# Depth Optimized Circuits for Lattice Based Voting with Large Candidate Sets

Oskar Goldhahn<sup>1</sup> and Kristian Gjøsteen<sup>1</sup>

NTNU: Norwegian University of Science and Technology, Trondheim Norway {oskar.goldhahn, kristian.gjosteen}@ntnu.no

**Abstract.** Homomorphic encryption has long been used to build voting schemes. Additively homomorphic encryption only allows simple counting functions. Lattice-based fully (or somewhat) homomorphic encryption allows more general counting functions, but the required parameters quickly become impractical if used naively. It is safe to leak information during the counting function evaluation, as long as the information could be derived from the public result. To exploit this observation, we design a flexible framework for using somewhat homomorphic encryption for voting that incorporates random input and allows controlled leakage of information. We instantiate the framework using novel circuits with low but significant multiplicative depth exploiting the fact that, in the context of voting, leakage of certain information during homomorphic evaluation can be permitted. We also instantiate the framework with a circuit that uses random input to shuffle without the use of mixnets.

## 1 Introduction

Most existing approaches to post-quantum voting use mixnets [1] or additively homomorphic encryption [3, 7, 2]. Another plausible approach would be to use blind signatures [10]. Mixnet-based approaches are flexible, but they are in general vulnerable to italian style attacks where voters hide identifying information in their vote, enabling coercion or vote-selling. Known lattice based mixnets also require very large proofs, making them less practical for all but small elections. Approaches based on additively homomorphic encryption are inflexible, only supporting certain counting functions, and only support small candidate sets since plaintext size grows linearly with the number of candidates or options rather than the entropy of the vote, if multiple candidates are supported at all. Approaches based on anonymous channels are in general vulnerable to traffic analysis.

If we build our homomorphic scheme with post-quantum primitives like BGV [4] we can also multiply homomorphically without much additional work. This essentially allows us to evaluate *arithmetic circuits*, which may lead to more space efficient or flexible voting schemes that can tally complex counting functions privately. The challenge is that depending on the nested number of multiplications, *the multiplicative depth*, we will need larger parameters, keys and ciphertexts.

Making complex counting functions tally privately and efficiently can be a challenge, even in the classical setting, often requiring a tradeoff between leaking some tally details and efficiency [9].

*Our Contributions* Our results are based on the observation that in voting the result is more public than the tally, so leaking some information during the tally is acceptable as long as the leakage could also be computed from the result. In this case an observer cannot learn anything from the tally they would not have learned from the result anyways. This idea allows us to optimize a homomorphic tally computation by carefully leaking information during the computation in a way that reduces the multiplicative depth of the circuits involved.

We also make the observation that with somewhat homomorphic encryption we can use randomization, just like in mixnets, by instead passing in additional random inputs.

Using these observations we design a framework, *Randomized Circuit Trees* or *RCTs*, for somewhat homomorphic encryption with random inputs and leakage, that permits more efficient branching than with pure arithemtic circuits, both in operation count and in multiplicative depth.

We hand-craft RCTs of low depth for secure e-voting which can scale to greater parameters than with known homomorphic voting schemes, but with similar privacy, offering a privacy advantage over mixnet-based schemes, which only offer indistinguishability of permutations of votes, rather than indistinguishability of votes leading to the same result.

We do not claim that the two observations that lead to our results are novel, but that our applications of them are.

Related work Our first circuits evaluate a sorting algorithm. There is earlier work on sorting with low multiplicative depth [5] but they require encrypted output, which only allows them to achieve multiplicative depth  $\lceil \log \ell + 1 \rceil + \lceil \log n - 1 \rceil + 1$ , where  $\ell$  is the bit-length of the elements and n is the number of elements compared to the  $\lceil \log \ell \rceil$  and  $\lceil \log n + 1 \rceil$  two of our RCTs achieve.

There is previous work on using multiplicative homomorphic encryption for post-quantum voting [6]. Their scheme uses bootstrapping, which is expensive, and though they manage to compress plaintext votes to be linear in the entropy rather than the number of options they still have intermediate ciphertexts where the underlying plaintexts are linear in the number of options, so this still cannot handle a large number of options, unlike our approaches.

There is also existing work on computing complex tally functions privately, called *tally-hiding* in the literature, which usually involves viewing tallying as an auditable or universally verifiable MPC protocol. Existing approaches tend to instantiate with classical primitives [9, 11], which is no surprise considering the gap in efficiency between classical and post-quantum MPC. Some of these could be instantiated with post-quantum primitives instead, but it is unclear how practical they would be in that setting. Our work can be seen as a continuation of this line of work that specializes to MPC based on fully homomorphic encryption,

and optimizes for BGV in particular by minimizing the multiplicative depth of the circuits used to avoid bootstrapping.

Structure of paper We start by introducing our RCT framework. Next we introduce some low depth RCTs for various counting functions, such as sorting and instant runoff voting. Then we define the cryptographic primitive, a somewhat homomorphic encryption scheme with verifiable encryption and decryption, which we use to keep the internal values of the RCT hidden during evaluation. Next we define a template for using RCTs to construct voting schemes using these building blocks. Finally we discuss the parameters necessary and estimate the runtime and artifact size of some of the schemes.

## 2 Randomized Circuit Trees

While homomorphic encryption can keep internal values hidden we use controlled leakage of internal values for efficiency. Values can be obtained or verified using verifiable distributed decryption.

*Efficient Branching* In arithmetic circuits branches generally have to be handled by evaluating both sides of the branch and then using the branch condition to discard a result. This can waste a lot of work and means that the circuit computing the branch condition always adds to the depth.

In voting schemes we use distributed trust to have the parties doing the computation also be the parties responsible for doing decryption, so we can optimize branches by leaking the branch conditions. For this to preserve privacy we need to make sure that the branch conditions we leak do not leak any information beyond the result of the election.

**Definition 1.** A function  $f : A \to B$  reduces to  $g : A \to C$  if there exists a polynomial time function  $h : C \to B$  such that f = hg.

The algorithm we want to model is one where we look at the outputs of previous circuits to decide which circuit to compute next. The individual outputs can be thought of as *hints* to what the final output is, and the full sequence of hints should be sufficient to compute the final output while providing no additional information.

**Definition 2.** A circuit tree C is a function taking an integer n, and returning a function from lists of values to circuits taking n inputs. We define the *i*-repeated evaluation of  $C_n$  at  $l \in V^n$  recursively as follows, where  $L_0$  is the empty list, and

$$L_{i+1} = L_i \| \mathcal{C}_n(L_i)(l).$$

In the limit this produces a possibly finite sequence of hint values,  $C(l) = L_{\infty}$ . Given a set  $A \subset V^*$  we say that C directly computes a function  $f : A \to V^*$  if for all  $l \in A$ , C(l) = f(l). We say that C computes f if f reduces to a function directly computed by C, and that C opaquely computes f if this function reduces to f. If a circuit tree opaquely computes a function f, then seeing all its internal outputs gives the same information as seeing just the final output of f.

Randomization Efficient privacy preserving computation can be made easier by randomization. E.g., we can use a random permutation to erase information about the order of a list. We extend our definition of circuit tree to capture this.

**Definition 3.** A randomized circuit tree (RCT)  $\mathcal{RC}$  is a distribution  $\delta$  over  $V^n$ , a function  $f: V^n \times V^* \to V^*$ , a circuit tree, C, that directly computes it and a collapsing function  $g: V^* \to A$  such that for all  $x, y \in \text{supp}(\delta)$  and  $v \in V^*$  we have g(f(x,v)) = g(f(y,v)). Using this equation we can define  $\mathcal{RC}$  to directly compute the function  $h: V^* \to A$  such that h(v) = g(f(x,v)) for all  $v \in V^*$ . We say that  $\mathcal{RC}$  opaquely computes h if C opaquely computes f and for every  $v, w \in V^*$  such that h(v) = h(w), given a  $\delta$ -distributed random variable X, f(X, v) and f(X, w) have the same distribution.

The procedure that randomly generates input using  $\delta$  and then evaluates C from an opaque RCT will not leak anything more than the output of h even if we reveal all the hints during the evaluation of C.

*Efficient Validation* RCTs can be used to compute, but for verification we can often do better by using a circuit to *check* that an input yields a certain output.

**Definition 4.** Let  $C_n$  be the set of circuits taking inputs of length n. Given an integer n, a family of functions  $\mathcal{V}_n : V^* \to V^* \times C_n$  validates a circuit tree C at n if for all  $x \in V^*$  and  $l \in V^n$ , for  $(v, c) = \mathcal{V}_n(x)$  we have c(l) = v iff C(l) = x. We say that the family validates C if  $\mathcal{V}_n$  validates it at n for all n. We also call such a family a validator for C. A validator is opaque if every circuit in its image reduces to the function computed by C.

For an opaque circuit tree there always exists a trivial opaque validator that uses the original circuit tree directly. We write  $\mathcal{V}_{\mathcal{C}}$  for a validator for  $\mathcal{C}$ .

#### 3 Some Randomized Circuit Trees for Voting

We start with an RCT computing the counting function that sorts the votes. This is very powerful because this makes it impossible to distinguish honest voters, though it might not sufficiently hide their votes.

We use the first RCT as inspiration for a specialized RCT for instant runoff voting that allows hiding more information than the sorting function, preventing italian style attacks.

Finally we describe an RCT computing sort that uses shuffling to improve on the efficiency of the first RCT in some cases.

Only the last RCT will exploit the randomness, and only the others will exploit controlled leakage of branch conditions. We leave it to further research to investigate whether there exist approaches using both randomness and controlled leakage that are more efficient or support additional counting functions. All our RCTs are over  $\mathbb{Z}_p[X]/\langle \Phi_N(X) \rangle$ , though we will simplify to working with its factor fields and omit the details of parallelization and movement between the factor fields to simplify the exposition. Most of the circuits will go even further and only work on 0 and 1 in the field. We call such values *binary* and lists of them *bitstrings* in the rest of the text.

Our Circuits and RCTs will have *parameters* that are public and can be used in branches and *arguments* that are private and can mainly be used by circuit operations. The former are written as subscripts while the latter are in parentheses. Sometimes the output of a circuit is used in a parameter, in which case this output is a hint for the selection of the next circuit. Outputs that are only used as arguments and in circuit operations constitute the internals of an implicitly defined circuit. The length of an argument list is always public.

#### 3.1 Depth Optimized Sorting for Generic Counting

Many existing homomorphic voting schemes work by counting the number of votes for each option. This is essentially equivalent to doing counting sort. They work by representing the votes as binary vectors with a single non-zero entry, ensuring this is the case for the ballots using zero-knowledge proofs. The counting can then simply add all the ballots and the result for each option can be read from their index in the decrypted sum. With this method the length of the votes is linear in the number of options, or exponential in the entropy of the options.

We propose RCTs consisting of circuits of low multiplicative depth where the length of the votes is linear in the entropy of the valid votes. This makes other counting functions practical for homomorphic counting, functions which have traditionally been limited to shuffle based schemes.

We encode options into bitstrings of length  $\ell$ , which circuits operate on. This means that formally our counting function will not sort the votes but rather output *the binary votes* in a sorted list together with some information about the nonbinary votes. We use bitstrings because binary equality can be done with fewer and shallower multiplications than equality in the full field.

We determine which votes are properly encoded in binary using the circuit ValidateBin(b) that for each claimed bit  $b_i$  computes  $(b_i - 1)b_i$  and outputs the resulting values. This outputs a 0 vector iff all the  $b_i$ 's are bits. This has multiplicative depth 1, requires 2|b| circuit operations and is easily parallelizable.

Improperly encoded votes (non-binary) are ignored by the rest of the RCT. Thus who cast a valid vote and who did not becomes public, which to some extent is unavoidable. Improperly encoded votes also leak details of their encoding.

The main building block of our counting algorithm is  $\mathsf{Match}_{b,S}(v)$ , a family of circuits that check whether two bitvectors share values at certain indices.

$Match_{b,S}(v)$	$Count_{v,c}(\mathbf{v})$
$\overline{\mathbf{return}} (\prod (\mathbf{if} \ b_i \ \mathbf{then} \ v_i \ \mathbf{else} \ (1-v_i)))$	$\mathbf{if} \ c = 0$
$i \in S$	return []
$\frac{CountMatch_{b,S}(\mathbf{v})}{\mathbf{return} \sum_{v \in \mathbf{v}} Match_{b,S}(v)}$	$\mathbf{if} \  v  = \ell$
	$\mathbf{return} \ (v,c)$
	$c_0 \leftarrow CountMatch_{v \parallel 0, [ v ]}(\mathbf{v})$
	return $Count_{v\parallel 0,c_0}(\mathbf{v}) \ Count_{v\parallel 1,c-c_0}(\mathbf{v})$

**Lemma 1.** Match<sub>b,S</sub> has multiplicative depth  $\lceil \log(\#S) \rceil$  and uses at most 2#S-1 operations.

*Proof.* Since S and b are parameters the circuits reduce to a multiplication of #S values where each value needs at most one subtraction. The multiplicative depth is achieved by using divide and conquer on the multiplications.

Using the matching circuits we build family of circuits,  $CountMatch_{b,S}(\mathbf{v})$  that count the number of votes matching a given bit pattern. Note that the output of this circuit when applied to all valid votes does not care about the order of those votes, which means that we, independently of b and S, can always leak the output of these circuits without leaking any private data.

CountMatch only computes a sum, which does not add to the multiplicative depth beyond what Match adds. It has at most 2#Sn - 1 operations. It should be noted that we need the field to have characteristic greater than the number of votes for this to give the correct result.

Our main algorithm, **Count**, works by recursively adding to a prefix while continuously checking how many votes start with that prefix, terminating if there are none and returning the prefix and count once it has the same length as a vote. One might think of it as doing binary search on the candidates with a positive number of votes.

The initial **Count** is used on the valid votes, with c being the number of valid votes and v being the empty bitstring. The output will then be a list of pairs  $(v_i, c_i)$ , where  $c_i$  is the number of occurrences of the vote  $v_i$  in the valid votes.

Observe that many of the different CountMatch circuits used in the RCT have overlapping sub-circuits of Match because parts of b are the same, which allows us to optimize the number of circuit operations in practice by using memoization.

**Theorem 1.** Count<sub>v,c</sub> opaquely computes a function that given binary input returns them in sorted order. It has multiplicative depth  $\lceil \log \ell \rceil$  and uses at most  $\hat{v}\ell(\lceil \log \ell \rceil n + n - 1)$  circuit operations, where  $\hat{v}$  is the number of unique votes.

*Proof.* The first part follows from the discussion above and the fact that only the outputs of CountMatch are hints. For the second part the multiplicative depth is  $\lceil \log \ell \rceil$  from the maximal size of the S used in Match. To bound the number of operations we start by counting at most  $\lceil \log \ell \rceil$  for each Match with memoized multiplications, which leads to  $\lceil \log \ell \rceil n+n-1$  operations in CountMatch. Finally

we bound the number of nontrivial recursive branches in Count by  $\hat{v}$  and the depth of nontrivial recursion by  $\ell$  which bounds the number of CountMatch uses by their product.

We can trivially bound  $\hat{v}$  by  $\min(n, 2^{\ell})$  but depending on the application this might be much smaller in practice. In practice the uses of Match can also be paralellized over the different voters.

Note that there are  $O(\ell \log \ell)$  circuit operations rather than linearly many in terms of  $\ell$ . We can get linearity by having parallel elections where connections between votes of different elections are obtained from hash chains. This requires the scheme to support at least  $\ell > 256$ . Using this approach also bounds the multiplicative depth.

We can also construct a more efficient validator for this RCT that only does the initial binarity check and checks for matches with the claimed votes, avoiding the need to check short prefixes.

We have used the fact that parts of the computation are done in plain to optimize the branching by aborting early on branches with no votes. Depending on the details of an election we can take this even further. In many real elections using plurality voting it is preferable to count candidates that are expected to have many votes first and the result can often be calculated without counting all the votes. With our circuit one can also greedily choose the branches that have the most votes.

#### 3.2 Applications to Structured Votes

The algorithm in the previous section applies to a generic electoral system but the techniques used can also be specialized to specific systems for increased efficiency and privacy. In particular it mitigates against italian style attacks, where one encodes an identifier into the vote in a way that has minimal effect on the result but can allow voters to prove what they voted for. In return we need a specialized RCT that is less flexible.

We exemplify this by using the techniques from the previous section to build a more specialized RCT for instant runoff voting.

Instant Runoff Voting Instant Runoff Voting is an electoral system where the voters cast a vote with an ordered list of options. The counting works in rounds where we keep track of a list of eliminated options and we count the number of votes for each non-eliminated option of which the lower end is eliminated. This continues until the remaining options reach below a certain threshold.

The options are encoded as  $\ell$  length bitstrings, which are concatenated to form a  $c_p$  length list of options. Formally the counting function will output the counts of the first non-eliminated option in the binary votes for each round and some information about the non-binary votes.

As before we start by filtering out the non-binary votes. For the rest we need a modified family of matching circuits, FirstFilteredMatch<sub>e,b,S</sub>(**v**) that match with respect to the first non-eliminated vote in the list of votes, **v** for a single voter with which the RCT can proceed in a similar fashion as the generic method.

FirstFilteredMatch<sub>e,b,S</sub>( $\mathbf{v}$ )

```
\mathsf{ShuffleCount}(\mathbf{d}, \mathbf{v})
```

 $\begin{array}{ll} \mathbf{if} \ |\mathbf{v}| = 1 & \mathbf{r} \leftarrow [] \\ b_e \leftarrow 1 - \Sigma_{e \in \mathbf{e}} \mathsf{Match}_{e,[|e|]}(v_1) & \mathbf{for} \ \mathbf{i} \in \{0,1\}^{\ell_{\mathrm{id}}} \\ b_m \leftarrow \mathsf{Match}_{v,S}(v_1) & v_{\mathbf{i}} \leftarrow \Sigma_{j \in [n]}(v_j \cdot \mathsf{Match}_{\mathbf{i},[\ell_{\mathrm{id}}]}(\mathbf{d}_j)) \\ \mathbf{return} \ b_e, b_e b_m & \mathbf{r} \leftarrow \mathbf{r} \| v_{\mathbf{i}} \\ m \leftarrow \lfloor |\mathbf{v}|/2 \rfloor & \mathbf{return} \ \mathbf{r} \\ b_{e,1}, b_{m,1} \leftarrow \mathsf{FirstFilteredMatch}_{\mathbf{e},b,S}([v_1, \dots, v_{m-1}]) \\ b_{e,2}, b_{m,2} \leftarrow \mathsf{FirstFilteredMatch}_{\mathbf{e},b,S}([v_m, \dots]) \\ b_e \leftarrow b_{e,1} + b_{e,2} - b_{e,1}b_{e,2} \\ b_m \leftarrow b_{m,1} + (1 - b_{e,1})b_{m,2} \\ \mathbf{return} \ b_e, b_p \end{array}$ 

The circuits recurse on the list of votes of a given voter. Here  $\mathbf{e}$  is the list of eliminated options,  $b_e$  is 0 if all the votes in the current subsequence are eliminated and  $b_m$  is 1 if there is a non-eliminated vote and the first is a match.

The circuit has multiplicative depth  $\lceil \log c_p \rceil + \lceil \log \ell \rceil$  and uses at most  $2\ell c_p \# \mathbf{e} + 2c_p \# S - 7 + 6 \cdot 2^{\lceil \log(c_p) \rceil}$  circuit operations.

Because of the significant worst case impact from the rounds, where in the worst case we have as many rounds as options, we will not be using the binary search method for counting. Instead we will have a small list of options that we explicitly check for. We start with an empty list of eliminated options that we add to as options get eliminated. Each round we count the votes for each non-eliminated option.

**Theorem 2.** The modified Count described above has multiplicative depth  $\lceil \log c_p \rceil + \lceil \log \ell \rceil$  and operation count at most  $2\ell c_p n 2^{\ell} + 3n \cdot 2^{2\ell + \lceil \log(c_p) \rceil}$ .

*Proof.* We can memoize Match for each vote-option pair. We can even memoize the sum  $1 - \sum_{e \in \mathbf{e}} \mathsf{Match}_{e,[|e|]}(v_1)$ . This yields  $2\ell c_p n 2^{\ell}$  operations. The remaining operations come from recursively processing the  $b_m$  and  $b_e$  and summing them up across voters at the end. We end up with at most  $6 \cdot 2^{\lceil \log(c_p) \rceil} - 1$  operations for the processing in one use of FirstFilteredMatch. Multiplying by n for the number of voters and adding n for the operations required for the sum yields  $6n \cdot 2^{\lceil \log(c_p) \rceil}$ . We get our final value by bounding the number of rounds by  $2^{2\ell-1}$  in the worst case where only one option is eliminated each round.

The RCT has a multiplicative depth  $\lceil \log c_p \rceil + \lceil \log \ell \rceil$ , which is comparable to the  $\lceil \log c_p \ell \rceil$  we would get from lifting to the generic method.

The number of circuit operations can be much greater than with the generic method since we have to perform multiple rounds, which the generic method would do in the plain without circuits. It should be noted that the number of operations in both cases are only rough bounds since we only focus on feasibility for the runtime.

#### 3.3 Sorting with Shuffling

The previous examples have not used any random input for the RCT. The idea for our third RCT is to exploit random input to replicate what shuffle based voting is doing.

The distribution for the RCT,  $\delta$ , is the uniform distribution over n length lists of distinct  $\ell_{id}$  length bitstrings such that  $n \leq 2^{\ell_{id}}$ . This will serve as a random permutation of the votes. A sensible choice here is  $\ell_{id} = \lceil \log n \rceil$  but using one greater than this can improve the efficiency of the rejection sampling we need to do to actually compute  $\delta$ .

The circuit tree for the RCT, ShuffleCount, takes a random input **d**, which should come from  $\delta$ , and a list of votes **v**. It sorts the votes using the elements of **d** as sorting keys, which with random, hidden and duplicate-free **d** is the same as shuffling.

This RCT is more flexible in that it can handle non-binary votes, so the votes are  $\ell$ -length *vectors* this time. The RCT also boils down to evaluating only a single circuit.

**Theorem 3.** ShuffleCount is opaque and computes the function that sorts the list of votes,  $\mathbf{v}$ , and has multiplicative depth  $\lceil \log \ell_{id} + 1 \rceil$  and operation count less than  $2n(\ell + 1)2^{\ell_{id}}$ .

*Proof.* Let **d** be the list of distinct bitstrings sampled from  $\delta$ . The output of the circuit is only the permuted list with some extra zero-votes inserted. The extra zero-votes are located at indices where there are no **i** values in **d**, which are uniformly random and independent of the ordering of the  $\mathbf{d}_i$ . Opaqueness follows from the transitivity of permutation and the fact that list that sort to the same result are permutations of each other. The multiplicative depth follows from the  $\ell_{\rm id}$  multiplications used in Match and the additional  $v_j$  multiplication, which we can use divide and conquer to optimize. Note that since  $v_j$  is a vector this is actually many multiplications but these can be done in parallel. For the memoization of Match there is a real time-memory tradeoff. A reasonable choice here is to memoize the prefixes since those will change less often, which lets us easily remove older memoized values once they will no longer be used. With this choice we get  $n2^{\ell_{\rm id}} + 1$  total multiplications in the Match across all voters and **i**, an additional  $n\ell 2^{\ell_{\rm id}}$  multiplications from  $v_j$  and less than  $n\ell 2^{\ell_{\rm id}}$  additions from the sum.

With this RCT the multiplicative depth scales with the number of voters rather than the number of options. This could thus be more suitable than our first RCT for elections where there are orders of magnitude more options than voters. This RCT is also immediately linear in  $\ell$  without the need to resort to hash chains, and since we do not need binary votes there is a potential for fitting more options in less space. Additionally all additions in the RCT will only add up to at most 1, which means that we are free to use a low p.

## 4 RCT Homomorphic Encryption

To implement Randomized Circuit Trees for voting we will use a Homomorphic Encryption scheme with Verifiable Encryption and Verifiable Distributed Decryption, which we will call *RCT Homomorphic Encryption*. We use security notions that are near-identical to the security notions of the voting scheme.

Though the RCTs we constructed in section 3 are built with BGV in mind we do not we do not prove that a Homomorphic Encryption primitive like BGV can be used to construct a secure RCT Homomorphic Encryption Scheme. Future work may want to use a different scheme entirely.

**Definition 5.** A RCT Homomorphic Encryption Scheme for  $n_t$  parties is a set of circuits C, one key generation protocol and 5 algorithms:

- $\mathcal{KG}$  Trusted key generation outputs decryption keys  $\mathsf{sk}_0, \ldots, \mathsf{sk}_{n_t-1} \in \mathcal{SK}$ , an evaluation key  $\mathsf{evk} \in \mathcal{EVK}$ , a validation key  $\mathsf{vk} \in \mathcal{VK}$  and an encryption key  $\mathsf{pk} \in \mathcal{PK}$ .
- $\mathcal{E}$  Verifiable encryption takes the public key  $\mathsf{pk} \in \mathcal{PK}$ , a message  $v \in V$ and outputs a ciphertext  $c \in \mathcal{CT}$  and a proof  $\pi_c \in \Pi$
- $\mathcal{D}$  Verifiable partial decryption takes a decryption key  $\mathsf{sk}_i \in \mathcal{SK}$  and a ciphertext  $c \in \mathcal{CT}$  and outputs a partial decryption  $p_i \in V$  and a proof  $\pi_{p,i} \in \Pi$ .
- $\mathcal{H}$  Homomorphic evaluation takes an evaluation key  $\mathsf{evk} \in \mathcal{EVK}$ , a circuit  $\mathcal{C} \in \mathsf{C}$  and a ciphertext list  $\mathbf{c} \in \mathcal{CT}^{\#\mathsf{arg}\mathcal{C}}$  and outputs a ciphertext list  $y \in \mathcal{CT}$ .
- $\mathcal{V}$  Validation takes a validation key  $\forall \mathbf{k} \in \mathcal{VK}$ , a token  $t \in \{\mathcal{E}, \mathcal{D}\}$ , additional data dependent on the token and a proof  $\pi : \Pi$  and outputs  $\perp$  or  $\top$ .
- CS Decryption share combination takes a vector of  $p_i \in V$  and outputs a single value  $v \in V$ .

We use three different games for the security definitions that correspond to the scenarios where everyone is honest, where no one is honest and where one decryption key holder is honest. The two first games are in Fig. 1.

First we have the correctness game that supposes that everything validates and the output is computed according to the circuit if everyone is honest.

**Definition 6.** The Voting HE scheme is correct if for  $\mathbf{v}$  and  $\mathcal{C}$  such that  $|\mathbf{v}| = \# \arg(\mathcal{C})$  all the assertions in Correctness( $\mathcal{C}, \mathbf{v}$ ) always hold.

Next we have the integrity game, where the adversary wins if they can produce valid inputs that make decryption fail to validate or if every validation passes, but we still get the wrong result. If any of the assertions fail the adversary immediately loses.

Note that this definition requires the identity circuit, ID, to be in C.

Finally we have the privacy game in Fig. 2, which we frame as a distinguishing game. The adversary has access to an oracle,  $\mathsf{OPriv}_{b,\mathsf{pk},\mathsf{sk},\mathsf{evk}}$ , and can perform two kinds of queries, an encryption query which encrypts two votes and only returns one encryption according to the secret bit, b, and a evaluate and decrypt

$Correctness(\mathcal{C}, \mathbf{v})$		Integ	$\operatorname{grity}(\mathcal{A})$
1: sk, evk, pk, vk $\leftarrow \mathcal{KG}$		1:	$sk, evk, pk, vk \leftarrow \mathcal{KG}$
2: for $i \in [ \mathbf{v} ]$		2:	$\mathbf{c}, \pi_{\mathbf{c}}, \mathbf{p}, \pi_{\mathbf{p}}, \mathcal{C} \leftarrow \mathcal{A}(sk, evk, pk, vk)$
3: $c_i, \pi_{\mathbf{c},i} \leftarrow \mathcal{E}_{\mathcal{KG}}(v_i)$		3:	<b>assert</b> $\forall i,  \mathbf{c}  =  \pi_{\mathbf{c}}  =  p_i  = \# \operatorname{arg}(\mathcal{C})$
4: assert $\mathcal{V}_{vk,\mathcal{E}}(c_i,\pi_{\mathbf{c},i})$		4:	$\mathbf{assert}  \left  \mathbf{p} \right  = \left  \pi_{\mathbf{p}} \right  = n_t$
5: $\mathbf{y} \leftarrow \mathcal{H}_{evk}(\mathbf{c}, \mathcal{C})$		5:	$\mathbf{for}\;i\in[ \mathbf{c} ]$
6: for $i \in [ sk ]$		6:	$\mathbf{assert}  \mathcal{V}_{vk,\mathcal{E}}(c_i,\pi_{\mathbf{c},i})$
7: $p_i, \pi_{p,i} \leftarrow \mathcal{D}_{sk_i}(\mathbf{y})$		7:	$\mathbf{y} \leftarrow \mathcal{H}_{evk}(\mathbf{c}, \mathcal{C})$
8: assert $\mathcal{V}_{vk,\mathcal{D},i}(\mathbf{y},p_i,\pi_{p,i})$	)	8:	$\mathbf{for}\;i\in[n_t]$
9: assert $C(\mathbf{v}) = CS(\mathbf{p})$		9:	$\mathbf{if}  \mathcal{V}_{vk,\mathcal{D},i}(\mathbf{y},\mathcal{D}_{sk_i}(\mathbf{y}))$
L		10:	$\mathbf{return} \perp$
		11:	$\mathbf{assert}  \mathcal{V}_{vk,\mathcal{D},i}(\mathbf{y},p_i,\pi_{\mathbf{p},i})$
		12:	$\mathbf{y}' \gets \mathcal{H}_{evk}(\mathbf{c},ID)$
		13:	$\mathbf{for}i\in[ sk ]$
		14:	$p_i', \pi_{\mathbf{p}',i} \leftarrow \mathcal{D}_{sk_i}(\mathbf{y}')$
		15:	$\mathbf{return} \ \neg \mathcal{C}(\mathcal{CS}(\mathbf{p}')) = \mathcal{CS}(\mathbf{p})$

Fig. 1: Correctness and integrity games for RCT Homomorphic Encryption

Priv	$\operatorname{acy}(\mathcal{A})$	0
1:	$sk, evk, pk, vk \leftarrow \mathcal{KG}$	as
2:	$b \leftarrow \$ \{0,1\}$	fo
3:	$b' \leftarrow \mathcal{A}(sk_{\{2,\ldots\}},evk,pk,vk,OPriv_{b,pk,sk,evk})$	
4:	$\mathbf{return}   b = b'$	
	1: 2: 3:	$\begin{array}{l} \hline \text{Privacy}(\mathcal{A}) \\ \hline 1:  sk, evk, pk, vk \leftarrow \mathcal{KG} \\ 2:  b \leftarrow \$ \ \{0, 1\} \\ 3:  b' \leftarrow \mathcal{A}(sk_{\{2, \ldots\}}, evk, pk, vk, OPriv_{b, pk, sk, evk}) \\ 4:  \mathbf{return} \ b = b' \end{array}$

 $\mathsf{OPriv}_{b,\mathsf{pk},\mathsf{sk},\mathsf{evk}}(\mathcal{E},v_0,v_1)$ 

 $c_{0}, \pi_{c_{0}} \leftarrow \mathcal{E}_{\mathsf{pk}}(v_{0})$  $l_{0} \leftarrow l_{0} ||(c_{0}, \pi_{c_{0}})$  $c_{1}, \pi_{c_{1}} \leftarrow \mathcal{E}_{\mathsf{pk}}(v_{1})$  $l_{1} \leftarrow l_{1} ||(c_{1}, \pi_{c_{0}})$  $return c_{b}, \pi_{c_{b}}$ 

 $\mathsf{Priv}_{b,\mathsf{pk},\mathsf{sk},\mathsf{evk}}(\mathcal{D},\mathbf{c},\pi_{\mathbf{x}},\mathcal{C})$ ssert  $|\mathbf{c}| = |\pi_{\mathbf{c}}| = \# \operatorname{arg}(\mathcal{C})$ or  $i \in [|\mathbf{c}|]$ assert  $\mathcal{V}_{\mathsf{vk},\mathcal{E}}(c,\pi_{\mathbf{c},i})$ if  $(c_i, \pi_{\mathbf{c},i}) \in l_b$  $c_i', \pi_{\mathbf{c}',i} = l_{1-b,i}$ else $c_i', \pi_{\mathbf{c}',i} = c_i, \pi_{\mathbf{c},i}$  $c \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{c}, \mathcal{C})$  $c' \gets \mathcal{H}_{\mathsf{evk}}(\mathbf{c}', \mathcal{C})$ for  $i \in [|\mathsf{sk}|]$  $p_i, \pi_{p,i} \leftarrow \mathcal{D}_{\mathsf{sk}_i}(c)$ assert  $\mathcal{V}_{\mathsf{vk},\mathcal{D},i}(c,p_i,\pi_{p,i})$  $p'_i, \pi_{p',i} \leftarrow \mathcal{D}_{\mathsf{sk}_i}(c')$ assert  $\mathcal{V}_{\mathsf{vk},\mathcal{D},i}(c',p'_i,\pi_{p',i})$  $v, v' \leftarrow \mathcal{CS}(\mathbf{p}), \mathcal{CS}(\mathbf{p}')$ assert v = v'return  $\mathbf{p}, \pi_{\mathbf{p}}$ 

Fig. 2: Privacy game for RCT Homomorphic Encryption

query which evaluates a circuit on a list of ciphertexts and decrypts the result for the adversary if swapping the ciphertexts in the input that were returned from the encryption query would yield the same result. In the game the adversary is given all the keys except one of the secret keys, and can make as many queries as they want until they guess the secret bit.

Note that we do not capture attacks that involve asking for decryptions on circuits evaluated on non-fresh ciphertexts. Nor do we protect against replay attacks. This means that we always have to validate the freshness of the ciphertexts we homomorphically evaluate and decrypt, as well as remove duplicates. Techniques such as embedding identities into the zero knowledge proofs can be used with a modified definition.

If there is an assertion failure, we immediately end the game and flip a random coin to decide if the adversary wins or loses. The goal of the adversary is to do better than a random coin flip at guessing which bit the oracle has.

Using RCT Homomorphic Encryption we can perform distributed evaluation of circuit trees using the protocol in Fig. 3 while keeping inputs hidden and only revealing the hints. Here we use a Share sub-protocol, which collects a value from each of the protocol participants and returns it. This can be implemented with standard digital signatures and can be initialized as part of the trusted setup. The lemma follows directly from definitions.

```
\label{eq:constraint} \begin{split} \frac{\mathsf{HEEvalCircuitTree}_{i,\mathcal{C}}(\mathbf{x})}{\mathbf{h},n,j\leftarrow[],|x|,0}\\ \mathbf{loop}\\ \mathbf{y}\leftarrow\mathcal{H}_{\mathsf{evk}}(\mathbf{x},\mathcal{C}_n(\mathbf{h}))\\ \mathbf{p}_i,\pi_{\mathbf{p}_i}\leftarrow\mathcal{D}_{\mathsf{sk}_i}(\mathbf{y})\\ \mathbf{p}\leftarrow\mathsf{Share}_i(\mathbf{p}_i)\\ \mathbf{if}\ \mathcal{CS}(\mathbf{p})=[]\\ \mathbf{break}\\ \mathbf{h}\leftarrow\mathbf{h}\|\mathcal{CS}(\mathbf{p})\\ \mathbf{y}\leftarrow\mathcal{H}_{\mathsf{evk}}(\mathbf{x},\mathcal{V}_{\mathcal{C}}(\mathbf{h})_1)\\ \mathbf{p}_i,\pi_{\mathbf{p}_i}\leftarrow\mathcal{D}_{\mathsf{sk}_i}(\mathbf{y})\\ \mathbf{p},\pi_{\mathbf{p}}\leftarrow\mathsf{Share}_i(\mathbf{p}_i,\pi_{\mathbf{p}_i})\\ \mathbf{return}\ \mathbf{h},\mathbf{p},\pi_{\mathbf{p}} \end{split}
```

Fig. 3: Protocol for evaluating a circuit tree using a RCT HE Scheme

**Lemma 2.** When using a correct HE Scheme honestly executed HEEvalCircuitTree produces the same result as evaluating the circuit tree if the result is finite.

## 5 Cryptographic Voting

Before we can introduce our voting scheme we have to define what they are. We will be using an extractable voting scheme to simplify the security definitions and proofs. This corresponds to our choice for homomorphic encryption to always include the identity circuit.

**Definition 7.** An Extractable Voting Scheme is a tallying function f, one setup protocol, one casting algorithm, one tallying protocol, one verification algorithm and one extraction algorithm:

- Setup Trusted setup outputs tallying keys tk<sub>i</sub> ∈ TK, a validation key vk ∈ VK and a casting key ck ∈ CK.
- Cast Verifiable casting takes the casting key  $\mathsf{ck} \in \mathcal{CK}$ , a vote  $v \in \mathcal{V}$  and outputs a ballot  $c \in \mathcal{B}$  and a proof  $\pi_c \in \Pi$
- Tally Tallying takes a list of ballots  $c_i \in \mathcal{B}$  with proofs  $\pi_{c_i} \in \Pi$ , a partial tallying key  $\mathsf{tk}_i \in \mathcal{TK}$  and outputs a result  $r \in \mathcal{R}$  and a proof  $\pi_r$
- Extract Extraction takes all the tallying keys  $\mathsf{tk} \in \mathcal{TK}$ , a ballot  $c \in \mathcal{B}$ , a validation key  $\mathsf{vk} \in \mathcal{VK}$ , and outputs a vote  $v \in \mathcal{V}$  or  $\bot$ .
- Validate Validation takes a validation key vk  $\in \mathcal{VK}$ , a token  $t \in \{\text{Cast}, \text{Tally}\}$ , additional data dependent on the token and a proof  $\pi \in \Pi$  and outputs  $\perp/\top$ .

The correctness game (Fig. 4) supposes that honest execution passes all the validations, the ballots produced extract properly and the result is the same as we would get when using the counting function directly.

**Definition 8.** A Voting scheme is correct if all the assertions in  $Correctness(\mathbf{v})$  always pass for all lists of votes  $\mathbf{v}$ .

In the integrity game (4) the adversary wins if they can produce valid ballots that fail to extract, where an honest tally fails to validate or can use them to produce a validated tally result that is inconsistent with the counting function on the extracted votes.

The reason we need extractability is to be able to peek inside the ballots for the integrity game. Some definitions allow both honest and adversarially generated ballots and only require a result consistent with the honest votes. This does not work for our scheme because deciding whether the output of a generic tallying function is consistent with a certain partial input can be inefficient.

If any of the assertions fail the adversary immediately loses.

In the privacy game (5) the adversary wins if they can distinguish between tallies yielding same result involving one of two sets of votes.

Note that this definition does not capture replay attacks. We can protect against replay attacks by filtering equal votes before tallying. We could also prevent replay attacks by mixing authentication data with the casting proofs, but to simplify exposition we do not.

If there is an assertion failure, we immediately end the game and flip a random coin to decide if the adversary wins or loses. The goal of the adversary is to do better than a random coin flip at guessing which bit the oracle has.



Fig. 4: Correctness and integrity games for a voting scheme

$\operatorname{Privacy}(\mathcal{A})$		
1:	$tk, ck, vk \leftarrow \mathcal{KG}$	
2:	$b \leftarrow \$ \{0, 1\}$	
3:	$\mathbf{c}, \pi_{\mathbf{c}} \leftarrow \mathcal{A}(tk_{\{2,\ldots\}}, ck, vk, OVPriv_{\mathit{b}, ck})$	
4:	$\mathbf{assert}  \left  \mathbf{c} \right  = \left  \pi_{\mathbf{c}} \right $	
5:	$l_0, l_1 \leftarrow OVPriv_{b,ck}, \ \mathbf{c}' \leftarrow \mathbf{c}, \ \pi_{\mathbf{c}'} \leftarrow \pi_{\mathbf{c}}$	
6:	for $i \in [ \mathbf{c} ]$	
7:	$\mathbf{assert} \ Validate_{vk,Cast}(c_i,\pi_{c_i})$	
8:	$\mathbf{for} j \in [ l_b ]$	
9:	$\mathbf{if}(c_i,\pi_{\mathbf{c},i})=l_b[j]$	
10:	$(c'_i, \pi_{\mathbf{c}', i}) \leftarrow l_{b-1}[j]$	
11:	$r, \pi_r \leftarrow Tally_{tk_1}(l_b \  \mathbf{c}, \mathcal{A})$	
12:	$r', \pi_{r'} \leftarrow Tally_{tk_1}(l_{b-1} \  \mathbf{c}, \mathcal{A})$	
13:	assert Validate <sub>vk,Tally</sub> $(r, l_b \  \mathbf{c}, \pi_r)$	
14:	$\textbf{assert Validate}_{vk,Tally}(r',l_{b-1} \  \mathbf{c}, \pi_{r'})$	
15:	$\mathbf{assert} \ r = r'$	
16:	$b' \leftarrow \mathcal{A}(r, \pi_r, r', \pi_{r'})$	
17:	$\mathbf{return}   b = b'$	

 $\mathsf{OVPriv}_{b,\mathsf{ck}}(v_0,v_1)$ 

$$\begin{split} c_0, \pi_{c_0} &\leftarrow \mathsf{Cast}_{\mathsf{ck}}(v_0) \\ l_0 &\leftarrow l_0 \| (v_0, c_0, \pi_{c_0}) \\ c_1, \pi_{c_1} &\leftarrow \mathsf{Cast}_{\mathsf{ck}}(v_1) \\ l_1 &\leftarrow l_1 \| (v_1, c_1, \pi_{c_1}) \\ \mathbf{return} \ c_b, \pi_{c_b} \end{split}$$

Fig. 5: Privacy game for a voting scheme

All our voting schemes follow the same template. Given an opaque randomized circuit tree  $\mathcal{RC}$  implementing the tallying function and a RCT Homomorphic Encryption Scheme  $\mathcal{HE} = (\mathcal{KG}, \mathcal{E}, \mathcal{D}, \mathcal{H}, \mathcal{V}, \mathcal{CS})$  capable of evaluating all the circuits in  $\mathcal{RC}$  and the ones used in HEGetRandom, we build the voting scheme as described in Fig. 6.

Setup	$Cast_{ck}(v)$
$sk, evk, pk, vk \leftarrow \mathcal{KG}$	$\mathbf{c}, \pi_{\mathbf{c}} \leftarrow \mathcal{E}_{pk}(v)$
$\mathbf{b} \gets \!\!\!\! s  \delta$	return c, $\pi_c$
$\mathbf{d}, \pi_{\mathbf{d}} \leftarrow \mathcal{E}_{pk}(\mathbf{b})$	
$tk_i \gets sk_i, evk, \mathbf{d}$	$Tally_{tk_i}(\mathbf{c})$
$ck \gets pk$	$\mathbf{h}, \mathbf{p}, \pi_{\mathbf{p}} \leftarrow HEEvalCircuitTree_{i, \mathcal{C}}(\mathbf{d} \  \mathbf{c})$
$vk \gets evk, vk, \mathbf{d}$	$r \leftarrow g(\mathbf{h})$
${f return}$ tk, ck, vk	$\pi_r \leftarrow \mathbf{h}, \mathbf{p}, \pi_\mathbf{p}$
	return $r, \pi_r$
$Extract_{tk}(c)$	$Validate_{vk,Cast}(c,\pi_c)$
$\mathbf{y} \leftarrow \mathcal{H}_{evk}(c,ID)$	$\textbf{return} \; \mathcal{V}_{vk,\mathcal{E}}(c,\pi_c)$
for $i \in [ tk ]$	
$p_i, \pi_{\mathbf{p},i} \leftarrow \mathcal{D}_{i,tk_i}(\mathbf{y})$	Validate <sub>vk,Tally</sub> ( $\mathbf{c}, r, \pi_r$ )
$\mathbf{assert}  \mathcal{V}_{vk,\mathcal{D},i}(p_i,\pi_{\mathbf{p},i},\mathbf{y})$	$\mathbf{y} \leftarrow \mathcal{H}_{evk}(\mathbf{d} \  \mathbf{c}, \mathcal{V}_{\mathcal{C}}(\mathbf{h})_1)$
$v \leftarrow \mathcal{CS}(\mathbf{p})$	for $i \in [n_t]$
	/ -
$\pi_v \leftarrow \mathbf{p}, \pi_\mathbf{p}$	$\mathbf{p}, \pi_{\mathbf{p}} \leftarrow \pi_{r,i}$
$\pi_v \leftarrow \mathbf{p}, \pi_\mathbf{p}$ return $v, \pi_v$	$\mathbf{p}, \pi_{\mathbf{p}} \leftarrow \pi_{r,i} \ \mathbf{assert} \ \mathcal{V}_{vk,\mathcal{D},i}(\mathbf{p},\pi_{\mathbf{p}},\mathbf{y})$

Fig. 6: Voting Scheme Template

The proofs for the following claims can be found in appendix C. They are trivial because when substituting the tally the games are nearly identical, except that for the voting scheme the adversary we are more restricted in which circuits can be used.

**Theorem 4.** The voting scheme is correct if the RCT Homomorphic Encryption Scheme is correct.

**Theorem 5.** Given an adversary  $\mathcal{A}$  against the integrity game for the voting scheme, there exists an adversary against the integrity game for the RCT Homomorphic Encryption Scheme, with equal or greater advantage.

**Theorem 6.** Given an adversary  $\mathcal{A}$  with advantage A against the privacy game for the voting scheme there exists a trivial wrapper adversary using  $\mathcal{A}$  with at

least as much advantage against the privacy game for the RCT Homomorphic Encryption Scheme.

#### 6 Instantiating the HE Scheme

Our main goal with this section is to demonstrate viability. Choosing optimal parameters for leveled homomorphic encryption with BGV is an active research area [8], and combining this with verifiable encryption and decryption does not make it easier. We leave optimal parameter selection to future work.

As mentioned we will be using BGV [4] for encryption. We refer to Aranha *et al.* [1] for how to do verifiable encryption and verifiable distributed decryption. For our comparison we use 11000 voters as in Black *et al.* [2]. This means that our schemes using counting will need to instantiate BGV with a plaintext ring of characteristic at least 11000. We choose 65537. We will provide parameters for votes with 256 bits of entropy, which is enough for most names and can fit instant runoff voting votes with  $c_p = 16$  and  $\ell = 4$ . Our generic RCT using counting is woefully inefficient in the worst case when every voter votes differently, so we omit its analysis, though it could be interesting to investigate whether this limitation matters in practice. For the shuffling RCT we use  $\ell_{id} = 14$ , which is enough for 11000 voters.

Using the parameters for multiplicative depth 9 from Gouert *et al.* [8], which uses a ring of characteristic 65537 we get 58  $\mu$ s per multiplication when amortizing over the 3840 slots, and since our RCTs are highly parallel we can run most operations over multiple slots. This gives us around  $1.48 \cdot 10^9$  multiplications a day, and additions are faster than multiplications. The parameter choice uses  $\mathbb{F}_{65537^6}$  as the field for the circuits, which gives each slot an entropy of over 96 bits, so  $\ell = 3$  allows 288 bits of entropy for our shuffling RCT.

The ciphertexts with this parameter set have a size of  $23040 \cdot 360$  bits, or 1.04 MB. The decryption proofs can be batched by using a random linear combination of the ciphertexts with their decryptions subtracted, so the size of this is negligible compared to the size of the ballots. The proofs of valid encryption are around the same order of magnitude as the ciphertexts.

Counting function	entropy	mult. depth	
Instant Runoff	256	6	$158 \times 10^{6}$
Generic w/ Shuffling	$\sim 288$	5	$1442 \times 10^6$

Fig. 7: Efficiency estimates for some of our voting RCTs

In conclusion, these schemes yield runtime and artifact size that are large but practical. We improve on Black *et al.* [2] by providing a lattice based voting scheme that can handle instant runoff voting with 16 candidates and more.

#### References

- Diego F. Aranha, Carsten Baum, Kristian Gjøsteen, and Tjerand Silde. "Verifiable Mix-Nets and Distributed Decryption for Voting from Lattice-Based Assumptions". In: ACM CCS 2023: 30th Conference on Computer and Communications Security. Ed. by Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda. Copenhagen, Denmark: ACM Press, Nov. 2023, pp. 1467–1481. DOI: 10.1145/3576915.3616683.
- [2] Ian Black, Emma McFall, Juliet Whidden, Bryant Xie, and Ryann Cartor. Practical Quantum-Safe Voting from Lattices, Extended. Cryptology ePrint Archive, Report 2022/1686. 2022. URL: https://eprint.iacr.org/2022/ 1686.
- [3] Xavier Boyen, Thomas Haines, and Johannes Mueller. Epoque: Practical End-to-End Verifiable Post-Quantum-Secure E-Voting. Cryptology ePrint Archive, Report 2021/304. 2021. URL: https://eprint.iacr.org/2021/ 304.
- [4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ITCS 2012: 3rd Innovations in Theoretical Computer Science*. Ed. by Shafi Goldwasser. Cambridge, MA, USA: Association for Computing Machinery, Jan. 2012, pp. 309–325. DOI: 10.1145/2090236.2090262.
- [5] Gizem S. Çetin, Yarkin Doröz, Berk Sunar, and Erkay Savas. "Depth Optimized Efficient Homomorphic Sorting". In: Progress in Cryptology -LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America. Ed. by Kristin E. Lauter and Francisco Rodríguez-Henríquez. Vol. 9230. Lecture Notes in Computer Science. Guadalajara, Mexico: Springer, Cham, Switzerland, Aug. 2015, pp. 61–80. DOI: 10.1007/978-3-319-22174-8\_4.
- [6] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "A Homomorphic LWE Based E-voting Scheme". In: Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016. Ed. by Tsuyoshi Takagi. Fukuoka, Japan: Springer, Cham, Switzerland, Feb. 2016, pp. 245– 265. DOI: 10.1007/978-3-319-29360-8\_16.
- [7] Rafaël del Pino, Vadim Lyubashevsky, Gregory Neven, and Gregor Seiler.
  "Practical Quantum-Safe Voting from Lattices". In: ACM CCS 2017: 24th Conference on Computer and Communications Security. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1565–1581. DOI: 10.1145/3133956. 3134101.
- [8] Charles Gouert, Rishi Khan, and Nektarios Georgios Tsoutsos. Optimizing Homomorphic Encryption Parameters for Arbitrary Applications. Cryptology ePrint Archive, Report 2022/575. 2022. URL: https://eprint.iacr. org/2022/575.
- [9] Nicolas Huber et al. "Kryvos: Publicly Tally-Hiding Verifiable E-Voting". In: ACM CCS 2022: 29th Conference on Computer and Communications Security. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine

Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 1443–1457. DOI: 10.1145/3548606.3560701.

- Guillaume Kaim, Sébastien Canard, Adeline Roux-Langlois, and Jacques Traoré. "Post-quantum Online Voting Scheme". In: FC 2021 Workshops. Ed. by Matthew Bernhard et al. Vol. 12676. Lecture Notes in Computer Science. Virtual Event: Springer, Berlin, Heidelberg, Germany, Mar. 2021, pp. 290–305. DOI: 10.1007/978-3-662-63958-0\_25.
- [11] Kim Ramchen, Chris Culnane, Olivier Pereira, and Vanessa Teague. "Universally Verifiable MPC and IRV Ballot Counting". In: FC 2019: 23rd International Conference on Financial Cryptography and Data Security. Ed. by Ian Goldberg and Tyler Moore. Vol. 11598. Lecture Notes in Computer Science. Frigate Bay, St. Kitts and Nevis: Springer, Cham, Switzerland, Feb. 2019, pp. 301–319. DOI: 10.1007/978-3-030-32101-7\_19.

### A Untrusted Setup

While our scheme uses trusted setup practical applications often require untrusted setup. We show how to efficiently perform the setup in an untrusted setting.

"Verifiable Mix-Nets and Distributed Decryption for Voting from Lattice-Based Assumptions" [1] already touches upon how to perform distributed setup for BGV. The interesting part of the setup is the part that samples an encrypted random element from the distribution of the RCT. For many distributions, such as the distribution used in our shuffle scheme, we can do rejection sampling homomorphically to hide the random element from the trustees given that at least one trustee is honest. We can also use homomorphic encryption with zero knowledge proofs to prove that the resulting ciphertext is an encryption of an element in the support.

We start with showing how to do rejection sampling with circuits. With our RCT for shuffling in mind we also want to sample the bits independently and reject based on previously sampled bits because this allows us to sample one bitstring in the list of distinct bitstrings at a time and reject if there is a collision.

**Definition 9.** A Rejection Sequence S is a finite sequence of circuits taking a strictly increasing number of arguments and returning a single boolean result such that there exists a **b** such that  $S_0(\mathbf{b}) = 1$  and for any **b** such that  $S_{i-1}(\mathbf{b}) = 1$ , there exists a **b**' such that  $S_i(\mathbf{b}||\mathbf{b}') = 1$ . A rejection sequence S defines the following algorithm for rejection sampling using a random tape r;

```
\begin{aligned} & \frac{\text{RejectionSample}_{\mathcal{S}}(r)}{i, l, \mathbf{b} \leftarrow 0, 0, []} \\ & \text{while } i < |\mathcal{S}| \\ & \mathbf{b}' \leftarrow \text{get}_r(\# \text{arg}(\mathcal{S}_i) - l) \\ & \text{if } \mathcal{S}_i(\mathbf{b} \| \mathbf{b}') = 1 \\ & i, l, \mathbf{b} \leftarrow i + 1, \# \text{arg}(\mathcal{S}_i), \mathbf{b} \| \mathbf{b}' \end{aligned}
return \mathbf{b}
```

We say that the rejection sequence implements the distribution induced by this algorithm. A rejection sequence is uniform if for all  $S_i$  and  $\mathbf{b} \in \{0, 1\}^{\# \operatorname{arg}(S_{i-1})}$  such that  $S_{i-1}(\mathbf{b}) = 1$  the number of  $\mathbf{b}'$ -s such that  $S_i(\mathbf{b} || \mathbf{b}') = 1$  is independent of  $\mathbf{b}$ . For uniform rejection sequences we define the accept probability of S at  $i, \mathcal{P}_S^i$ , as the proportion of  $\mathbf{b}'$  where this happens.

Note that the support of the distribution induced by S is the set of **b** such that  $S_i(\mathbf{b}) = 1$  for all *i*.

**Theorem 7.** A uniform rejection sequence implements a uniform distribution.

*Proof.* For a uniform rejection sequence the probability of sampling any given element is either zero or  $\prod_i \mathcal{P}_S^i$ 

**Theorem 8.** For a uniform rejection sequence S, the distribution of outputs of RejectionSample<sub>S</sub> and the distribution of outputs of the internal  $S_i$  calls are independent.

*Proof.* The outputs of the internal  $S_i$  calls are 1 with probability  $\mathcal{P}_{S}^{i}$  and 0 otherwise. This probability only depends on the index, which depends only on the output of previous  $S_i$  calls.

This tells us that the outputs of S calls in a uniform rejection sequence do not leak anything about the output other than that it is in the support.

Using these rejection sequences we can generate random input for circuit trees.

### A.1 Validation

Similarly to with RCTs we can validate the output of a rejection sequence by checking that a value is in its support instead of reusing the rejection sequence to recompute it.

**Definition 10.** Let S be a rejection sequence, then a circuit C with #arg(S) arguments and one output validates S if it outputs 1 precisely for the support of S.

We can always construct a validator by combining S with ValidateBin. We use  $\mathcal{V}_S$  for an arbitrary validator for S.

#### A.2 Homomorphic Evaulation

To homomorphically evaluate rejection sequences we first need to evaluate a random tape to use as input for the rejection sequence. The random tape works by having all the parties XOR random binary inputs.

Because computing XOR adds multiplicative depth we reset the final bitstring by creating a fresh sharing of it. This does mean that we don't have a sampled value, but rather a sharing of a sampled value, and the RCTs need their circuits augmented to add together the shares first.

> $XOR(\mathbf{a}, \mathbf{b})$  $\mathsf{HEGetRandom}_{i,\mathsf{pk},\mathsf{vk},\mathsf{sk}_i,\mathsf{evk}}(n)$  $\mathbf{b}_i \leftarrow \{0,1\}^n$ for  $i \in [|\mathbf{a}|]$  $\mathbf{c}_i, \pi_{\mathbf{c}_i} \leftarrow \mathcal{E}_{\mathsf{pk}}(\mathbf{b}_i)$  $c_i \leftarrow a_i + b_i - 2 * a_i * b_i$  $\mathbf{c}, \pi_{\mathbf{c}} \leftarrow \mathsf{Share}_i(\mathbf{c}_i, \pi_{\mathbf{c}_i})$ return c for  $i \in [|\mathbf{c}|]$ assert  $\mathcal{V}_{\mathsf{vk},\mathcal{E}}(\mathbf{c}_i,\pi_{\mathbf{c},i})$  $\mathbf{r} \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{r}, \mathbf{c}_i, \mathsf{XOR})$  $\mathbf{e}_i \leftarrow V$  $\mathbf{d}_i, \pi_{\mathbf{d}_i} \leftarrow \mathcal{E}_{\mathsf{pk}}(\mathbf{e_i})$  $\mathbf{d}, \pi_{\mathbf{d}} \leftarrow \mathsf{Share}_i(\mathbf{d}_i, \pi_{\mathbf{d},i})$ for  $\mathbf{d}_i \in \mathbf{d}$ assert  $\mathcal{V}_{\mathsf{vk},\mathcal{E}}(\mathbf{d}_i, \pi_{\mathbf{d}_i})$  $\mathbf{r} \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{r}, \mathbf{d}_i, \mathsf{SUB})$  $\mathbf{p}_i, \pi_{\mathbf{p}_i} \leftarrow \mathcal{D}_{\mathsf{sk}_i}(\mathbf{r})$  $\mathbf{p}, \pi_{\mathbf{p}} \leftarrow \mathsf{Share}_i(\mathbf{p}_i, \pi_{\mathbf{p}_i})$ for  $\mathbf{p}_i \in \mathbf{p}$ assert  $\mathcal{V}_{\mathsf{vk},\mathcal{D},i}(\pi_{\mathbf{p}_i},\mathbf{p}_i,\mathbf{r})$  $\mathbf{x} \leftarrow \mathcal{CS}(\mathbf{p})$ return  $\mathbf{d}, \mathbf{x}, \pi_{\mathbf{d}}$

**Theorem 9.** With a correct HE scheme and honest key generation all the assertions in an honestly executed HEGetRandom succeed and it produces plaintexts  $\mathbf{x}$ , ciphertexts  $\mathbf{d}_i$  and encryption proofs  $\pi_{\mathbf{d},i}$  such that the encryption proofs validate the  $\mathbf{d}_i$ -s and summing  $\mathbf{x}$  and the  $e_i$  used to get the  $\mathbf{d}_i$ -s element wise yields a n-length bitstring  $\mathbf{b}$ .

*Proof.* The assertions involving  $\mathcal{V}$  succeed because the HE scheme is correct and all the inputs are honestly generated. For the rest we need to keep track of the plaintexts used to generate the ciphertexts. The  $b_i$  are random *n*-length bitstrings, so the  $\mathbf{c}_i$  are *n*-length encryptions of random bitstrings. The plaintext in  $\mathbf{r}$ , let us call this  $\mathbf{b}$ , is the XOR of the random bitstrings, which is also a *n*length random bitstring. For the rest we can verify that  $\mathbf{x} = \mathbf{r}' - \sum_i \mathbf{e}_i$  so  $\mathbf{x} + \sum_i \mathbf{e}_i = \mathbf{b}$ , which is a *n*-length bitstring. **Theorem 10.** If every party except one in HEGetRandom samples  $\mathbf{b}_i$  non-uniformly and the output validates, then the output encodes a uniformly random bitstring.

*Proof.* If the xor of the semi-honest parties is binary, then xoring this with a uniform bitstring yields a uniform bitstring. If the xor is not binary, then using the xor circuit on it and a bitstring cannot produce a bistring, so the validation is guaranteed to fail. If the validation succeeded then we must get a uniform bitstring.

Using the random tape, a rejection sequence and an RCT HE we build a rejection sampling protocol, HERejectionSample in Figure 8.

```
\mathsf{HERejectionSample}_{i,\mathcal{S},\mathsf{pk},\mathsf{vk},\mathsf{sk}_i,\mathsf{evk}}
j \leftarrow 0
\mathbf{d} \leftarrow []
\pi_{\mathbf{d}} \leftarrow []
\mathbf{x} \leftarrow []
 while j < |\mathcal{S}|
         \mathbf{d}', \mathbf{x}', \pi_{\mathbf{d}'} \leftarrow \mathsf{HEGetRandom}_{i,\mathsf{pk},\mathsf{vk},\mathsf{sk}_i,\mathsf{evk}}(\#\mathsf{arg}\mathcal{S}_j)
         \mathbf{y} \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{d}', \mathbf{x}', \mathsf{ADD})
         \mathbf{y} \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{y}, \mathcal{S}_j)
         \mathbf{p}_i, \pi_{\mathbf{p}_i} \leftarrow \mathcal{D}_{\mathsf{sk}_i}(\mathbf{y})
         \mathbf{p}, \pi_{\mathbf{p}} \leftarrow \mathsf{Share}_i(\mathbf{p}_i, \pi_{\mathbf{p}_i})
         for \mathbf{p}_k \in \mathbf{p}
                 assert \mathcal{V}_{\mathsf{vk},\mathcal{D},i}(\mathbf{p}_k,\pi_{\mathbf{p}_k},\mathbf{y})
         \mathbf{v} \leftarrow \mathcal{CS}(\mathbf{p})
         if \mathbf{v} = 1
                j \leftarrow j + 1
                 \mathbf{d} \leftarrow \mathbf{d} \| \mathbf{d}'
                 \pi_{\mathbf{d}} \leftarrow \pi_{\mathbf{d}} \| \pi_{\mathbf{d}'}
                 \mathbf{x} \gets \mathbf{x} \| \mathbf{x}'
\mathbf{y} \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{d}, \mathbf{x}, \mathcal{V}_\mathcal{S})
\mathbf{p_i}, \pi_{\mathbf{p_i}} \leftarrow \mathcal{D}_{\mathsf{sk}_i}(\mathbf{y})
 \mathbf{p}, \pi_{\mathbf{p}} \leftarrow \mathsf{Share}_i(\mathbf{p}_i, \pi_{\mathbf{p}_i})
return d, x, p, \pi_d, \pi_p
```

Fig. 8: Homomorphic Rejection Sampling

**Lemma 3.** With a correct HE scheme and honest key generation all the assertions in an honestly executed HERejectionSample succeed and it produces plaintexts  $\mathbf{x}$ , ciphertexts  $\mathbf{d}_i$  and encryption proofs  $\pi_{\mathbf{d},i}$  such that the encryption proofs

validate the  $\mathbf{d}_i$  and summing  $\mathbf{x}$  and the  $\mathbf{e}_i$  used to get  $\mathbf{d}_i$  element wise yields a bitstring  $\mathbf{b}$  such that evaluating  $S_i$  on prefixes of  $\mathbf{b}$  always returns 1

*Proof.* From Theorem 9 and its proof we know that all the assertions involving  $\mathcal{V}$  will succeed and that for each iteration of the while loop there exist plaintexts  $\mathbf{e}'_i$  encrypting to the  $\mathbf{d}'_i$  and a  $\# \arg \mathcal{S}_j$ -length bitstring  $\mathbf{b}'$  such that  $\mathbf{x}' + \sum_i \mathbf{e}'_i = \mathbf{b}'$ . For the final assertion and properties of the outputs we proceed by induction on the length of  $\mathcal{S}$ . In the base case everything is length zero, so the assertion at the end is trivial, and the lemma holds vacuously.

For the induction step the induction hypothesis on S with the last circuit removed means that before the last iteration of the while loop the proofs  $\pi_{\mathbf{d}}$  are valid for  $\mathbf{d}$  and there exists a binary vector  $\mathbf{b}$  satisfying the lemma on all but the last element of S.

Since honest evaluation and decryption corresponds to evaluating the circuit on the underlying plaintext  $\mathbf{v} = 1$  iff  $S_j(\mathbf{b} \| \mathbf{b}') = 1$ . After executing the contents of the if statement we have  $\mathbf{x} + \sum_i \mathbf{e}_i \| \mathbf{e}'_i = \mathbf{b} \| \mathbf{b}'$  where  $\mathbf{e}_i \| \mathbf{e}'_i$  encrypts to  $\mathbf{d}_i$ .

**Corollary 1.** In the above lemma  $\mathcal{V}_{\mathcal{S}}(\mathbf{b}) = \mathbf{1}$ , and  $\mathbf{p}$  are shares combining to  $\mathbf{1}$ .

By using HERejectionSample we can do distributed setup with the same primitives and trust assumptions we need anyways.

#### **B** RCT for Limited Voting

Limited Voting is an electoral system where the voters get to vote for a small set of different options at once. The voter submits a list of votes. The result is a list of options with the number of votes they got.

Options are encoded the same way as our first RCT, except now each vote will have a  $c_p$  length list of distinct options. We leak information about the votes with invalid encoding. Formally our counting function produces a sorted list of the nonbinary votes from voters that did not vote for the same option multiple times in addition to some information about votes from voters with duplicated or non-binary votes.

As with the first RCT the we start by filtering out voters with non-binary votes using ValidateBin. The next step is to check for duplicated votes among the votes of each of the remaining voters. To do this we use the following circuit to check for equality of  $\ell$ -length bitstrings;

$$\frac{\mathsf{Eq}(\mathbf{v}, \mathbf{v}')}{\mathsf{return}} \prod_{i \in [\ell]} 1 - (v_i - v_i')^2} \quad \frac{\mathsf{CheckDup}(\mathbf{v})}{\mathsf{return}} \sum_{0 \le i < j < |\mathbf{v}|} \mathsf{Eq}(v_i, v_j)$$

The circuit outputs 1 if they are equal and 0 if they are not. It needs  $4\ell - 1$  circuit operations and has multiplicative depth  $\lceil \log(\ell) \rceil + 1$ .

We then use this circuit to check that all pairs of votes of a given voter are distinct. For efficiency we instead count the number of pairs that are equal, which reveals information about the votes of voters with invalid votes but gives the same result for all voters with valid votes. **Lemma 4.** CheckDup has multiplicative depth  $\lceil \log \ell \rceil + 1$  and uses  $2\ell c_p(c_p-1)-1$  operations.

*Proof.* The multiplicative depth comes solely from Eq, which multiplies  $\ell$  values that have already been multiplied once. The number of operations can be easily counted.

It is also easy to parallelize across different voters.

After filtering non-binary votes and removing duplicates we can concatenate the votes from each individual voter and count them as in Section 3.1.

**Theorem 11.** The count described above has multiplicative depth  $\lceil \log \ell \rceil$  and uses at most  $\hat{v}\ell(\lceil \log \ell \rceil nc_p + nc_p - 1)$  circuit operations.

*Proof.* This follows immediately from instantiating Count with  $c_p n$  votes.

As with generic method the validation circuits are dwarfed by the final counting circuits for reasonable elections.

Looking at just the cost of counting we can compare the efficiency of the specialized limited voting RCT to what we would get by lifting to the generic RCT by considering lists of options as options in their own right, which yields multiplicative depth  $\lceil \log c_p \ell \rceil$  and operation count at most  $\hat{v}\ell c_p(\lceil \log \ell c_p \rceil n + n - 1)$ . The multiplicative depth of the specialized method is significantly better. The operation count is also better but the comparison somewhat misleading since the number of unique votes,  $\hat{v}$ , will be different for both counting methods. With the specialized method it is bounded by  $\min(n, 2^{\ell+c_p})$ , which makes the specialized method much more suited for elections with the number of options on the order of the number of voters. The specialized method might also have more duplicated votes in practice.

Similarly to the generic method we can construct a validator that only checks the binarity and lack of duplicates and matches with the full options that are actually present in the votes.

## C Proofs from Section 5

**Theorem 12.** If the underlying HE scheme is correct, then the voting scheme is also correct.

*Proof.* Correctness of the underlying HE scheme guarantees that all honestly generated ciphertexts validate, all honestly generated decryptions from those ciphertexts validate and that the result after decryption is the same as the circuit applied to the plaintexts used to generate the encryptions.

We investigate the assertions in the correctness game for the voting scheme.

The cast validation just validates the honest encryption, so it succeeds. The extract validation checks that we get back the underlying vote when we use the identity circuit, which succeeds. The tally validation checks that the decryptions computed in the tally are valid, which succeed. It also checks that the validator validates the output, which succeeds from the definition of validator and the computation of  $\mathcal{CS}(\mathbf{p})$ , and  $g(\mathbf{h}) = r$ , which is trivial in the honest case. The final assertion in the game checks that the result of the tally corresponds to the tallying function, which succeeds because the result of the tally comes from evaluating a circuit tree that evaluates the tallying function.

IntegrityReduction<sub>A</sub>(sk, evk, pk, vk)

$$\begin{split} \mathbf{b} &\leftarrow \$ \, \delta \\ \mathbf{d}, \pi_{\mathbf{d}} \leftarrow \mathcal{E}_{\mathsf{pk}}(\mathbf{b}) \\ \mathsf{tk}_i \leftarrow \mathsf{sk}_i, \mathsf{evk}, \mathbf{d} \\ \mathsf{ck} \leftarrow \mathsf{pk} \\ \mathsf{vk} \leftarrow \mathsf{evk}, \mathsf{vk}, \mathbf{d} \\ \mathbf{c}, \pi_{\mathbf{c}}, \pi_r \leftarrow \mathcal{A}(\mathsf{tk}, \mathsf{ck}, \mathsf{vk}) \\ \mathsf{for} \, i \in [|\mathbf{c}|] \\ \quad \mathsf{if} \, \mathsf{Extract}_{\mathsf{tk}}(c_i) = \bot \\ \mathbf{y} \leftarrow \mathcal{H}_{\mathsf{evk}}(\mathbf{c}, \mathcal{C}) \\ \quad \mathsf{for} \, i \in [|\mathsf{sk}|] \\ p_i, \pi_{\mathbf{p},i} \leftarrow \mathcal{D}_{\mathsf{sk}_i}(\mathbf{y}) \\ c_i, \pi_{\mathbf{c},i}, \mathbf{p}, \pi_{\mathbf{p}}, \mathsf{ID} \\ \mathsf{return} \, \mathbf{d} \| \mathbf{c}, \pi_{\mathbf{d}} \| \pi_{\mathbf{c}}, \mathbf{p}, \pi_{\mathbf{p}}, \mathcal{V}_{\mathcal{C}}(\mathbf{h})_1 \end{split}$$



**Theorem 13.** Given an adversary  $\mathcal{A}$  with advantage A against the integrity game for the voting scheme IntegrityReduction<sub> $\mathcal{A}$ </sub><sup>1</sup> is an adversary with at least as much advantage against the integrity game for the RCT Homomorphic Encryption Scheme.

*Proof.* Consider the case where  $\mathcal{A}$  wins the game. In the first case they win by making extraction fail, which means that the ballot that made extraction fail was a valid encryption. In this case IntegrityReduction<sub> $\mathcal{A}$ </sub> wins by making honest decryption of evaluated identity circuit fail to validate. In the second case  $\mathcal{A}$  wins by making the honest tally result fail to validate, in which case IntegrityReduction<sub> $\mathcal{A}$ </sub> wins by making honest decryption of the evaluated validation circuit fail to validate. In the final case  $\mathcal{A}$  wins by making the final result inconsistent with the result, at which point all the decryptions and encryptions are valid. In this case IntegrityReduction<sub> $\mathcal{A}$ </sub> wins by making the final equality in the game not hold. Since  $\mathcal{A}$  winning always means that IntegrityReduction<sub> $\mathcal{A}$ </sub> wins we can deduce that the latter has at least as much advantage.

 $<sup>^{1}</sup>$  from Figure 9

**Theorem 14.** Given an adversary  $\mathcal{A}$  with advantage A against the privacy game for the voting scheme there exists a trivial wrapper adversary using  $\mathcal{A}$  with at least as much advantage against the privacy game for the RCT Homomorphic Encryption Scheme.

*Proof.* We do not provide an explicit reduction this time. The idea is the similar as with integrity. Use the game for the RCT Homomorphic Encryption Scheme to generate the inputs for  $\mathcal{A}$  by performing most of the honest setup. Use the oracle to do the tally and encryptions, and find that the resulting adversary always wins when  $\mathcal{A}$  does.