Multiple Secret Leaders

Authors here

August 22, 2023

1 Preliminaries

1.1 Threshold FHE

[JRS17, BGG+18] defines threshold FHE encryption. For the sake of completeness, we will define it here.

Definition 1.1 (TFHE [JRS17])**.** Let $P = \{ P_1, ..., P_N \}$ be a set of N parties and $\mathbb{S}$ be a class of access structures on P. A TFHE scheme for $\mathbb{S}$ is a tuple of PPT algorithms

$$(\texttt{TFHE.Setup}, \texttt{TFHE.Encrypt}, \texttt{TFHE.Eval}, \texttt{TFHE.PartDec}, \texttt{TFHE.FinDec})$$

such that the following specifications are met

- $(pk, sk_1, ..., sk_N) \leftarrow \texttt{TFHE.Setup}(1^\lambda, 1^s, \mathbb{A})$: Takes as input a security parameter λ, a depth bound on the circuit, and a structure $\mathbb{A} \in \mathbb{S}$. Outputs a public key pk and a secret key sk_i for each party P_i.
- $\texttt{ct} \leftarrow \texttt{TFHE.Encrypt}(pk, \mu)$: Takes as input a public key and a message $\mu \in \{ 0, 1 \}$ and outputs a ciphertext $\texttt{ct}$.
- $\hat{\texttt{ct}} \leftarrow \texttt{TFHE.Eval}(C, \texttt{ct}_1, ..., \texttt{ct}_k)$: Takes as input a circuit C of depth at most d and k ciphertexts $\texttt{ct}_1, ..., \texttt{ct}_k$. Outputs a ciphertext $\hat{\texttt{ct}} = C(\texttt{ct}_1, ..., \texttt{ct}_k)$.
- $p_i \leftarrow \texttt{TFHE.PartDec}(\texttt{ct}, sk_i)$: Takes as input a ciphertext $\texttt{ct}$ and a secret key sk_i and outputs a partial decryption p_i.
- $\hat{\mu} \leftarrow \texttt{TFHE.FinDec}(B)$: Takes as input a set $B = \{ p_i \}_{i \in S}$ for some $S \subseteq [N]$ and deterministically outputs a message $\hat{\mu} \in \{ 0, 1, \perp \}$.

Further, we remember the definitions of evaluation correctness, semantic security, and simulation security as outlined in, [BEHG20].

Definition 1.2 (Evaluation Correctness [JRS17])**.** . We have that $\texttt{TFHE}$ scheme is correct if for all λ, depth bounds d, access structure $\mathbb{A}$, circuit $C : \{ 0, 1 \}^k \to \{ 0, 1 \}$ of depth at most d, $S \in \mathbb{A}$, and $\mu_i \in \{ 0, 1 \}$, we have the following. For $(pk, sk_1, ..., sk_N) \leftarrow \texttt{TFHE.Setup}(1^\lambda, 1^d, \mathbb{A})$, $\texttt{ct}_i \leftarrow \texttt{TFHE.Encrypt}(pk, \mu_i)$ for $i \in [k]$, $\hat{\texttt{ct}} \leftarrow \texttt{TFHE.Eval}(pk, C, \texttt{ct}_1, ..., \texttt{ct}_k)$,

$$\mathbf{Pr}\left[\texttt{TFHE.FinDec}(pk, \{ \texttt{TFHE.PartDec}(pk, \hat{\texttt{ct}}, sk_i) \}_{i \in S}) = C(\mu_1, ..., \mu_k)\right] = 1 - \texttt{negl}(\lambda).$$

Definition 1.3 (Semantic Security [JRS17])**.** We have that a $\texttt{TFHE}$ scheme satisfies semantic security for for all λ, and depth bound d if the following holds. There is a stateful PPT algorithm $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ such that for any PPT adversary $\mathcal{A}$, the following experiment outputs 1 with negligible probability in λ:

1. On input 1^λ and depth 1^d, the adversary outputs $\mathbb{A} \in \mathbb{S}$
2. The challenger runs $(pk, sk_1, ..., sk_N) \leftarrow \texttt{TFHE.Setup}(1^\lambda, 1^d, \mathbb{A})$ and provides pk to $\mathcal{A}$.
3. $\mathcal{A}$ outputs a set $S \subseteq \{ P_1, ..., P_N \}$ such that $S \notin \mathbb{A}$.
4. The challenger provides $\{ sk_i \}_{i \in S}$ and $\texttt{TFHE.Encrypt}(pk, \mu)$ to $\mathcal{A}$ where $\mu \xleftarrow{\$} \{ 0, 1 \}$.

5. $\mathcal{A}$ outputs a guess μ'. The experiment outputs 1 if $\mu' = \mu$.

Definition 1.4 (Simulation Security [JRS17])**.** We say that a TFHE scheme is simulation secure if for all λ, depth bound d, and access structure $\mathbb{A}$ if there exists a stateful PPT simulator, $\mathcal{S}$, such that for any PPT adversary $\mathcal{A}$, we have that the experiments $\texttt{Expt}_{\mathcal{A},\text{Real}}(1^\lambda, 1^d)$ and $\texttt{Expt}_{\mathcal{A},\text{Sim}}(1^\lambda, 1^d)$ are statisically close as a function of λ. The experiments are defined as follows:

- $\texttt{Expt}_{\mathcal{A},\text{Real}}(1^\lambda, 1^d)$:
  1. On input the security parameter 1^λ and depth bound d, the adversary outputs $\mathbb{A} \in \mathbb{S}$.
  2. Run $\texttt{TFHE.Setup}(1^\lambda, 1^d, \mathbb{A})$ to obtain $(pk, sk_1, ..., sk_N)$. The adversary is given pk.
  3. The adversary outputs a set $S \subseteq \{ P_1, ..., P_N \}$ such that $S \notin \mathbb{A}$ together with plaintest messages $\mu_1, ..., \mu_k \in \{ 0, 1 \}$. The adversary is handed over $\{sk_i\}_{i\in S}$
  4. For each μ_i, the adversary is given $\texttt{TFHE.Encrypt}(pk, \mu_i) \to \texttt{ct}_i$.
  5. The adversary issues a polynomial number of queries, $(S_i \subseteq \{ P_1, ..., P_N \}, C_i)$. for circuites $C_i : \{ 0, 1 \}^k \to \{ 0, 1 \}$. After each query the adversary receuves for $l \in S_i$ the value
  $$\texttt{TFHE.PartDec}(\texttt{TFHE.Eval}(C_i, \texttt{ct}_1, ..., \texttt{ct}_k), sk_l) \to p_l$$
  6. $\mathcal{A}$ outputs $\texttt{out}$, the experiment's output.
- $\texttt{Expt}_{\mathcal{A},\text{Sim}}(1^\lambda, 1^d)$:
  1. On input the security parameter 1^λ and depth bound d, the adversary outputs $\mathbb{A} \in \mathbb{S}$.
  2. Run $\texttt{TFHE.Setup}(1^\lambda, 1^d, \mathbb{A})$ to obtain $(pk, sk_1, ..., sk_N)$. The adversary is given pk.
  3. $\mathcal{A}$ outputs a set $S^* \subseteq \{ P_1, ..., P_N \}$ such that $S \notin \mathbb{A}$ and plaintexts $\mu_1, ..., \mu_k \in \{ 0, 1 \}$. The simulator is given $pk, \mathbb{A}, S^*$ as input and outputs $\{ sk_i \}_{i\in S^*}$ and the state $\texttt{state}$. The adversary is given $\{sk_i\}_{i\in S^*}$
  4. For each μ_i, the adversay is given $\texttt{TFHE.Encrypt}(pk, \mu_i) \to \texttt{ct}_i$.
  5. $\mathcal{A}$ issues a polynomial number of queries of the form $(S_i \subseteq \{ P_1, ..., P_N \}, C_i)$
  6. for circuits $C_i : \{ 0, 1 \}^k \to \{ 0, 1 \}$. After each query, the simulator computes
  $$\texttt{Sim}_{\texttt{TFHE}}(C_i, \{ \texttt{ct}_l \}_{l=1}^k, C_i(\mu_1, ..., \mu_k), \texttt{state}) \to \{ p_l \}_{l\in S_i}$$
  and sends $\{ p_l \}_{l\in S_i}$ to the adversary.
  7. $\mathcal{A}$ outputs $\texttt{out}$, the experiment's output.

1.2 Non Interactive Zero-Knowledge

The following definition is taken almost verbatim from [CFG22]. A non-Interactive Zero-Knowledge (NZIK) proof system for relationship Rel is a tuple of PPT algorithms $(\texttt{NZIK.G}, \texttt{NZIK.P}, \texttt{NZIK.V})$ such that $\texttt{NZIK.G}$ generates a common reference string, $\texttt{NZIK.crs}$, $\texttt{NZIK.P}(\texttt{NZIK.crs}, x, w)$ given $(x, w) \in Rel$, outputs a proof π, and $\texttt{NZIK.V}(\texttt{NZIK.crs}, x, \pi)$ outputs 1 if $(x, w) \in Rel$ and 0 otherwise. A NZIK is correct is for every $\texttt{NZIK.crs}$ and all $(x, w) \in Rel$, we have that $\texttt{NZIK.V}(\texttt{NZIK.crs}, x, \texttt{NZIK.P}(\texttt{NZIK.crs}, x, w)) = 1$ holds with probability 1. We also require that our NZIKs satisfy the notions of weak simulation extractability [Sah99] and zero-knowledge [FLS90].

Weak simulation extractability guarentees the extractability of proofs produced by the adversary that are not equal to proofs previously observed. Thus, we make each proof "unique" by implicitly adding a session ID to the statement. For more of a commentary, see section 2.7 of [CFG22]. We will not detail how to handle these session IDs.

We recall the notion of simulation security from [Sah99]:

Definition 1.5 (NZIK simulation security)**.** We say that a proof system for relationship *Rel* is simulation secure if proof system $(\mathtt{NZIK.G}, \mathtt{NZIK.P}, \mathtt{NZIK.V})$ is a non-interactive proof system and $\mathrm{Sim}_1, \mathrm{Sim}_2$ are PPT algorithms such that for all PPT adversaries $\mathcal{A}_1, \mathcal{A}_2$ we have that $\left|\mathbf{Pr}[\mathtt{Expt}_{\mathcal{A},\mathrm{Real}}(\lambda) = 1] - \mathbf{Pr}[\mathtt{Expt}_{\mathcal{A},\mathrm{Sim}}(\lambda) = 1]\right|$ is negligible in λ. The experiments are defined as follows:

- $\mathtt{Expt}_{\mathcal{A},\mathrm{Real}}$:
  1. $\mathtt{NZIK.crs} \leftarrow \mathtt{NZIK.G}(1^\lambda)$
  2. $\mathcal{A}_1$ outputs $(x, w, \mathtt{state}_1)$
  3. $\pi \leftarrow \mathtt{NZIK.P}(\mathtt{NZIK.crs}, x, w)$
  4. return $\mathcal{A}_2(\pi, \mathtt{state}_1)$
- $\mathtt{Expt}_{\mathcal{A},\mathrm{Sim}}$:
  1. $\mathtt{NZIK.crs} \leftarrow \mathrm{Sim}_1(1^\lambda)$
  2. $\mathcal{A}_1$ outputs $(x, w, \mathtt{state})$
  3. $\pi \leftarrow \mathrm{Sim}_2(\mathtt{NZIK.crs}, x, w)$
  4. return $\mathcal{A}_2(\pi, \mathtt{state})$

We also have that a NZIK has ideal functionality $\mathcal{F}^{Rel}_{\mathtt{NZIK}}$ which is defined as follows:

The $\mathcal{F}^{Rel}_{\mathtt{NZIK}}$ functionality for zero-knowledge:
Upon receiving $(\mathtt{prove}, sid, x, w)$ from P_i, with sid being used for the first time, if $(x, w) \in Rel$, broadcast $(\mathtt{proof}, sid, i, \pi)$, otherwise broadcast $\perp$.

1.3 Data Independent Priority Queue

In this work, we will use 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]).

Definition 1.6 (Word RAM model [MZ14])**.** In the word RAM model, the RAM has a constant number of public and secret registers and can perform arbitrary operations on a constant number of registers in constant time.

Definition 1.7 (Data Independent Data Structure [MZ14])**.** In the word RAM model, a data independent data structure is a collection of algorithms where all the algorithms uses RAM such that the RAM can only set its control flow based on registers that are public.

Data independent queues are especially useful as they allow for efficient computation within MPC and FHE as control flow is not dependent on underlying ciphertexts data. 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.

1.4 Resevoir Sampling

Resevoir sampling is an online algorithm which allows for randomly selecting k elements from a stream of n elements while using $\tilde{O}(k)$ space. Algorithm R ([Vit85]) is a simple algorithm which relies on a priority queue with interface:

- $\texttt{Resevoir.Init}(k)$ initialize the resevoir sampling data structure $\mathcal{R}$ such that $\mathcal{R}.\texttt{PQ}$ is an empty priority queue.
- $\texttt{Resevoir.Insert}(\mathcal{R}, \mu_i, e_i, \texttt{coin}_i)$ where μ_i is i-th item, e_i is independentally sampled randomness, and $\texttt{coin}_i$ is a random coin with probability $1/m$ of equaling 1.
  - If $i \leq k$, insert the item into the queue along with e_i as its tag via $\mathcal{R}.\texttt{PQ.Insert}(e_i, \mu_i)$.
  - If $i > k$ and $\texttt{coin}_i = 1$, replace the smallest labeled item in the queue with the new item if the coin is 1.
  - If $i > k$ and $\texttt{coin}_i = 0$, do nothing.
  - Return $\mathcal{R}$
- $\mu_{a_1}, \mu_{a_2}, ..., \mu_{a_k} \leftarrow \texttt{Resevoir.Output}(\mathcal{R})$ where $a_1, ..., a_k$ are a uniformly random ordered susbset of $[n]$ if $e_i, \texttt{coin}_i$ are independentally sampled uniformly at random for all $i \in n$ where n is the number of `Resevoir.Insert` calls.
  - Call `PQ.ExtractFront` k times setting μ_{a_ℓ} to the ℓ-th call to `PQ.ExtractFront` where $\ell \in [k]$.

2 Data Independent Priority Queue

Here we introduce a data independent priority queue. We will assume that there are no items in the queue with equal priority. In practice, we can break ties by adding a unique identifier to each item.

Algorithm 1 MergeFill

Input: $\texttt{PQ.Head}, \mathcal{P}_{\texttt{count}}$ where $|\texttt{PQ.Head}| = \sqrt{n}$
$e_1, ..., e_\gamma \leftarrow \texttt{sort}(\texttt{PQ.Head}, \mathcal{P}_{\texttt{count}})$ where $\gamma = |\texttt{PQ.Head}| + |\mathcal{P}_{\texttt{count}}|$
$\texttt{PQ.Head}' \leftarrow \{e_1, ..., e_{\sqrt{n}}\}$
$\mathcal{P}'_{\texttt{count}} \leftarrow \{e_{\sqrt{n}+1}, ..., e_\gamma\}$
return $\texttt{PQ.Head}', \mathcal{P}'_{\texttt{count}}$

Algorithm 2 Fill

Input: $\mathcal{P}_{\texttt{count}}, \texttt{PQ.Stash}$
$e_1, ..., e_\gamma \leftarrow \mathcal{P}_{\texttt{count}} \bigcup \texttt{PQ.Stash}$ where $\gamma = |\mathcal{P}_{\texttt{count}}| + |\texttt{PQ.Stash}|$
$\mathcal{P}'_{\texttt{count}} \leftarrow \{e_1, ..., e_{\min(\sqrt{n}, \gamma)}\}$
$\texttt{PQ.Stash}' \leftarrow \{e_{\min(\gamma, \sqrt{n}+1)}, ..., e_\gamma\}$
return $\mathcal{P}'_{\texttt{count}}, \texttt{PQ.Stash}'$

2.1 Invariants

Let $i, j \in \mathbb{N}$ be the total number of insertions and extractions respectively. Note that if $i - j \leq \sqrt{n}$ and in the prior $i+j$ operations, $i-j$ never exceeded $\sqrt{n}$, then the priority queue is trivially correct because all elements remain in the head which is itself a priority queue. Thus, we will assume that at some point in the sequence of insertions and extractions, we had $i - j > \sqrt{n}$.

Before specifying our invariants, we will add some notation. Let $\mathcal{G}$ ($\mathcal{G}$ for "good") be the smallest set of elements in the head. More formally, $\mathcal{G}$ is the largest subset of $\texttt{PQ.Head}$ such that $\forall e \in \texttt{PQ.Head}, e \notin \mathcal{G}$, $a < e, \forall a \in \mathcal{G}$. Note that $|\mathcal{G}| \leq \sqrt{n}$. Further, let g (the "good" element) be the smallest element in $\texttt{PQ}$ such that $\forall a \in \mathcal{G}, g > a$ and $\forall e \in \texttt{PQ} \setminus \mathcal{G}, g \leq e$. Notice that g is not in $\texttt{PQ.Head}$.

We will show that the following invariants hold:

- The "good" element is not in the prior $\sqrt{n} - |\mathcal{G}|$ iterated over partitions: $g \notin \bigcup_{q \in [\texttt{count} - \sqrt{n} + |\mathcal{G}|, \texttt{count})} \mathcal{P}_q$

Proof. We proceed by joint induction on i, j where $i - j \leq n$. We will first prove the base case where $i - j = \sqrt{n} + 1$ and $i - j \leq \sqrt{n}$ for all prior operations. Note that the last operation had to be an insertion in-order for $i - j$ to increase. Thus, the head contains $\sqrt{n}$ elements, all of which are smaller than the element evicted by $\texttt{PQ.ExtractLargest}$ (line 11) in the insertion operation. Note that all partitions, $\mathcal{P}_\alpha$ for $\alpha \in [\sqrt{n}]$, were empty before this operation and thus the $\texttt{PQ.order}$ add the element p' evicted from the head to $\mathcal{P}_{\texttt{count}}$. Thus the good element, g ends up in the first partition. As $|\mathcal{G}|$ after the insert is $\sqrt{n}$, we have that the invariants hold trivially.

Now for the inductive case, assume that the invariants hold for insertion count i and extraction count j. We will show that they hold for $i + 1, j$ and $i, j + 1$.

Algorithm 3 Data Independent Priority Queue

function PQ.Init
  count $\leftarrow 0$
  PQ.Head $\leftarrow$ PQ.Init$_{\sqrt{n}}$
  $\mathcal{P}_1, \cdots, \mathcal{P}_{\sqrt{n}} \leftarrow \emptyset$
  PQ.Stash $\leftarrow \emptyset$
function PQ.Insert(p)
  if $|$PQ.Head$| < \sqrt{n}$ **then**
    PQ.Head $\leftarrow$ PQ.Insert(PQ.Head, p)
  else
    PQ.Head $\leftarrow$ PQ.Insert(PQ.Head, p)
    PQ.Head, $p' \leftarrow$ PQ.Head.ExtractLargest
    PQ.Stash $\leftarrow$ PQ.Stash $\bigcup\{p'\}$
  Call Order()
function PQ.ExtractFront
  PQ.Head, $p \leftarrow$ PQ.Head.Front
  $p' \leftarrow$ GetLargest(PQ.Stash)
  if $p > p'$ **then** Remove the front most element, p, from PQ.Head and set $r = p$.
  else Remove p' from the stash and set $r = p'$
  Call Order()
  return r
function PQ.Order
  $\mathcal{P}_{\texttt{count}}$, PQ.Stash $\leftarrow$ Fill($\mathcal{P}_{\texttt{count}}$, PQ.Stash)
  PQ.Head, $\mathcal{P}_{\texttt{count}} \leftarrow$ MergeFill(PQ.Head, $\mathcal{P}_{\texttt{count}}$)
  count $\leftarrow$ count $+ 1 \mod \sqrt{n}$

Case 1: $i+1, j$ Let p be the inserted element. Note that if $p \le a$ for some $a \in \mathcal{G}$, then the new "good" set, $\mathcal{G}'$ has size $|\mathcal{G}'| \leftarrow \min(\sqrt{n}, |\mathcal{G}|+1)$. If $p > a$ for all $a \in \mathcal{G}$, then the new "good" set, $\mathcal{G}'$ has size $|\mathcal{G}|$ or $|\mathcal{G}|+1$ depending on whether $\mathcal{P}_{\texttt{count}}$ contains g or not. In either case, we have that $|\mathcal{G}'| \ge |\mathcal{G}|$. If $|\mathcal{G}'| = \sqrt{n}$, then the invariants hold trivially.

Otherwise, we have that either $|\mathcal{P}_{\texttt{count}}|$ is 0 or non-empty. If $|\mathcal{P}_{\texttt{count}}| = 0$, then $g \notin \mathcal{P}'_{\texttt{count}}$ where $\mathcal{P}'_{\texttt{count}}$ is the updated $\mathcal{P}_{\texttt{count}}$ after running the insertion. In the case that $|\mathcal{P}_{\texttt{count}}| > 0$, note that because we call `Order`() at the end of insertion (line 13) and $|\mathcal{G}| < \sqrt{n}$, `PQ.Head` either has an empty slot or an element β such that $\beta > g$. Thus after calling `PQ.Order`, we have that g is moved into `PQ.Head`$'$ if $g \in \mathcal{P}_{\texttt{count}}$ We can then see that g is not in $\mathcal{P}'_{\texttt{count}}$ and thus the invariants hold that $g \notin \bigcup_{q \in [\texttt{count}+1-\sqrt{n}+|\mathcal{G}|, \texttt{count}+1)} \mathcal{P}_q$.

Case 2: $i, j+1$ Note that on an extraction from `PQ.Head`, $|\mathcal{G}|$ decreases by 1. Thus for the new "good" set, $\mathcal{G}'$, $|\mathcal{G}'| < \sqrt{n}$. For the case where $|\mathcal{G}| = \sqrt{n}$, we must show that g is not in the updated current partition, $\mathcal{P}'_{\texttt{count}}$ after `PQ.Order` is called. This holds because we call `MergeFill` on the head and the current partition, $\mathcal{P}_{\texttt{count}}$. I.e. if `MergeFill` introduced element g into the head, then $|\mathcal{G}'| = \sqrt{n}$ and thus the invariants hold trivially. Otherwise, we have that $|\mathcal{G}'| = \sqrt{n} - 1$ and thus there exists at least one element in the head, β, such that $\beta > g$. As `MergeFill` guarantees that $\forall e \in \mathcal{P}'_{\texttt{count}}, e > \beta$, we have that g is not in $\mathcal{P}'_{\texttt{count}}$ and thus the invariants hold.

Similar to **case 1**, we either have that $|\texttt{PQ.Head}| = \sqrt{n}$ or has empty slots. In either case, after calling merge sort, we have that forall $e \in \mathcal{P}'_{\texttt{count}}$, $e > g$ as if $g \in \mathcal{P}_{\texttt{count}}$, then g is moved into the head. By the inductive hypothesis and the above, we can see that $g \notin \bigcup_{q \in [\texttt{count}+1-\sqrt{n}+|\mathcal{G}|-1, \texttt{count}+1)} \mathcal{P}_q$. Thus, we then have that invariant holds.

□

2.2 Correctness Proof

Assuming that the invariants hold in section 2.1, we will show that the algorithm is correct. Note that if the total number of inserted elements in the queue is less than $\sqrt{n}$, correctness holds trivially. Otherwise, we will show that $\mathcal{P}_{\texttt{count}}$ is always "close enough" to the next "good" element. Whenever `PQ.ExtractFront` is called, we have that $|\mathcal{G}|$ may decrease by 1. If $|\mathcal{G}|$ does decrease by 1, we still have that the partition containing the next good element, g, will be reached in at most $|\mathcal{G}| - 1$ `PQ` operations. So, by the call of `MergeFill` in `PQ.Order`(), we will have that g is in the head or the stash after $|\mathcal{G}| - 1$ calls. We can then guarantee that the smallest element in the head and stash are at least as small as g. If no insertions were called with p where $p < g$, then g is indeed the smallest element in the priority queue. Otherwise we will have p in the head and thus p will be the smallest element in the priority queue. Thus for every call to `PQ.ExtractFront`, we can return the smallest item in `PQ`.

2.3 Time Complexities

Fist, we will show that $|\texttt{PQ.Stash}| \le \sqrt{n}$, note that we are guaranteed that the total number of elements in `PQ` is at most n and that the capacity of each partition is $\mathcal{P}_\alpha = \sqrt{n}$. Thus, the total capacity of all partitions is n and we then have at least $\sqrt{n}$ empty slots in the partitions as the head is guaranteed to contain $\sqrt{n}$ elements. So, we have that no element which is added to the stash remains in the stash for more than $\sqrt{n}$ calls to `PQ.Order`() as each call attempts to place an element from the stash into the current partition. So then, after performing an initial $\sqrt{n}$ insertions, we have for each subsequent operation, we are guaranteed to be able to cumulatively remove $i+j-\sqrt{n}$ elements from the stash where $i+j$ is the total number of operations. So, because we can add at most i elements to the stash and remove at least $i+j-\sqrt{n}$ elements from the stash, we have that $|\texttt{PQ.Stash}| \leq \sqrt{n}$.

3 Ideal MSLE Functionality

We use a similar notion of ideal functionality for a multi-secret leader election from the ideal functionality of single secret leader election of [CFG22] except that we add a `register_elect` phase for each election.

The MSLE functionality $\mathcal{F}_{\text{MSLE}}$: Set $\mathcal{E} \leftarrow \emptyset$ to denote the set of finished elections. Fix some $k \in \mathbb{N}$ to denote the number of rounds. Upon receiving,

- `register_elect`(eid) from party P_i. If $eid \in \mathcal{E}$ send $\perp$ to P_i and do nothing. If S_{eid} is not defined, set $S_{eid} \leftarrow \{i\}$. Otherwise, set $S_{eid} \leftarrow S_{eid} \cup \{i\}$ and store S_{eid}.
- `elect`(eid) from all honest participants.
  If $|S_{eid}| \geq k$, randomly sample $W^{eid} \subseteq S_{eid}$ where $|W^{eid}| = k$. Then, assign a random ordering to W^{eid} to get ordered set E^{eid}. Next, send $(\texttt{outcome}, eid, \ell)$ to P_j if $E^{eid}_\ell = j$ and $(\texttt{outcome}, eid, \perp)$ to P_i if $i \notin E^{eid}$. Store E^{eid} and set $\mathcal{E} \leftarrow \mathcal{E} \cup \{eid\}$.
- `reveal`(eid, ℓ) from P_i: If E^{eid} is not defined, send $\perp$ to P_i and do nothing. Otherwise, retrieve E^{eid}. If $i = E^{eid}_\ell$, broadcast $(\texttt{result}, eid, \ell, i)$. Otherwise, broadcast $(\texttt{rejected}, eid, \ell, i)$.

Figure 1: Description of the Multi Secret Leader Election functionality

.

4 Semi Honest Reservoir Sampling Based Protocol

We outline a semi-honest protocol in fig. 2 in the common reference string model for the MSLE functionality.

We define the setup, `setup`, for the protocol as follows:

- Set $k_{\mathtt{TFHE}} \xleftarrow{\$} \mathbb{F}_q$.
- Publish CRS, $CRS = \mathtt{Enc}_{\mathrm{TFHE}}(k_{\mathtt{TFHE}})$.

The MSLE Protocol, Protocol, π_i, for party P_i in the semi-honest setting:
Each party has as input a TFHE secret key share sk_i of secret key sk with associated public key pk. Initialize a set of finished elections $\mathcal{E}$.

- ($\texttt{register\_elect}, eid$) called by party P_i
  - Sample $s_i, r_i \xleftarrow{\$} \mathbb{F}_q$, $c_i \leftarrow \texttt{comm}(s_i, r_i)$, $\texttt{ct}_i \leftarrow \texttt{Enc}_{\text{TFHE}}(c_i)$
  - Broadcast ($\texttt{register\_elect\_add}, eid, \texttt{ct}_i$) to all parties and run ($\texttt{register\_elect\_add}, eid, \texttt{ct}_i$)
- ($\texttt{register\_elect\_add}, eid, \texttt{ct}_j$) from party P_j for $j \in [n]$
  - If b_{eid} is not stored, set $b_{eid} = 0$
  - $b_{eid} \leftarrow b_{eid} + 1$
  - If $\mathcal{R}_{eid}$ is not stored, $\mathcal{R}_{eid} \leftarrow \texttt{Resevoir.Init}(k)$
  - Set $\texttt{Enc}_{\text{TFHE}}(e_i), \texttt{Enc}_{\text{TFHE}}(\texttt{coin}_i) \leftarrow \texttt{TFHE.Eval}(C_{PRF}, CRS, \texttt{ct}_j)$ where C is the PRF evaluation circuit and CRS is the key to the PRF.
  - Call $\mathcal{R}_{eid} \leftarrow \texttt{TFHE.Eval}(C, \mathcal{R}_{eid}, \texttt{ct}_j, \texttt{Enc}_{\text{TFHE}}(e_i), \texttt{Enc}_{\text{TFHE}}(\texttt{coin}_i))$ where C is the reservoir sampling circuit for $\texttt{Resevoir.Insert}$.
  - Store $\mathcal{R}_{eid}$.
- ($\texttt{elect}, eid$) called by party P_i when $b_{eid} = n$
  - For $\ell \in [k]$, set $\texttt{ct}_\ell \leftarrow \texttt{TFHE.Eval}(C_{\texttt{Resevoir.Output}}, \mathcal{R}_{eid})$
  - Set $p_i \leftarrow \texttt{TFHE.PartDec}(\texttt{ct}_1, ..., \texttt{ct}_k, sk_i)$
  - Broadcast ($\texttt{elect\_done}, eid, p_i$) and call ($\texttt{elect\_done}, eid, p_i$)
- ($\texttt{elect\_done}, eid, p_j$) from party P_j
  - Retrive $\mathcal{P}^{eid}$ if it is stored, otherwise set $\mathcal{P}^{eid} \leftarrow \emptyset$.
  - Set $\mathcal{P}^{eid} \leftarrow \mathcal{P}^{eid} \bigcup \{p_j\}$
  - If $|\mathcal{P}^{eid}| = n$, set $c_{a_1}, ..., c_{a_\ell} \leftarrow \texttt{TFHE.FinDec}(\mathcal{P}^{eid})$
  - Store $\mathcal{P}^{eid}$.
- ($\texttt{reveal}, eid, \ell$) from party P_i if $c_{a_\ell} = c_i$.
  - Broadcast ($\texttt{result}, eid, \ell, i$)

Figure 2: Description of the MSLE protocol

4.1 Semi Honest Simulation Security

We will now show that the above protocol is semi-honest simulation secure.

Theorem 4.1 (Semi-honest simulation security)**.** *Assuming the existence of a PRF, a* TFHE *scheme with semantic security (definition 1.3) and simulation security (definition 1.4), then the protocol outline in fig. 2 is semi-honest simulation secure.*

Proof. We will show that the simulator outlined in fig. 3 is a simulator for the view of corrupt parties $C \subset [n]$ and $|C| < t$ by showing that the simulator's view and the protocol's view are indistinguishable for all calls via the following three lemmas.

Lemma 4.2. *The views of* `register_elect`, `register_elect_add` *in the protocol and simulator are indistinguishable.*

Proof. The simulator firsts simulates the view of the protocol for all calls to `register_elect` from P_i, $i \in C$. Note that the simulator uses the same inputs, s_i, r_i as in the protocol and thus the view is identical. Moreover, the view of $(\texttt{register\_elect\_add}, eid, \texttt{ct}'_i)$ is identical to the real protocol as $\texttt{ct}'_i = \texttt{ct}_i$. For $j \notin C$, note that $c'_j \overset{c}{\equiv} c_j$ by the hiding property of commitments and thus $\texttt{ct}'_j \overset{c}{\equiv} \texttt{ct}_j$. Then, assuming that prior calls to `register_elect` by P_j are simulated by the simulator, the view of $(\texttt{register\_elect\_add}, eid, \texttt{ct}'_j)$ is indistinguishable to the real protocol as `register_elect_add` is completely determined by prior calls to `register_elect_add` and $\texttt{ct}'_j$.

Let $\mathcal{R}_{eid,\texttt{Sim}_C}$ denote the reservoir sampling data-structure in the simulator. We then note that, after α, for $\alpha \in [n]$, calls to `register_elect_add`, the simulator's $\mathcal{R}_{eid,\texttt{Sim}_C} \overset{c}{\equiv} \mathcal{R}_{eid}$ of the protocol as $\mathcal{R}$ is completly determined by the calls to `register_elect_add`. $\square$

Lemma 4.3. *The views of* `elect`, `elect_done` *in the protocol and simulator are indistinguishable.*

Proof. When `elect` is called by P_i for $i \in C$, we have that for $\texttt{ct}'_\ell = \texttt{TFHE.Eval}(C_{\texttt{Resevoir.Output}}, \mathcal{R}_{eid,\texttt{Sim}_C})$, $\texttt{ct}'_\ell \overset{c}{\equiv} \texttt{ct}_\ell$ for all $\ell \in [k]$. This is because $\mathcal{R}_{eid,\texttt{Sim}_C}$ is indistinguishable from that of the real protocol. We also have that, by the ideal functionality of `elect`, $\{ a'_\ell \}_{\ell \in [k]}$ is a random subset of $[n]$ with random order with size k. Then, note that $e_i, \texttt{coin}_i = PRF(k_{\texttt{TFHE}}, c_i)$ in the protocol is indistinguishable from a randomly sampled $e_i, \texttt{coin}_i$ by the semantic security of PRFs. Thus, by the correctness of $\texttt{Resevoir.Output}(\mathcal{R})$, $\{ a_\ell \}_{\ell \in [k]}$ is a randomly chosen ordered subset of $[n]$. So then, $\{ a'_\ell \}_{\ell \in [k]} \overset{c}{\equiv} \{ a_\ell \}_{\ell \in [k]}$. Remembering that $c'_\beta \overset{c}{\equiv} c_\beta$ for all $\beta \in [n]$, we have that $c'_{a_1}, ..., c'_{a_k} \overset{c}{\equiv} c_{a_1}, ..., c_{a_k}$. By the simulation security of `TFHE`, we have that the decryption shares, $p'_1, ..., p'_n = \texttt{Sim}_{\texttt{TFHE}}(C_{\texttt{Ident}}, (\{ ct'_\ell \}, c'_{a_1}, ..., c'_{a_k}), [n], \texttt{st})$, for all $i \in [n]$ are simulated by the simulator such that $p'_i \overset{c}{\equiv} p_i$ in the real protocol. And thus, all calls to `elect_done` are indistinguishable from the real protocol as `elect_done` is completely determined by $\{ p_i \}_{i \in [n]}$. $\square$

Lemma 4.4 (`reveal` is simulation secure)**.** *Proof.* Finally, note that `reveal` in the simulator is identical to that of the real protocol as in the semi-honest setting, only parties which have won an election will call `reveal`. $\square$

We thus have that the view of the simulator is identical to that of the real protocol by the above three lemmas. $\square$

Simulator for Threshold MSLE $\texttt{Sim}_C$ where $C \subseteq [n]$ and $|C| < t$:
Initializie a set of finished elections $\mathcal{E}$ and set a random tape for the simulator.

- $(\texttt{register\_elect}, eid)$ for party P_j
  - If $j \in C$ Follow the protocol's $\texttt{register\_elect}$ using P_j's $s_j, r_j, c_j, \texttt{ct}_j$ from the real protocol and call $(\texttt{register\_elect\_add}, eid, \texttt{ct}_j)$. Set $c'_j \leftarrow c_j$ and store c'_j.
  - If $j \notin C$, $s'_j, r'_j \xleftarrow{\$} \mathbb{F}_q, c'_j \leftarrow \texttt{comm}(s'_j, r'_j), \texttt{ct}'_j \leftarrow \texttt{Enc}_{\text{TFHE}}(c'_j)$ Store c'_j and follow the protocol by calling $(\texttt{register\_elect\_add}, eid, \texttt{ct}'_j)$.
- $(\texttt{elect}, eid, \texttt{out})$ from party P_j where $\texttt{out}$ is $\mathcal{F}_{\text{MSLE}}.\texttt{elect}$ output $(\texttt{outcome}, eid, q)_i$ for all $i \in [n]$ where $q \in \{1, ..., k, \perp\}$.
  - For $\ell \in [k]$, set $\texttt{ct}'_\ell \leftarrow \texttt{TFHE.Eval}(C_{\texttt{Resevoir.Output}}, \mathcal{R}_{eid})$
  - Let $a'_1, ..., a'_k \in [n]$ such that if $q_i = \ell$ then $a'_\ell = i$.
  - Call $p'_1, ..., p'_n \leftarrow \texttt{Sim}_{\texttt{TFHE}}(C_{\texttt{Ident}}, (\{\texttt{ct}'_\ell\}, c'_{a'_1}, ..., c'_{a'_\ell}), [n], \texttt{st})$
  - Call $(\texttt{elect\_done}, eid, p_j)$ from the protocol.
- $(\texttt{reveal}, eid, \ell)$ for P_j and $\mathcal{F}_{\text{MSLE}}.\texttt{reveal}$ output $(\texttt{result}, eid, \ell, j)$ or $(\texttt{reject}, eid, \ell, j)$.
  - Broadcast $(\texttt{result}, eid, \ell, j)$

Figure 3: Description of the MSLE protocol

5 Malicious Adversary Secure Protocol

Before describing the protocol, we need to define two relationships which will be used in conjunction with two different NZIKs. A tuple $((pk, \texttt{ct}), \mu) \in Rel_{\texttt{Encr}}$ if $\texttt{ct} = \texttt{Enc}_{\text{TFHE}}(\mu)$ under TFHE public key pk. A tuple $((p_i, \texttt{Inps}, \texttt{Enc}_{\text{TFHE}}(s_{\texttt{TFHE}})), sk_i) \in Rel_{\texttt{Prot}}$ if

The MSLE Protocol π_{MSLE}: Initialize $b = 0$. Fix some $k \in \mathbb{N}$ to denote the number of rounds. Initialize an empty set of tickets, R, an empty lookup of sets of parties in each election, S, an empty lookup of reservoir sampling data structures $\mathcal{R}$, and a set of finished elections $\mathcal{E}$. Initialize an empty lookup of election results, E.

- $(\texttt{register\_elect}, eid)$ called by party P_i Every honest party does the following:
  - Sample $s_i, r_i \xleftarrow{\$} \mathbb{F}_q, c_i \leftarrow \texttt{comm}(s_i, r_i), \texttt{ct}_i \leftarrow \texttt{Enc}_{\text{TFHE}}(c_i)$. Store c_i and $\texttt{ct}_i$ in a lookup table, C.
  - Generate $\pi_i^{\texttt{Encr}} \leftarrow \texttt{NZIK.P}(CRS, \texttt{ct}_i, c_i)$ to show $(c_i, \texttt{ct}_i) \in Rel_{\texttt{Encr}}$ where $(w, x) \in Rel_{\texttt{Encr}}$ if $x = \texttt{Enc}_{\text{TFHE}}(w)$.
  - Broadcast $(\texttt{register\_elect\_add}, eid, \texttt{ct}_i, \pi_i^{\texttt{Encr}})$ and run $(\texttt{register\_elect\_add}, eid, \texttt{ct}_i, \pi_i^{\texttt{Encr}})$.
- $(\texttt{register\_elect\_add}, eid, \texttt{ct}_j, \pi_j^{\texttt{Encr}})$ called by party P_j.
  - If $eid \in \mathcal{E}$ or $\texttt{NZIK.V}(\texttt{NZIK.crs}, \texttt{ct}_j, \pi_j^{\texttt{Encr}}) \neq 1$, return and do nothing.
  - If b_{eid} is not stored, $b_{eid} \leftarrow 0$
  - $b_{eid} \leftarrow b_{eid} + 1$
  - If $\mathcal{R}_{eid}$ is not stored, $\mathcal{R}_{eid} \leftarrow \texttt{Resevoir.Init}(k)$
  - $\texttt{Enc}_{\text{TFHE}}(e_i), \texttt{Enc}_{\text{TFHE}}(\texttt{coin}_i) \leftarrow \texttt{TFHE.Eval}(C_{PRF}, CRS, \texttt{ct}_j)$
  - If $\texttt{Inps}$ is not stored, $\texttt{Inps} \leftarrow [\,]$.
  - $\texttt{Inps}[j] \leftarrow \texttt{Enc}_{\text{TFHE}}(c_j)$
  - Call $\mathcal{R}_{eid} \leftarrow \texttt{TFHE.Eval}(C, \mathcal{R}_{eid}, \texttt{ct}_j, \texttt{Enc}_{\text{TFHE}}(e_i), \texttt{Enc}_{\text{TFHE}}(\texttt{coin}_i))$ where C is the reservoir sampling circuit for $\texttt{Resevoir.Insert}$.
- $(\texttt{elect}, eid)$. called by party P_i. (TODO: add conditions)
  - For $\ell \in [k]$, $\texttt{ct}_\ell \leftarrow \texttt{TFHE.Eval}(C_{\texttt{Resevoir.Output}}, \mathcal{R}_{eid})$
  - Set $p_i \leftarrow \texttt{TFHE.PartDec}(\texttt{ct}_1, ..., \texttt{ct}_k, sk_i)$
  - Generate $\pi_i^{\texttt{Prot}} \leftarrow \texttt{NZIK.P}(CRS, (p_i, \texttt{Inps}), sk_i)$ for the relationship $Rel_{\texttt{Prot}}$ where $(p_i, \texttt{Inps}) \in Rel_{\texttt{Prot}}$ if... TODO:
  - Broadcast $(\texttt{elect\_done}, eid, p_i, \pi_i^{\texttt{Prot}})$ and call $(\texttt{elect\_done}, eid, p_i, \pi_i^{\texttt{Prot}})$
- $(\texttt{elect\_done}, eid, p_j, \pi_j^{\texttt{Prot}})$ called by party P_j
  - If $\texttt{NZIK.V}(CRS, (\texttt{Inps}, p_j), \pi_j^{\texttt{Prot}}) \neq 1$, do nothing and return
  - If $\texttt{elect\_done}_j$ is stored, do nothing and return.
  - Store $\texttt{elect\_done}_j$

- Retrieve $\mathcal{P}^{eid}$ if it is stored, otherwise set $\mathcal{P}^{eid} \leftarrow \emptyset$.
- Set $\mathcal{P}^{eid} \leftarrow \mathcal{P}^{eid} \bigcup \{p_j\}$
- If $|\mathcal{P}^{eid}| \geq t$, set and store $c_{a_1}, ..., c_{a_\ell} \leftarrow \texttt{TFHE.FinDec}(\mathcal{P}^{eid})$
- Store $\mathcal{P}^{eid}$.
- Send $(\texttt{prove}, sid, (p_i, \texttt{Inps}, \texttt{Enc}_{\text{TFHE}}(s_{\texttt{TFHE}})), sk_i)$ to $\mathcal{F}^{Rel}_{\texttt{NZIK}}$.

- $\texttt{reveal}(eid, \ell, \texttt{Enc}_{\text{TFHE}}(c_i'))$ from P_i
  - P_i submits a proof that they know the opening to c_{a_ℓ}. If this proof verifies, send out $(\texttt{result}, eid, \ell, i)$ to all parties. Otherwise, send out $(\texttt{rejected}, eid, \ell, i)$ to all parties.

5.1 Static Malicious Adversary Security

Consider parties P_i to be corrupted where $i \in C$ and $C \subseteq [n]$ such that. $|C| < t$.

Lemma 5.1 (`register_elect` is simulation secure)**.** *Idea: if $\perp$, just do same If: $j \notin C$, just do same If: $j \in C$, then use the NIZK to extract out the commitment from the FHE ciphertext or bot Then: S follows protocol*

Lemma 5.2 (`elect` is simulation secure)**.** *Idea: if $\perp$, just do same If: $j \notin C$, just do same Output list from streaming sampler using TFHE sim, set to correct sequence (irrespective of advers) Ask advers to generate decryption shares for each corrupted party and broadcast proof For non-corrupted parties, use TFHE sim to generate decryption shares, simulate proofs For corrupted parties, ask advers to generate decryption shares and proofs If proofs check, use them, if not discard the decryption shares*

Lemma 5.3 (`reveal` is simulation secure)**.** *If an honest party sends the req, we simulate a commitment opening according to the stored commitment (that we know) If corrupted party sends req, pretending to know opening to not their commitment, we say $\perp$ (this is likely as we have the hiding property) If corrupted party sends req correctly, we use the de-commitment that they send*

References

[BEHG20] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 12–24, 2020. 1.1

[BGG+18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*, pages 565–596. Springer, 2018. 1.1

[CFG22] Dario Catalano, Dario Fiore, and Emanuele Giunta. Adaptively secure single secret leader election from ddh. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 430–439, 2022. 1.2, 3

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs based on a single random string. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 308–317. IEEE, 1990. 1.2

[JRS17] Aayush Jain, Peter MR Rasmussen, and Amit Sahai. Threshold fully homomorphic encryption. *Cryptology ePrint Archive*, 2017. 1.1, 1.1, 1.2, 1.3, 1.4

[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. 1.3, 1.3

[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. 1.3, 1.6, 1.7

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 543–553. IEEE, 1999. 1.2

[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. 1.3

[Vit85] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985. 1.4