of the MSLE functionality, heavily based on the description of SSLE in LS: CITE Dario and CFG21

.

We wil realize MSLE via the following protocol:

4.1 Correctness

4.2 Simulation Security

To show simulation security, we will prove that, given a party's input and output in the ideal model, a simulator can simulate the distribution of the view in the protocol for the party. Note that because the protocol is deterministic, it suffices to prove simulation for only the party's output.

More formally we will show that for party i,

$$\text{Sim}_i\left(s_i, r_i, \mathcal{F}_{\text{MSLE}}.\texttt{register}_i\right) \overset{\text{c}}{\approx} \texttt{view}_{\texttt{register}_i}((s_0, r_0), ..., (s_n, r_n)), \tag{3}$$

$$\text{Sim}_i\left(s_i, b_i, r_i, \mathcal{F}_{\text{MSLE}}.\texttt{elect}(eid)_i\right) \overset{\text{c}}{\approx} \texttt{view}_{\texttt{elect}(eid)_i}((s_0, b_0), ..., (s_n, b_n)), \tag{4}$$

and

$$\text{Sim}_i\left(b_{a,1}, \ldots, b_{a,k}, s_i, r_i, \mathcal{F}_{\text{MSLE}}.\texttt{reveal}(eid, \ell)_i\right) \overset{\text{c}}{\approx} \texttt{view}_{\texttt{reveal}(eid,\ell)_i}(), \tag{5}$$

Lemma 4.1. *We will first show that eq. (3) is simulation secure.*

Proof. □

Lemma 4.2. *We will now show that eq. (4) is simulation secure.*

Proof. The view of each party i for $\texttt{elect}(eid)$ can be expressed as

$$(r_i, s_i, \texttt{Enc}_{\text{TFHE}}(b_1), \ldots \texttt{Enc}_{\text{TFHE}}(b_n), \texttt{Enc}_{\text{TFHE}}(r'_1), \ldots, \texttt{Enc}_{\text{TFHE}}(r'_n),$$
$$\texttt{Enc}_{\text{TFHE}}(b_{a_1}), \ldots, \texttt{Enc}_{\text{TFHE}}(b_{a_k}), \sigma_1, \ldots \sigma_t, b_1, \ldots, b_n)$$

where r'_i is the randomness from the streaming sampler and $\sigma_1, \ldots \sigma_t$ are the decryption shares. Then, we have the simulator proceed in the following manner:

1. The simulator sets a random, local tape
2. The simulator samples some randomness r and computes $\texttt{Enc}_{\text{TFHE}}(e_1), \ldots, \texttt{Enc}_{\text{TFHE}}(e_n)$
3. The simulator creates an encrypted priority queue $\texttt{EncPQ}$ and simulates the encrypted streaming sampler for all inputs $\texttt{Enc}_{\text{TFHE}}(c_j)$ for $j \in [i]$.
4. The simulator runs the encrypted streaming sampler to get the outputs $\texttt{Enc}_{\text{TFHE}}(c_{a_1}), ...\texttt{Enc}_{\text{TFHE}}(c_{a_k})$.
5. The simulator then chooses a random, ordered subset $S \subseteq [n]$ where $|S| = k$. If there is some w such that $S_w = i$, then set $y = w$. Otherwise, set $y = \perp$.
6. The simulator then sets the decryptions of $\texttt{Enc}_{\text{TFHE}}(c_{a_\ell})$ for $\ell \neq y$ to b_ℓ where b_ℓ is a commitment to a random value. The simulator also creates shares σ_j such that the shares decrypt to the b_ℓ.
7. The simulator then outputs y is P_i won an election

We now use a sequence of hybrids to show that the view of the real protocol is indistinguishable from that of the simulated one

□

Lemma 4.3. *We will now show that eq. (5) is simulation secure.*

Proof. □

References

[MDPB23] Sahar Mazloom, Benjamin E Diamond, Antigoni Polychroniadou, and Tucker Balch. An efficient data-independent priority queue and its application to dark pools. *Proceedings on Privacy Enhancing Technologies*, 2:5–22, 2023. 3

[MZ14] John C Mitchell and Joe Zimmerman. Data-oblivious data structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014. 3

[Tof11] Tomas Toft. Secure data structures based on multi-party computation. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 291–292, 2011. 3

The MSLE Protocol π_{MSLE}: Initialize $n = 0$. Fix some $k \in \mathbb{N}$ to denote the number of rounds. Also, sample ; this will be the key FHE to the PRF.

- `initialize`
  - Set $n = 0$
  - Sample some secret $s_{\texttt{TFHE}}$ and publicly publicly publish $\texttt{Enc}_{\text{TFHE}}(s_{\texttt{TFHE}})$. This will be the key to the PRF
  - Initialize an empty set C, containing all party's registered commitments.
- `register`$(\texttt{Enc}_{\text{TFHE}}(c_i))$ from P_i where $c_i = \texttt{comm}(s_i, r_i)$ for secret s_i and secret randomness r_i,
  - If party P does not already have a TFHE share, create a TFHE share for P_i.
  - Add $\texttt{Enc}_{\text{TFHE}}(c_i)$ to C.
  - $n \leftarrow n + 1$
- `elect`(eid) k
  - An encrypted streaming sampler will be publicly initialized
  - Public randomness, r will be sampled to and used in the encrypted PRF to get random $\texttt{Enc}_{\text{TFHE}}(e_1), ..., \texttt{Enc}_{\text{TFHE}}(e_n) = \texttt{Enc}_{\text{TFHE}}(\texttt{PRF}(r, s_{\texttt{TFHE}}))$
  - All $c_i \in C$ will be fed to the encrypted streaming sampler along with randomness $\texttt{Enc}_{\text{TFHE}}(e_i)$
  - The encrypted streaming sampler will output a list of k messages: $\texttt{Enc}_{\text{TFHE}}(c_{a_1}), \texttt{Enc}_{\text{TFHE}}(c_{a_2}), ..., \texttt{Enc}_{\text{TFHE}}(c_{a_k})$.
  - Then, at least t parties will submit decryption shares to get $c_{a_1}, ..., c_{a_k}$.
  - Each party will then check if they won an election by seeing if their commitment is in the list of decrypted messages.
- `reveal`$(eid, \ell, \texttt{Enc}_{\text{TFHE}}(c_i'))$ from P_i
  - P_i submits a zero knowledge proof that they know the opening to c_{a_ℓ}. If this proof verifies, remove $\texttt{Enc}_{\text{TFHE}}(c_i)$ from C and add $\texttt{Enc}_{\text{TFHE}}(c_i')$. Then send out $(\texttt{result}, eid, \ell, i)$ to all parties. Otherwise, send out $(\texttt{rejected}, eid, \ell, i)$ to all parties.

Figure 2: Description of the MSLE protocol