Multiple Secret Leaders

Authors here

January 13, 2024

1 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 `Fill`

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

Algorithm 2 `MergeFill`

Input: $\texttt{Head}, \mathcal{P}_{\texttt{count}}$
$e_1, ..., e_\gamma = \texttt{sort}(\texttt{Head}, \mathcal{P}_{\texttt{count}})$ where $\gamma = |\texttt{Head}| + |\mathcal{P}_{\texttt{count}}|$ ▷ Via a data-independent sort
$\texttt{Head} = \{e_1, ..., e_{\min(\sqrt{n},\gamma)}\}$
$\mathcal{P}_{\texttt{count}} = \{e_{\sqrt{n}+1}, ..., e_\gamma\}$
return $\texttt{Head}', \mathcal{P}_{\texttt{count}}$

Algorithm 3 `ExtractSmallest`

Input: List ℓ of size k
$i = \texttt{argmin}(\ell)$
$\ell = \ell_1, ..., \ell_{i-1}, \ell_{i+1}, ...\ell_k$
return $\ell, \ell[i]$

1.1 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, the size of `DIPQ` 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 that $|\texttt{DIPQ}| > \sqrt{n}$.

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

We will show that the following invariants hold:

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

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 $|\texttt{DIPQ}|$ is less than or equal to $\sqrt{n}$ for all prior operations. Note that the operation has 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 `PQ.ExtractLargest` (line 10) in the insertion

Algorithm 4 Data Independent Priority Queue (DIPQ)

function INIT
  $\texttt{count} \leftarrow 0$
  $\texttt{Head} \leftarrow []$
  $\mathcal{P}_0, \cdots, \mathcal{P}_{\sqrt{n}-1} \leftarrow \emptyset$
  $\texttt{Stash} \leftarrow \emptyset$
function INSERT(p, v)
  if $|\texttt{Head}| < \sqrt{n}$ **then**
    Append (p, v) to Head
  else
    $i, (p', v') = \texttt{GetLargest}(\texttt{Head})$ where i is the index of (p', v') in Head
    $I = 1$ if $p' < p$ else 0
    $\texttt{Stash} = \texttt{Stash} :: (I \cdot (p, v) + (1 - I) \cdot (p', v'))$
    $\texttt{Head}[i] = I \cdot (p', v') + (1 - I) \cdot (p, v)$
  Call Order()
function EXTRACTFRONT
  $j, (p, v) = \texttt{GetSmallest}(\texttt{Stash})$ where j is the index of (p', v') in Stash
  $k, (p', v') = \texttt{GetLargest}(\texttt{Head})$ where k is the index of (a, b) in Head
  $I = 1$ if $p' < p$ else 0
  $\texttt{Stash}[j] = (1 - I) \cdot (p', v') + I \cdot (p, v)$ ▷ Conditionally swap (p, v) and (p', v')
  $\texttt{Head}[k] = I \cdot (p', v') + (1 - I) \cdot (p, v)$ ▷ Conditionally swap (p', v') and (p, v)
  $\texttt{Head}, (p_r, v_r) = \texttt{ExtractSmallest}(\texttt{Head})$
  Call Order()
  return (p_r, v_r)
function ORDER
  $\mathcal{P}_{\texttt{count}}, \texttt{Stash} = \texttt{Fill}(\mathcal{P}_{\texttt{count}}, \texttt{Stash})$
  $\texttt{Head}, \mathcal{P}_{\texttt{count}} = \texttt{MergeFill}(\texttt{Head}, \mathcal{P}_{\texttt{count}})$
  $\texttt{count} \leftarrow \texttt{count} + 1 \mod \sqrt{n}$

operation. We thus have that $|\mathcal{G}| = \sqrt{n}$ after the insertion. We can then see that the invariant holds trivially as $\texttt{count} - \sqrt{n} + |\mathcal{G}| = \texttt{count} - \sqrt{n} + \sqrt{n} = \texttt{count}$ and thus $\bigcup_{q \in [\texttt{count} - \sqrt{n} + - \sqrt{n}, \texttt{count})} \mathcal{P}_q = \emptyset$.

Now, to prove the inductive case we need to show that after an operation, the new good set, $\mathcal{G}'$ has size at least $|\mathcal{G}| - 1$ and that $g \notin \mathcal{P}_{\texttt{count}+1}$. Then, as we have the inductive hypothesis that $g \notin \bigcup_{q \in [\texttt{count} - \sqrt{n} + |\mathcal{G}|, \texttt{count})} \mathcal{P}_q$, we can show that $g \notin \bigcup_{q \in [\texttt{count}+1 - \sqrt{n} + |\mathcal{G}| - 1, \texttt{count}+1)} \mathcal{P}_q$.

We will proceed by assuming that the invariant hold for insertion count i and extraction count j. We will show that they hold for $i+1, j$ and $i, j+1$.

Case 1: $i+1, j$ Let (p, v) be the inserted element. We will now show that the new good set, $\mathcal{G}'$ has size at least $|\mathcal{G}|$. If $p < a$ for some $a \in \mathcal{G}$, then $p \in \mathcal{G}'$. As at most one element is evicted from $\mathcal{G}$, we have that $|\mathcal{G}'| \geq |\mathcal{G}| - 1 + 1$. If $p > a$ for all $a \in \mathcal{G}$, then the $\mathcal{G}'$ does not lose any elements from $\mathcal{G}$ and so $|\mathcal{G}'| \geq |\mathcal{G}|$.

Now, note that if $|\mathcal{G}'| = \sqrt{n}$, then the invariants hold trivially as $\bigcup_{j \in [\texttt{count}+1 - \sqrt{n} + \sqrt{n}, \texttt{count}+1)} \mathcal{P}_j = \emptyset$. So, we assume that $|\mathcal{G}'| < \sqrt{n}$. Note that because we call `Order` at the end of insertion (line 14) and $\sqrt{n} > |\mathcal{G}'| \geq |\mathcal{G}|$, `Head` either had an empty slot or an element β such that $\beta > g$ prior to insertion. If $p < g$, then $g' \leftarrow p$ (where g' is the updated g) and (p, v) are moved into the head. Otherwise, after calling `Order`, we have that g is moved into $\texttt{Head}'$ if $g \in \mathcal{P}_{\texttt{count}}$ by the functionality of `MergeFill`. So, we can then see that $g' \notin \mathcal{P}'_{\texttt{count}}$. Finally, as $g' \leq g$ and no element smaller than or equal to g is in $\bigcup_{q \in [\texttt{count} - \sqrt{n} + |\mathcal{G}|, \texttt{count})} \mathcal{P}_q$ by the inductive hypothesis, $g' \notin \bigcup_{q \in [\texttt{count}+1 - \sqrt{n} + |\mathcal{G}|, \texttt{count}+1)} \mathcal{P}_q$.

Case 2: $i, j+1$ Note that on an extraction from `Head`, $|\mathcal{G}|$ may decrease by at most 1. We can then see that the new good set $\mathcal{G}'$ has size at least $|\mathcal{G}| - 1$. Again, for the case where the new good set, $\mathcal{G}'$, has size $\sqrt{n}$, the invariants hold trivially. So, we assume that $|\mathcal{G}'| < \sqrt{n}$.

We must now show that g is not in the updated current partition, $\mathcal{P}'_{\texttt{count}}$, after `Order` is called. We can show this 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, $g \notin \mathcal{P}'_{\texttt{count}}$ and thus the invariants hold. Otherwise, if `MergeFill` did not introduce g into the head, then $g \notin \mathcal{P}_{\texttt{count}}$ and so $g \notin \mathcal{P}'_{\texttt{count}}$ as elements in $\mathcal{P}_{\texttt{count}}$ can only increase after `MergeFill`. By the inductive hypothesis and the above, we can see that $g \notin \bigcup_{q \in [\texttt{count}+1 - \sqrt{n} + |\mathcal{G}| - 1, \texttt{count}+1)} \mathcal{P}_q$ via an identical line of reasoning as in **case 1**. Thus, we then have that invariant holds.

By induction, we have that the invariant holds for all i, j when the number of elements in the priority queue passed $\sqrt{n}$ at some point in the sequence of operations. □

1.2 Correctness Proof

`ExtractFront` Correctness

Assuming that the invariants hold in section 1.1, we will show that the algorithm is correct. Note that if the total number of elements in the queue never exceeded $\sqrt{n}$ elements, correctness holds as elements are never moved out of the head which is itself a priority queue. Otherwise, we will show that $\mathcal{P}_{\texttt{count}}$ is always "close enough" to the next "good" element. Whenever `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 g will be reached in at most $|\mathcal{G}| - 1$ `DIPQ` operations. So, by the call of `MergeFill` in `DIPQ.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 element e where $e < g$, then after $|\mathcal{G} - 1|$ operations, we will either have that our new $\mathcal{G}$ set, $\mathcal{G}'$, will have size of at least $|\mathcal{G}| - 1 - (|\mathcal{G}| - 1) + 1 = 1$ as g will be moved into the head or g will be in the stash. Thus, calling `ExtractFront` will return the smallest element as we always have that $|\mathcal{G}| > 0$ or, if $|\mathcal{G}| = 0$, g, the smallest element, is in the stash. Otherwise, if e, such that $e < g$, is inserted within the $|\mathcal{G}| - 1$ operations, then e will be in the head and be a part of the new $\mathcal{G}$ set, $\mathcal{G}'$. Similarly, after $|\mathcal{G}| - 1$ operations, we will have that $|\mathcal{G}'| \geq 1$ and thus the smallest element will be returned by `ExtractFront`.

Size of Stash

We will show that $|\texttt{Stash}| \leq \sqrt{n}$. As, $|\texttt{Stash}|$ and $\mathcal{P}_\alpha$, for all $\alpha \in [\sqrt{n}]$, does not grow if $|\texttt{Head}| < \sqrt{n}$, we will assume that $|\texttt{Head}| = \sqrt{n}$. Note that we are guaranteed that the total number of elements in `DIPQ` is at most n and that the capacity of each partition is $\mathcal{P}_\alpha = \sqrt{n}$. So, the total capacity of all partitions is n and, we have that there is always at least $\sqrt{n}$ empty slots in the partitions. Thus, we have that no element which is added to the stash remains in the stash for more than $\sqrt{n}$ calls to `DIPQ.Order()` as each call attempts to place an element from the stash into the current partition. We can also note that the stash can only grow by at most 1 element per call to `Insert`. Thus, assume towards contradiction that $|\texttt{Stash}| > \sqrt{n}$. Then, we have that an element remained in the stash for more than $\sqrt{n}$ calls to `DIPQ.Order()` which is a contradiction. We can then see that $|\texttt{Stash}| \leq \sqrt{n}$.

Data Independence

We will now show that the priority queue is data independent. We can do this by simply noting that every operation is data independent.

1.3 Complexity

Assume that we have a data independent sorting algorithm which takes $O(f(\alpha))$ time where $f(\alpha)$ is some function of α, the number of sorted elements. Now, we can show that the time complexity of `Insert` and `ExtractFront` is $O(\sqrt{n} + f(\sqrt{n}))$. Note that `Order` makes a call to `MergeFill` which does a sort and that the rest of the operations take at most $O(\sqrt{n})$ time. So, `Order` takes $O(\sqrt{n} + f(\sqrt{n}))$ time. We can also see that in `Insert` and `ExtractFront`, all non-`Order` operations take at most $O(\sqrt{n})$ time. So, `Insert` and `ExtractFront` take $O(\sqrt{n} + f(\sqrt{n}))$ time.

2 Adopting to an Oblivious Priority Queue

Here we will adopt the above data structure into an oblivious one.

Algorithm 5 Oblivious Priority Queue (DIPQ)

function INIT
  $\texttt{count} \leftarrow 0$
  $\texttt{Head} \leftarrow [(\text{inf}, \perp), ..., (\text{inf}, \perp)]$ where $|\texttt{Head}| = \sqrt{n}$
  $\mathcal{P}_0, \cdots, \mathcal{P}_{\sqrt{n}-1} \leftarrow [(\text{inf}, \perp), ..., (\text{inf}, \perp)]$ where $|\mathcal{P}_\alpha| = \sqrt{n}$
  $\texttt{Stash} \leftarrow [(\text{inf}, \perp), ..., (\text{inf}, \perp)]$ where $|\texttt{Stash}| = \sqrt{n}$
function ACCESS$(p, v, op \in OPS)$ ▷ where $OPS = \{\texttt{Insert}, \texttt{ExtractFront}\}$
  Append (p, v) to Head
  $i, (p', v') = \texttt{GetLargest}(\texttt{Head})$ where i is the index of (p', v') in Head
  $\texttt{flag}_p = 1$ if $p' < p$ else 0
  $\texttt{Stash} = \texttt{Append}(\texttt{Stash}, \texttt{flag}_p \cdot (p, v) + (1 - \texttt{flag}_p) \cdot (p', v'))$
  $\texttt{Head}[i] = \texttt{flag}_p \cdot (p', v') + (1 - \texttt{flag}_p) \cdot (p, v)$
  $\texttt{Head}_{ins}, (p_r, v_r)_{ins} = \texttt{Head}, (0, 0)$
  $\texttt{Head}_{extr}, (p_r, v_r)_{extr} = \texttt{ExtractSmallest}(\texttt{Head})$
  $\texttt{flag}_{ins} = 1$ if $op = \texttt{Insert}$, otherwise 0 when $op = \texttt{ExtractFront}$
  $\texttt{Head} = \texttt{flag}_{ins} \cdot \texttt{Head}_{ins} + (1 - \texttt{flag}_{ins}) \cdot \texttt{Append}(\texttt{Head}_{extr}, \text{inf})$
  $(p_r, v_r) = \texttt{flag}_{ins} \cdot (p_r, v_r)_{ins} + (1 - \texttt{flag}_{ins}) \cdot (p_r, v_r)_{extr}$
  Call Order()
  return (p_r, v_r)
function ORDER
  $\mathcal{P}_{\texttt{count}}, \texttt{Stash} = \texttt{Fill}(\mathcal{P}_{\texttt{count}}, \texttt{Stash})$ ▷ See algorithm 1
  $\texttt{Head}, \mathcal{P}_{\texttt{count}} = \texttt{MergeFill}(\texttt{Head}, \mathcal{P}_{\texttt{count}})$ ▷ See algorithm 2
  $\texttt{count} \leftarrow \texttt{count} + 1 \mod \sqrt{n}$

2.1 Correctness

We can note that we have the same invariants hold as in the data independent case. We can also see that when we call Access with operation ExtractFront where $p = \text{inf}$, then we have that the only possible change before the call to Order is that we append an element with priority inf to the stash. As we are removing the smallest element from the queue but adding back an element with priority inf, we mantain the size of the queue without changing its correctness.

2.2 Security and Obliviousness

We can see that the memory access pattern for Access is deterministic and independent of the input. We thus have that algorithm 5 is oblivious.

3 Building a Dequeue

Given black box access to a data independent (or oblivious) priority queue, we can build a data independent (resp. oblivious) double sided queue. With a double sided queue, we can build a stack and heap as well.

3.1 Data Independent (Oblivious) Dequeue

Assume that we have a data independent (resp. oblivious) priority queue, `DIPQ`. Then, in algorithm 4, we show how to build a data independent (resp. oblivious) double sided queue, `Dequeue`.

Algorithm 6 Data Independent Dequeue (`Dequeue`)

function INIT
  $\texttt{DIPQ}_{\texttt{front}} \leftarrow \texttt{DIPQ.Init}()$ where $\texttt{DIPQ}_{\texttt{front}}$ is a min priority queue
  $\texttt{DIPQ}_{\texttt{back}} \leftarrow \texttt{DIPQ.Init}()$ where $\texttt{DIPQ}_{\texttt{back}}$ is a max priority queue
  $\texttt{counter} \leftarrow 0$
  $\texttt{size} \leftarrow 0$
function ACCESS$(v, op \in OPS)$ ▷ where $OPS = \{\texttt{Ins}, \texttt{ExtrBack}, \texttt{ExtrFront}\}$
  $p_{\texttt{front}} = \texttt{counter}$ if $op = \texttt{Ins}$ else inf
  $p_{\texttt{back}} = \texttt{counter}$ if $op = \texttt{Ins}$ else $-\inf$
  $\texttt{DIPQ}_{\texttt{back}}.\texttt{Insert}(p_{\texttt{back}}, v)$ ▷ If $op \neq \texttt{ins}$, then this insert will be ignored
  $\texttt{DIPQ}_{\texttt{front}}.\texttt{Insert}(p_{\texttt{front}}, v)$ ▷ If $op \neq \texttt{ins}$, then this insert will be ignored
  $p'_{\texttt{back}} = -\inf$ if $op = \texttt{ExtrBack}$ and inf otherwise
  $p'_{\texttt{front}} = \inf$ if $op = \texttt{ExtrFront}$ and $-\inf$ otherwise
  $\texttt{DIPQ}_{\texttt{back}}.\texttt{Insert}(p'_{\texttt{back}}, \perp)$ ▷ dummy element which will be removed if $op \neq \texttt{extrBack}$
  $\texttt{DIPQ}_{\texttt{front}}.\texttt{Insert}(p'_{\texttt{front}}, \perp)$ ▷ dummy element which will be removed if $op \neq \texttt{extrFront}$
  $\texttt{count} = \texttt{count} + 1$ if $op = \texttt{ins}$ and `count` otherwise
  $(p, v)_{\texttt{back}} \leftarrow \texttt{DIPQ}_{\texttt{back}}.\texttt{ExtractBack}()$
  $(p, v)_{\texttt{front}} \leftarrow \texttt{DIPQ}_{\texttt{front}}.\texttt{ExtractFront}()$
  return $(p, v)_{\texttt{back}}, (p, v)_{\texttt{front}}$

3.2 Correctness

We assume that the number of insertions is always greater than or equal to the number of extractions. Thus $\texttt{DIPQ}_{\texttt{front}}$ and $\texttt{DIPQ}_{\texttt{back}}$ will never output the same element.

To show correctness, we must show that `ExtractBack` returns the "newest" element in the queue and `ExtractFront` returns the "oldest" element inserted into the queue. We can do this by simply noting that every inserted item is associated with a unique counter value which increases with every operation. Thus, extracting from the front of $\texttt{DIPQ}_{\texttt{front}}$ will always return the oldest element and extracting from the back of $\texttt{DIPQ}_{\texttt{back}}$ will always return the newest element.

We can also see that when we have a non-insert operation, we are either calling a front or back extract operation. If we are calling a front extraction, then we insert a dummy element into the back queue which will be removed by the extract operation associated with $(p, v)_{back}$. If we are calling a back extraction, then we insert a dummy element into the front queue which will be removed by the extract operation. Thus, we have that calling the "dummy" extractions will not change the state of the queue.

3.3 Obliviousness

Assuming that `DIPQ` is oblivious, we can see that `Dequeue` is also oblivious as the memory access of `Access` is deterministic.

4 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

.

5 Using iO

We can adopt the $i\mathcal{O}$-based version of secret leader election outlined in [BEHG20] in a rather straightforward manner. We can then show semi-honest security of the protocol in the CRS model, where the $i\mathcal{O}$ obfuscated program is the CRS.

Algorithm 7 $i\mathcal{O}$ based Multi Secret Leader Election

function $P^k_{\texttt{ELECT}}((\text{pk}_0, \ldots, \text{pk}_{n-1}), i, n, R)$
  $s \leftarrow R, \text{pk}_0, \ldots, \text{pk}_{n-1}$
  $(w_1, r_1) \ldots (w_k, r_k), r' \leftarrow F(K, s)$
  $b_\ell \leftarrow 1$ if $i = w_\ell$ and 0 otherwise
  $c_\ell \leftarrow \texttt{comm}(b_\ell, r_e ll)$
  $\texttt{ct} \leftarrow \text{PKE.Encrypt}(\text{pk}_i, (r_1, \ldots, r_k); r')$
  return $(c_1, \ldots, c_k, \texttt{ct})$

The above is nearly identical to [BEHG20] except that we have F be a different size which scales linearly with the number of elections.

5.1 Sketch of Semi-Honest Simulation Security

We sketch the simulator for the semi-honest security of the above protocol. First, we have that `register_elect` behaves identically to the $i\mathcal{O}$-based single secret leader election protocol (we do not show this above). The only difference is that the parties also send a public key pk_i. Given a simulation secure public-key encryption scheme, the simulator can simulate these public keys.

Then, a call to elect can be simulated in a rather straightforward way. As we have that R is public randomness, we can show that $F(K, s) \overset{\text{c}}{\equiv} s'$ where $s' \overset{\$}{\leftarrow} U$. We can see this as $P^k_{\texttt{elect}} \overset{\text{c}}{\equiv} P^k_{\texttt{elect}}(\{s\})$ where the latter program punctures F on s and hardcodes the output of $F(K, s)$ as a constant which we will call z'. Then, $P^k_{\texttt{elect}}(\{s\}) \overset{\text{c}}{\equiv} P^k_{\texttt{elect}}(\{s\})'$ where the constant s is replaced with a random value s'. Then, if an adversary can distinguish between the two, we can construct an adversary which can distinguish the output on the punctured point from a uniformly random distribution.

So, given the output of the ideal functionality, a simulator can sample K, R from a uniformly random distribution. Then, the simulator can create the obfuscated program TODO:compl.

Next, given output $\texttt{elect}(eid)$ as $(\texttt{outcome}, eid, W_i = \{\alpha_1, \ldots, \alpha_k\})$ the simulator can draw a random $\bar{z}$ and get z' by XORing $\bar{z}$ such that $w_\ell = i$ if $\ell \in W_i$. Note that z' is still indistinguishable from a random string as we have $w_\ell = i$ with uniformly random probability. Then, we can see that the simulator has $P^k_{\texttt{elect}}(\{s\})' \overset{\text{c}}{\equiv} P^k$.

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. 5, 5

[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. 4