Complex Elections via Threshold (Fully) Homomorphic Encryption

Charlotte Bonte[1], Georgio Nicolas[2], and Nigel P. Smart[1,2]

[1] Zama, Paris, France.
[2] COSIC, KU Leuven, Leuven, Belgium.
charlotte.bonte@zama.ai,
georgio.nicolas@esat.kuleuven.be,
nigel.smart@kuleuven.be/nigel@zama.ai.

Abstract. We discuss how Fully Homomorphic Encryption (FHE), and in particular the TFHE scheme, can be used to define an e-voting scheme for the Alternative Vote (AV) election system. This system has a more complex tallying phase than traditional First-Past-The-Post (FPTP) election variants. Previous work on e-voting schemes that used homomorphic encryption has focused on FPTP systems only, and utilised mainly linearly homomorphic encryption. We show, by using FHE, that more complex electoral systems such as AV can also be supported by homomorphic encryption. We show security of our protocol by considering it as a simple MPC functionality, and we also show practicality by presenting some experimental runtimes using the **tfhe-rs** library.

Table of Contents

1 Introduction

Using Threshold Linear Homomorphic Encryption (TLHE) to enable electronic voting has a long history. The classical blueprint was defined by Cramer et al in [CFSY96], where the requirements of *universal verifiability*, *privacy*, and *robustness* were set out. The work of [CFSY96] built upon earlier work of [Ben87, BY86, CF85, SK94].

The basic template for such voting systems is as follows. Voters *encode* and then encrypt their ballot using a linearly homomorphic encryption scheme, along with a zero-knowledge proof of *both* correctness of the vote format *and* knowledge of the underlying encryption. Then the ballots are "added up", after the associated zero-knowledge proof is checked, using the linearly homomorphic property of the underlying encryption scheme. Finally, the tally is decrypted using the threshold decryption property of the TLHE scheme. If the threshold decryption operation is *verifiable* then the entire voting scheme is verifiable, as any voting party can check that their ballot has been counted, and that the tally is correct.

Using pre-quantum linearly homomorphic encryption schemes such as Paillier encryption [Pai99] or ElGamal "in the exponent" one can implement such an election relatively efficiently. Indeed the popular Helios voting system [Adi08, AdMP] is based on this paradigm. There have been some works which extend this paradigm to the post-quantum setting (relying on lattice based assumptions) such as [BHM21, CGGI16b, HSS24, KCRT21, dPLNS17].

The encoding step is particularly important and in some sense limits the applicability of the methodology. It is perhaps illustrative to explain the encoding methodology for a standard First-Past-The-Post (FPTP) election amongst n candidates with v voters. Firstly a linear homomorphic encryption scheme is chosen with message space which allows addition of integer values (with no wrap-around) up to size $N > B^{\mathsf{n}}$ where B is some integer such that $B > v$.

A vote is encoded as an encryption of an element in the set $\{1, B, B^2, \ldots, B^{\mathsf{n}-1}\}$. Note, that only n elements out of the total space of N plaintext elements are valid encodings of votes. When the ballots are added up one obtains an encryption of an integer

$$T = v_1 + v_2 \cdot B + \cdots + v_{\mathsf{n}} \cdot B^{\mathsf{n}-1} \tag{1}$$

where $v_1 + v_2 + \cdots v_{\mathsf{n}} = v$ and hence $T < B^{\mathsf{n}} < N$. By decrypting T using the threshold decryption protocol one can obtain the number of voters v_1 for candidate one, the number of voters v_2 for candidate two, the number of voters v_3 for candidate three, and so on. In such an FPTP election it is considered standard that the values $(v_1, v_2, \ldots, v_{\mathsf{n}})$ are made public, and not just the eventual winner.

Zero-knowledge proofs are needed for three reasons:

1. To ensure independence of inputs. This protects against, say, a dishonest player re-randomizing an honest player's ballot in order to deduce information about that player's vote from the final tally.
2. To ensure that dishonest players have actually encrypted their input correctly, i.e. that the ciphertext is indeed a valid ciphertext. This is important as, for many homomorphic encryption schemes, it is impossible to determine if a ciphertext actually encodes *any* message just from looking at it.
3. To ensure that the ballot encodes a valid vote. The latter property is important, as demonstrated above, when the message space of the encryption scheme is much larger than the space which encodes a single vote.

This voting mechanism is relatively fast, easy to implement, and works well for FPTP elections with around 10 candidates and voters in the few thousands. However, the encoding of the final tally via equation (1) requires the underlying encryption scheme to support a plaintext space of integers modulo $N = O(v^{\mathsf{n}})$. If a Paillier based scheme is used the plaintext space can be extended using the techniques of [DJ02, DJ03]. However, if ElGamal "in the exponent" is used then in order to obtain the tally T one needs to execute a work effort of $\sqrt{N}$, in order to extract the tally from the exponent. In such a situation decoding using a work effort of $\sqrt{N}$ soon becomes impractical for reasonable values of v and n.

1.1 Beyond First-Past-The-Post

However, as soon as more complex election methodologies are required this simple encoding mechanism quickly breaks down. It is well known that FPTP elections have a number of drawbacks[3]. In particular, they are not very expressive of voters' intentions, and can lead to outcomes which the majority of electors disagree with. There are many different ways to solve the problems that come with FPTP. One solution could be the introduction of more proportional methodologies in multi-outcome elections (for example when electing a group of people from different constituencies for a Parliament). Alternatively, voters could be allowed to express more complex voting intentions. In this paper, we examine in detail one such simple solution: the so-called Alternative Vote (AV) system (in the US this is called the Ranked-Choice Voting (RCV) system with "instant run-off"). Nevertheless, our discussion and methodology can be applied to many similar systems.

In a simple AV system, a voter produces a list which ranks possible election outcomes, instead of choosing a single outcome. To simplify our discussion, algorithms, and analysis, we do not allow under-voting (each voter ranks all candidates), nor do we allow voters to equal match candidates. Thus, a ballot consists of a ranked list of *all* the candidates. In other words, a ballot is a permutation on the set of candidates. If an election has n candidates, then the total number of possible votes is then $\mathsf{n}!$, which can grow rather large as n increases.

In an AV election round, one first considers all the first choice votes of all electors. If a single candidate obtains more than one-half of the votes, then that candidate is selected as the winner. Otherwise, the candidate with the least number of votes is eliminated, and the voters which chose that candidate as their top preference have this choice replaced by their next preferred candidate. The process is repeated until one candidate has a majority of the votes. Thus, if a winner is not selected in round one, then one passes to round two. As there are n candidates in total, a winner must be found after $\mathsf{n} - 1$ rounds[4]. The public outcome of the election is (usually) not just the winner, but the various tallies of elector choices for each of the rounds of the election, until the winner is found. In other words, the sum of the votes for all the candidates in round one, i.e. the sum of the votes of all first preference votes. Then the sum of the votes for all candidates in round two (if there is one), i.e. the sum of the votes for preferred candidates after one candidate has been eliminated and so forth. We call such intermediate tallies the "partial tallies" in what follows.

The above illustrates AV in the case where just one candidate is elected. The case where multiple winners are selected is more complicated[5]. In the case of elections with multiple winners, we need

[3] See for example `https://aceproject.org/main/english/es/` for a summary of the advantages and disadvantages of various election systems.

[4] With some form of decision making process made for ties in both the final round, and in the elimination stages.

[5] Often a multiple winner AV system, as we describe it, is called a Single Transferable Vote (STV) system. It should be noted that some authors refer to what we call AV as STV, and reserve AV to denote a single winner system in which each voter only casts first and second preference votes, and not third, fourth, fifth, etc. preferences.

to replace the quota of receiving one-half of the votes with another quota (the minimum number of votes that each of the candidates must receive to be elected, and to exclude the possibility of any additional candidate being elected by the remaining votes). In addition, if one candidate is selected with a number of votes greater than the quota, the surplus number of votes that candidate received has to be distributed over the next preferred candidates. This must be done in a manner to ensure that only the fraction of the votes needed to reach the quota goes to the elected candidate and the remaining part of the vote of that ballot goes to the next preferred candidate on that ballot. In this paper, we concentrate solely on single-winner AV elections in order to remove the complexity of dealing with surplus votes.

A single-winner AV election can be carried out by re-purposing our simple FPTP system described above. We simply encode each permutation as a single possible vote, and thus the total number of (virtual) candidates becomes $\mathsf{n}!$. However, this requires us to assume $N > B^{\mathsf{n}!}$, which becomes prohibitive when n becomes more than a handful of candidates. For example, if we repurposed Helios to deal with an AV system then a single vote, for five candidates, would require over one MByte of data, however in our system such a vote only requires 19 KBytes. In addition the information leakage from the final tally, when repurposing an FPTP system, is more than the leakage that arises from a standard AV election. More specifically one determines the number of voters who prefer each of the $\mathsf{n}!$ possible orderings.

Thus, for complex election methodologies, such as AV, something more elaborate than using linear homomorphic encryption is needed, since the tallying mechanism is inherently non-linear in nature. Luckily, such a technology exists, called *Fully Homomorphic Encryption* (FHE).

1.2 Fully Homomorphic Encryption

Since its creation by Gentry [Gen09], the development of FHE has been rapid. There are now a plethora of schemes to choose from: BGV [BGV12], BFV [Bra12, FV12], CKKS [CKKS17], GSW [GSW13], FHEW [DM15], and TFHE [CGGI16a, CGGI20, BdBB$^+$25], to name but a few. Each scheme comes with its own advantages and disadvantages, and particular applications for which it is well suited. In this work, we concentrate on the TFHE scheme, as it supports the computation of arbitrary depth functions quite efficiently.

Current FHE schemes are now fast enough to be deployed in real life scenarios. In particular bootstrapping is no longer a bottleneck. For example, bootstrapping for the TFHE encryption scheme can be done in under 20 milliseconds [CJL$^+$20]. In addition, distributed decryption can be done in either a few milliseconds or around half a second [DDK$^+$23] (depending on the type of FHE scheme, the parameters and the number of parties).

1.3 Our Contribution

We present a Threshold FHE (ThFHE) based protocol which implements an AV election, based on a threshold version of the TFHE scheme [CGGI16a, CGGI20, BdBB$^+$25]. Our protocol not only outputs the winner, but also the results of partial tallies at the different rounds of an election, until the winner is found. By basing our methodology on ThFHE, we are able to easily address the non-linearity inherent to the tallying of AV elections.

Our protocol follows very closely the blueprint of Cramer et al in [CFSY96]: Ballots are encrypted and proven correct, then a homomorphic tally is performed, and finally the result is revealed. However, we differ from previous techniques in that we do not use zero-knowledge proofs to prove

the correct *encoding* of a vote. We use zero-knowledge proofs to only prove correct *encryption*, hence plaintext knowledge, and to ensure independence of inputs. We propose three methodologies to ensure that the encoding of a vote is correct:

- The first method requires interaction between the tally centres to show that a ballot is valid. In particular this interactive method requires $2 \cdot n$ threshold decryptions to be performed per vote.
- The other two methods utilise the fully homomorphic encryption scheme, and ensure that **any** plaintext encrypted by the voter is guaranteed to correspond to a valid vote once homomorphically decoded.

The choice between which methodology is more efficient, or practical, in any given instantiation will depend on various factors; thus we leave the precise choice of methodology to the reader.

1.4 Prior Use of TFHE in E-Voting

In [CGGI16b], the authors utilise the TFHE-related scheme called FHEW [DM15] to implement a Fast-Past-The-Post e-voting scheme. This scheme is very different from our approach. In particular the scheme in [CGGI16b] dispenses with zero-knowledge proofs of knowledge in order to "prove correctness" of ballots. Instead, the authors rely on the fact that even if one bootstraps an invalidly formed ciphertext then the output is a valid ciphertext. Thus, by bootstrapping the authors claim that all input ciphertexts can be made to be correct, even in the presence of dishonest voters.

Whilst this intuition is technically correct, it does not take into account the other reason why homomorphic encryption based voting schemes make use of zero-knowledge proofs. To obtain simulation security any simulator needs to be able to extract the underlying messages encrypted by a dishonest voter. Thus, the knowledge extractor, implicit in the proof of knowledge definition, is needed in order to prove simulation security. In terms of the underlying ideal functionalities inherent in such simulation-based proofs, this is captured by the property that the adversarial inputs need to be independent of the honest parties' inputs. This technical proof artifact may seem to be obscure, but without it, there are simple attacks which can be mounted.

Imagine we have three voters, and one voter (voter C) is dishonest. If there were no zero-knowledge proofs of knowledge used, voter C could take voter A's ballot, re-randomise it, and then submit it as their own. The (public) result of the election would be the candidate which voter A voted for, since this candidate would now have at least two votes. This would strictly be more information than could be determined if voter C's input was independent of voter A's input. Homomorphic voting schemes always allow a form of re-randomization (by simply adding an encryption of zero to any voting ciphertext), thus this attack is always possible. However, by insisting that each voter appends a zero-knowledge proof of not only the correct encryption, but also knowledge of the underlying plaintext, such attacks are prevented.

One could obtain independence of inputs by enabling a commit and open methodology to the ballots. Thus, voting proceeds in two phases: In the first phase ALL voters commit to their homomorphically encrypted ballot, and in the second phase the commitments are opened. This does enable independence of inputs, but does not allow extraction of the dishonest parties' votes, which would be needed to produce a valid simulation of the final tally in a security proof. For example, if voter C voted independently for candidate one, and then in the final tally produced by the simulation, candidate one obtained no votes, then voter C would know it was working in a

simulation. For the simulation to be perfect, the simulator needs a method to extract the plaintext underlying voter C's ballot.

In our protocol we utilise a simulation-based security definition, and the above problems are excluded by the use of applying zero-knowledge proofs of knowledge.

1.5 Other Methodologies

We end this background overview by noting that there are other methodologies which can enable electronic voting, without the need for homomorphic encryption. A popular alternative mechanism is to use Mix-Nets. When using Mix-Net based voting, ballots are first encrypted and then batches of ballots are shuffled via a Mix-Net in order to remove the linkage between the ballot and the voter. Finally the ballots are opened, via threshold decryption, and tallied. Again this strategy can be implemented in both pre-quantum [Nef01] or post-quantum [ABG+21, ABGS23] settings.

However, this method leaks more than the partial tallies at each stage of the tallying process, since it leaks the content of all of the ballots; but not the linkage between a voter and a ballot. This creates an avenue for voter coercion when combining the Mix-Net approach with AV voting, see [BMN+09] for a discussion of this. The paper [BMN+09] then goes on to consider a combination of linearly homomorphic encryption, along with a Mix-Net style approach, in order to prevent this coercion. The authors of [BMN+09] esitmate that with their method an election with five candidates per region and 3 million voters divided into eight regions (i.e. about 375,000 voters per region) would take 10,000 hours to compute the final eight results.

Our use of Fully Homomorphic Encryption enables both coercion resistance and a significant decrease both in the amount of data which needs to be stored on the bulletin board compared to the method in [BMN+09], but also a significant reduction in the computational time needed for elections. In our experiments we can cope with a five candidate election with 10,000 voters in about 1000 seconds. This means the above eight elections with 375,000 voters would be expected to take

$$8 \cdot \frac{375000}{10000} \cdot \frac{1000}{60 \cdot 60} \approx 83.3$$

hours.

2 Preliminaries

In recapping the following preliminaries, which are all building blocks on which we build our voting protocol, the reader should keep in mind the following variables and notations.

- The number of voters is denoted by v.
- The number of candidates in the election will be denoted with n.
- There is a set $\mathbb{T}$ of T tallying parties, of which at most t are assumed corrupt.

For a set S, we denote by $a \leftarrow S$ the process of drawing a from S with a uniform distribution on the set S. If D is a probability distribution, we denote by $a \leftarrow D$ the process of drawing a with the given probability distribution. For a probabilistic algorithm A, we denote by $a \leftarrow A$ the process of assigning a the output of algorithm A, with the underlying probability distribution being determined by the random coins of A.

2.1 Fully Homomorphic Encryption

We consider an FHE scheme with plaintext space a ring $\mathcal{R}$, which one can think of either as a finite field $\mathbb{F}$ or a finite ring such as $\mathbb{Z}/(p^k)$ (with $\mathbb{Z}/(2^k)$ being a ring of particular interest). We assume (for simplicity) that the integer zero is encoded to the additive identity (i.e. zero) in $\mathcal{R}$, and the integer one is encoded to the multiplicative identity (i.e. one) in $\mathcal{R}$.

An FHE scheme, with plaintext space a ring $\mathcal{R}$, is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add}, \mathsf{Mult})$ and a space $\mathcal{E}$ of "valid" encryptions; where the set $\mathcal{E}$ is a subset of a larger set $\mathbb{E}$. Detecting membership of $\mathbb{E}$ can be done publicly, detecting membership of $\mathcal{E}$ publicly is difficult. These algorithms have the following signatures:

- $\mathsf{KeyGen}(1^\kappa)$: On input of the security parameter κ, this randomised algorithm produces a public/private key pair $(\mathfrak{pk}, \mathfrak{sk})$.
- $\mathsf{Enc}(m, \mathfrak{pk}; r)$: On input of $m \in \mathcal{R}$, a public key, and randomness from a space of random coins Coins, this produces a ciphertext $\mathfrak{ct} \in \mathcal{E}$.
- $\mathsf{Dec}(\mathfrak{ct}, \mathfrak{sk})$: On input of an element $\mathfrak{ct} \in \mathcal{E}$ and a secret key $\mathfrak{sk}$ this outputs the corresponding plaintext $m \in \mathcal{R}$.
- $\mathsf{Add}(\mathfrak{ct}_1, \mathfrak{ct}_2, \mathfrak{pk})$: On input of two elements $\mathfrak{ct}_1, \mathfrak{ct}_2 \in \mathcal{E}$ which decrypt to $m_1, m_2 \in \mathcal{R}$ this deterministically produces a ciphertext $\mathfrak{ct}_3 \in \mathcal{E}$ which decrypts to $m_1 + m_2$.
- $\mathsf{Mult}(\mathfrak{ct}_1, \mathfrak{ct}_2, \mathfrak{pk})$: On input of two elements $\mathfrak{ct}_1, \mathfrak{ct}_2 \in \mathcal{E}$ which decrypt to $m_1, m_2 \in \mathcal{R}$ this deterministically produces a ciphertext $\mathfrak{ct}_3 \in \mathcal{E}$ which decrypts to $m_1 \cdot m_2$.

The correctness conditions of an FHE scheme are obvious; i.e. every element in $\mathcal{E}$ should be decryptable to the correct value.

To ease notation, we write $\mathsf{Add}(\mathfrak{ct}_1, \mathfrak{ct}_2, \mathfrak{pk})$ as $\mathfrak{ct}_1 + \mathfrak{ct}_2$ and $\mathsf{Mult}(\mathfrak{ct}_1, \mathfrak{ct}_2, \mathfrak{pk})$ as $\mathfrak{ct}_1 \cdot \mathfrak{ct}_2$. We also assume that scalars (i.e. non-encrypted values in $\mathcal{R}$) can be added and multiplied into ciphertexts at will. Thus one can form arbitrary arithmetic expressions combining ciphertexts and plaintexts, with the output being a ciphertext if any of the input variables are ciphertexts.

In practice, we will use the TFHE scheme, as implemented in the tfhe-rs library[6]. To aid interpretation of the benchmarking results, a bit of background on the data representation in tfhe-rs is given here. The tfhe-rs library uses an encoding where space for the encrypted value is reserved in the most significant bits and the random coins of the encryption in the least significant bits. The space reserved for the encrypted value is split up into a carry subspace and a message subspace. The message subspace is the space reserved for the messages of freshly encrypted ciphertexts, usually this is set to two bits per ciphertext. Reserving some extra space for carry propagation allows to perform some linear operation without the need for a PBS, usually the space for carry propagation is also set to two bits. A schematic representation is given in Figure 1[7].

As such, the base plaintext space is $\mathbb{Z}/(2^4)$; with each ciphertext encoding a two bit plaintext, along with a potential two-bit carry. This means an encryption of a $2 \cdot \ell$-bit integer is performed using ℓ ciphertexts. In order to deal with tallyies of at most v binary values, we will require $v < 2^{2 \cdot \ell}$, i.e. we can perform exact arithmetic on integers of size bounded by v.

When encrypting a uint8 integer value to obtain an encrypted FheUint8 value, the encrypter obtains four ciphertexts. Each ciphertext encrypting two bits of the original uint8 integer. Integer addition and multiplication of integer values, and all other homomorphic operations are carried out via a limited number of additions of ciphertexts followed by a so called Programmable Bootstrapping

[6] Available from `https://github.com/zama-ai/tfhe-rs`.
[7] Figure taken from [BdBB+25].

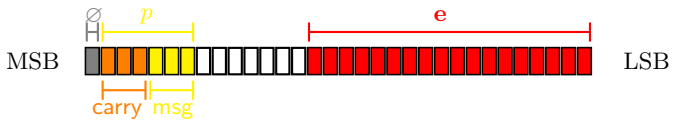

Fig. 1. The gray bit is a padding bit required for technical reasons for the evaluation of look-up tables in the PBS. The space reserved for the encrypted value held by a ciphertext is given by p (orange+yellow). This space is split up into a message subspace msg (yellow) where the freshly encrypted values are put and a carry subspace carry (orange) reserved to put the bits of the carry propagation of linear operations. The random coins of the encryption are represented by e (red).

(PBS) operation. Using the TFHE PBS operation, one can, for example, also homomorphically perform bit extraction on the bit-representation of the integers in one operation. To extract all bits, one can execute the PBS operations in parallel. See [BdBB+25] where this operation is denoted by[8]

$$\{\mathfrak{ct}_i\}_{i=0}^{\mathsf{length}-1} = \mathsf{ExtractBits}\left(\mathfrak{ct}, 0, \mathsf{length}\right),$$

which extracts the first length bits of the integer held in ciphertext $\mathfrak{ct}$.

We also define a CMux operation which, given an encrypted bit $\mathfrak{ct}_b$ and two ciphertexts $\mathfrak{ct}_x$ and $\mathfrak{ct}_y$, will produce the ciphertext

$$\mathsf{CMux}(\mathfrak{ct}_x, \mathfrak{ct}_y, \mathfrak{ct}_b) = (\mathfrak{ct}_y - \mathfrak{ct}_x) \cdot \mathfrak{ct}_b + \mathfrak{ct}_x.$$

This outputs an encryption of the value encrypted by $\mathfrak{ct}_x$ if the bit encrypted by $\mathfrak{ct}_b$ is zero, otherwise it outputs an encryption of the value encrypted by $\mathfrak{ct}_y$. Using the CMux operation, we can define a conditional swap operation, which we denote by $\mathsf{Swap}(\mathfrak{ct}_x, \mathfrak{ct}_y, \mathfrak{ct}_b)$. The swap operation executes the following three operations in sequence

$$\begin{aligned} t &\leftarrow \mathfrak{ct}_x, \\ \mathfrak{ct}_x &\leftarrow \mathsf{CMux}(\mathfrak{ct}_x, \mathfrak{ct}_y, \mathfrak{ct}_b), \\ \mathfrak{ct}_y &\leftarrow \mathsf{CMux}(\mathfrak{ct}_y, t, \mathfrak{ct}_b). \end{aligned}$$

The above syntax for Enc, Add, Mult etc, can be extended to emulate encryption, decryption and operations on $2 \cdot \ell$-bit integers, in which case we consider the plaintext space to be $\mathcal{R}^\ell$. We will abuse notation in this way to aid readability in what follows.

2.2 Distributed Decryption

Formally we define threshold decryption via two ideal functionalities. The first $\mathcal{F}_{\mathsf{KeyGen}}$, in Figure 2, acts as a set-up assumption for our protocol. This is needed to enable a UC proof of security of the threshold decryption. $\mathcal{F}_{\mathsf{KeyGen}}$ generates a key pair, and secret shares the secret key among the T vote tallying parties $\mathbb{T}$ using a linear secret sharing scheme $\langle\cdot\rangle$ which tolerates up to t adversaries. Party $\mathcal{T}_i$'s share of a secret shared value x is denoted by $\langle x\rangle_i$.

One can realise this functionality using a generic MPC protocol. Note that, despite wanting active security, the ideal functionality does not "complete" adversarially input shares into a complete sharing (as is often done in such situations). Thus any implementing protocol for $\mathcal{F}_{\mathsf{KeyGen}}$ does not need to take as input the adversarial shares of the secrets.

[8] The second argument to $\mathsf{ExtractBits}\left(\mathfrak{ct}, 0, \mathsf{length}\right)$ is technical, and can be ignored in our application.

$\mathcal{F}_{\mathsf{KeyGen}}$

$\mathsf{Init}(\mathsf{sid})$:

1. Execute $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$ for the underlying FHE encryption scheme.
2. Generate a secret sharing $\langle \mathfrak{sk} \rangle$ of the secret key amongst the T players in $\mathbb{T}$.
3. Send $(\mathsf{sid}, \mathfrak{pk})$ to all players (including the adversary), and send $(\mathsf{sid}, \langle \mathfrak{sk} \rangle_i)$ to player $\mathcal{T}_i$ (including adversarially controlled players).

Fig. 2. The ideal functionality for distributed key generation

The key functionality we want to implement is $\mathcal{F}_{\mathsf{KeyGenDec}}$ given in Figure 3. Note that this functionality always returns the correct result, irrespective of what the adversary does. A protocol which realises $\mathcal{F}_{\mathsf{KeyGenDec}}$ in the $\mathcal{F}_{\mathsf{KeyGen}}$-hybrid model is given in [DDK+23] when $t < T/3$. Here the distributed decryption protocol can be executed with TFHE in under one second. We note that the distributed decryption operation in [DDK+23] is not publicly verifiable, in that an external party can not verify that the distributed decryption was performed correctly. Whilst theoretically possible, such public verifiability would be very expensive in terms of computational resources required to implement.

If (efficient) public verifiability is required then the threshold decryption protocol from [ABGS23] can be deployed. This latter protocol works when $t < T$, but does not provide robustness; namely if a party misbehaves then the protocol simply aborts. In addition the described implementation is only efficient in an amortised setting, where lots of threshold decryptions need to be performed at once. A recent publicly verifiable robust protocol, see [BFM+25], can deal with up to $t < 2 \cdot T/3$ where $t + 1$ honest parties are needed for threshold decryption to work. However, this third option has not been fully implemented, and thus its behaviour in practical applications is currently unknown.

$\mathcal{F}_{\mathsf{KeyGenDec}}$

$\mathsf{Init}(\mathsf{sid})$:

1. Execute $(\mathfrak{pk}, \mathfrak{sk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$ for the underlying FHE encryption scheme.
2. Send $(\mathsf{sid}, \mathfrak{pk})$ to all players, including the adversary and store the value $\mathfrak{sk}$.

$\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$: For a ciphertext $\mathfrak{ct} \in \mathcal{E}$

1. Compute $m \leftarrow \mathsf{Dec}(\mathfrak{ct}, \mathfrak{sk})$.
2. Output $(\mathsf{sid}, \mathfrak{ct}, m)$ to all players in $\mathbb{T}$.

Fig. 3. The ideal functionality for distributed key generation and decryption

2.3 ZKPoKs

We will apply ZKPoKs to prove the specific NP-relation R_{enc} given by

$$R_{\mathsf{enc}} = \left\{ (\mathfrak{ct}, (m, r)) \ : \ m \in \{0,1\}^{\mathsf{len}}, \quad r \in \mathsf{Coins}, \quad \mathfrak{ct} = \mathsf{Enc}(m, \mathfrak{pk}; r) \right\},$$

with len being the length of the message m as specified in Section 3.1 for our three vote encoding methods. The ZKPoK proves that ciphertexts are validly formed. The zero-knowledge proofs will be defined by two algorithms.

- $\mathsf{Prv}^{\mathsf{enc}}(\mathfrak{ct}, (m, r))$: A prover algorithm which, on input of a statement $\mathfrak{ct}$ and a witness (m, r) for the relation R_{enc}, will produce a proof π.
- $\mathsf{Ver}^{\mathsf{enc}}(\mathfrak{ct}, \pi)$: A verifier algorithm which, on input of a statement $\mathfrak{ct}$ and a proof π, will output true or false.

For the FHE schemes mentioned earlier (BGV, BFV, CKKS and TFHE) the encryption algorithm Enc is of the following form: It generates some "small" random integer values (which one can consider as sums of bits, where the bits are part of the random coins in r), then two values $(\mathbf{a}, \mathbf{b})$ are constructed which are *linear combinations* of the random bits, the message, and the values in the public key $\mathfrak{pk}$. The final ciphertext is the pair $(\mathbf{a}, \mathbf{b})$. One can therefore interpret the NP-relation R_{enc} as a (multiple) subset-sum relation over the hidden bits in the random coins r and the bits making up the message m. Thus the post-quantum techniques given in [FMRV22, FR23] can be applied directly to this situation resulting in a very simple proof technique. Both these papers are based on specializations of the general KKW [KKW18] method for MPC-in-the-Head; and utilise the random oracle model.

As our implementation is based on the tfhe-rs library we will instead utilise the pre-quantum (i.e. pairing-based) proofs based on vector commitment schemes which are contained in this library. Note, that the pre-quantum assumptions only affect the soundness property of the proofs, and so our overall voting still is still post-quantum resistant to harvest-now/decrypt-later attacks (since the votes are encrypted using a cryptosystem based on variants of LWE, which is a post-quantum assumption). The zero-knowledge proofs in tfhe-rs are described in [Lib24] and [BEL$^+$24], and rely on the Common Reference String (CRS) model, the Random Oracle Model (ROM) and the Algebraic Group Model (AGM) [FKL18]. In order to simulate the proofs, the simulator makes use of a trapdoor in the CRS, whereas the knowledge extractor makes use of either the ability to program the random oracles, or the ability to observe the group oracle queries in the underlying AGM.

The zero-knowledge proof does not require operations on the encrypted value, hence there are no operations that will fill the carry subspace. Thus, the tfhe-rs library is able to amortise the proof of correct encryption of many ciphertexts into a smaller number of proofs; thus aiding efficiency. Going back to our discussion above, re-encrypting a uint8 value to obtain an encrypted FheUint8 value. Instead of proving knowledge/valid encryption of the four ciphertexts which encode a FheUint8 value, the tfhe-rs library instead compresses the four ciphertexts into two, and then provides a proof of correctness of the two ciphertexts. Upon verification, the two ciphertexts are then expanded (using two parallel PBS operations) into the four ciphertexts representing a FheUint8 value. Due to this compression, the prover time depends mainly on the CRS size, which is the extra public data that needs to be generated for the proof [9]. However, the downside is that verification times increase as one encrypts larger and larger integers, since we need to decompress from the representation used in the proof. Further compression is achieved on encryption by using the property of the public key encryption method from [Joy24], which enables the encryption of many payloads using a single Ring-LWE "mask"-value.

[9] For more details on the proof and the corresponding CRS, see Section 4 of [Lib24].

2.4 Permutations

A permutation is a bijection from the set $\{1, \ldots, \mathsf{n}\}$ to itself, sometimes represented in the following notation

$$\begin{pmatrix} 1 & 2 & 3 & \ldots & \mathsf{n} \\ U_1 & U_2 & U_3 & \ldots & U_{\mathsf{n}} \end{pmatrix}$$

It is common to represent such a permutation σ as a product of disjoint cycles. For example, the mapping on $\{1, 2, 3, 4, 5\}$ given by $1 \to 2 \to 3$ and $4 \to 5$, i.e. the permutation

$$\begin{pmatrix} 1\,2\,3\,4\,5 \\ 2\,3\,1\,5\,4 \end{pmatrix},$$

can be represented by

$$\sigma = (1, 2, 3)(4, 5).$$

Another standard representation of a permutation is as a matrix in which each row and column has exactly one element; a so-called permutation matrix. With this notation we have the above permutation being represented by

$$\sigma = \begin{pmatrix} 0\,1\,0\,0\,0 \\ 0\,0\,1\,0\,0 \\ 1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,1 \\ 0\,0\,0\,1\,0 \end{pmatrix}$$

as $\sigma \cdot (1, 2, 3, 4, 5)^{\mathsf{T}} = (2, 3, 1, 5, 4)$. It is well known that a permutation can be also written as a product of transpositions, however this representation is not unique. For example, we have $\sigma = (1, 2)(2, 3)(4, 5) = (1, 3)(1, 2)(4, 5)$.

The following Lemma will be important for one of our encoding of votes methodologies

Lemma 2.1. *All permutations can be written in the following form*

$$\prod_{j=2}^{\mathsf{n}} (j, k_j)$$

for some values $k_j \in \{1, \ldots, j\}$.

Proof. Read the sequence of transpositions from the right to the left, i.e. in the order that the permutation is applied. The rightmost transposition will place the correct value into position n. The penultimate transposition places the correct value into position $\mathsf{n} - 1$, and since the correct position has already been placed in position n this transposition will be between $j = \mathsf{n} - 1$ and a value $1 \leq k_j \leq \mathsf{n} - 1$. Continuing in this way, the Lemma follows. □

As an example, we can write the permutation $(1462)(53)$ as $(21)(41)(53)(64)$.

Note that if $k_j = j$, then we can drop this permutation from the product. This means that we can write every permutation as

$$\prod_{j=2}^{\mathsf{n}} \left(\prod_{k=1}^{j-1} (j, k)^{b_{j,k}} \right)$$

where $b_{j,k} \in \{0,1\}$. Thus, we have a redundant encoding of every permutation as a sequence of $\sum_{j=2}^{\mathsf{n}}(j-1) = \frac{\mathsf{n}\cdot(\mathsf{n}-1)}{2}$ bits. This might not seem that interesting as our encoding in n^2 bits via a permutation matrix is unique. However, being able to map any set of $\frac{\mathsf{n}\cdot(\mathsf{n}-1)}{2}$ bits into a permutation means that we always output a permutation upon decoding an integer of this encoding, whilst for the encoding directly via the permutation matrix, i.e. via n^2 bits, most sequences of bits do not correspond to a valid permutation matrix.

Note, there is a similar (but more compact) encoding of permutations into numbers via a Lehmer Code (see [Leh60] and [Knu97][Page 66, Algorithm P]). This encodes a permutation as an integer in the range $[0,\ldots,\mathsf{n}!)$, and thus requires $\lceil \log_2(\mathsf{n}!) \rceil$ bits to represent it. The encoding algorithm from a permutation given in the form

$$\begin{pmatrix} 1 & 2 & 3 & \ldots & \mathsf{n} \\ U_1 & U_2 & U_3 & \ldots & U_{\mathsf{n}} \end{pmatrix}$$

to an integer, and the decoding algorithm from the integer to a permutation matrix representing the same permutation are given in Figure 4. The function $\mathsf{quo\text{-}rem}(n,d)$ used in decoding returns the quotient q and remainder r upon Euclidean division of n by d, i.e. $n = q \cdot d + r$ with $r \in [0,\ldots,d)$. Note that Lehmer-Decode will return a valid permutation matrix, even if the input value x is not in the range $[0,\ldots,\mathsf{n}!)$.

Lehmer Encoding and Decoding

Lehmer-Encode$(U_1,\ldots,U_{\mathsf{n}})$:

On input of a permutation as a sequence $(U_1,\ldots,U_{\mathsf{n}})$ of distinct elements, this outputs an integer in $[0,\ldots,\mathsf{n}!)$.

1. $r \leftarrow \mathsf{n}$, $x \leftarrow 0$.
2. While $r > 1$ do:
   (a) $s \leftarrow \arg\max(U_1,\ldots,U_r)$.
   (b) $x \leftarrow r \cdot x + s - 1$.
   (c) Swap U_r and U_s.
   (d) $r \leftarrow r - 1$.
3. Return x.

Lehmer-Decode(x):

On input of an integer $x \in [0,\ldots,\mathsf{n}!)$ this outputs a permutation matrix P representing the permutation encoded by x.

1. Let P denote the identity matrix of size n.
2. For $r \in [2,\ldots,\mathsf{n}]$ do:
   (a) $(x,s) \leftarrow \mathsf{quo\text{-}rem}(x,r)$.
   (b) Swap columns r and $s+1$ in matrix P.
3. Return P.

Fig. 4. Lehmer Encoding and Decoding

3 Our AV Protocol

The blueprint for TLHE-based voting protocols of [CFSY96] comes in three phases. In the first phase, the tally centres $\mathbb{T}$ set up their threshold secret key, and distribute the public key to voters via a bulletin board. In the second phase, a ballot is encrypted and posted on the bulletin board along with an associated proof of correctness (and knowledge) of the encryption and encoding of the vote. In the third phase the votes are tallied and the result is decrypted using the threshold decryption of the TLHE scheme.

In our protocol, we proceed however in four phases. This is because we do not utilise zero-knowledge proofs in order to prove correctness of the *encoding* of the vote. We only use zero-knowledge proofs to provide correctness of the encryption (and knowledge of the underlying plaintext). Thus, we use an additional phase which guarantees the correctness of the encoding.

We provide three methods to demonstrate correctness of encoding; one which is interactive and requires $2 \cdot \mathsf{n}$ threshold decryptions per vote, the other two are non-interactive and make use of the underlying FHE scheme we use (namely TFHE), as well as Lemma 2.1/Lehmer decoding. We start by explaining our vote encoding methods, and then go on to discuss the four phases of our protocol.

3.1 Vote Encoding

Each voter's vote V can be encoded as an $\mathsf{n} \times \mathsf{n}$ binary permutation matrix $\sigma_1 = \{v_{j,k}\}_{j=1,k}^{\mathsf{n}}$, with $v_{j,k} \in \{0,1\}$. The first column has a one in the row corresponding to the voter's first preference, the second column has a one in the row corresponding to the second preference and so on. As explained in the discussion following Lemma 2.1, we can also express σ as a product of transpositions of the form

$$\sigma_2 = \prod_{j=2}^{\mathsf{n}} \left(\prod_{k=1}^{j-1} (j,k)^{b_{j,k}} \right)$$

where $b_{j,k} \in \{0,1\}$. The two representations are related by the relation

$$\sigma_1 = \sigma_2 \cdot I_{\mathsf{n}}$$

where each transposition (i,j) in σ_2 acts on the matrix on its right via column transposition. Finally, one can also represent the vote V as a sequence of distinct integers $(u_1, \ldots, u_{\mathsf{n}})$, as also explained in the preliminaries section. The encoding of a vote will use one of these three methodologies, however when tallying all ballots we will use the (encrypted) representation as a permutation matrix.

Our three encoding methods take a vote V and then encode it as an integer in the range $[0, \ldots, 2^{\mathsf{n}^2-1}]$ (resp. $[0, \ldots, 2^{\mathsf{n}\cdot(\mathsf{n}-1)/2-1}]$ or $[0, \ldots, \mathsf{n}!]$) via the maps

$$\begin{aligned}
\mathsf{encode\text{-}v1}(V) &= \sum_{j=1}^{\mathsf{n}} \sum_{k=1}^{\mathsf{n}} v_{j,k} \cdot 2^{(j-1)\cdot\mathsf{n}+k-1}, \\
\mathsf{encode\text{-}v2}(V) &= \sum_{j=2}^{\mathsf{n}} \sum_{k=1}^{j-1} b_{j,k} \cdot 2^{(j-1)\cdot(j-2)/2+k-1}, \\
\mathsf{encode\text{-}v3}(V) &= \mathsf{Lehmer\text{-}Encode}((u_1, \ldots, u_{\mathsf{n}})).
\end{aligned}$$

We let len denote the bit-length of this encoding, i.e. n^2 for $\mathsf{encode\text{-}v1}(V)$, $\frac{\mathsf{n}\cdot(\mathsf{n}-1)}{2}$ for $\mathsf{encode\text{-}v2}(V)$ and $\lceil \log_2 \mathsf{n}! \rceil$ for $\mathsf{encode\text{-}v3}(V)$.

3.2 Voting Protocol: Phase One

In the first phase, just as in [CFSY96], we initiate the ThFHE scheme. See Figure 5 for a description, which is executed via the tallying centres $\mathbb{T}$ calling the ideal functionality $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{Init}(\mathsf{sid})$.

AV Voting: Phase 1

$\mathsf{Init}(\mathsf{sid})$:

1. The parties in $\mathbb{T}$ call $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{Init}(\mathsf{sid})$, hence obtaining $\mathfrak{pk}$.
2. The value $(\mathsf{sid}, \mathfrak{pk})$ is posted to a bulletin board.

Fig. 5. The initialization phase for AV elections based on threshold FHE

3.3 Voting Protocol: Phase Two

In our second phase, a voter encodes their vote in the clear, encrypts it using the ThFHE scheme, and then proves the encryption is correct; this phase uses the simple zero-knowledge proofs based on the relation R_{enc}. The ballot is posted to the bulletin board, at which point all interested parties (including the tallying parties) can verify its correctness, and if not reject it from the election. See Figure 6 for a more detailed description, in this description we use the notation $\mathsf{encode\text{-}v}*$ to denote either the method $\mathsf{encode\text{-}v1}$ or $\mathsf{encode\text{-}v2}$ or $\mathsf{encode\text{-}v3}$ above.

Note that the verification step only checks whether the encryption is valid, it does not check whether the vote has been encoded correctly. In the case of $\mathsf{encode\text{-}v1}$ it may not have been correctly encoded. For the case of $\mathsf{encode\text{-}v2}$ and $\mathsf{encode\text{-}v3}$, any valid encryption will correspond to some valid vote upon homomorphic decoding. These issues will be dealt with in Phase 3.

AV Voting: Phase 2

$\mathsf{Vote}(\mathsf{sid}, \mathcal{P}_i, V)$:

This algorithm is called by a voter in order to create a ballot.

1. $x \leftarrow \mathsf{encode\text{-}v}*(V)$.
   Recall this produces x as a len*-bit integer.*
2. $r \leftarrow \mathsf{Coins}$.
3. $\mathfrak{ct}_i \leftarrow \mathsf{Enc}(x, \mathfrak{pk}; r)$.
4. $\pi_i \leftarrow \mathsf{Prv}^{\mathsf{enc}}(\mathfrak{ct}_i, (x, r))$.
5. Post $\{\mathfrak{ct}_i, \pi_i\}$ to the bulletin board.

$\mathsf{Verify}(\mathsf{sid}, (\mathfrak{ct}_i, \pi_i))$:

This algorithm can be called by anyone to verify that a ballot has been correctly encrypted on the bulletin board.

1. If $\mathsf{Ver}^{\mathsf{enc}}(\mathfrak{ct}_i, \pi_i) = \mathsf{false}$ then output $\perp$ and reject the ballot from player $\mathcal{P}_i$.

Fig. 6. The voting phase for AV elections based on threshold FHE

3.4 Voting Protocol: Phase Three

In this phase, the tally centres $\mathbb{T}$ take the encrypted ballot $\mathfrak{ct}_i$ from player $\mathcal{P}_i$ and homomorphically decode it into an encryption of a permutation matrix, see Figure 7. In other words, from $\mathfrak{ct}_i$ we obtain ciphertexts $\mathfrak{ct}_{i,j,k}$, with $1 \leq j, k \leq \mathsf{n}$, such that $\mathfrak{ct}_{i,j,k}$ is guaranteed to encrypt only a single bit $b_{i,j,k}$ and that the matrix is a permutation matrix.

If encode-v1 has been used in Phase 2, then, if the voter is adversarial, it may be the case that Phase Three results in the tally centres rejecting the ballot. This is done by forming the row and column sums of the underlying bit-matrix, and checking that they are equal to one. If they are not, then the underlying matrix is not a permutation matrix, and hence not a validly encoded vote. If the sums are equal to one, then the underlying matrix is a permutation matrix, and hence a validly encoded vote. Abusing notation again by extending the syntax of encryption and decryption to matrices of values, we have

$$\begin{aligned}\mathsf{Dec}(\mathsf{Hom\text{-}decode}(\mathsf{Enc}(\mathsf{encode\text{-}v1}(V), \mathfrak{pk}), \mathfrak{sk}) &= \mathsf{Dec}(\mathsf{Hom\text{-}decode}(\mathsf{Enc}(\mathsf{encode\text{-}v2}(V), \mathfrak{pk}), \mathfrak{sk}) \\ &= \mathsf{Dec}(\mathsf{Hom\text{-}decode}(\mathsf{Enc}(\mathsf{encode\text{-}v3}(V), \mathfrak{pk}), \mathfrak{sk}) \\ &= V,\end{aligned}$$

for all votes V.

The advantage of using encoding method encode-v1 is that the required homomorphic operations to homomorphically decode a ballot are relatively simple. One has to simply form $2 \cdot \mathsf{n}$ sums, of integers which have bit-length at most $\lceil \log_2 \mathsf{n} \rceil$. Thus, we require $O(\mathsf{n}^2 \cdot \log_2 \mathsf{n})$ PBS operations to perform these additions. The implied constant in this big-Oh can be reduced by performing the additions using a tree like addition structure, in which the underlying integer bit-size increases (when required) as one passes up layers of the tree. This decreases the number of PBS operations needed. The disadvantage of using encoding method encode-v1 is the need for the threshold openings, which can in practice be more expensive than the homomorphic operations.

The advantage of using encoding method encode-v2 or encode-v3 is that there is no need for any threshold decryption to check the correctness of the encoding. In addition, even if the encoding is invalid, i.e. not an output of the encoding method in the case of encode-v2 or not an integer in the range $[0, \ldots, \mathsf{n}!)$ in the case of encode-v3, the decoding method will always output a valid permutation. Thus, a voter who encodes incorrectly, still encrypts a valid vote.

Using the latter two methods, we first (Step 1) decode the encrypted integers into an encrypted permutation, and then (Step 2) convert that permutation into a permutation matrix. Step 2 is identical for both methods (see lines 2d and 3e). This utilises n^2 Comparison and CMux operations between plaintext and ciphertext values, i.e. $O(\mathsf{n}^2)$ PBS operations.

Performing Step 1 with encode-v2 requires $\frac{\mathsf{n} \cdot (\mathsf{n}-1)}{2} \cdot \mathsf{Swap}$ operations, each of which requiring two CMux operations bringing us to $(n^2 - n) \cdot \mathsf{CMux}$. Therefore, performing both steps using encode-v2 requires $O(\mathsf{n}^2)$ PBS operations, which is its main advantage as demonstrated in Figure 15.

The disadvantage of encode-v2 over encode-v3 is that the encoding itself is less compact. This leads to larger expanded ciphertexts (see Figure 14, Table 1) and slower verification of ZKPoKs in Phase 2 (see Figure 12).

Our method to homomorphically apply Lehmer decoding given in Figure 7 is more efficient than the direct application of the classical method in Figure 4 mapped to the encrypted domain. This method is inspired by the method given in [Kor08, Section 5.3], but modified to make it more amenable to encrypted computation. The idea here is that the loop in line 3c converts the

AV Voting: Phase 3

$\mathsf{Hom\text{-}decode}(\mathsf{sid}, \mathfrak{ct}_i)$:

This takes the encrypted ballot and decodes it to an encrypted permutation matrix, or it outputs $\perp$, signalling the vote was incorrectly encoded. The protocol is run by the tally centres $\mathbb{T}$.

1. If $\mathsf{encode\text{-}v}* = \mathsf{encode\text{-}v1}$ then
   (a) $\{\mathfrak{ct}_{i,j,k}\}_{j,k=1}^{\mathsf{n}} \leftarrow \mathsf{ExtractBits}\left(\mathfrak{ct}_i, 0, \mathsf{n}^2\right)$.
   (b) For $j \in [1, \ldots, n]$ do:
      i. $\mathfrak{ct}_c \leftarrow \sum_{k=1}^{\mathsf{n}} \mathfrak{ct}_{i,j,k}$.
      ii. $\mathfrak{ct}_r \leftarrow \sum_{k=1}^{\mathsf{n}} \mathfrak{ct}_{i,k,j}$.
      iii. $c \leftarrow \mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct}_c)$.
      iv. $r \leftarrow \mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct}_r)$.
      v. If $r \neq 1$ or $c \neq 1$ then output $\perp$ and reject this ballot $\mathfrak{ct}_i$.
2. Else if $\mathsf{encode\text{-}v}* = \mathsf{encode\text{-}v2}$ then
   (a) $\{\{\mathfrak{ct}'_{i,j,k}\}_{k=1}^{j}\}_{j=2}^{\mathsf{n}} \leftarrow \mathsf{ExtractBits}\left(\mathfrak{ct}_i, 0, \frac{\mathsf{n}\cdot(\mathsf{n}-1)}{2}\right)$.
   (b) Let $\mathbf{a}$ denote a list of (trivial) encryptions of $[1, \ldots, \mathsf{n}]$.
   (c) For $j \in [\mathsf{n}, \ldots, 2]$ do:
      i. For $k \in [j-1, \ldots, 1]$ do $\mathsf{Swap}(\mathbf{a}_j, \mathbf{a}_k, \mathfrak{ct}'_{i,j,k})$.
   (d) For $j, k \in [1, \ldots, \mathsf{n}]$ do $\mathfrak{ct}_{i,j,k} \leftarrow (\mathbf{a}_j = k)$.
3. Else
   (a) For $j \in [1, \ldots, \mathsf{n}-1]$ do $\mathfrak{ct}''_{i,j} \leftarrow (\mathfrak{ct}_i / j!) \mod (j+1)$.
   This, by homomorphically evaluating the remainder operation for a plaintext denominator, will output the encryptions of the s values from Figure 4.
   (b) Let $\mathbf{a}$ denote a list of (trivial) encryptions of $[1, \ldots, \mathsf{n}]$.
   (c) For $j \in [1, \ldots, \mathsf{n}-1]$ do:
      i. For $k \in [1, \ldots, \mathsf{n}-1]$ do:
         A. $\mathfrak{ct}'_k \leftarrow (k \geq \mathfrak{ct}''_{i,j})$.
         This homomorphically computes the greater-than-or-equal-to operator between a clear and an encrypted value.
         B. $\mathsf{Swap}(\mathbf{a}_k, \mathbf{a}_{k+1}, \mathfrak{ct}'_k)$.
      ii. $\mathfrak{ct}'_{i,j} \leftarrow \mathbf{a}_{\mathsf{n}}$.
   (d) $\mathfrak{ct}'_{i,\mathsf{n}} \leftarrow \mathbf{a}_1$.
   (e) For $j, k \in [1, \ldots, \mathsf{n}]$ do $\mathfrak{ct}_{i,j,k} \leftarrow (\mathfrak{ct}'_{i,j} = k)$.
   i.e. we homomorphically compute the equality operator between a ciphertext and a clear value.
4. Return $\{\mathfrak{ct}_{i,j,k}\}_{j,k=1}^{\mathsf{n}}$.

Fig. 7. The homomorphic decoding phase for AV elections based on threshold FHE

ciphertexts $\mathfrak{ct}''_{i,j}$ into a permutation $\mathfrak{ct}'_{i,j}$ by performing swaps on the vector $\mathbf{a}$. The element we wish to extract is pushed to the end of the vector and then it is taken and placed in $\mathfrak{ct}'_{i,j}$. Finally, from the encrypted permutation $(\mathfrak{ct}'_{i,1}, \ldots, \mathfrak{ct}'_{i,n})$ a permutation matrix is extracted in line 3e.

The main disadvantage of encoding method $\mathsf{encode\text{-}v3}$ is, again, the need to apply arithmetic operations in lines 3a and 3c and line 3e. The modular remainder in line 3a requires $O((\log_2 \mathsf{n})^2)$ PBS operations and needs to be evaluated $\mathsf{n}-1$ times. In lines 3c and 3e the arithmetic operations require only $O(\log_2 \mathsf{n})$ PBS operations, but they need to be executed $(\mathsf{n}-1)^2$ times. Thus, this homomorphic decoding method requires $O(\mathsf{n}^2 \cdot \log_2 \mathsf{n})$ PBS operations in total.

The homomorphic decoding method of $\mathsf{encode\text{-}v1}$ could be implemented using a levelled FHE scheme such as BGV or BFV, if the encrypted votes were just encryptions of bits as opposed to being packed into a single encrypted integer as in our above description. The decoding method in this case would be simple homomorphic additions, in lines 1(b)i and 1(b)ii of Figure 7. These

homomorphic additions can be accomplished in BGV and BFV without consuming any levels. However, for encoding methods encode-v2 and encode-v3, even if the votes where encrypted as encrypted bits, the CMux operations would consume levels when implemented with BGV or BFV, as the CMux gate involves a multiplication operation.

3.5 Voting Protocol: Phase Four

Finally, having each vote presented as an encrypted permutation matrix, we can perform the tallying phase of our AV election. See Figure 10 for details.

During the tallying part of the protocol, the voters' vote matrix is updated with the current 'active' vote always being the first column. As a candidate is eliminated, and since we do not allow undervoting, all vote matrices will be shifted. This is demonstrated in Figure 9 and Figure 8: starting from the leftmost vote matrix, we eliminate a candidate. The middle matrix demonstrates the intermediate state after shifting the columns (see line 2(b)iv). The red values representing the candidate's row (orange box) and the last column of the matrix may be trimmed or ignored. The remaining values form an $(\mathsf{n}-1 \times \mathsf{n}-1)$ matrix which can be used in the next tally round.

In case of a tie, we remove the candidate with the smallest candidate number[10].

$$\begin{pmatrix} 0&0&0&1\\ 0&1&0&0\\ 1&0&0&0\\ 0&0&1&0 \end{pmatrix} \begin{pmatrix} 0&0&1&1\\ 0&1&0&0\\ 1&0&0&0\\ 0&0&1&0 \end{pmatrix} \begin{pmatrix} 0&0&1\\ 0&1&0\\ 1&0&0 \end{pmatrix}$$

Fig. 8. Elimintation of Candidate 4 in Phase 4

$$\begin{pmatrix} 0&0&0&1\\ 0&1&0&0\\ 1&0&0&0\\ 0&0&1&0 \end{pmatrix} \begin{pmatrix} 0&0&1&1\\ 1&0&0&0\\ 1&0&0&0\\ 0&1&0&0 \end{pmatrix} \begin{pmatrix} 0&0&1\\ 1&0&0\\ 0&1&0 \end{pmatrix}$$

Fig. 9. Elimination of Candidate 3 in Phase 4

Notice how the partial tallies (see the introduction for what these are) for each round are output by the protocol. But that no other information is output by the protocol.

As before, if one tried to implement this with BGV or BFV the homomorphic addition in line 2(a)iiA of Figure 10 will be essentially trivial (as the plaintext modulus can be larger than v). However, line 2(b)ivB of the Tally procedure will, in the worst case, produce an equation of degree $2^{\mathsf{n}-1}$ if no winner is found until the very last round[11]. Thus, using levelled-BGV/BFV would limit the number of candidates to the number of levels. In addition, if using encoding method encode-v2

[10] This is purely for simplicity of presentation, and could easily be modified.

[11] If on input to a round $\mathsf{ct}_{i,j,r}$ has degree d in the inputs, then after the CMux it will have degree $2 \cdot d$, as the CMux is a multiplication gate essentially. As we have a maximum of $\mathsf{n}-1$ rounds, this means the worst case degree is $2^{\mathsf{n}-1}$.

AV Voting: Phase 4

$\mathsf{Tally}(\mathsf{sid}, \{\mathfrak{ct}_{i,j,k}\}_{i,j,k})$:

1. $\mathsf{winner} \leftarrow 0$, $\mathsf{round} \leftarrow 1$ and $\mathsf{exclude} \leftarrow \mathbf{0} \in \{0,1\}^{\mathsf{n}}$.
2. While $\mathsf{winner} = 0$ do:

   Find a winner if there is one

   (a) For $c \in [1, \ldots, \mathsf{n}]$ do:
      i. $T_{c,\mathsf{round}} \leftarrow 0$.
      ii. If $\mathsf{exclude}_c = 0$ then
         A. $\mathfrak{ct} \leftarrow \sum_{i=1}^{v} \mathfrak{ct}_{i,c,1}$.
         B. $T_{c,\mathsf{round}} \leftarrow \mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$.
         C. If $T_{c,\mathsf{round}} > v/2$ then $\mathsf{winner} \leftarrow c$.

   What to do if no winner found in this round

   (b) If $\mathsf{winner} = 0$ then

      First find one candidate to eliminate, who has the lowest number of votes

      i. $m \leftarrow v$
      ii. For $c \in [1, \ldots, \mathsf{n}]$ do: if $\mathsf{exclude}_c = 0$ and $T_{c,\mathsf{round}} < m$ then $\mathsf{remove} \leftarrow c$, $m \leftarrow T_{c,\mathsf{round}}$.

      Mark candidate as removed

      iii. $\mathsf{exclude}_{\mathsf{remove}} \leftarrow 1$.

      Now remove the candidate, by selectively rotating the votes one column to the left

      iv. For $i \in [1, \ldots, v]$, $k \in [1, \ldots, \mathsf{n} - \mathsf{round}]$ do:
         A. $\mathfrak{ct}_{\mathsf{shift}} = \mathfrak{ct}_{i,\mathsf{remove},k}$
         B. For $j \in [1, \ldots, n] \setminus \{\mathsf{remove}\}$, $r \in [k, \ldots, \mathsf{n} - \mathsf{round}]$ do:
            Exclude CMux *operations which assign values from the eliminated candidate's column.*
            - $\mathfrak{ct}_{i,j,r} \leftarrow \mathsf{CMux}(\mathfrak{ct}_{i,j,r}, \mathfrak{ct}_{i,j,r+1}, \mathfrak{ct}_{\mathsf{shift}})$.

      Optionally trim the last row and column of each $\mathfrak{ct}_i$ if memory consumption is a concern.

3. Output winner, $\{T_{c,round}\}$.

Fig. 10. The tallying phase for AV elections based on threshold FHE

or encode-v3, the decoding method would also add to the number of consumed levels. Indeed, the worst degree for this phase, using the encode-v2 method, will be $2^{\mathsf{n}-1}$ as well, as you at most move one column of the identity matrix $\mathsf{n} - 1$ times. Each move costing one one multiplication level. To support arbitrary numbers of candidates, for fixed parameters over all possible elections, one would need to utilise an FHE scheme which supports bootstrapping. This explains our choice of using TFHE, as this provides relatively small parameters and an efficient bootstrapping procedure.

In each round, one candidate is eliminated. The total amount of homomorphic computation in round round is given by

1. The creation of $(\mathsf{n} - \mathsf{round} + 1)$ sums of v bit values, in line 2(a)iiA
2. The threshold decryption of $(\mathsf{n} - \mathsf{round} + 1)$ integer values of bit-length $\log_2 v$, in line 2(a)iiB. Since tfhe-rs works in standard sizes for integer values, this corresponds to integer values of bit-length 8, 16, 32, etc.
3. A total of $O(v \cdot (\mathsf{n} - \mathsf{round}) \cdot \mathsf{n}^2)$ evaluations of the CMux operation on bit values in line 2(b)ivB.

The cost is dominated (in practice) by the last two of these operations; threshold decryption because it requires an expensive Switch-n-Squash operation, and the CMux operation. If the election terminates after R rounds of vote tallying, then the total cost, in terms of basic PBS operations, of all the CMux operations on line 2(b)ivB, is

$$O\left(v \cdot \mathsf{n}^2 \cdot \sum_{\mathsf{round}=1}^{R} (\mathsf{n} - \mathsf{round})\right) = O\left(v \cdot \mathsf{n}^3 \cdot R\right).$$

3.6 Security Analysis

There are many different ways of presenting security analysis of voting protocols. Many adopt a game-based security formulation, see [BCG+15, CGK+16]. We adopt a more UC-style definition, in that we interpret a voting protocol as an instance of a Multi-Party Computation protocol in which the input parties have private input (their ballots), and the MPC parties wish to compute the functionality which outputs the election tally.

One can consider the above voting protocol as an instantiation of the general FHE-based MPC protocol analysed in [Sma23]. The MPC protocol in [Sma23] is between a set of input parties $\mathbb{I}$, computing parties $\mathbb{C}$ and output parties $\mathbb{O}$. The protocol given in [Sma23] enables the evaluation of what is called an F-Circuit; which loosely corresponds to the evaluation of an arithmetic circuit in which certain intermediate values are revealed during the execution. The F-Circuit is (essentially) the representation of the underlying ideal functionality, one is trying to emulate in the MPC protocol, via the basic operations enabled by both the FHE-scheme and the ability to open intermediate results. The resulting protocol is UC-secure against static active adversaries, and provides robustness (a.k.a. guaranteed output delivery), assuming the ideal functionality $\mathcal{F}_{\mathsf{KeyGenDec}}$ also is robust.

Treating a voting protocol as an instance of an MPC protocol instantly captures many of the desired properties of a voting scheme:

- Privacy of votes (all that can be obtained about the votes of honest parties is exclusively what can be determined from the output of the election).
- Independence of inputs (no dishonest party can make their vote a function of the honest parties' votes).
- Correctness of the tally if all parties are honest.
- Robustness of the output (assuming a given number of tallying parties are honest and the MPC protocol is robust, then the output is correct irrespective of adversarial behaviour).

The only standard aspect of voting protocols which is not captured directly by treating the protocol as an MPC protocol is that of verifiability. We shall return to verifiability below, after considering the privacy and correctness of our protocol.

Applying the protocol analysis in [Sma23] to our protocol is relatively simple. The input parties $\mathbb{I}$ clearly correspond to our voters. The computing parties $\mathbb{C}$ correspond to our tally centres $\mathbb{T}$. In [Sma23], to ensure robustness, it is assumed that a majority of the entities in $\mathbb{C}$ are honest. The output parties $\mathbb{O}$ represent anyone who has an interest in the output of the election. The protocol in [Sma23] uses the ideal functionality $\mathcal{F}_{\mathsf{KeyGenDec}}$ given in Figure 3, the tally centres can implement this using the protocol described in [DDK+23]; which, as we remarked above, will be robust as long as at most $|\mathbb{C}|/3 = T/3$ are dishonest.

Thus, the only issue in applying the methodology in [Sma23] is the "leakage" of information which is inherent to executing the underlying F-Circuit. The F-Circuit is in virtual one-to-one correspondence with the pseudo-code given in Figure 6–Figure 10. The data "leaked" by the MPC protocol are the values output from the calls to $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$, excluding leakage from badly formed encryptions which are captured by the zero-knowledge proof verifications. We see the "leakage" is of two types:

1. When $\mathsf{encode\text{-}v}* = \mathsf{encode\text{-}v1}$ we "leak" that the row and column sum of a vote matrix is equal to one. Since this must be true for a valid vote, such a leakage is actually not a leak at all.
2. In the tallying phase we "leak" the values $T_{c,\mathsf{round}}$, this is the tally of votes for candidate c on round round of the tallying. This is strictly more than the final winner of the election. However, as explained in the introduction such "partial tallies" are often considered to be part of the output of an AV election; so again such leakage is actually no leakage at all.

Thus we see that our protocol respects the privacy of voters, as the only information which is leaked about valid votes is that which can be derived from the output of an ideal election.

We now return to the issue of verifiability. We assume in what follows that the following modifications are made to the above protocol outline

- Voters sign their votes, so that external parties can verify that only valid voters vote.
- All calls to the functionality $o \leftarrow \mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$ are recorded on the bulletin board with both the input $\mathfrak{ct}$ and the output o; signed by the parties who actually implemented the operations underlying $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$.

With these assumptions we can also, trivially, obtain the following desirable properties of an electronic voting scheme:

- **Individual Verifiability:** Each voter can check that their ballot was received and utilised by checking that their ballot is recorded on the bulletin board, assuming the bulletin board is honest.
- **Eligibility Verifiability:** By checking the signatures of voters against a list of valid voters, any party can verify that only valid voters were able to vote.
- **Ballot Verifiability:** Any party (including external auditors) can verify the zero-knowledge proofs, attached to each ballot, to check that a ballot was correctly encrypted. To check that a vote was correctly encoded the verifying party, in the case when $\mathsf{encode\text{-}v}* = \mathsf{encode\text{-}v1}$ needs to rely on the fact that the $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$ operation was performed correctly; this requires assuming that the parties executing this protocol had a suitable honest subset[12].
- **Universal Verifiability:** Anyone can check, in much the same way, that the final tally is correct; again assuming a suitable subset of honest parties amongst the entities who implement $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$.

The verifiability could be further enhanced if the execution of $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$ was itself publicly verifiable. Such verifiability is possible, using generic zero-knowledge techniques, but when applied to the robust protocol from [DDK+23] such techniques will be very expensive to execute. Alternatively, as mentioned in Section 2.2, if one is willing to give up on robustness and accept active-with-abort security, then by passing to the protocol from [ABGS23], one can achieve public verifiability of $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$. Alternatively, the (currently theoretical) protocol from [BFM+25] could be deployed if both robustness and public verifiability are required.

[12] Note, the input $\mathfrak{ct}$ to $\mathcal{F}_{\mathsf{KeyGenDec}}.\mathsf{DistDecrypt}(\mathsf{sid}, \mathfrak{ct})$ can be verified to be correct publicly, as it can be computed by anyone given the input ciphertexts.

4 Implementation Details

We implemented the main components of our voting protocol using the Rust library tfhe-rs (v1.3.2) mentioned earlier. The only aspects we did not implement are the associated networking related to posting ballots by voters on a bulletin board, the retrieval of such ballots by the tally centres, and the networking associated to threshold decryption.

The threshold decryption protocol given in [DDK+23] is dominated by the Switch-n-Squash algorithm needed to ensure the TFHE ciphertext noise is small enough to enable efficient threshold decryption. This operation can take between 300ms and 500ms to execute on a standard processor, and is provided in the tfhe-rs library via the member function *ct.squash_noise()*. Once the Switch-n-Squash method has been applied to a ciphertext, a threshold decryption can be accomplished over real networks in under 10ms, plus the associated network ping time (and this can be amortised by executing many threshold decryptions in parallel if possible). Thus, approximating the total time for threshold decryption by only counting the time needed to apply the Switch-n-Squash method is a good first order approximation of the actual running time.

Our experiments were run on a MacBook Pro laptop running macOS 15.6, with a 12-core M4 Pro Processor and 24 GB of available RAM. We restricted our experiments to exclusively utilise the 8 performance cores available which run at 4.5 GHz.

Recall an encoded vote requires n^2 bits to represent it, if one is using encoding method encode-v1, $\frac{\mathsf{n}\cdot(\mathsf{n}-1)}{2}$ bits to represent it, if one is using encoding method encode-v2, and $\lceil \log_2 \mathsf{n}! \rceil$ if using encoding method encode-v3. As we encode each vote, during encryption, as an integer within tfhe-rs we utilise the data types in Table 1 for encrypting ballots.

	Encoding Method		
n	encode-v1	encode-v2	encode-v3
3	FheUint10	FheUint4	FheUint4
4	FheUint16	FheUint6	FheUint6
5	FheUint32	FheUint10	FheUint8
6	FheUint64	FheUint16	FheUint10
7		FheUint32	FheUint14
8			FheUint16
9	FheUint128	FheUint64	FheUint32
10			

Table 1. The used tfhe-rs datatypes for each encoding method and number of candidates.

4.1 Phase 2: Voting

We first discuss the run-times for Phase 2, as described in Figure 6. This consists of two methods, first the voter needs to encode their vote, encrypt the ballot and compute the zero-knowledge proof. After the ballot is received by the bulletin board, the zero-knowledge proof needs to be verified. These two run-times are presented in Figure 11 and Figure 12, with the run-times given in milliseconds.

Recall, the discussion on compressed proofs from earlier. This results in a relatively constant running time for the voting phase, as seen in Figure 11. However, the runtime of the verification phase increases as the underlying datatype size increases, as seen in Figure 12.

Recall that tfhe-rs, when performing public key encryption, and providing proofs of encryption, compresses the data. Thus, the ciphertexts in Phase 2 output by the Vote algorithm are smaller than the ciphertexts output by the Verify algorithm. In graph Figure 13 (resp. Figure 14) we can also see the sizes of the compressed (resp. expanded) encrypted votes.

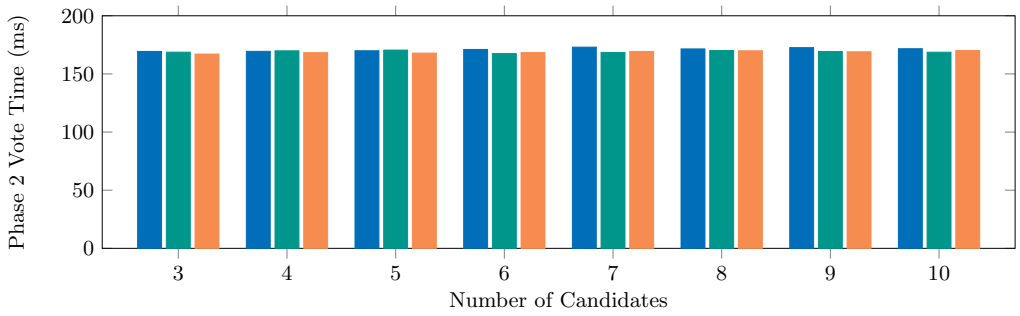

Fig. 11. Average time over 100 runs for the voter to execute the Vote step of Phase 2 of our protocol. Blue is when using encode-v1, green is when using encode-v2 and orange is when using encode-v3.

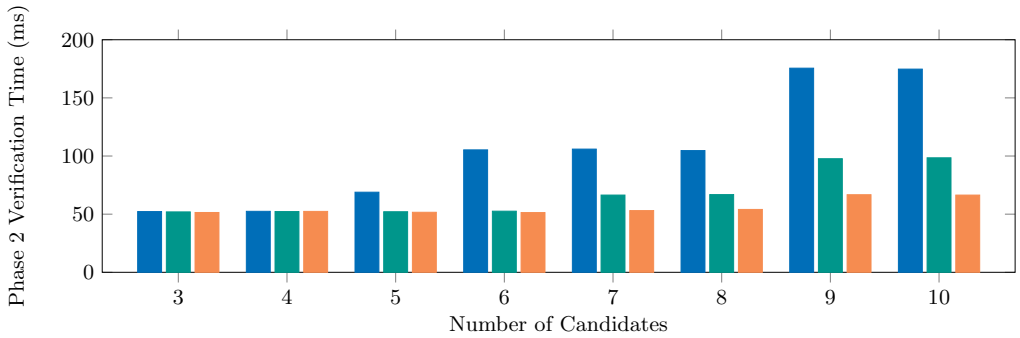

Fig. 12. Average time over 100 runs for the receiver to verify the ZKPoKs associated with each ballot in Phase 2 of our protocol. Blue is when using encode-v1, green is when using encode-v2 and orange is when using encode-v3.

4.2 Phase 3: Homomorphic Vote Decoding

In Phase 3, see Figure 7, the encrypted vote is decoded from an encrypted integer into an encrypted permutation matrix. In the case of encoding method encode-v1, this requires the evaluation of $2 \cdot \mathsf{n}$ threshold decryption operations, each of which requiring $2 \cdot \mathsf{n}$ executions of the Switch-n-Squash operation. The actual final threshold decryption, after the Switch-n-Squash operations, can be performed in parallel. Hence, this step is negligible as remarked above. These threshold decryptions are needed in order to verify that the encrypted integer actually corresponds, under the encoding scheme, to a valid permutation matrix. For encoding method encode-v2 or encode-v3 there is no need for any threshold decryption operations in order to determine whether the encrypted integer corresponds to a valid encoding of a permutation matrix, since *all* integers correspond to *a* permutation matrix.

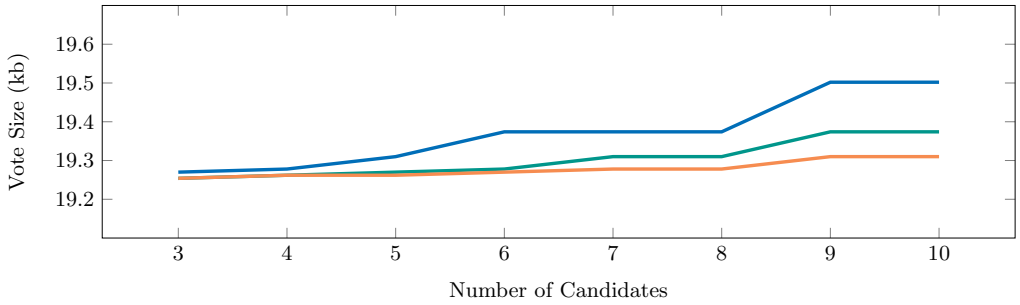

Fig. 13. Combined size of the compressed encrypted votes and ZKPoKs resulting from the execution of the Vote phase of Phase 2. Blue is when using encode-v1, green is when using encode-v2 and orange is when using encode-v3.

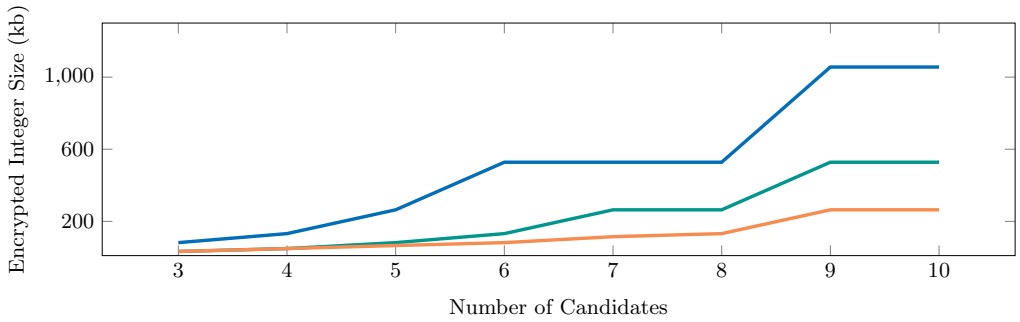

Fig. 14. Size of expanded integer ciphertexts at tally centres after the Verify phase of Phase 2. Blue is when using encode-v1, green is when using encode-v2 and orange is when using encode-v3.

See Figure 15 for the execution times of this method, for various numbers of candidates. Here we present the run times in seconds. The figure supports the expectations from the complexity estimations, namely that encoding method encode-v2 is the most efficient. However, the reader should be reminded that encode-v2 is less compact than encode-v3 (see Figure 14), so depending on the application it might be more interesting to use encode-v3. Also note, we are only taking into account computational time, and there is of course added complexity (both in terms of time and code complexity) related to needing to perform threshold decryption during the decoding when using encode-v1. This in addition with the fact that encode-v1 has the least compact representation (see Figure 14), renders encode-v1 the least prefered option when using the fully homomorphic TFHE scheme. However, with a levelled FHE scheme like BGV or BFV, encode-v1 will likely be the preferred option.

4.3 Phase 4: Homomorphic Tallying

In examining run times for Phase 4 we examined election sizes consisting of 3 to 10 candidates, with a voter population size of 10, 100, 1,000 and 10,000. In total, for each election size, we ran 10 different elections in order to determine average running times.

The voters' votes in our elections were randomly selected from all permutations. This obviously produced elections different from those normally seen, i.e. it is rare for everyone to differ in opinion so much in terms of preferential order. Whilst the maximum number of rounds of execution, given n candidates, in Phase 4 is bounded by $\mathsf{n}-1$, it is not the case that all elections require $\mathsf{n}-1$ rounds.

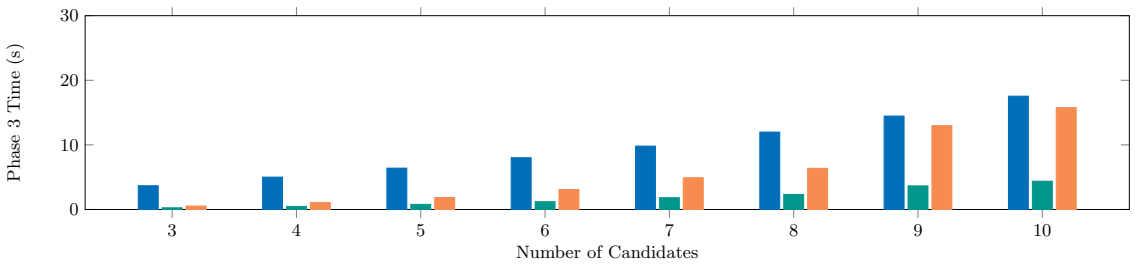

Fig. 15. Average time over 100 runs for the receiver to apply Phase 3 of our protocol in order to extract a permutation matrix from the underlying encrypted integer. Blue is when using encode-v1, green is when using encode-v2, and orange is when using encode-v3.

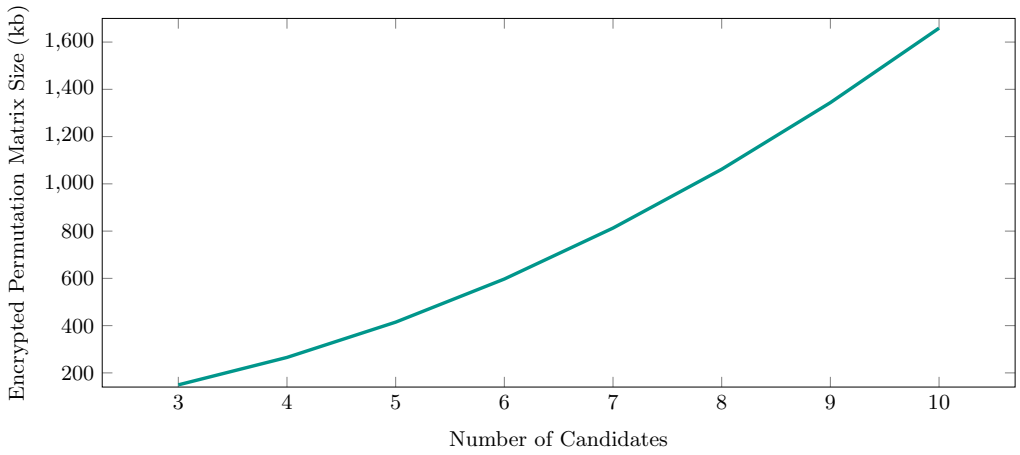

Fig. 16. Size of extracted encrypted permutation matrices after executing Phase 3.

Indeed, if the majority of voters agree on one candidate as their first choice then the election will terminate after one round. For our randomly chosen permutation votes, such a distribution of voters' first choices is unlikely to occur. In contrast, as the votes are randomly selected from all permutations, then the worst case, of always needing $\mathsf{n}-1$ rounds to determine a winner for the election, is likely to always be the case.

We examined average run times (given in Figure 17–Figure 20), for the different rounds of the protocol, when they were executed. Note that, except when we have small number of voters, the average time to execute each round decreases as the round increases. The variation in this rule, in Figure 17 and Figure 18, for the larger number of candidates is simply due to the fact that for such configurations, for small numbers of voters, we are more likely to terminate our execution on an earlier round than round $\mathsf{n}-1$. This termination skews the average runtimes as we only give averages over ten runs.

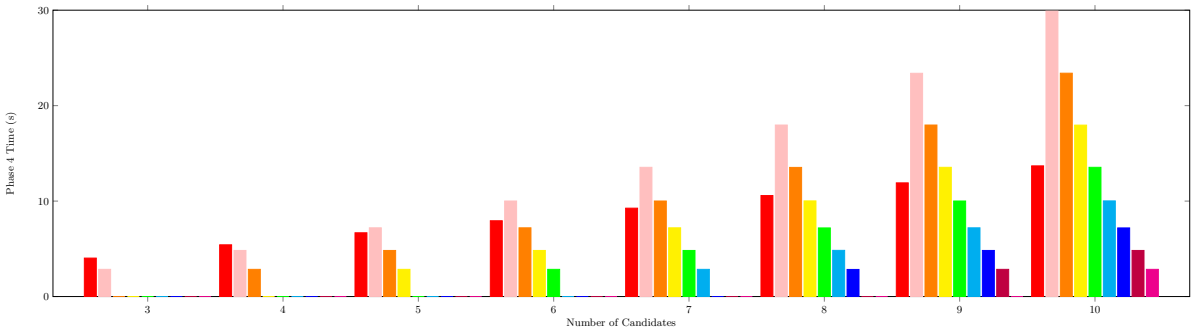

Fig. 17. Average per round time over 10 runs for the tally centres to apply Phase 4 of our protocol with 10 voters. The coloured bars refer to the time needed to execute a specific round of Phase 4. From red being round one, through to magenta being round nine. Recall if there are n candidates, then there are at most $\mathsf{n}-1$ rounds.

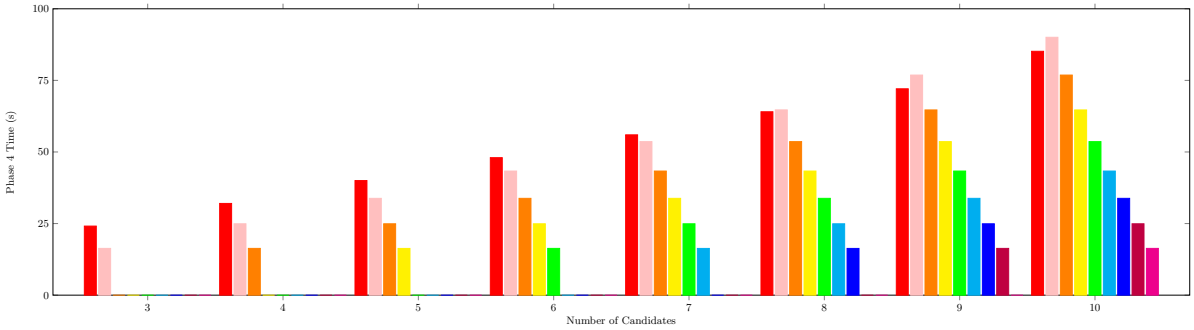

Fig. 18. Average per round time over 10 runs for the tally centres to apply Phase 4 of our protocol with 100 voters. The coloured bars refer to the time needed to execute a specific round of Phase 4. From red being round one, through to magenta being round nine. Recall if there are n candidates, then there are at most $\mathsf{n}-1$ rounds.

In addition we give the total runtime (in Figure 21) across all elections. We see that the run times behave roughly as one would predict from the complexity estimates. Namely that the overall

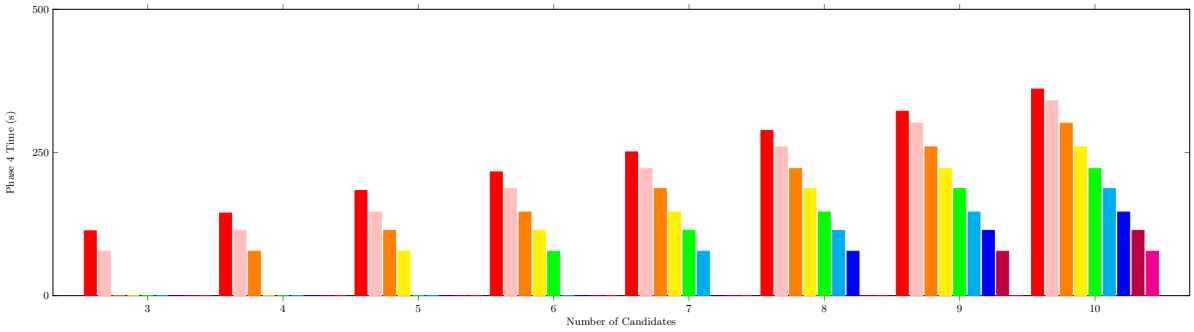

Fig. 19. Average per round time over 10 runs for the tally centres to apply Phase 4 of our protocol with 1,000 voters. The coloured bars refer to the time needed to execute a specific round of Phase 4. From red being round one, through to magenta being round nine. Recall if there are n candidates, then there are at most $\mathsf{n} - 1$ rounds.

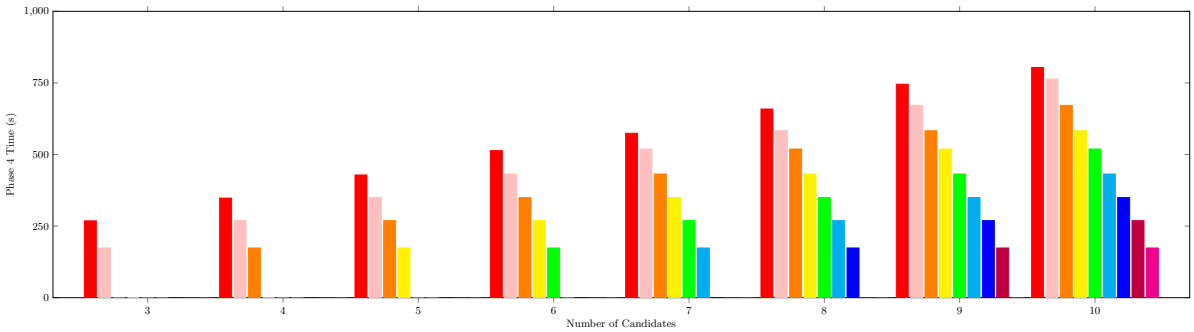

Fig. 20. Average per round time over 10 runs for the tally centres to apply Phase 4 of our protocol with 10,000 voters. The coloured bars refer to the time needed to execute a specific round of Phase 4. From red being round one, through to magenta being round nine. Recall if there are n candidates, then there are at most $\mathsf{n} - 1$ rounds.

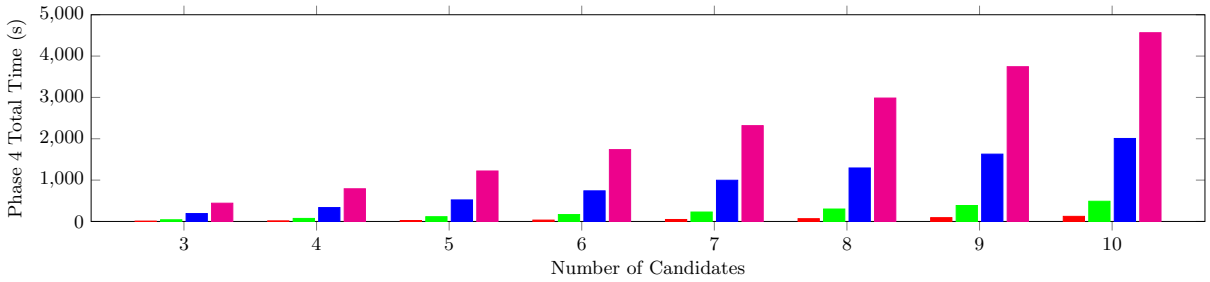

Fig. 21. Average total time over 10 runs for the tally centres receiver to apply Phase 4 of our protocol with 10 voters (red), 100 voters (green), 1000 voters (blue) and 10,000 voters (magenta).

complexity is

$$O\left(v \cdot \mathsf{n}^3 \cdot R\right).$$

in terms of the number of voters v, the number of candidates n and the number of executed rounds R. The per round complexity, in round R, being

$$O\left(v \cdot \mathsf{n}^2 \cdot (\mathsf{n} - R)\right).$$

We see that such elections are practical if the number of candidates is limited, say less than 10. But as one scales the number of candidates up, the execution time, especially when there is a need to execute multiple rounds, becomes prohibitive. However, in real elections the votes are not uniformly distributed, and so larger elections may be feasible, as the tallying phase will terminate after only a few rounds.

Acknowledgements

The work of the second and third author was supported by CyberSecurity Research Flanders with reference number VR20192203, and by the FWO under an Odysseus project GOH9718N.

References

ABG+21. Diego F. Aranha, Carsten Baum, Kristian Gjøsteen, Tjerand Silde, and Thor Tunge. Lattice-based proof of shuffle and applications to electronic voting. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 227–251, Virtual Event, May 17–20, 2021. Springer, Cham, Switzerland.

ABGS23. Diego F. Aranha, Carsten Baum, Kristian Gjøsteen, and Tjerand Silde. Verifiable mix-nets and distributed decryption for voting from lattice-based assumptions. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 1467–1481, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

Adi08. Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *USENIX Security 2008: 17th USENIX Security Symposium*, pages 335–348, San Jose, CA, USA, July 28 – August 1, 2008. USENIX Association.

AdMP. Ben Adida, Olivier de Marneffe, and Olivier Pereira. Helios voting system. `http:/heliosvoting.org`.

BCG+15. David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. SoK: A comprehensive analysis of game-based ballot privacy definitions. In *2015 IEEE Symposium on Security and Privacy*, pages 499–516, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

BdBB+25. Mathieu Ballandras, Mayeul de Bellabre, Loris Bergerat, Charlotte Bonte, Carl Bootland, Benjamin R. Curtis, Jad Khatib, Jakub Klemsa, Arthur Meyre, Thomas Montaigu, Jean-Baptiste Orfila, Nicolas Sarlin, Samuel Tap, and David Testé. TFHE-rs: A (Practical) Handbook, 2025.

BEL+24. Olivier Bernard, Sarah Elkazdadi, Benoît Libert, Arthur Meyre, Arthur Meyre, Jean-Baptiste Orfila, and Nicolas Sarlin. Faster Short Pairing-Based NIZK Proofs for Ring LWE Ciphertexts. In submission, October 2024.

Ben87. Josh Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, 1987. `https://www.cs.yale.edu/publications/techreports/tr561.pdf`.

BFM+25. Zvika Brakerski, Offir Friedman, Avichai Marmor, Dolev Mutzari, Yuval Spiizer, and Ni Trieu. Threshold FHE with efficient asynchronous decryption. Cryptology ePrint Archive, Paper 2025/712, 2025.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. Association for Computing Machinery.

BHM21. Xavier Boyen, Thomas Haines, and Johannes Müller. Epoque: Practical end-to-end verifiable post-quantum-secure E-voting. In *2021 IEEE European Symposium on Security and Privacy*, pages 272–291, Vienna, Austria, September 6–10, 2021. IEEE Computer Society Press.

BMN+09. Josh Benaloh, Tal Moran, Lee Naish, Kim Ramchen, and Vanessa Teague. Shuffle-sum: coercion-resistant verifiable tallying for STV voting. *IEEE Trans. Inf. Forensics Secur.*, 4(4):685–698, 2009.

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

BY86. Josh Cohen Benaloh and Moti Yung. Distributing the power of a government to enhance the privacy of voters (extended abstract). In Joseph Y. Halpern, editor, *5th ACM Symposium Annual on Principles of Distributed Computing*, pages 52–62, Calgary, Alberta, Canada, August 11–13, 1986. Association for Computing Machinery.

CF85. Josh D. Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme (extended abstract). In *26th Annual Symposium on Foundations of Computer Science*, pages 372–382, Portland, Oregon, October 21–23, 1985. IEEE Computer Society Press.

CFSY96. Ronald Cramer, Matthew K. Franklin, Berry Schoenmakers, and Moti Yung. Multi-autority secret-ballot elections with linear work. In Ueli M. Maurer, editor, *Advances in Cryptology – EUROCRYPT'96*, volume 1070 of *Lecture Notes in Computer Science*, pages 72–83, Saragossa, Spain, May 12–16, 1996. Springer Berlin Heidelberg, Germany.

CGGI16a. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany.

CGGI16b. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. A homomorphic LWE based E-voting scheme. In Tsuyoshi Takagi, editor, *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016*, pages 245–265, Fukuoka, Japan, February 24–26, 2016. Springer, Cham, Switzerland.

CGGI20. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

CGK+16. Véronique Cortier, David Galindo, Ralf Küsters, Johannes Mueller, and Tomasz Truderung. SoK: Verifiability notions for E-voting protocols. In *2016 IEEE Symposium on Security and Privacy*, pages 779–798, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

CJL+20. Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. CONCRETE: Concrete Operates oN Ciphertexts Rapidly by Extending TFHE. In *8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC 2020)*, pages 57–63. Leibniz Universität IT Services, 2020.

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland.

DDK+23. Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-fhe using noise flooding. In *Workshop on Encrypted Computing & Applied Homomorphic Cryptography – WAHC 2023*, pages 35–46. ACM, 2023.

DJ02. Ivan Damgård and Mads Jurik. Client/server tradeoffs for online elections. In David Naccache and Pascal Paillier, editors, *PKC 2002: 5th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 125–140, Paris, France, February 12–14, 2002. Springer Berlin Heidelberg, Germany.

DJ03. Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In Reihaneh Safavi-Naini and Jennifer Seberry, editors, *ACISP 03: 8th Australasian Conference on Information Security and Privacy*, volume 2727 of *Lecture Notes in Computer Science*, pages 350–364, Wollongong, NSW, Australia, July 9–11, 2003. Springer Berlin Heidelberg, Germany.

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640, Sofia, Bulgaria, April 26–30, 2015. Springer Berlin Heidelberg, Germany.

dPLNS17. Rafaël del Pino, Vadim Lyubashevsky, Gregory Neven, and Gregor Seiler. Practical quantum-safe voting from lattices. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1565–1581, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

FKL18. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

FMRV22. Thibauld Feneuil, Jules Maire, Matthieu Rivain, and Damien Vergnaud. Zero-knowledge protocols for the subset sum problem from MPC-in-the-head with rejection. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 371–402, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland.

FR23. Thibauld Feneuil and Matthieu Rivain. Threshold linear secret sharing to the rescue of MPC-in-the-head. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 441–473, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore.

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `https://crypto.stanford.edu/craig`.

GSW13. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, Santa Barbara, CA, USA, August 18–22, 2013. Springer Berlin Heidelberg, Germany.

HSS24. Patrick Hough, Caroline Sandsbråten, and Tjerand Silde. More efficient lattice-based electronic voting from NTRU. *IACR Communications in Cryptology (CiC)*, 1(4):10, 2024.

Joy24. Marc Joye. TFHE public-key encryption revisited. In Elisabeth Oswald, editor, *Topics in Cryptology – CT-RSA 2024*, volume 14643 of *Lecture Notes in Computer Science*, pages 277–291, San Francisco, CA, USA, May 6–9, 2024. Springer, Cham, Switzerland.

KCRT21. Guillaume Kaim, Sébastien Canard, Adeline Roux-Langlois, and Jacques Traoré. Post-quantum online voting scheme. In Matthew Bernhard, Andrea Bracciali, Lewis Gudgeon, Thomas Haines, Ariah Klages-Mundt, Shin'ichiro Matsuo, Daniel Perez, Massimiliano Sala, and Sam Werner, editors, *FC 2021 Workshops*, volume 12676 of *Lecture Notes in Computer Science*, pages 290–305, Virtual Event, March 1–5, 2021. Springer Berlin Heidelberg, Germany.

KKW18. Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. Improved non-interactive zero knowledge with applications to post-quantum signatures. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 525–537, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

Knu97. Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

Kor08. Richard E. Korf. Linear-time disk-based implicit graph search. *J. ACM*, 55(6):26:1–26:40, 2008.

Leh60. D. H. Lehmer. Teaching combinatorial tricks to a computer. *Proceedings of Symposia in Applied Mathematics - Combinatorial Analysis*, 10:179–193, 1960.

Lib24. Benoît Libert. Vector commitments with proofs of smallness: Short range proofs and more. In Qiang Tang and Vanessa Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 14602 of *Lecture Notes in Computer Science*, pages 36–67, Sydney, NSW, Australia, April 15–17, 2024. Springer, Cham, Switzerland.

Nef01. C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001: 8th Conference on Computer and Communications Security*, pages 116–125, Philadelphia, PA, USA, November 5–8, 2001. ACM Press.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer Berlin Heidelberg, Germany.

SK94. Kazue Sako and Joe Kilian. Secure voting using partially compatible homomorphisms. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 411–424, Santa Barbara, CA, USA, August 21–25, 1994. Springer Berlin Heidelberg, Germany.

Sma23. Nigel P. Smart. Practical and efficient FHE-based MPC. In Elizabeth A. Quaglia, editor, *19th IMA International Conference on Cryptography and Coding*, volume 14421 of *Lecture Notes in Computer Science*, pages 263–283, London, UK, December 12–14, 2023. Springer, Cham, Switzerland.