A toolbox for verifiable tally-hiding e-voting systems

Véronique Cortier, Pierrick Gaudry, Quentin Yang

A toolbox for verifiable tally-hiding e-voting systems

Véronique Cortier, Pierrick Gaudry and Quentin Yang
Université de Lorraine, CNRS, Inria, France

***Abstract*—In most verifiable electronic voting schemes, one key step is the tally phase, where the election result is computed from the encrypted ballots. A generic technique consists in first applying (verifiable) mixnets to the ballots and then revealing all the votes in the clear. This however discloses much more information than the result of the election itself (that is, the winners) and may offer the possibility to coerce voters.**

In this paper, we present a collection of building blocks for designing tally-hiding schemes based on multi-party computations. As an application, we propose the first tally-hiding schemes with no leakage for four important counting functions: D'Hondt, Condorcet, STV, and Majority Judgment. We also unveil unknown flaws or leakage in several previously proposed tally-hiding schemes.

1. Introduction

Electronic voting is used in many countries and various contexts, from major politically binding elections to small elections for example among scientific councils. It allows voters to vote from any place and is often used as a replacement of postal voting. Moreover, it enables complex tally processes where voters express their preference by ranking their candidates (preferential voting). In such cases, the votes are counted using the prescribed procedure (*e.g.* Single Transferable Vote or Condorcet), which can be tedious to conduct by hand but can be easily handled by a computer.

Numerous electronic voting protocols have been proposed such as Helios [4], Civitas [11], or CHVote [17]. They all intend to guarantee at least two security properties: *vote secrecy* (no one should know how I voted) and *verifiability*. Vote secrecy is typically achieved through asymmetric encryption: election trustees jointly compute an election public key that is used to encrypt the votes. The trustees take part in the tally, to compute the election result. Only a coalition of dishonest trustees (set to some threshold) can decrypt a ballot and violate vote secrecy. Verifiability typically guarantees that a voter can check that her vote has been properly recorded and that a voter (and external auditor) can check that the result corresponds to the received votes. Then, depending on the protocol, additional properties can be achieved such as coercion-resistance or cast-as-intended. Various techniques are used to achieve such properties but one common key step is the tally: from the set of encrypted ballots, it is necessary to compute the result of the election, in a verifiable manner.

There are two main approaches for tallying an election in the context of electronic voting. The first one is the *homomorphic tally*. Thanks to the homomorphic property of the encryption scheme (typically ElGamal), the ballots are combined to compute the (encrypted) sum of the votes. Then only the resulting ciphertext needs to be decrypted to reveal the election result, without leaking the individual votes. For verifiability, each trustee produces a zero-knowledge proof of correct (partial) decryption so that anyone can check that the result indeed corresponds to the encrypted ballots. The second main approach is based on *mixnets*. The encrypted ballots are shuffled and re-randomized such that the resulting ballots cannot be linked to the original ones [17], [36]. A zero-knowledge proof of correct mixing is produced to guarantee that no ballot has been removed nor added. Several mixers are successively used and then each (rerandomized) ballot is decrypted, yielding the original votes in clear, in a random order.

Homomorphic tally can only be applied to simple vote counting functions, where voters select one or several candidates among a list and the result of the election is the sum of the votes, for each candidate. We note that even in this simple case, the tally reveals more information than just the winner(s) of the election. Mixnet-based tally can be used for any vote counting function since it reveals the (multi)set of the initial votes. On the other hand, mixnet-based tally reveals much more information than the result itself (the winner(s) of the election) and is subject to so-called Italian attacks. Indeed, when voters rank their candidates by order of preference, the number of possible choices is much higher than the number of voters. Hence a voter can be coerced to vote in a certain way by first selecting the first candidates as desired by the coercer and then "signing" her ballot with some very particular order of candidates, as prescribed by the coercer. The coercer will check at the end of the election that such a ballot appears.

Recent work have explored the possibility to design tally-hiding schemes, that compute the result of the election from a set of encrypted ballots, without leaking any other information. This can be seen as an instance of Multi-Party Computation (MPC). The context of voting adds some constraints. First, a voter should only produce one encrypted ballot that should remain of reasonable size and that should be computed with low resources (*e.g.* in JavaScript). The trustees can be assumed to have more resources. Yet, it is important to minimize the number of communications and the computation cost, whenever possible. In particular, voters should not wait for weeks before obtaining the result. Moreover, all proofs produced by the authorities need to be downloaded and verified by external, independent auditors.

It is important that verifying an election remains affordable.

Related work. Even when the winner(s) of the election is simply the one(s) that received the most votes, leaking the scores of each candidate can be embarrassing and can even lower vote privacy. This is discussed in [22] where the authors propose a protocol called Ordinos that computes the candidate who received the most votes, without any extra information. In case of preferential voting, where voters rank candidates, several methods can be applied to determine the winner(s). Two popular methods are Single Transferable Vote (STV) and Condorcet. STV is used in politically binding elections in several countries like Australia, Ireland, or UK. Condorcet has several variants and the Schulze variant is popular among several associations like Ubuntu or GnuGP. It is the counting methods offered by the voting platform CIVS [1] and used in many elections. Literature for tally-hiding schemes includes [19] which shows how to compute the result in Condorcet, while [35] and [7] provide several methods for STV. They all leak some partial information, but much less than the complete set of votes. Finally, Majority Judgment (MJ) is a vote system where voters give a grade to each candidate (typically between 1 and 6). The winner is, roughly, the candidate with the highest median rating. Since typically several candidates have the same median, the winner is determined by a complex algorithm that iteratively compares the highest median, then the second one and so on (see [5] for the full details). In [10], the authors show how to compute Majority Judgment in MPC. All these approaches except [19] rely on Paillier encryption since it is better suited than ElGamal for the arithmetic comparison of the content of two ciphertexts.

Our contribution. First, we revisit the existing work, exhibiting weaknesses and even flaws for some of them. Second, we provide new algorithms for computing vote counting functions, improving both the complexity and the leakage or proposing other trade-offs regarding the load for the voters and the trustees. In particular, we propose the first tally-hiding schemes with no leakage for four major counting functions: D'Hondt, STV, Condorcet, and Majority Judgment. We summarize our main contributions in the following table.

Single vote	- Fix shortcoming in [22] in case of equality - Adaptation to D'Hondt method
Majority Judgment	- Fix the fact that [10] fails in not-so-rare cases - Complete leakage-free algorithm, based on ElGamal
Condorcet	- Fix privacy issue in [19] - Several efficiency/leakage compromises - Original ballot encoding and ZKP by the voters - Complete leakage-free algorithm
STV	- Ideal STV has exponential worst-case complexity - Complete leakage-free algorithm, with fast arithmetic

One of our first findings is that even for complex counting functions, it is possible to use ElGamal encryption instead of Paillier. This offers several advantages. ElGamal encryption can be implemented on elliptic curves, yielding building blocks of much lower size than Paillier's encryption for the same security parameter. The computational time is similarly lower. Moreover, in the context of voting, it is important to split the decryption key among several trustees so that no single authority can break vote privacy. It is easy to set up threshold decryption in ElGamal, with an arbitrary threshold of k trustees among n needed for decryption [12]. The situation is more complex in Paillier. The general threshold key distribution scheme [20] is of high complexity (about 80 rounds of communications and 90Go of exchanged data just for 5 authorities). A more efficient scheme exists [26], but only if the threshold is large enough ($k > n/2$).

Another reason for preferring ElGamal could be that the underlying security assumption (Decisional Diffie Hellman) can be considered as more standard than the one for Paillier (Decisional n-Residuosity). Finally, from a practical point of view, it is also easier to find standard software libraries that include support for ElGamal encryption.

We have considered several families of counting methods, that include complex ones (e.g. STV, MJ), to demonstrate that it is possible to build efficient MPC schemes for such vote counting functions, often using standard ElGamal encryption.

Single vote. A first class of counting functions applies to the case where the voters simply select one (or several) candidate(s). The typically way to determine the s winners is to count the number of votes for each candidate and select the s candidates with the most votes. This is exactly the case covered by Ordinos [22]. There is however a shortcoming in case of equalities: the function implemented in Ordinnos may return more winners than the number of seats. We correct this shortcoming by providing an algorithm that computes exactly the s winners according to the election rule, without leaking any extra information. Moreover, we show here that actually, it is possible to rely on ElGamal with the associated benefits discussed earlier, thanks to an adapted algorithm. This lowers the size of a ballot for voters at a similar cost for the authorities (less exponentiations, messages of smaller size, but more communications).

Things get more complex when voters select a candidate list instead of a candidate. Then the s seats need to be shared among the candidates of the different lists, according to number of votes received by each list. One popular technique to compute how to "share" the seats is the D'Hondt method, that can be adapted to several variants depending on whether the election system wishes to favor big or small parties. We extend the approach initiated by Ordinos to the case of D'Hondt, building on two main ideas: the use of a more advanced sorting algorithm and a more efficient algorithm for comparison, inspired from algorithms on circuits (but with different constraints) in order to reduce the cost of the resulting algorithm. We propose two different compromises in terms of computations and communications cost. We study the cost of relying on either Paillier and ElGamal and in this case, ElGamal is a key ingredient for designing a practical tally-hiding scheme.

Majority Judgment. The idea of Majority Judgment [5] is that candidates should not be ranked but instead should each be judged independently. We found out that [10] ac-

tually only implements a simplified version of the Majority Judgment method, called majority gauge. When the majority gauge returns a winner, then it is indeed a MJ winner. Unfortunately, in small elections, there is a rather high probability that the simplified algorithm does not provide any result. For example, in an election with 100 voters, [10] would fail with probability 20%, which not only is inconvenient (imagine an election that must be canceled because no winner is declared!) but also leaks some information (there is no winner according to the majority gauge).

To repair the approach, one issue is that the complexity of the MJ algorithm depends (linearly) in the number of voters, which may be large. Hence, [5] devises an alternative (complex) algorithm that no longer depends on the number of voters. We propose a variant of this algorithm and use it as a basis to derive a tally-hiding procedure for MJ. Our resulting algorithm remains of a complexity similar to [10] while they implement a much simpler algorithm. Then we show that it is actually possible to adapt our algorithm to ElGamal encryption. Interestingly, the format remains unchanged for the voter (hence the resulting ballot is even easier to compute). A key idea is to work in the bit-encoding of integers, which allows us to perform all the needed operations (additions, comparisons) on ElGamal encryptions. The load for the trustees increases (since comparisons are more complex) but our study shows that it remains reasonable since the extra operations are more or less compensated by the fact that computations are faster in ElGamal.

Condorcet. A Condorcet winner is a candidate that would win against each of her opponents. In some cases, there is no Condorcet winner, and several variants exist to further determine a winner in such a case. In [19] a tally-hiding algorithm is proposed for the Condorcet-Schultze variant. However, we found out that [19] is subject to a major privacy flaw. Indeed, everyone learns, for each voter, how many candidates have been placed at equality. Hence Alice completely loses her vote privacy if she votes blank, which cannot be acceptable. This flaw has been acknowledged by the authors.

To repair this flaw, we had to solve a difficult question: voters need to prove, at a reasonable cost, that they encrypted a meaningful ballot, that is, a ballot that corresponds to an order. We considered two main ingredients here. First, we devised a new encoding for ballots. Second, we used mixnet in an original way: a voter proves that her ballot is valid by showing that her ballot can be obtained as a permutation of a valid (public) ballot. Here, the permutation hides the voter's choice and the voter is her own mixer. We then devise several algorithms (all based on ElGamal) with different compromises in terms of load balance between the voters and the trustees and in terms of leakage.

STV. In a first round of STV, if a candidate has been ranked in the first place sufficiently enough (more than a quota), then the candidate obtains a seat. However, if he obtained more votes than the quota, the exceeding votes should not be lost. Instead, they should be transferred to the next candidate. Hence a fraction of votes (which corresponds to the exceeding votes) is transferred to the second preferred candidate of each voter. The process is repeated until all the seats are filled. Since it is not easy to compute by hand the fractions that need to be transferred, many variants of STV exist where the fractions can be rounded or where the votes can be transferred to randomly selected ballots.

Our first goal was to implement a tally-hiding algorithm for the ideal STV, with no rounding. However, we discovered that actually, even without any cryptography, the pure STV algorithm is exponential and far from being practical. On real data elections from the South New Wales election in Australia [2], the pure STV algorithm (on clear votes) would take about one month on a personal computer to compute the result according to the real STV. We believe that this issue was not well understood.

Given that ideal STV cannot be efficiently computed even in the clear, we considered a variant with rounding. In [7], [35], there are in total three techniques to compute the STV winners, all with some leakage (for example, the current score of the selected candidate). Note that [35] computes the ideal STV (with no rounding) but probably because the authors did not realize that it would quickly be impractical. We propose a fully tally-hiding algorithm, with no leakage, at a cost similar to [7], [35]. To keep the cost reasonable, we re-used techniques of hardware circuits (*e.g.* to reduce the length of the carry-chains in additions).

In addition to improving or even fixing several results of the literature, we have developed several techniques that we believed can be reused for other counting methods. In particular, we identified several basic operations that can be computed in ElGamal at a reasonable cost. We have also re-used classical or advanced algorithmic techniques from various provenance, and shown that they can be highly relevant in the context of MPC.

Detailed algorithms are given in Appendix.

2. Building blocks

We focus on the tally phase, common to most voting schemes. We assume a public ballot box that contains the list of encrypted ballots where all the traditional issues up to here have been handled: eligibility, validity of ballots, revoting policy if applicable, and so on. We concentrate on the counted-as-recorded property. We do not assume that the encrypted ballots are anonymous: for example, they could be signed by voters.

Our goal is to compute the winners of the election, while preserving the privacy of the voters, namely with no additional leakage of information about the tally. The decryption key is assumed to be shared among trustees, with a threshold scheme, and we wish the procedure to produce a transcript such that: 1) if at least a threshold of trustees is honest, the result will be obtained and only the result is known (no side-information); 2) even if all the trustees are dishonest, the result is guaranteed to be correct.

This does not come for free and usually involve heavy computations and communications between trustees.

2.1. MPC toolbox

The MPC implementation of counting functions rely on several common building blocks that we define below, such as addition, multiplication, comparison. For each of them, we study their cost. All these costs are summarized in Figure 9 in Appendix. Regarding the computation cost, we count the number of exponentiations. For the communications, sometimes all the trustees need to broadcast their share of the computations, and sometimes they need to perform a round of communications, where one trustee contributes to the data they receive from the previous one in the loop. We will count these two types of communications separately. An important information is also the size of the transcript that is created during the process and that should be checked, for example by auditors, to guarantee that the result is correct.

Beyond defining our needed building blocks, we believe that this study is of independent interest since it could be used in other contexts than voting. It has required to study a rich literature, first on zero-knowledge proofs [9], [23], [29], [36] and MPC [6], [24], [30], [31], [32] but also on hardware circuits [8]. Interestingly, we distinguish between the functionality (*e.g.* addition) and the algorithm that realizes it since different algorithms may be considered, leading to different trade-offs in terms of communications and computations. For a few building blocks, we even propose our own algorithms, that improve existing propositions.

Threshold key generation. This initial part of all our algorithm is considered slightly out of the scope of our study. It is not specific to e-voting and the literature on the topic is abundant. In the ElGamal setting, a key generation technique following Pedersen's is very cheap [12] compared to everything we describe in our work. For Paillier, this is much more complicated and it could be not negligible at all, with hundreds of thousands of exponentiations to be performed by the trustees when we have a dozen of trustees [20].

In both case, however, the output is a classical encryption key that can easily be used by the voters, and decrypting a single ciphertext is cheap, with a constant number of exponentiations per trustee and a constant number of communications [14], [27].

Homomorphic property. Both Paillier and ElGamal are homomorphic encryption schemes. This means that multiplication or division of ciphertexts correspond to addition or subtraction of the corresponding cleartexts. We denote these functions `Add` and `Sub`; they cost no communications nor exponentiations. This allows re-randomization, by multiplying with $\texttt{Enc}(0)$. If the encrypted value is a bit, by dividing $\texttt{Enc}(1)$ by it, this allows to flip the encrypted bit. We denote $\texttt{Not}(B)$ this function that is therefore essentially for free.

Encoding of encrypted integers. An integer can be directly encrypted. This is simple and we call this *natural encoding* in this paper. It allows to directly add and subtract encrypted values. However, in the ElGamal setting, most of the other operations (comparison, multiplication, ...) are more difficult, or even impossible.

The alternative is to encrypt each bit of the integer separately; we call this the *bit-encoding* of an encrypted integer and we denote it $X^{\text{bits}} = (X_0, \ldots, X_{m-1})$, where 2^m is a bound on the integer represented by X, and X_i is the encryption of the i-th bit of the binary expansion (index 0 for the least significant bit). Converting an integer in bit-encoding to natural encoding is easily done using the homomorphic property and the Horner scheme. The other direction is harder (in the Paillier setting) or impossible (in the ElGamal setting).

Branch-free tools. In MPC, the algorithms must be implemented in a branch-free setting, because the result of a test cannot be revealed (unless we allow a partial leakage). The classical building-blocks for this are conditional operations.

- $\texttt{CondSetZero}(X, B)$, $\texttt{CondSetZero}^{\text{bits}}(X^{\text{bits}}, B)$: conditionally set to zero. This function returns (a re-encryption of) X if B is an encryption of 1, or $\texttt{Enc}(0)$ if B is an encryption of 0. In the bit-encoding setting, each bit of X is re-encrypted or set to zero.
- $\texttt{Select}(X, Y, B)$, $\texttt{Select}^{\text{bits}}(X^{\text{bits}}, Y^{\text{bits}}, B)$: select according to bit. This function returns (a re-encryption of) X if B is an encryption of 0 and Y if B is an encryption of 1.
- $\texttt{SelectInd}([X_i], [B_i])$: select in array according to bits. This function returns (a re-encryption of) X_i such that B_i is an encryption of 1. This requires that the encrypted bit array $[B_i]$ is such that that there is only one index i for which B_i is 1.

The `CondSetZero` function is the main primitive from which all the others can be easily derived using the homomorphic property. For instance, $\texttt{Select}(X, Y, B)$ can be implemented as $\texttt{Add}(\texttt{CondSetZero}(\texttt{Sub}(Y, X), B), X)$. In the context of ElGamal encryption, it costs one round of communication at each use.

Arithmetic. As already said, by homomorphy, addition and subtraction of encrypted values are built-in functionalities when the natural encoding is used. However the same operations with the bit-encoding become more involved. Several variants can be considered, the most classical being to have all the operations defined modulo 2^m where m is the number of encrypted bits. Sometimes it is useful to return the final carry / borrow bits. Comparison of two integers is denoted by `LT`. In bit-encoding, it can be seen as a subtraction where only the final borrow is needed, but in the natural-encoding, the borrow is not available, and a dedicated algorithm must be designed, only available in the Paillier scheme. Similarly we define the `Mul` function that can be applied to integers in both encoding, with the exception that the natural-encoding multiplication is available only in the Paillier scheme. Finally, a frequent operation is to compute the sum of many encrypted values each containing a bit, typically to get the total number of votes for a given option. We call this operation `Aggreg`. Again with homomorphic encryption, this is very cheap. However, especially in the ElGamal setting, it could be that the result is wanted in the bit-encoding format. Then a dedicated tree-based algorithm, with variable bit-precision can be designed to improve the

complexity compared to naively using the Add function with the maximum precision.

- $\texttt{Add}(X, Y)$, $\texttt{Add}^{\text{bits}}(X^{\text{bits}}, Y^{\text{bits}})$: addition of X and Y, given in any encoding. In the bit-encoding, the result is taken modulo 2^m.
  $\texttt{Sub}(X, Y)$, $\texttt{Sub}^{\text{bits}}(X^{\text{bits}}, Y^{\text{bits}})$ is similarly the subtraction of X and Y, given in any encoding.
- $\texttt{Aggreg}([X_i])$, $\texttt{Aggreg}^{\text{bits}}([X_i])$: sum of n binary values X_i. In the bit-encoding, the output contains $\log n$ encrypted bits.
- $\texttt{LT}(X, Y)$, $\texttt{LT}^{\text{bits}}(X^{\text{bits}}, Y^{\text{bits}})$: comparison of X and Y given in any encoding. In natural-encoding, this is available only in the Paillier setting.
- $\texttt{EQ}(X, Y)$, $\texttt{EQ}^{\text{bits}}(X^{\text{bits}}, Y^{\text{bits}})$: equality test of X and Y given in any encoding. In natural-encoding, this is available only in the Paillier setting.
- $\texttt{BinExpand}(X)$: binary expansion of X. This function returns X^{bits}. This is available only in the Paillier setting.
- $\texttt{Mul}(X, Y)$, $\texttt{Mul}^{\text{bits}}(X^{\text{bits}}, Y^{\text{bits}})$: multiplication of X and Y given in any encoding. In natural-encoding, this is available only in the Paillier setting.

Since $\texttt{CondSetZero}(X, B)$ can be seen as an And gate when X is just a bit, with the additional homomorphic operations (Add and Not), this allows to build any arithmetic circuit with bits as input and output. Building all the arithmetic functions with the bit-encoding is therefore a matter of optimizing the circuit design with respect to the number of exponentiations and communications. We will discuss more thoroughly the impact of these optimizations in Section 6. In the natural-encoding, the strategy is different and available only in the Paillier setting, where it is possible to extract the bits of naturally-encoded integers with an MPC procedure based on masking [31]. This gives an algorithm for BinExpand. Hence, using this conversion, it is possible to compute all the arithmetic operations even if the input are in natural-encoding.

Shuffle and mixnet. A tool that is of great use in our context is verifiable shuffling, leading to mixnets. In electronic voting, the typical use of a verifiable mixnet is during the tally phase, just before decrypting all the ballots, one by one. Our tally-hiding schemes will actually make a thorough use of mixnets, not only on the trustees side but also on the voter's side, as we will see for example in Section 5.

The first building block is $\texttt{Shuffle}([X_i])$. It takes as input an array of encrypted values and output the same (re-encrypted) values in another order that remains secret, together with a zero-knowledge proof that everything was done correctly. As such, this is not an MPC primitive: this is an operation done by just one entity. Chaining a sequence of applications of this procedure by all the trustees, in turn, leads to the $\texttt{Mixnet}([X_i])$ function, that outputs an array of the same re-encrypted values in order that is known to nobody as soon as at least one trustee is honest.

A variant is to shuffle ballots containing a pairwise comparison matrix. Then, the (secret) permutation used to shuffle the columns should be the same as the one used to shuffle the rows. This leads to the $\texttt{ShuffleMatrix}([M_{i,j}])$ and the $\texttt{MixnetMatrix}([M_{i,j}])$ procedures. All these functionnalities come with their variants in the bit-encoding.

2.2. Paillier vs elliptic ElGamal

We discuss aspects of efficiency that must be taken into account, and propose a coarse estimate of a cost of an exponentiation for each of the Paillier and elliptic ElGamal settings. In general, an algorithm based on the Paillier scheme requires less exponentiations that when based on ElGamal; however, exponentiations are more costly.

Parameter sizes. For a voting system, a 128-bit level of security seems to be a reasonable choice. While 112-bit level is probably acceptable for the next decade, many certification body will ask for 128 bits or more. In the case of an elliptic ElGamal this translates readily into a curve over a base field of 256 bits. Furthermore, base fields that are prime finite fields are usually preferred.

For the Paillier scheme, the security relies on a supposedly hard problem that it not harder than integer factorization of an RSA number n. The complexity of the best known factoring algorithm, the Number Field Sieve, being hard to evaluate, there is no strict consensus about the size of n giving a 128-bit security level, but generally this goes around 3072 bits, and this is what we are going to consider in our discussion.

Availability and efficiency. On the voter side, this is irrelevant, but for the trustees, it might be interesting to consider the possibility of using dedicated cryptographic coprocessors. In the Paillier setting, one needs arithmetic modulo n^2, and this is probably not so easy to re-use RSA-accelerators that are dedicated to smaller modular arithmetic. Elliptic curves being more and more deployed in security products, it is more likely to find dedicated hardware. The main building block used in our work is the exponentiation which is a very standard operation in elliptic curve cryptography.

A voting system needs to be implemented on a wide variety of platforms: computing server for the trustees, and smartphone, web browser, or native applications under various OS/hardware, for the voters. It is hard to optimize the code everywhere, and a good approach is to re-use widely available libraries. This is also much safer, especially if the software libraries are standard ones which have been scrutinized by the community for a long time. On this particular aspect, there is a clear advantage of elliptic ElGamal compared to Paillier. Popular elliptic curves like NIST P-256 or Curve25519 are now ubiquitous in cryptographic libraries, while there is in general no support for Paillier. We believe that for e-voting implementations that want to go beyond an academic prototype, this might be decisive.

Cost of operations.

Typical exponentiations in the elliptic ElGamal setting are 256-bit scalar multiplications, where the group law is the elliptic group law, which amounts to a handful of multiplications in the base field $\mathbb{F}_p$ of 256-bits. Here, usually

	Paillier	Elliptic ElGamal	Ratio
Native (server-side)	200	50,000	250
In browser (voter-side)	2	5,000	2,500

Figure 1. Number of exponentiations per second for 128-bit security.

the number p is a pseudo-Mersenne number so that reduction modulo p is cheap.

Typical exponentiations in the Paillier setting are 3072-bit long exponentiations in $\mathbb{Z}/n^2\mathbb{Z}$, where n^2 has 6144 bits. The number n is an RSA-modulus and has no special structure that could speed-up the reduction modulo n^2. Also, in our distributed situation where nobody knows the prime factors of n, the CRT optimizations are not available.

In many cases, heavy optimizations can be done, leading to tremendous speed-ups (see for instance [21]). However, most of them are specific to the types of exponentiations used in a plain encryption, while in our setting many operations are for constructing or verifying zero-knowledge proofs. Tracking which of these can indeed be optimized and which one remains is out of the scope of this work (see [18] for such a careful work in the specific case of a verifiable mixnet in the ElGamal setting).

In Figure 1, we give estimates based on a medium level of optimization, for a native implementation on a modern processor (based on RSA in OpenSSL), and for a javascript implementation running in a modern web browser (based on `libsodium.js`).

3. Single-choice voting

Context. Voters give their choice among a list of k possibilities. The choices that get the more votes get the seats. Sometimes voters can select more than one choice, specially when the number of seats s is large. The basic situation is when choices are precisely the candidates. Another frequent situation is when the voter's choices are lists of candidates. Then one needs a rule to decide how to assign the seats according to the number of votes obtained by each list. For this later case, we will study the D'Hondt method since it is widely used in practice for politically binding elections.

Basic counting. The s winners are the first s candidates who obtained the most votes. This is the situation covered by Ordinos [22]. Their algorithm proceeds as follows. A ballot contains an encrypted bit for each choice (and a proof that the number of set bits follows the rules of the election). Then the ballots are aggregated to get the encrypted number of votes obtained by each candidate. After that, each value is compared to all the others (thus requiring a quadratic number of comparisons between encrypted values). For each candidate, the number of defeats can then be computed and compared to the number of seats s. A bit is set according to this last comparison, and decrypted, thus revealing whether the candidate should get a seat. Hence in case of equality between several values, more than s candidates can be elected. Assume for example that there are 10 seats but that the 10th and 11th candidates have received exactly the same number of votes. The toolkit of Ordinos provides only two options: either the outcome will be the all 11 candidates, with no information on who are the two last ones, or the outcome is the ordered list of the 11 candidates, which leaks more information than needed.

List-voting. The method of D'Hondt parametrized by a sequence of distinct weights $w_1, \ldots, w_s$ proceeds as follows. Each voter votes for one list among k lists. At the end of the election, each list i has received c_i votes. Then the coefficients (c_i/w_j) for $1 \leq i \leq k$ and $1 \leq j \leq s$ are computed and the s largest values are selected so that the seats can be assigned accordingly: a list i gets one seat for each selected coefficient of the form (c_i/w). The encoding of the ballots and the aggregation process is the same as above, in order to get the encrypted values of $c_1, \ldots, c_k$. The algorithmic question is therefore similar to the one in the basic situation, except that there is a need to handle the fractions, and that the length of the list in which to select the s largest encrypted values is increased from k to ks.

Breaking ties. We consider that an algorithm where the output could give less or more than the exact number of available seats is not acceptable for a practical use, in our context where only the result should be leaked. For instance, if there are 3 seats available and 4 candidates A,B,C,D are output, it is impossible to decide which candidate to eliminate, because it could be that all the candidates have the same number of votes, or that A and B got much more votes than C and D who are in a tie. In this later situation, eliminating A or B would not be acceptable. Therefore, the breaking of the tie must be done inside the hidden phase but with a verifiable tallying procedure.

A simple way to decide which candidate or list gets the seat when the number of votes are equal is to have a tie-breaking ordering of the choices to use specifically for that case. This ordering can be public and known in advanced, based on an arbitrary rule (age of the candidate, alphabetical order, ...). It is then possible to modify each value c in the list of values to be compared as $c \leftarrow 2^e c + r$, where r is the rank of the candidate for the tie-breaking ordering, and e is such that $r < 2^e$. Therefore, the values become pairwise distinct, and using the usual integer comparison will lead to a lexicographical order using the tie-breaking ordering only for equal values. This modification is cheap, even if it must be done on encrypted values.

We consider also the case where the tie-breaking ordering is supposed to be random and hidden. In this case, a mixnet can be used to transform any ordering into a uniformly random, hidden ordering (as soon as one of the trustees is honest). Then we proceed as with a known ordering by including the tie-breaking information in the low-significant bits of the values to compare.

Various MPC algorithms for the best s values. Let $c_1, \ldots, c_K$ be a list of encrypted distinct integers, from which to select the s largest ones. Efficient sorting is not as easy to design as it seems, due to the difficulty to emulate pointers or accessing an array at a hidden position. Allowing

Version	Leak-age	EG/P	Voters # exp.	Authorities # exp.	Authorities # comm.	Transcript size
Basic counting [22]	[i]	P	$5k$	$(27m + 146 \log m)k^2a$	$(4R + 26B) \log m$	$(28m + 50 \log m)ak^2$
ours (exact winners)	$\varnothing$	P	$5k$	$(54m_1 + 292 \log m_1)ksa$	$\log\left(\frac{k}{s}\right) \log m_1(2R + 13B)$	$(56m_1 + 100 \log m_1)ksa$
ours (exact winners)	$\varnothing$	EG	$8k$	$60nak + 160m_1ksa$	$(m^2 + 2 \log\left(\frac{k}{s}\right) \log m_1)R$	$18nak + 144m_1ksa$
D'Hondt (comm.)	$\varnothing$	P	$4k$	$(27m_2 + 146 \log m_2)(ks)^2a$	$(4R + 26B) \log m_2$	$(28m_2 + 50 \log m_2)a(ks)^2$
D'Hondt (comp.)	$\varnothing$	EG	$8k$	$60nak + 160m_3ks^2a$	$(m^2 + 32ms'a)R$ $+2 \log k \log m_3 R$	$54nak + 144m_3ks^2a$

[i] More than s winners can be output in case of tie.

Figure 2. Leading terms of the cost of MPC implementations of various single choice systems. s: number of seats, k: number of lists, a: number of authorities, n: number of voters, $m = \lceil \log(n+1) \rceil$, $m_1 = m + \log k$, $m_2 = m + 2 \log k$, $m_3 = m_1 + \log(\mathrm{lcm}(2, \cdots, s))$, $s' = \log(\mathrm{lcm}(2, \cdots, s))$, R: round of communications, B: broadcasts.

a quadratic number of comparisons is therefore a good option for simplicity and can be efficient enough if K is small. In Ordinos, they actually first compute all the possible comparisons independently, and then accumulate the results. We call this technique "naive" in what follows, but in terms of MPC rounds of communications, this is highly efficient.

We propose an alternative that reduces considerably the number of comparisons that we call s-insertion. The idea is to use insertion sort, keeping a sorted list of the current highest values. However, since in an MPC setting the cost of swapping is negligible compared to the cost of comparisons, it makes sense to use a dichotomy (that costs $\log s$ comparisons) to first find the position of the new value to maybe insert, and then to shift the elements in the list to make room for the new value. In pure MPC, this would require s calls to `Select`, and therefore annihilate our efforts to get a good complexity. Therefore, the result of the comparisons will be decrypted, so that the insertion position and the modification of the list of encrypted values can be done at no additional cost, with pointer arithmetic. In order to avoid any information leakage, the K values must be shuffled with a mixnet. Hence revealing the results of the comparisons does not leak any information. This leads to a cost corresponding to a mixnet for the K values, and then $K \log s$ comparisons for the insertions. The main drawback of this method is the number of communications ($\Omega(K \log s)$).

A compromise between both methods is another algorithm that we call s-merge. Here, we separate the K values in K/s blocks of s values and we sort all these blocks. Then, with a sub-routine that can merge two sorted lists of size s and extract the top s elements, we can build a tree of logarithmic depth to merge all the blocks with a reduced number of communications rounds. The s-merge approach comes itself with several variants, depending on which algorithm is used to perform the first sorting of the blocks of size s (use a quadratic sorting algorithm, in one round of communication, or a fast $s \log s$ one, like heapsort, with many rounds), and depending on which algorithm is used for merging the lists (a naive approach, or the s-insertion method). This leads to many trade-offs, and there seems to be no one-size-fits-all solution.

Comparisons. Comparing two encrypted integers can be done with various algorithms, depending on the ElGamal or Paillier setting, and whether the inputs are in the bit-encoding format or not. In Appendix, we present a list of various cost vs communications trade-offs, that we select depending on our needs.

The question of how to handle fractions in the D'Hondt method must also be addressed. A textbook approach using pairs of integers to store the numerators and the denominators would require to compute products each time we want to make a comparison. One option to avoid this additional cost is to compute the $(s'_{i,j} = c_i w_j)$'s instead of $(s_{i,j} = c_i / w_j)$'s. Then, the boolean $s_{i_i,j_i} < s_{i_2,j_2}$ is exactly $s'_{i_i,j_2} < s'_{i_2,j_1}$. In a quadratic setting where all the comparisons are made, this is a nice solution. Otherwise, this would require to keep track of the indexes to be compared and would leak information. Therefore we resort to another option, namely multiplying all the $s_{i,j}$ by the lcm of the weights w_j, to have only integers. In the case where the weights are just $1, 2, 3, \ldots, s$, this lcm grows like $\exp(s(1 + o(1)))$, so this adds $O(s)$ bits to the integers to manipulate. If s is of a size comparable to the logarithm of the number of voters, this is probably faster than to deal with the numerators and denominators separately.

Summary. The choices of the algorithms to use depend on many practical questions and it is impossible to propose a universally best solution. A first element to consider is the choice between ElGamal and Paillier. If many voters are involved, then, with ElGamal, the aggregation of the ballots become very costly for the trustees both in computations and in communications, and Paillier might be the only realistic solution. Otherwise, ElGamal is very attractive for the reasons mentioned in Section 2.2 and the much easier key generation step (DKG).

In Figure 2 we propose two choices for basic counting, one with ElGamal and one with Paillier, in order to compare to Ordinos [22]. The cost of the DKG is not included, even though it is not at all negligible in the case of Paillier. In both cases, we assume that the number of candidates is small enough, so that the quadratic algorithm for selecting the s best is appropriate. For these figures as well as all the following ones, we only count the *leading terms of the cost*. For example, we neglect ak^2 if there is a term of the form a^2k^2. The unit of the *transcript size* is the key length,

typically 3072 bits in Paillier and 256 bits in ElGamal.

Next, we propose two options for computing a D'Hondt tally with weights $1, 2, \ldots, s$. The first option is with Paillier, and with the objective of reducing the amount of communications between them. Therefore, we use the quadratic selection, and a communication-efficient comparison function. The second option is with ElGamal, where we propose what we believe to be a good compromise between computations and communications for the trustees. We use the s-merge algorithm with quadratic algorithm for merging two s-lists, and the communication-efficient integer-comparison function. For comparing the fractions, in the Paillier option, we use the idea of crossing the indexes, while in the ElGamal setting we multiply by the lcm.

4. Majority Judgment

In the Majority Judgment (MJ) approach [5], voters give a grade to each candidate, such as Excellent, Very Good, Poor, *etc.* Each grade is translated into a numerical value, typically from 1 to 6, where 1 is the highest grade. At the end of the election, each candidate c has received a list L_c of grades. The *list of medians* $\mathsf{med}(c)$ associated to candidate c is the sequence formed by first the median grade m of L_c, *i.e.* the highest grade m in the list such that at least a half of the grades are greater or equal to m, then the median of $L_c \backslash \{m\}$ and so on. For example, if Alice received $1, 2, 2, 4, 4, 5$, her list of medians is $2, 4, 2, 4, 1, 5$. Then a candidate c_1 is ranked above a candidate c_2 if $\mathsf{med}(c_1) > \mathsf{med}(c_2)$ in the lexicographical order (taking care of the fact that a low integer value means a higher grade). Intuitively, c_1 wins over c_2 if she has a higher median, or, in case of a draw, a higher second median, *etc.* The winner according to MJ is the greater candidate. There is equality between candidates only if they received exactly the same grades.

A simplified algorithm. While the algorithm to determine the MJ winner(s) is simple, its naive implementation yields a complexity that depends on the number of voters, which could be very costly when done in MPC. Hence, the authors of [10] propose an MPC implementation of a simplification of the MJ algorithm, where whenever two candidates have the same median, only their number of grades higher and smaller than the median are compared. It has been shown that this technique is sound [5]: if a winner can be determined with this approach, it is indeed a MJ winner. However, it may also fail to conclude. An experiment run in [5] on real ballots of a political election with 12 candidates is reassuring: the simplified approach fails only with probability 0,001 for an election of 100 voters. However, this is due to the fact that in this political election, there was a high correlation between candidates (if a voter likes a candidate, he is likely to also like other candidates from similar political parties). In case the number of candidates is smaller and if the distribution of votes is uniform, then the probability of failure raises to 22%, as shown in Figure 4. In any case, the approach of [10] leaks more information about the ballots than just the result, with non negligible probability, since it reveals whether the result can be determined with the simplified algorithm.

Algorithm 1: Majority Judgment

Require: a the aggregated matrix, d the number of grades, n the number of voters
Ensure: C the set of MJ winner(s)

Let $m = \max\{m_i \mid m_i \text{ is the median of candidate } i\}$
Let C be the set of candidates with m as median grade.
Let $I^- = 1$ and $I^+ = 1$ be counters.
Let $s = 1$.
for $i \in C$ **do**
  $p_i = \sum_{j=1}^{m-1} a_{i,j}$, $q_i = \sum_{j=m+1}^{d} a_{i,j}$,
  $m_i^- = \lfloor \frac{n}{2} \rfloor - p_i$, $m_i^+ = \lfloor \frac{n}{2} \rfloor - q_i$
while $(\mid C \mid > 1) \wedge (s \neq 0)$ **do**
  for $i \in C$ **do**
    if $m_i^- \leq m_i^+$ **then**
      $s_i = p_i$
    else
      $s_i = -q_i$
  $s = \max\{s_i \mid i \in C\}$
  $C = \{i \in C \mid s_i = s\}$.
  if $s \geq 0$ **then**
    for $i \in C$ **do**
      $m_i^+ = m_i^+ - m_i^-$, $m_i^- = a_{i,m-I^-}$
      $p_i = p_i - a_{i,m-I^-}$
    $I^- = I^- + 1$
  else
    for $i \in C$ **do**
      $m_i^- = m_i^- - m_i^+$, $m_i^+ = a_{i,m+I^+}$
      $q_i = q_i - a_{i,m+I^+}$
    $I^+ = I^+ + 1$
Return C.

An alternative MJ algorithm. Our first contribution is Algorithm 1 that computes MJ winner(s), with a complexity that does not depend on the number of voters. Another algorithm was also proposed in [5] but our algorithm is easier to adapt in MPC and we prove it to correctly implement the MJ definition. We assume that each vote is encoded as a matrix where each line corresponds to a candidate and contains exactly one 1 in the column corresponding to the selected grade. The *aggregated matrix* a is then defined as the sum of all the votes, that is a sum of matrices. The coefficient $a_{i,j}$ represents the number of grade "j" received by candidate i. We show that it is sufficient to work on the aggregated matrix. Intuitively, we examine the grades by "stacks": whenever two competing candidates have the same number of grades higher than their median (resp. smaller), we remove a stack of grades (the one closest to the median) and we proceed.

Version	Leak-age	Voters # exp.	Authorities # exp.	Authorities # comm.	Transcript size
[10]	[i]	$5kd$	$kma(224k + 58d)$	$(4m + d)R$	$kma(280k + 62d)$
ours (P)	$\varnothing$	$5kd$	$kda(75m + 146 \log m + 20d)$	$d(2R + 13B) \log m \log k$	$kda(78m + 50 \log m + 22d)$
ours (EG)	$\varnothing$	$8kd$	$60nkda + 40kmda(10 + d)$	$m(m+1)+$ $d(6m + 2 \log k \log m)$	$18kda(3n + 20m + 2d)$

[i] [10] leaks whether the winner can be determined with the simplified algorithm.

Figure 3. Leading terms of the cost of MPC implementations of Majority Judgment. n: number of voters, $m = \lceil \log(n+1) \rceil$, k: number of candidates, d: number of grades, a: number of authorities.

Number of voters	10	100	1000
uniform distribution over 5 candidates	0.384	0.220	0.080
political distribution [5]	N/A	0.001	N/A

Figure 4. Estimated probability that the algorithm of [10] fails to determine the MJ winner(s).

MPC implementations.

MPC with Paillier. Despite the fact that Algorithm 1 is complex, it can easily be translated into an MPC algorithm using the building blocks described in Section 2. We assume here that each voter produces a ballot that is formed of a matrix of encrypted 0 and 1, that encodes her choice, together with a zero-knowledge proof that each line contains exactly one 1. Thanks to the homomorphic property of Paillier encryption, the (encrypted) aggregated matrix can easily be obtained from the votes. Then our algorithm essentially consists of comparisons, selections, additions or subtractions and has been written in order to have an easy correspondence with an MPC algorithm. The main difference is the treatment of the "if then else" and "while" constructions. We do no want to leak when the condition is true. Hence each construction of the form "if B then P_1 else P_2" is replaced (intuitively) by $\texttt{Select}(P_2, P_1, B)$ where P_1 and P_2 are slightly rewritten to update exactly the same variables. The most delicate modification is for the "while" loop: we cannot leak the number of iterations. Instead, we can show that the number of iteration is always bounded by d, the number of grades and we simply replace the "while" loop by a "for". Interestingly, the cost remains similar to the (leaky) MPC implementation of [10], except for the number of communications that increases (see Figure 3).

MPC with ElGamal. It is also possible to implement our MJ algorithm based on ElGamal encryption. The encoding of ballots remains unchanged for voters: each voter produces the matrix of her encrypted choices. Hence the cost is even lower for the voter since ElGamal encryption is smaller (and faster). Then the computation of the aggregated matrix requires more care since we cannot compare natural numbers encrypted with ElGamal. So instead, we can compute the bit-encoding of the aggregated matrix using $\texttt{Add}^{\text{bits}}$. This part is linear in the number of voters but could be done "on the fly" during the election. Then the same algorithm can be used, on the bit-encoding, yielding a similar complexity than the Paillier's version, with the advantages of ElGamal as discussed in Section 2.2.

Hence not only it remains practical to implement the full Majority Judgment function in MPC but surprisingly, the simple ElGamal encryption is well suited in this case.

5. Condorcet-Schulze

The Condorcet approach is one popular technique to determine a winner when voters rank candidates by order of preference, possibly with equalities. A Condorcet winner is a candidate that is preferred to every other candidate by a majority of voters. More formally, we can consider the *matrix of pairwise preferences* d where $d_{i,j}$ is the number of voters that prefer (strictly) candidate i over j. Then a Condorcet winner is a candidate i such that $d_{i,j} > d_{j,i}$ for any $j \neq i$. Such a Condorcet winner may not exist. In that case, several variants can be applied to compute the winner. We will mainly focus here on the Schulze method, used for example for Ubuntu elections [3]. The Schulze method first considers by "how much" a candidate is preferred, which can be reflected into the *adjacency matrix* a defined as

$$a_{i,j} = \begin{cases} d_{i,j} - d_{j,i} & \text{if } d_{i,j} > d_{j,i} \\ 0 & \text{otherwise} \end{cases}$$

Then a weighted directed graph is derived from the adjacency matrix, where each candidate i is associated to a node and there is an edge from i to j with weight $a_{i,j}$. This itself induces an order relation between the candidates by comparing the "strength" of the paths between i and j. The exact algorithm can be found in [33]. Note that there may be several winners according to Condorcet-Schulze.

We propose several MPC implementations of Condorcet-Schulze, depending on the accepted leakage and on the load balance between the voters and the authorities. The different approaches are summarized in Figure 5.

Ballots as matrices. A first approach is to encode each vote as a *preference matrix* m where

$$m_{i,j} = \begin{cases} 1 & \text{if } c_i < c_j \\ 0 & \text{if } c_i = c_j \\ -1 & \text{otherwise} \end{cases}$$

The voters then simply encode their ballot as an encrypted preference matrix M. Note that this requires k^2 encryptions (one encryption for each coefficient of the matrix). But voters also need to prove that their (encrypted) preference

Version	Leakage	Voters # exp.	Authorities # exp.	Authorities # comm.	Size of the transcript
[19]	adj. matrix **privacy breach** [i]	$10k^2$	$18ank^2$	2	$10ank^2$
ballots as list of integers (partial MPC)	adj. matrix	$8k \log k$	$30nak^2 \log k$	$2 \log k$	$27nak^2 \log k$
ballots as list of integers (full MPC)	$\varnothing$	$8k \log k$	$10nak^2(3 \log k + 5m) + 120mak^3$	$m(m + 4k)$	$9nak^2(3 \log k + 5m) + 108mak^3$
ballots as matrices	adj. matrix	$\frac{51}{2}k^2$	$\frac{51}{2}nk^2$	0	$\frac{29}{2}nk^2$
ballots as matrices (naive, for comparison)	adj. matrix	$20k^3$	$20nk^3$	0	$20nk^3$

[i] [19] leaks, for each ballot, the number of candidates ranked at equality. In particular, who voted blank is known to everyone.

Figure 5. Leading terms of the cost of MPC implementations of Condorcet winners. n: number of voters, $m = \lceil \log(n+1) \rceil$, k: number of candidates, a: number of authorities.

matrix is well-formed, that is, corresponds to a total order (with equalities). This requires for example to prove that if the voter prefers i over j and j over k then she must prefer i over k, that is:

$$(m_{i,j} = 1) \wedge (m_{j,k} = 1) \Rightarrow (m_{i,k} = 1)$$

and similar relations when $m_{i,j}$ and $m_{j,k}$ are equal to 0 or -1, yielding $O(k^3)$ statements.

Previous work [19]. To discharge the voter from such a proof effort, in [19] the authorities shuffle each preference matrix in blocks (using `ShuffleMatrix`$([M_{i,j}])$) and then decrypt it to check that it was indeed well formed. However, this yields a privacy breach, unnoticed by the authors, as explained in introduction: for each voter, everyone learns the number of candidates placed at equality. In particular, everyone learns who voted blank since in that case all candidates are placed at equality.

Our approach. A naive but costly way to repair [19] is to let the voters prove the relations with zero-knowledge proofs, yielding a cost of $O(k^3)$ exponentiations to build and to check a ballot. We propose an alternative approach in $O(k^2)$. Assume first that a voter prefers candidate 1 over candidate 2, that is preferred over candidate 3 and so on. Then the corresponding preference matrix is:

$$m^{\mathsf{init}} = \begin{pmatrix} 0 & 1 & \cdots & 1 \\ -1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ -1 & \cdots & -1 & 0 \end{pmatrix}.$$

We consider a fixed encryption M^{init} of this matrix:

$$M^{\mathsf{init}}_{i,j} = \begin{cases} E_1 & \text{if } i < j \\ E_0 & \text{if } i = j \\ E_{-1} & \text{otherwise} \end{cases}$$

where E_α is the ElGamal encryption of α with "randomness" 0. Everyone can check that M^{init} is formed as prescribed, at no cost, since we use a constant "randomness".

Assume now that a voter wishes to rank the candidates in some order, which is a permutation σ of $1, 2, \ldots, k$. Then our core idea is that the voter can simply shuffle M^{init} (using `ShuffleMatrix`) using permutation σ. The associated zero-knowledge proof guarantees that the resulting matrix is indeed a permutation of M^{init}, hence is well formed. Interestingly the secret vote σ is not encoded in the initial matrix but in the permutation used to shuffle it. Applying [36], this requires $O(k^2)$ exponentiations for the voter.

Our approach only covers cases where voters rank the candidate in a strict order. We now explain how to account for candidates that have an equal rank. The voter still shuffles M^{init} according to a permutation σ, consistent with her preference order, that is such that $\sigma(i) < \sigma(j)$ implies that $c_i \leq c_j$. But beforehand, she sends an additional vector B of encrypted bits (b_i), where $b_i = 1$ if candidates $\sigma^{-1}(i)$ and $\sigma^{-1}(i+1)$ have equal rank and $b_i = 0$ otherwise. The voter will then modify the matrix M^{init} (with cleartexts (m_{ij})) into a transformed matrix M' (with cleartexts (m'_{ij})), using B, so that M' can be shuffled into her preference matrix. For example, if two consecutive candidates are equal, the corresponding coefficient is set to 0, that is for all i: $m'_{i,i+1} = m_{i,i+1} - b_i$. Then equalities need to be propagated, each diagonal in turn. This propagation can be rewritten as a combination of additions and multiplications on cleartexts, for which the voter builds ciphertexts and zero-knowledge proofs that they follow the algorithm. The resulting cost is in $O(k^2)$ (since k^2 coefficients need to be updated) instead of $O(k^3)$ for the naive approach.

Now that voter ballots are already encoded as preference matrices, the (encrypted) adjacency matrix can be computed by simply multiplying all ballots. This matrix is then (provably) decrypted by the authorities and Condorcet-Schulze as well as many variants can be applied. The main cost for the authorities lies in the verification of the proofs for each ballot. We could also avoid leaking the adjacency matrix by computing the Condorcet-Schulze winner(s) in MPC. However, the cost for the authorities would be in $O(k^3)$. If this is considered as affordable, then we can alleviate the charge of the voters, as we shall explain now.

Ballots as list of integers. To minimize computation cost on the voter's side, we can simply ask each voter to encrypt

an integer c_i for each candidate i representing their order of reference (possibly with equalities). To allow for ElGamal encryption, we will directly use the bit representation of each integer and encrypt each bit separately. If there are k candidates, we need $\log k$ bits to encode each candidate, hence a ballot will contain $k \log k$ ciphertexts, together with zero-knowledge proofs that the ciphertexts encrypt only 0 or 1. This is to be compared with the k^2 encryptions when ballots are encoded as a preference matrix.

Our first goal is to transform back each ballot into a preference matrix, except that we will consider the *positive preference matrix*, obtained from the preference matrix by setting negative coefficients to 0. If C_i denotes the bitwise encryption of c_i then the encrypted positive preference matrix M can be computed by the authorities as:

$$M_{i,j} = \texttt{LT}(C_i, C_j).$$

Summing up the (encrypted) matrix M_v for each voter v, we immediately obtain the (encrypted) pairwise preferences matrix D. From there, we have two options. Either the authorities may compute the encrypted adjacency matrix A from D, and decrypt it, revealing the adjacency matrix (with proof of correct decryption). Then the Condorcet winner can be computed applying various techniques, including Schulze. Or the authorities may apply the Schulze method in MPC from D. Despite the fact that the Schulze method is a complex algorithm on graphs, it can be implemented with an algorithm from Floyd-Warshall [15], [34], that mostly consists in computations of min/max. This can be translated into an MPC algorithm using the building blocks presented in Section 2. We have also considered MPC implementations of other Condorcet variants such as Ranked Pairs.

The advantage of this solution is that the load for voters remains very reasonable, with $O(k \log k)$ exponentiations in total. However, transforming each ballot into the (encrypted) preference matrix M_v is of cost $O(k^2 \log k)$ for each authority and has to be repeated for each voter.

To summarize, when the number of candidates and voters remain reasonable, it is actually possible to compute the Condorcet winners with no leakage. Interestingly, the costly operations performed by the trustees can be done on-the-fly, while voters submit their ballots. Note that unless the number of candidates is really large w.r.t. the number of voters, a fully-hiding tally scheme is not really more expensive than schemes leaking the adjacency matrix.

6. Single Transferable Vote

Choosing one version of STV. Many flavors of STV election methods exist. In all of them, a ballot cast by a voter contains an ordered list of candidates, starting with the most preferred one. Along the counting process, if the candidate in the first line has been selected to get a seat or eliminated, then it should be erased from the ballot, so that the candidate on the second line becomes the most preferred at this stage. However, when a candidate gets a seat, this must "consume" some of the ballots who voted for him. From this comes the notion of quota and the transfer mechanism. In our case, we used the so-called Droop quota, which sets the value of a seat at $q = \lceil n/(s+1) \rceil + 1$. Here s is the number of seats, and n is the number of valid ballots. If a candidate is in the first line of a (weighted) number of ballots that is larger than q, then she gets elected. Otherwise, we take the candidate that gets the least votes and we eliminate her. In case of equality, we use a predefined arbitrary ordering (as in Section 3). The transfer is implemented as follows: each ballot starts with a weight set to one. When a candidate is elected, the surplus of votes is transferred to the next candidates. Namely, all the ballots where this candidate was listed first have their weight multiplied by a transfer coefficient $t = (c - q)/c$ where c is the sum of the weights of such ballots.

Fractions vs approximations. All along the STV algorithm, the weights of the ballots and the transfer coefficient are rational numbers that can be stored as pairs of integers. While this looks as the cleanest approach, we noticed that this leads to an exponential worse-case complexity. Indeed, the transfer coefficient t_i at a round i where a candidate is selected is computed from the sum c_i of the weights at round i, so that t_i has the same height as c_i. Then the weights are updated by multiplying them by t_i, so that their sum c_{i+1} at round $i+1$ has a height that is twice the height of that of c_i. In other words, the heights of the fractions double at each round where a candidate is selected, and we get a complexity that is exponential in the number of seats. This assumes that there is no lucky cancellations between numerators and denominators, and that all the seats get assigned by selection, and not elimination of the competitors.

This observation is a major problem in an MPC setting where the worst complexity must always be done, in order to hide every side-information. However, we realized that this is also a problem outside any cryptographic consideration. Running an ideal implementation of STV based on fractions becomes quickly impractical when the number of seats grows. For instance, we ran the algorithm on the publicly available ballots of the 2019 Legislative Council of New South Wales in Australia [2]. There are 21 seats, 346 candidates and 3.5 millions of ballots were cast. Our basic implementation using Sagemath shows that indeed, the size of the fractions roughly doubles at each selection, so that one would require about 30 GB of central memory for storing all of them.

In real elections, and due to the fact that elections were initially counted by hand, approximations of fractions are used instead, and fix-point arithmetic is the simplest solution, especially in our MPC context. As shown in [16], this is not guaranteed to give the correct answer. Outside cryptographic considerations, interval arithmetic could provide a proof that rounding did not alter the result, but for our MPC goal, we do not see any way to get around unproven approximations, apart from providing a rigorous bound on the required precision, which is out of the scope of this work. We therefore represent fractions with a fix-point arithmetic, allowing r binary digits after the radix point.

Version	Leakage	P/EG	Voters	Authorities		Transcript
			# exp.	# exp.	# comm.	size
[7, Sec. II]	[i]	EG	$10k^2$	$62nak^2$	$9kR$	$19nak^2$
[35]	[ii]	P	$5k^2$	$22nk^2am$	$2nkmR$	$11nak^2m$
[7, Sec. III.B]	[iii,iv]	EG	$10k^2$	$62nak^2$	$9kR$	N/A
ours (naive arith.)	$\varnothing$	EG	$6k \log k$	$20nak(4k \log k + 3m'k')$	$2k(m'(m + 2r + \log k) + \frac{1}{2}k \log k)R$	$18nak(4k \log k + 3m'k')$
ours (optimized arith.)	$\varnothing$	EG	$6k \log k$	$10nak(9k \log k + m'k' \log m')$	$k((2m + \frac{7}{2}r) \log m' + (\log k)^2)R$	$9nak(9k \log k + m'k' \log m')$

[i] Score of all candidates at each turn
[ii] Score of selected candidates at each turn
[iii] Selected or eliminated candidates and approximation of transfer coefficient at each turn
[iv] Trustees learn the score of all candidates at each turn

Figure 6. Leading terms of the cost of MPC implementations of STV. n: number of voters, k: number of candidates, $m = \lceil \log(n+1) \rceil$, a: number of authorities, r: precision in power of 2, $m' = m + r$, $k' = k + r$.

To leak or not to leak. The two main approaches toward a tally-hiding STV algorithm in the literature are [35] and [7]. In [35], mixnets are applied between each round of the algorithm, so that some information can be decrypted and revealed, without disclosing the list of the complete original ballots. The information that is leaked is whether the round was a selection or an elimination, and in the latter case, the score of the selected candidates. We remark also that their technique involves a very sequential first phase with a number of communications that is proportional to the number of ballots. In [7], some information is also revealed between each round of the STV algorithm, in particular the score of all the candidates, which is much more than in [35]. It is however highly efficient. The authors acknowledge that revealing the intermediates scores might be too much; in particular, they propose realistic scenarios where a coercer could successfully use this information. In [7], a variant is proposed where the most crucial information is leaked only to the trustees, while preserving verifiability of the result. For external observers, their approach leaks essentially the same information as [35], and also an approximation of the transfer coefficient at each round.

In what follows, we present an approach without any leakage, even to the trustees. The intrinsic drawback is that the number of rounds and the number of updates that must be computed at each step must be hidden, so that the worst case is always reached. The number of rounds can be $k-1$, when during the first $s-1$ rounds a candidate is selected, and then all the other rounds are candidate elimination. The number of rounds of updates can be as large as nk, when all ballots include the selected candidate in first position. Therefore, the goal is to reach a $O(nk^2)$ complexity.

Description of our MPC algorithm. We identify candidates by integers between 1 and k, and we add a fake candidate numbered 0 that can never get a seat. A data structure is initialized for all candidate number i, containing a bit H_i that indicates if i is still in the running, a bit W_i if i got selected, and a scalar S_i containing the number of votes for i at the current stage. All of these are kept encrypted; only the W_i bits are decrypted and published at the end.

Another data structure contains the (encrypted) ballots, initialized with the ballots sent by the voters, and modified during the algorithm. A ballot contains a list B_j of $k+1$ integers representing the candidates in the preferred order. The candidate numbered 0 is interpreted as all subsequent candidates are not mentioned and must be ignored. Hence a blank vote is encoded by any ballot that contains 0 at the top of the list. All ballots also come with a weight v that is first initialized to 1. To prove that their ballots are valid, voters must provide a zero-knowledge proof that this is a shuffle of the public list of integers $\{0, 1, \ldots, k\}$.

The algorithm then runs $k-1$ rounds, each having the 5 following steps:

1) **Finished?** From the candidate data structure, compute the number of candidates (apart from candidate 0) that got a seat or are still in the running. If this is equal to the number of available seats s, then mark as selected all the candidates that were still in the running.
2) **Count votes.** For each ballot B, take the candidate in the first rank, and add the weight of the ballot to the number of votes S_i of this candidate. In MPC, this is done with a loop on all candidates i, and conditionally adding the weight of the ballot to S_i, depending on whether B_0 is equal to i.
3) **Search for min and max.** Compute i and j the indexes such that $S_i = \max(s_k)$ and $S_j = \min(s_k)$. If the candidate i gets a seat, *i.e.* $S_i \geq q$, set e to 1, c to i and the transfer ratio t to $(S_i - q)/S_i$. Otherwise, the candidate j will be eliminated and set e to 0, c to j, and t to 1.
4) **Select / delete.** Mark the candidate number c as selected or eliminated: set $H_c = 0$, and if e is 1, then set $W_c = 1$. Also, for all ballots, remove the candidate c. This is done in one pass over the list of preferences of each ballot. At the time, remember for each ballot if c was in first position.
5) **Transfer.** For each ballot for which c was in first position, multiply its weight by the transfer value t.

While we have written the algorithm with "if" statements, it can easily be converted to a branch-free version using the `Select` statement, at the price of passing through all data.

In terms of complexity, Steps 2 and 4 are the most costly, since they involve $O(kn)$ arithmetic operations, because they involve two nested loops. This leads to $O(nk^2)$ operations for the whole algorithm with the k rounds. In terms of communications, during Step 4, it is directly possible to handle all ballots in parallel. In Step 2, this requires to organize the sums of n terms in a tree like in the `Aggreg` procedure to reduce the number of communications.

Arithmetic optimizations Let r be the number of binary digits after the radix point, so that all our computations are done with a fix-point precision of 2^{-r}. A bound on the real numbers manipulated during the algorithm is given by the number of voters n, so that we need $m = \lceil \log(n+1) \rceil$ bits for the mantissa. Hence, the operations reduce to integer arithmetic with $m' = m + r$ bits. While this looks small (a few dozens of bits), using textbook algorithms with a naive carry-propagation would lead to a number of rounds of communications that grow linearly with m' for additions and quadratically with m' for multiplications.

For carry-propagation during additions, this is a classical problem in hardware arithmetic circuits. The depth of the circuit translates more or less immediately into the number of communications in our MPC setting. An important difference with hardware considerations is that bounding the fan-in / fan-out of the gates is not relevant for us. The general idea is to rewrite the addition (or subtraction, or comparison) with the help of an associative operator acting on bits, so that a tree of height $\log(m')$ can be constructed. The Appendix contains the details of how, following this strategy, we managed to strongly reduce the communication rounds at a moderate increase in terms of exponentiations. This also yields big savings for multiplications and divisions, since they are built upon additions.

Efficiency considerations. In Figure 6, we give a summary of the various costs for our algorithm and the ones from the literature. Comparing the two last lines demonstrates the advantages of optimizing the arithmetic, since the last one is a very good compromise. While it is difficult to draw conclusions without knowing the context, we consider that with our algorithm, requiring a perfectly tally-hiding is not the criterion that will make the solution turns from practical to impractical. In fact, from the voter's side, our scheme is more efficient than existing solutions, with a quasi-linear number of exponentiations instead of quadratic. The costs for the authorities is certainly terribly high and is not yet realistic for a large scale election, but we consider that this is not much more than the previous solutions which leak partial information.

7. Lessons learned

Our study shows that it is possible to compute the result of an election without leaking any additional information on the original ballots, often at a realistic cost. This requires however to carefully design the corresponding algorithm for each different tally function. We have provided in this paper several techniques that can reduce the cost. This was applied to several well-known complex voting systems, and we developed a toolbox that can be re-used in other contexts. We list here the main questions that a designer should consider when implementing another counting function.

Think ElGamal. While Paillier is the Swiss-Army knife for MPC implementations, our study has shown that ElGamal can often suffice, even when encrypted integers need to be compared or multiplied. This can be a big advantage in terms of efficiency and availability of software libraries.

Rethink the encoding of ballots. The encoding of a ballot can have a huge impact on the cost of the rest of the procedure. For example, encoding integers in their bit representation adds an initial cost that can later save a lot of computation. It can allow to use ElGamal rather than Paillier. The encoding of ballots also typically offers different tradeoffs in terms of load balance between voters and authorities, as seen for example for the Condorcet voting function where a more complex ballot can alleviate the authorities task.

Verifiable mixnets are a versatile tool. The typical use of a mixnet is to mix and re-randomize encrypted ballots before decrypting them and applying the counting function on the cleartexts. However, verifiable mixnets are also useful for tally-hiding implementations, to discharge some computations (*e.g.* verifications) on the cleartexts. More advanced mixing can be used to ensure for example that the same permutation is applied to several components. We have proposed an original usage of mixnet in the context of Condorcet, where the voters themselves use a verifiable shuffle to encode their vote as a (secret) permutation of a fixed public matrix, thus proving well-formedness.

Consider the full algorithmic toolbox. When designing an MPC algorithm, the constraints are rather non standard. The worst case always needs to be considered, and all branches need to be always visited, like in the circuit complexity model. In fact, this circuit point of view is highly relevant, and we borrowed some algorithms from the hardware literature. While these are not exactly the same notions, the depth of the circuit is related to the number of communication rounds; but limits on the fan-in or fan-out of a gate are irrelevant.

Some rather advanced algorithms like the MJ counting functions or the Floyd-Warshall shortest path algorithm can be translated rather easily. On the other hand, some basic tasks can be way too costly if one chooses the wrong algorithm for them. For instance, sorting a list of integers becomes quadratic for more than a few quasi-linear classical algorithms when converted to MPC. Indeed, many classical algorithms assume that accessing the i^{th} value of an array $T[i]$ takes constant time, even when i is a computed value, while in MPC this requires a linear time to pass through all the values and hide the value of i. Another typical example is addition of encrypted integers, where carry propagation can generate a chain of dependencies that translates into a linear number of communication rounds. Breaking the chain of carries as done in hardware circuits allows to reduce this to a logarithmic number of rounds.

References

[1] Condorcet Internet Voting Service (CIVS). https://civs.cs.cornell.edu/.

[2] NSWEC – Election results. NSW Electoral Commision, https://pastvtr.elections.nsw.gov.au/SG1901/LC/State/preferences. Accessed: 2020-08-05.

[3] Ubuntu IRC council position. https://lists.ubuntu.com/archives/ubuntu-irc/2012-May/001538.html, 2012.

[4] B. Adida. Helios: Web-based open-audit voting. In *17th USENIX Security Symposium (Usenix'08)*, 2008.

[5] M. Balinski and R. Laraki. *Majority Judgment: Measuring Ranking and Electing*. MIT Press, 2010.

[6] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, 1989.

[7] J. Benaloh, T. Moran, L. Naish, K. Ramchen, and V. Teague. Shufflesum: Coercion-resistant verifiable tallying for STV voting. *IEEE Transactions on Information Forensics and Security*, 2010.

[8] Brent and Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31(3), 1982.

[9] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy (S&P'18)*, 2018.

[10] S. Canard, D. Pointcheval, Q. Santos, and J. Traoré. Practical strategy-resistant privacy-preserving elections. In *European Symposium on Research in Computer Security (ESORICS'18)*. Springer, 2018.

[11] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy (S&P'08)*, 2008.

[12] V. Cortier, D. Galindo, S. Glondu, and M. Izabachene. Distributed ElGamal à la Pedersen - Application to Helios. In *Workshop on Privacy in the Electronic Society (WPES'13)*, 2013.

[13] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO'94*. Springer, 1994.

[14] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Public Key Cryptography (PKC'01)*. Springer, 2001.

[15] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6), 1962.

[16] R. Goré and E. Lebedeva. Simulating STV hand-counting by computers considered harmful: A.C.T. In *International Joint Conference on Electronic Voting (EVoteID'17)*. Springer, 2017.

[17] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. CHVote system specification. Cryptology ePrint Archive, Report 2017/325, 2017.

[18] R. Haenni and P. Locher. Performance of shuffling: Taking it to the limits. In *Financial Cryptography and Data Security (FC'20)*. Springer, 2020.

[19] T. Haines, D. Pattinson, and M. Tiwari. Verifiable homomorphic tallying for the Schulze vote counting scheme. In *Verified Software. Theories, Tools, and Experiments (VSTTE'19)*. Springer, 2019.

[20] C. Hazay, G. Mikkelsen, T. Rabin, and T. Toft. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. *Journal of Cryptology*, 2019.

[21] C. Jost, H. Lam, A. Maximov, and B. Smeets. Encryption performance improvements of the Paillier cryptosystem. Cryptology ePrint Archive, Report 2015/864, 2015. https://eprint.iacr.org/2015/864.

[22] R. Kuesters, J. Liedtke, J. Mueller, D. Rausch, and A. Vogt. Ordinos: A verifiable tally-hiding e-voting system. In *IEEE European Symposium on Security and Privacy (EuroS&P'20)*, 2020.

[23] H. Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. In *ASIACRYPT'03*. Springer, 2003.

[24] H. Lipmaa and T. Toft. Secure equality and greater-than tests with sublinear online complexity. In *Automata, Languages, and Programming (ICALP'13)*. Springer, 2013.

[25] B. L. Meek. Une nouvelle approche du scrutin transférable. *Mathématiques et Sciences humaines*, 25, 1969.

[26] T. Nishide and K. Sakurai. Distributed Paillier cryptosystem without trusted dealer. In *Information Security Applications (WISA 2010)*. Springer, 2010.

[27] T. P. Pedersen. A threshold cryptosystem without a trusted party. In *EUROCRYPT'91*. Springer, 1991.

[28] M. Pollack. The maximum capacity through a network. *Operations Research*, 8(5), 1960.

[29] G. Poupard and J. Stern. Security analysis of a practical "on the fly" authentication and signature generation. In *EUROCRYPT'98*. Springer, 1998.

[30] B. Schoenmakers and P. Tuyls. Practical two-party computation based on the conditional gate. In *ASIACRYPT'04*. Springer, 2004.

[31] B. Schoenmakers and P. Tuyls. Efficient binary conversion for Paillier encrypted values. In *EUROCRYPT'06*. Springer, 2006.

[32] B. Schoenmakers and M. Veeningen. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In *Applied Cryptography and Network Security (ACNS'15)*. Springer, 2015.

[33] M. Schulze. A New Monotonic, Clone-independent, Reversal Symmetric, and Condorcet-consistent Single-winner Election Method. *Social Choice and Welfare*, 36, 2011.

[34] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1), 1962.

[35] R. Wen and R. Buckland. Mix and test counting in preferential electoral systems. Technical report, University of New South Wales, 2008.

[36] D. Wikström. A commitment-consistent proof of a shuffle. In *Information Security and Privacy (ACISP'09)*. Springer, 2009.

Appendix: detailed algorithms and complexities

In this Appendix, we give as many details as we think are necessary to reproduce the data given in the various tables of the article. This also includes algorithms for the basic functionalities that can be combined to get trade-offs that are different from the one selected in the main body of the article.

The structure of the Appendix follows that of the article. In all our tables, the communication costs are expressed in terms of broadcast (denoted B) and full-rounds (denoted R). The unit of the transcript size is the key length. This corresponds to half the size of a ciphertext in both Paillier (typically 3072 bits) and ElGamal (typically 256 bits) settings. Finally, we use the notation $\texttt{Enc}(x)$ for any encryption of x with a fresh randomness, and E_x for a trivial encryption of x with a trivial randomness (*i.e.* 0 in the ElGamal setting, 1 in the Paillier setting).

Appendix A. Building blocks

A.1. ElGamal and Paillier encryptions

In this section, we recall the encryption algorithms in the Paillier and ElGamal cryptosystems. Both are additively homomorphic. This allows efficient addition, subtraction, negation (flipping an encrypted bit) and re-encryption, without resorting to MPC. These are extremely useful for various uses. We sum up their complexity in Figure 7.

ElGamal encryption.

In the ElGamal setting, G is a group of prime order q and public generator g. The public encryption key is a group element h, whose discrete logarithm is the corresponding decryption key. To encrypt a message $m \in \mathbb{Z}_q$ under h, one chooses $r \in_r \mathbb{Z}_q$ and compute

$$\texttt{Enc}(m, r) = (g^r, g^m h^r).$$

Note that this is different from the textbook ElGamal cryptosystem, since we encrypt g^m instead of m. Therefore, decrypting will require do solve a discrete logarithm problem and only small values of m can be efficiently decrypted. Hence we assume that computing g^m is negligible compared to the two other exponentiations. This modification grants the ElGamal cryptosystem the desired homomorphic property. The `Add` and `Sub` are simply point-wise multiplication and division of ciphertexts, and we will often just use the multiplication or division symbols in our algorithms, without explicitly mentioning that they encode `Add` and `Sub`. We also have an almost free `Not` operation (divide an encryption of 1 by the operand) and a cheap `ReEnc` primitive (multiply the operand by an encryption of 0). Note that `Not` can use a fixed (trivial) encryption of 1, while `ReEnc` needs a fresh encryption of 0. Therefore `Add`, `Sub` and `Not` are essentially for free, while `ReEnc` costs two exponentiations.

Paillier encryption.

In the Paillier setting, n is a RSA integer, coprime with it's Euler's totient value $\phi(n)$. In addition, $g \in \mathbb{Z}_{n^2}$ is an element of order n, for instance $g = 1 + n$. To encrypt a message $m \in \mathbb{Z}_n$ under the public key (n, g), one chooses $r \in \mathbb{Z}_n^\times$ and computes

$$\texttt{Enc}(m, r) = g^m r^n \mod n^2.$$

This encryption scheme is naturally homomorphic, which allows to derive the `Add`, `Sub`, `Not` and `ReEnc` primitives as above. Note that when m is small, computing an encryption of m only costs 1 exponentiation, as the other is either negligible or precomputable.

Functionality	Option	Exponentiations
`Enc`	P	1 or 2
	EG	2
`Not`	P/EG	0
`Add`/`Sub`	P/EG	0
`ReEnc`	P	1
	EG	2

Figure 7. Cost of non-MPC homomorphic operations. In the first line, when the plaintext is a small integer, the cost is only 1 exponentiation as the other is either precomputable or negligible.

A.2. Threshold decryption

We recall the distributed algorithms for threshold decryption in the Paillier and ElGamal setting. While threshold ElGamal is standard, there are several algorithms for threshold decryption in Paillier, and it is not straightforward to decide which one is the best. In the following, we consider the work of [14]. In both cases, the overall cost of the `Dec` decryption function is $5a$ exponentiations per authority (where a is the number of authorities), and it requires a single broadcast per authority.

ElGamal decryption.

In the ElGamal setting, the secret s such that $h = g^s$ is shared between the authorities using a Shamir secret sharing scheme, such as Pedersen's distribution scheme [27]. More precisely, there exists a polynomial P of degree t (where t is the threshold) such that $P(0) = s$, while authority i's share is $s_i = P(i)$. Each authority has a public commitment $h_i = g^{s_i}$ to their share, which allows to provide proofs of correct decryption.

In order to decrypt a ciphertext (x, y), each authority computes $w_i = x^{s_i}$ and provides a Zero Knowledge proof that $\log_x(w_i) = \log_g(h_i)$. The w_i are referred to as the *partial tally*. From any $t + 1$ valid partial tallies, the value x^s can be recovered using Lagrange's interpolation. Finally, the plaintext is $m = log_g(y/x^s)$. The operations performed by authority i are described in Algorithm 2.

Algorithm 2: Decryption algorithm for authority i in the ElGamal setting

Require: (g, h), $(h_1, \cdots, h_a)$, s_i, (x, y)
Ensure: m, a decryption of (x, y)

$w_i = x^{s_i}$
$\alpha_i \in_r \mathbb{Z}_q$
$e_{1,i} = g^{\alpha_i}$, $e_{2,i} = x^{\alpha_i}$
$d_i = \texttt{hash}(g||h||h_1||\cdots||h_a||x||y||w_i||e_{1,i}||e_{2,i})$
$r_i = \alpha_i + s_i d_i$
for $j = 1$ *to* a $(j \neq i)$ **do**
  $d_j = \texttt{hash}(g, h, h_1, \cdots, h_a, x, y, w_j, e_{1,j}, e_{2,j})$
  $b_j = (g^{r_j} h_j^{-d_j} == e_{1,j})$
  $b_j = b_j \wedge (g^{r_j} w_j^{-d_j} == e_{2,j})$
$S = \{i\} \bigcup \{j \in [1, a] \mid b_j = 1\}$
Compute Lagrange coefficients λ_k for set $S = \{j_1, \cdots, j_{t+1}\}$ (* keep only the first elements of S if it's larger *)
Return $\log_g \left(y \left(\prod_{k=1}^{t+1} w_{j_k}^{-\lambda_k} \right) \right)$

Paillier decryption.

In the Paillier setting, we use the approach from [14] as it provides a decryption algorithm which is similar to that of the ElGamal setting. In what follows, $g = (1 + n)$. Recall that n and $\phi(n)$ are coprime, so there exists a unique integer s in $\mathbb{Z}_{n\phi(n)}$ such that s is congruent to 1 modulo n and 0 modulo $\phi(n)$. This integer s is shared among the authorities using a Shamir secret sharing scheme, for instance using the work of [20], which can be generalized for an arbitrary number of authorities. Finally, we assume that a public random group element $g' \in_r \mathbb{Z}_{n^2}$ has been chosen, and that each authority has a public commitment $h_i = (g')^{s_i}$ to their share.

To decrypt a ciphertext C, each authority computes $w_i = C^{s_i}$ and provides a Zero Knowledge proof that w_i is well-formed (using the proof from [29]). Let $\Delta = a!$, where a is the number of authorities. For any $t + 1$ valid partial tallies, the value $D = C^{\Delta s}$ can be recovered using Lagrange's interpolation. Note that the Lagrange coefficients are multiplied by Δ because inverting an integer is infeasible modulo $n\phi(n)$, as $\phi(n)$ is unknown. Since Δ and n are coprime, Δ is invertible modulo n and we denote $u = \Delta^{-1} \bmod n$. We compute $D' = D^u \bmod n^2$ and cast D' into $\mathbb{Z}$ in order to derive $m = (D' - 1)/n$. The resulting m is the desired plaintext. The operations performed by authority i are described in Algorithm 3.

Note that another threshold scheme for the Paillier cryptosystem can be found in [26]. It is less similar to the ElGamal threshold scheme, and it requires an honest majority of authorities.

A.3. Zero Knowledge Proofs

Zero Knowledge proofs are ubiquitous in our algorithms. Already, the decryption algorithms we have just mentioned include proofs of correct decryption for the partial tallies. Not trying to be exhaustive, we recall two standard zero knowledge proofs, and give their complexity in terms of exponentiations for the prover, the verifier, as well as the size of the transcript.

Algorithm 3: Decryption algorithm for authority i in the Paillier setting

Require: n, g', $(h_1, \cdots, h_a)$, s_i, C, κ, Q, where κ is a security parameter (typically $\kappa = 128$) and Q an upper bound of $n\phi(n)$ (typically n^2)

Ensure: m, a decryption of C

$w_i = x^{s_i}$

$\alpha_i \in_r \mathbb{Z}_{2^{2\kappa}Q}$

$e_{1,i} = (g')^{\alpha_i}$, $e_{2,i} = C^{\alpha_i}$

$d_i = \texttt{hash}(n||g'||h_1||\cdots||h_a||C||w_i||e_{1,i}||e_{2,i})$

$r_i = \alpha_i + s_i d_i$

for $j = 1$ *to* a $(j \neq i)$ **do**

  $d_j = \texttt{hash}(n||g'||h_1||\cdots||h_a||C||w_j||e_{1,j}||e_{2,j})$

  $b_j = (0 < w < 2^{2\kappa}Q)$

  $b_j = b_j \wedge (g^{r_j} h_j^{-d_j} == e_{1,j})$

  $b_j = b_j \wedge (g^{r_j} w_j^{-d_j} == e_{2,j})$

$S = \{i\} \bigcup \{j \in [1, a] \mid b_j = 1\}$

Compute Lagrange coefficients λ_k for set $S = \{j_1, \cdots, j_{t+1}\}$ (* keep only the first elements of S if it's larger *)

$\Delta = a!$, $u = \Delta^{-1} \bmod n$.

$D = \left(\prod_{k=1}^{t+1} w_{j_k}^{-u\Delta\lambda_k}\right)$

Return $m = (D-1)/n$

All our Zero Knowledge proofs are made non-interactive with the Fiat-Shamir transformation, which requires a hash function $\texttt{hash}$. We decided to incorporate this function as an argument of the algorithms as some specific prefixes should be incorporated into the hash depending on the context. The precise specification of how the hashes should be prefixed to provide the correct level of security is out of the scope of our work, but still needs to be mentioned.

Standard $0/1$ encryption.

In all our algorithms, in particular on the voter side, it is extremely common to prove that some encryption is an encryption of either 0 or 1. We give Algorithm 4 which allows to produce such a proof given a bit b, a randomness r and an encryption $X = \texttt{Enc}(b, r)$. This proof has the form $\pi_{01} = (e_1, e_2, \sigma_1, \rho_1, \sigma_2, \rho_2)$. To verify such a proof, one can simply compute $d = \texttt{hash}(X||e_1||e_2)$ and check that the following equations are verified:

$$\begin{aligned}
&\sigma_1 + \sigma_2 = d\\
&\texttt{Enc}(0, \rho_1)(X/E_1)^{-\sigma_1} = e_1\\
&\texttt{Enc}(0, \rho_0)(X/E_0)^{-\sigma_0} = e_0.
\end{aligned}$$

Algorithm 4: ZKP01

Require: $R, \texttt{hash}, X, r, b \in \{0, 1\}$ such that $X = \texttt{Enc}(b, r)$

(* R is $\mathbb{Z}_q$ for ElGamal encryption, $\mathbb{Z}_n^\times$ for Paillier encryption *)

Ensure: π_{01}, a Zero Knowledge proof that X is an encryption of 0 or 1

$w \in_r R$

$e_b = \texttt{Enc}(0, w)$

$\sigma_{1-b}, \rho_{1-b} \in_r R$

$e_{1-b} = \texttt{Enc}(0, \rho_{1-b})(X/E_{1-b})^{-\sigma_{1-b}}$

$d = \texttt{hash}(X||e_1||e_2)$

$\sigma_b = d - \sigma_{1-b}$

$\rho_b = w + r\sigma_b$

Return $\pi_{01} = (e_1, e_2, \sigma_1, \rho_1, \sigma_2, \rho_2)$

Proof of a shuffle.

We consider a prover P which is given a list of ciphertexts $C_1, \cdots, C_n$. The prover wants to shuffle the ciphertexts and output $C'_1, \cdots, C'_n$, while providing a proof $\pi_{\texttt{shuffle}}$ that there exists a permutation φ such that for all i, $C'_i = \texttt{ReEnc}(C_{\varphi(i)})$. To do so, one can for instance apply the protocol from [36], which is a very standard approach in the ElGamal setting.

ZKP	P/EG	Exp. for Prover	Exp. for Verifier	Transcript size
π_{01}	EG	6	8	6
	P	4	4	6
$\pi_{\texttt{shuffle}}$	EG	$10n+5$	$9n+11$	$10n+10$
	P	$8n+4$	$8n+10$	$10n+10$

Figure 8. Cost of various Zero Knowledge Proofs.

This approach can be adapted in the Paillier setting. Note that a mixnet procedure can be derived from a proof of a shuffle, using one round of communication.

A.4. Encoding of encrypted integers for MPC

From now, we go on with proper MPC primitives and sum up their complexities in Figure 9. We will explain thoroughly how they can be obtained.

We stress that each functionality can be implemented in several manners, depending on the context. Choosing the implementation that best suits their need is left to the readers, and may imply implementations which are not presented in this section, as a more efficient replacement could exist in some specific context. For instance, we give a generic algorithm for adding two bit-wise encrypted integers (Algorithm 9). But when one of the operand is known in the clear, another implementation is available, which is twice more efficient (Algorithm 19). Such an optimisation is often possible for a very specific use, but we cannot anticipate every single specific situation.

A fundamental choice is however to decide how to encode integers. As explained in the main body of the article, in the ElGamal setting it is not possible to perform advanced arithmetic (in particular multiplication or comparison) if the integer is encrypted in the natural way (m is directly the integer to be dealt with). The *bit-encoding* means that each bit of the integer m is encoded individually. We recall that everything with an exponent $^{\text{bits}}$ means that this encoding is used. In the Paillier setting, the `BinExpand` function allows to convert an integer in the natural encoding to its bit-encoding. We postpone the description of this conversion; we will discuss it with other Paillier-specific algorithms (see Section A.9).

A.5. `CondSetZero` (abbreviated as `CSZ`) and selectors

The `CondSetZero` functionality is the basis of many other MPC primitives. Given two ciphertexts X and Y which encrypt x and y respectively where $y \in \{0,1\}$, it returns an encryption of xy. This algorithm is the basis of virtually all of our MPC algorithm in the ElGamal setting, but it could be used in the Paillier setting as well. We present two algorithms for it. Algorithm 5 is extracted from [30], where it is referred to as the *conditional gate*. It requires a commitment scheme, and we took the liberty to chose Pedersen's commitment which requires two independent generators g_0 and g_1, but any other commitment scheme could be used as well. In the Paillier case, there exists a more efficient and more general algorithm [32], that we present as Algorithm 6: this is a general multiplication algorithm that does not require y to be a bit. The costs of these two variants are given in Figure 9.

We remark that the `CGate` algorithm requires raising the ciphertext to the power $1/2$. This can be done by raising to the power $(q+1)/2$ in ElGamal, or $(n+1)/2$ in Paillier. In the latter case, this works because while the full-group order is unknown, the cleartexts belong to $\mathbb{Z}_n$. Therefore, even though `Mul` is a faster implementation of `CSZ` in the Paillier setting, `CGate` could be used as well.

In line 5, Algorithm 5 includes a Zero-Knowledge proof that X_i, Y_i, c_i are well formed. For this, the authority i can prove that she knows a preimage of X_i, Y_i, c_i by the morphism $b, r_1, r_2, r_3 \mapsto X_{i-1}^b \texttt{Enc}(0, r_1), Y_{i-1}^b \texttt{Enc}(0, r_2), g_0^b g_1^{r_3}$. This can be done as follows.

1) Choose $\alpha, \beta_1, \beta_2, \beta_3 \in_r \mathbb{Z}_q$ and compute $e_1 = X_{i-1}^{\alpha}\texttt{Enc}(0, \beta_1)$,$e_2 = Y_{i-1}^{\alpha}\texttt{Enc}(0, \beta_2)$ and $e_3 = g_0^{\alpha} g_1^{\beta_3}$.
2) Compute $d = \texttt{hash}(g||h||g_0||g_1||X_{i-1}||Y_{i-1}||X_i||Y_i||c_i||e_1||e_2||e_3)$ and $a_0 = \alpha + db$, $a_1 = \beta_1 + dr_1$, $a_2 = \beta_2 + dr_2$ and $a_3 = \beta_3 + dr_3$.
3) Return $(e_1, e_2, e_3, a_0, a_1, a_2, a_3)$.

To verify the proof, one can simply compute $d = \texttt{hash}(g||h||g_0||g_1||X_{i-1}||Y_{i-1}||X_i||Y_i||c_i||e_1||e_2||e_3)$ and check that

$$\begin{aligned} X_{i-1}^{a_0}\texttt{Enc}(0, a_1)X_i^{-d} &= e_1 \\ Y_{i-1}^{a_0}\texttt{Enc}(0, a_2)Y_i^{-d} &= e_2 \\ g_0^{a_0} g_1^{a_3} c_i^{-d} &= e_3. \end{aligned}$$

Since each authority has to check all the other authorities' proofs, this algorithm costs approximately $20a$ exponentiations per authority, where a is the number of authorities. The real value depends on the threshold, since a threshold decryption

Functionality	Option	Algorithm	Exp per trustee	Comm. cost	Transcript size
Dec	P/EG	Dec	$5a$	B	$4a$
RandBit	P/EG	RandBit	$3a+2$	R	$6a$
CSZ	EG	CGate	$20a$	$R+B$	$18a$
	P	Mul	$10a$	$2B$	$11a$
Select	P/EG	Select	CSZ	CSZ	CSZ
SelectInd	P/EG	SelectInd	nCSZ	CSZ	nCSZ
Neg^{bits}	P/EG	Neg^{bits}	$(m-1)$CSZ	$(m-1)$CSZ	$(m-1)$CSZ
Add^{bits}	P/EG	Add^{bits}	$(2m-1)$CSZ	$(2m-1)$CSZ	$(2m-1)$CSZ
	Sublinear P/EG	$\text{UFCAdd}^{\text{bits}}$	$m(\frac{3}{2}\log m+2)$CSZ	$2(1+\log m)$CSZ	$m(\frac{3}{2}\log m+2)$CSZ
Sub^{bits}	P/EG	Sub^{bits}	$(2m-1)$CSZ	$(2m-1)$CSZ	$(2m-1)$CSZ
	LT P/EG	$\text{SubLT}^{\text{bits}}$	$(2m-1)$CSZ	$(2m-1)$CSZ	$(2m-1)$CSZ
	LT+EQ P/EG	$\text{SubLT}^{\text{bits}}$	$(3m-2)$CSZ	$(2m+\log m)$CSZ	$(3m-2)$CSZ
	Sublinear P/EG	$\text{UFCSub}^{\text{bits}}$	$m(\frac{3}{2}\log m+2)$CSZ	$2(1+\log m)$CSZ	$m(\frac{3}{2}\log m+2)$CSZ
LT^{bits}	LT P/EG	$\text{SubLT}^{\text{bits}}$	$(2m-1)$CSZ	$(2m-1)$CSZ	$(2m-1)$CSZ
	LT+EQ P/EG	$\text{SubLT}^{\text{bits}}$	$(3m-2)$CSZ	$(2m+\log m)$CSZ	$(3m-2)$CSZ
	Sublinear P/EG	CLT^{bits}	$(4m-3)$CSZ	$2(1+\log m)$CSZ	$(4m-3)$CSZ
	Sublinear+EQ P/EG	CLT^{bits}	$(5m-4)$CSZ	$2(1+\log m)$CSZ	$(5m-4)$CSZ
EQ^{bits}	P/EG	EQ^{bits}	$(2m-1)$CSZ	$(1+\log m)$CSZ	$(2m-1)$CSZ
EQ	Precomp P	EQH	$21ma+75a$ $+4(m+1)$	$R+8B$	$(22m+28)a$
GT	Precomp P	GTH	$(27m+146\log m)a$ $+8m+9a+5\log m$	$(2R+13B)\log m$	$(28m+50\log m)a$ $+6a$
BinExpand	P	BinExpand	$12ma+53a+3m$	$R+2mB$	$(17m+21)a$
$\text{Aggreg}^{\text{bits}}$	EG	$\text{Aggreg}^{\text{bits}}$	$3n$CSZ	$(\log n+1)\log n$CSZ	$3n$CSZ
Mul^{bits}	P/EG	Mul^{bits}	$3m^2$CSZ	$2m^2$CSZ	$3m^2$CSZ
Div^{bits}	P/EG	Div^{bits}	$(3m-1)r$CSZ	$2mr$CSZ	$(3m-1)r$CSZ
$\text{MinMax}^{\text{bits}}$	naive P/EG	$\text{MinMax}^{\text{bits}}$	$n(6m+2\log n)$	$2m\log n$CSZ	$n(6m+2\log n)$CSZ
	sublinear P/EG	$\text{MinMax}^{\text{bits}}$	$n(10m+2\log n)$CSZ	$\log n(3+2\log m)$CSZ	$n(10n+2\log n)$CSZ
Mixnet	EG	[36]	$(9n+11)a$ $+n-6$	R	$10(n+1)a$
	P	[36]	$(8n+10)a$	R	$10(n+1)a$

Figure 9. Cost of various MPC primitives: basic functionalities for logic, integer arithmetic, and a few advanced functions. The Option column includes whether this is available in Paillier (P) or ElGamal (EG). The notations are a for the number of authorities, m for the bit-length of the operands, n for the number of operands, r for the precision (in the division). All logarithms are in base 2. The communication costs are expressed in terms of broadcast (denoted B) and full-rounds (denoted R). The unit of the transcript size is the key length. This corresponds to half the size of a ciphertext in both Paillier and ElGamal settings.

Algorithm 5: `CGate`

Require: X, Y such that X (resp. Y) is an encryption of x (resp. y), with $y \in \{0, 1\}$
Ensure: $Z = \texttt{Enc}(xy)$

Compute $Y_0 = E_{-1}Y^2$, set X_0 at X
for $i = 1$ *to* a **do**
  Authority i chooses $r_1, r_2, r_3 \in_r \mathbb{Z}_q$ and $b \in_r \{-1, 1\}$
  She computes $X_i = (X_{i-1})^b \texttt{Enc}(0, r_1)$, $Y_i = (Y_{i-1})^b \texttt{Enc}(0, r_2)$ and $c_i = g_0^b g_1^{r_3}$
  She reveals X_i, Y_i, c_i and a zero knowledge proof that X_i and Y_i are well formed
Each authority verifies the proof of the other authorities
$y_a = \texttt{Dec}(Y_a)$. If $y_a \notin \{-1, 1\}$, the authorities open their commitments c_i and any authority who fails to open her c_i into a value $b \in \{-1, 1\}$ is punished
Compute $Z' = (X_a)^{y_a}$
Return $Z = (XZ')^{\frac{1}{2}}$

Algorithm 6: `Mul`

Require: X, Y, Paillier encryptions of $x, y \in \mathbb{Z}_n$
Ensure: Z, an encryption of xy

Authority i chooses $s_i \in_r \mathbb{Z}_n$ and $r_i \in_r \mathbb{Z}_n$
The authorities simultaneously reveal $S_i = \texttt{Enc}(s_i, r_i)$, $Y_i = Y^{s_i}$ as well as a Zero Knowledge proof that S_i and Y_i are well formed
Each authority check the proof of the other authorities
$x' = \texttt{Dec}(X \prod_i S_i)$ (* $x' = x + \sum_i s_i$ *)
They compute $Z' = Y^{x'}$, then $Z = Z' / \prod_i Y_i$

is needed in line 7. The value $20a$ is a reasonable upper-bound. The communication cost is one round of communication and a broadcast.

In Algorithm 6, there is also a Zero Knowledge proof required for the well-formedness of Y_i, S_i. The authority i can proceed as follows.

1) Choose $\alpha, \beta \in_r \mathbb{Z}_n$ and compute $e_1 = \texttt{Enc}(\alpha, \beta)$ and $e_2 = Y^\alpha$
2) Compute $d = \texttt{hash}(g, n, Y, Y_i, S_i, e_1, e_2)$, $a_1 = \alpha + ds_i$ and $a_2 = \beta r_i^d$
3) Return (e_1, e_2, a_1, a_2)

To verify the proof, one can simply compute $d = \texttt{hash}(g, n, Y, Y_i, S_i, e_1, e_2)$ and check that

$$\texttt{Enc}(a_1, a_2) S_i^{-d} = e_1$$
$$Y^{a_1} Y_i^{-d} = e_2.$$

Since each authority has to check all the other authorities' proofs, the overall cost of the procedure is approximately $9a + 3$ exponentiation, where a is the number of authorities. The communication is also lower than in Algorithm 5 since it only requires broadcasts.

Selectors derived from `CSZ`

From the `CSZ` functionality, it is easy to build the `Select` function that allows to select a ciphertext among two, according to the value of an encrypted bit: Given two ciphertexts X and Y, and B an encryption of a bit b, this will return a reencryption of X if $b = 0$, of Y otherwise. This allows to remove branching by computing both branches and keeping only the relevant one, without revealing which one. An algorithm for `Select` is given in Algorithm 7. This can be generalized for bit-wise encrypted integers X^{bits}, Y^{bits} (simply return $(\texttt{Select}(X_i, Y_i, B))_i$), or for wider branches with more than two possibilities (see Algorithm 8). The cost of `Select` is the same as the cost of `CSZ`.

Algorithm 7: `Select`

Require: X, Y, B encryptions of x, y and b with $b \in \{0, 1\}$
Ensure: Z, an encryption of x if $b = 0$, of y otherwise.

Return $Z = X\texttt{CSZ}(Y/X, B)$

Algorithm 8: `SelectInd`

Require: $[X_i]$, an array of ciphertexts, $[B_i]$, an array of ciphertexts of the same size, such that one of them is an encryption of 1 while the others are encryptions of 0.
Ensure: Z, an encryption of x_i such that $b_i = 1$.
Return $Z = \prod_i \texttt{CSZ}(X_i, B_i)$

A.6. Basic integer arithmetic: addition, subtraction, comparison

Due to the homomorphic property, `Add` and `Sub` can be simply implemented by multiplication and division of the ciphertexts, when we want to work in the natural encoding. In bit-encoding, however, we need to build appropriate algorithms for these. We remark readily that comparing integers can be done with a subtraction, where we return the final borrow bit. However, we can sometimes do better.

Linear addition and subtraction.
We have in input the (encrypted) bits X^{bits} and Y^{bits} of x and y, where x and y are the m-bit plaintexts associated to X^{bits} and Y^{bits} respectively. For addition, *i.e.* computing Z^{bits}, an encryption of $x + y$ modulo 2^m, we reproduce in Algorithm 9 the method found in [30]. The idea is simply to reproduce the schoolbook algorithm for the addition, with four variables X_i, Y_i, Z_i and R which represent (encryptions of) the i^{th} bit of X and Y, the i^{th} bit of the sum and the current value of the carry. The value of z_i, the plaintext associated with Z_i, is simply $x_i \oplus y_i \oplus r_i$, and the new value of R can be obtained with a truth table from the three other variables.

Algorithm 9: $\texttt{Add}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1}), (Y_0, \cdots, Y_{m-1})$ bit-wise encryptions of x and y
Ensure: $Z_0, \cdots, Z_{m-1}$, bitwise encryption of $x + y$ modulo 2^m
$R = \texttt{CSZ}(X_0, Y_0)$
$Z_0 = X_0 Y_0 / R^2$ (* $x_0 \oplus y_0$ *)
for $i = 1$ *to* $m - 1$ **do**
  $A = X_i Y_i / \texttt{CSZ}(X_i, Y_i)^2$ (* $x_i \oplus y_i$ *)
  $Z_i = AR / \texttt{CSZ}(A, R)^2$ (* $x_i \oplus y_i \oplus r$ *)
  $R = (X_i Y_i R / Z_i)^{\frac{1}{2}}$
Return $Z_0, \cdots, Z_{m-1}$

A first approach for writing a subtraction algorithm that returns an encryption of $x - y \bmod 2^m$ is to modify Algorithm 9 as follows. Computing $x - y \bmod 2^m$ is the same as computing $x + (-y) \bmod 2^m$. Turning y to $-y \bmod 2^m$ is performed by flipping each bit (replacing y_i by $1 - y_i$) then adding 1. This gives Algorithm 10.

Algorithm 10: $\texttt{Sub}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1})$, $(Y_0, \cdots, Y_{m-1})$, bit-wise encryptions of x and y.
Ensure: $(Z_0, \cdots, Z_{m-1})$, bit-wise encryption of $x - y$ modulo 2^m.
$A = \texttt{CSZ}(X_0, Y_0)$
$Z_0 = (X_0 Y_0)/A^2$ (* $x_0 \oplus (1 - y_0) \oplus 1$ *)
$R = A\,\texttt{Not}(Y_0)$ (* $x_0 \vee \neg y_0$ *)
for $k = 1$ *to* $m - 1$ **do**
  $A = X_k\,\texttt{Not}(Y_k) / \texttt{CSZ}(X_k, \texttt{Not}(Y_k))^2$
  $Z_k = AR / \texttt{CSZ}(A, R)^2$
  $R = (X_k \texttt{Not}(Y_i) R / Z_k)^{\frac{1}{2}}$
Return $Z_0, \cdots, Z_{m-1}$

Algorithm 10 is interesting for its similarity with Algorithm 9, but another way to perform the subtraction is also to use the schoolbook algorithm, just as for Algorithm 9. The advantage is that the carry is then the classical borrow of the subtraction, and not an artificial carry in an equivalent addition modulo 2^m. Hence, if the last borrow bit is required in order to get a comparison algorithm from the subtraction, Algorithm 11 must be preferred.

When the required comparison is just an equality test, there is a simpler and cheaper approach. Indeed, testing whether two integers are equal is the same as testing whether all of their bits are equal, therefore the associativity of the logical $\wedge$

Algorithm 11: $\texttt{SubLT}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1})$, $(Y_0, \cdots, Y_{m-1})$, bit-wise encryption of x and y.
Ensure: $(Z_0, \cdots, Z_{m-1}), R$ where Z_i are bit-wise encryption of $x - y$ modulo 2^m and $R = \texttt{Enc}(x < y)$.

$A = \texttt{CSZ}(X_0, Y_0)$
$Z_0 = X_0 Y_0 / A^2$ (* $x_0 \oplus y_0$ *)
$R = Y_0 / A$ (* $y_0 \wedge \neg x_0$ *)
for $k = 1$ *to* $m - 1$ **do**
  $A = \texttt{CSZ}(Y_k, R)$
  $B = Y_k R / A^2$ (* $y_k \oplus r$ *)
  $C = \texttt{CSZ}(X_k, B)$
  $Z_k = X_k B / C^2$ (* $x_k \oplus y_k \oplus r$ *)
  $R = Y_k / (AC)$ (* $(y_k \wedge r) \vee [(y_k \vee r) \wedge \neg x_k]$ *)
Return $(Z_0, \cdots, Z_{m-1}), R$

operator can be exploited to parallelize the procedure. This gives Algorithm 12, the cost of which is $(2m-1)\texttt{CSZ}$ in term of transcript size and exponentiations per authority, but only $(1 + \log m)\texttt{CSZ}$ in term of communication cost, using a tree structure.

Algorithm 12: $\texttt{EQ}^{\text{bits}}$

Require: $X_0, \cdots, X_m$, $Y_0, \cdots, Y_m$ bit-wise encryptions of x and y.
Ensure: $Z = \texttt{Enc}(x == y)$, an encryption of 1 if $x = y$, of 0 otherwise.

For all i (in parallel), compute $A_i = \texttt{CSZ}(X_i, Y_i)$.
For all i (in parallel), compute $B_i = E_1 A_i^2 / (X_i Y_i)$ (* $1 - x_i \oplus y_i$ *)
Return $Z = \texttt{CSZ}(B_0, \cdots, B_{m-1})$

Therefore, adding, subtracting or comparing two m-bit integers have roughly the same cost of $(2m-1)\texttt{CSZ}$. The similarity between these algorithms can be exploited to build specialized algorithm which do several operations altogether.

For instance, if one need an algorithm that computes both the subtraction and the full comparison as a ternary value (1 if $x > y$, 0 if $x = y$ or -1 if $x < y$), we can combine these operations by first calling Algorithm 11, then using a $\vee$ composition to test whether all the bits of the output are 0. This leads to a cost of about $3m\texttt{CSZ}$ instead of $4m\texttt{CSZ}$ if done separately. This is useful, for instance in our version of Condorcet.

Finally, we remark that computing the opposite $-x$ of an integer x modulo 2^m can be done faster than using the subtraction algorithm between 0 and x. Algorithm 13 simply flips all bits and then add 1; this is simply a special case of Algorithm 19 which be introduced later on.

Algorithm 13: $\texttt{Neg}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1})$, a bit-wise encryption of x
Ensure: $(Z_0, \cdots, Z_{m-1})$, a bit-wise encryption of $-x$ mod 2^m

$Z_0 = X_0$
$R_0 = \texttt{Not}(X_0)$
for $i = 1$ *to* $m - 1$ **do**
  $R_i = \texttt{CSZ}(X_i, R_{i-1})$
  $Z_i = X_i R_{i-1} / R_i^2$
Return $Z_0, \cdots, Z_{m-1}$

A.7. Arithmetic with sublinear communication complexity

Apart from the equality test, all the previous arithmetic algorithms in the bit-encoding require a number of communication rounds that is proportional to the bit-size of the input integers. This is mostly due to carry and borrow propagations. In order to reduce the number of communication rounds, our idea is to use more sophisticated adder circuits, following the (now classical) approach of Brent and Kung [8]. We do not reproduce the full algorithm from Brent and Kung here but we sketch the key idea and give the resulting algorithms and their complexity (summarized in Figure 9). Recall that the

i^{th} bit of $x+y$ is simply $z_i = x_i \oplus y_i \oplus r_i$, where r_i is the i^{th} carry bit. The idea is to first compute all the $x_i \oplus y_i$ in parallel, then to compute all the r_i in parallel, so as to deduce the result. To perform the second step efficiently, Brent and Kung's approach consists of computing the variables (p_i, g_i) where $p_i = x_i \vee y_i$ and $g_i = x_i \wedge y_i$. Those variable are used to encode elements of a set $\Sigma = \{P, G, K\}$, where P is encoded by $(1,0)$, K by $(0,0)$ and G by $(0,1)$ and $(1,1)$. They represent the fact that the carry bit will be propagated, generated of killed in the i^{th} position. They define an operation $\circ$ as follows (which we slightly modify into an equivalent operation for the sake of presentation).

$$\begin{aligned} P \circ P &= P \\ G \circ P &= G \\ K \circ P &= K \\ x \circ G &= G \\ x \circ K &= K. \end{aligned}$$

In the boolean representation, the $\circ$ law can be computed with the following formula:

$$(p, g) \circ (p', g') = (p \wedge p', g' \vee (p' \wedge g)).$$

It is easy to show that $\circ$ is associative [8], which enables tree-based parallelism for computing all the prefixes of $(p_0, g_0) \circ \cdots \circ (p_{m-1}, g_{m-1})$, which gives essentially the i^{th} carry bit for all i. From here onward, we diverge from [8]'s work since we are not interested in designing hardware, so the unbounded fan-in is not an issue. We deduce the Unbounded Fan-in Composition algorithm, which can be instantiated to compute the addition (Algorithm 14). Algorithm 14 is highly efficient in term of communication since it only requires about $\log(m)$ times more round communications than the one required for $\circ$. However, this comes with an increase in term of computation as the number of calls to $\circ$ is about $\frac{1}{2} m \log(m)$, so the linear approach could be preferable in some cases. To evaluate the complexity, note that the worst-case scenario in term of computational cost is when m is a power of 2, in which case the number of calls to $\texttt{CSZ}$ is easy to derive.

Algorithm 14: $\texttt{UFCAdd}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1}), (Y_0, \cdots, Y_{m-1})$, bit-wise encryptions of x and y.
Ensure: $(Z_0, \cdots, Z_{m-1})$, bit-wise encryption of $x + y \bmod 2^m$

1 **for** $i = 0$ *to* $m-1$ *(* in parallel *)* **do**
2 $A_i = \texttt{CSZ}(X_i, Y_i)$
3 $B_i = X_i Y_i / A_i^2$ (* $x_i \oplus y_i$ *)
4 $P_i = X_i Y_i / A_i$ (* $x_i \vee y_i$ *)
5 $G_i = A_i$ (* $x_i \wedge y_i$ *)
6 $C_{i,j} = (P_j, G_j)$ for all $1 \leq i \leq \lceil \log m \rceil$ and $0 \leq j \leq m-1$
7 **for** $i = 1$ *to* $\lceil \log m \rceil$ **do**
8 **for** $j = 0$ *to* $\lceil m/2^i \rceil - 1$ *(in parallel)* **do**
9 **for** $k = 1$ *to* 2^{i-1} *(in parallel)* **do**
10 $(P, G) = C_{i-1, j2^i + 2^{i-1}}$
11 $(P', G') = C_{i-1, j2^i + 2^{i-1} + k}$ (* do not proceed for this k if $j2^i + 2^{i-1} + k \geq m$ *)
12 $T = \texttt{CSZ}(P', G)$
13 $C_{i, j2^i + 2^{i-1} + k} = (\texttt{CSZ}(P, P'), TG' / \texttt{CSZ}(T, G'))$
14 $Z_0 = B_0$
15 **for** $i = 1$ *to* $m-1$ *(in parallel)* **do**
16 $(\_, G_i) = C_{\lceil \log(i+1) \rceil, i+1}$
17 $Z_i = B_i G_i / \texttt{CSZ}(B_i, G_i)^2$
18 Return $Z_0, \cdots, Z_{m-1}$

The same algorithm can be used for computing subtraction; it only requires to change the initialization of the p_i and g_i. Indeed, we have initially $p_i = x_i \oplus y_i$ and $g_i = y_i \wedge \neg x_i$, so to obtain $\texttt{UFCSub}^{\text{bits}}$, one can just replace line 4 by $P_i = B_i$ and line 5 by $G_i = Y_i / A_i$ in Algorithm 14.

When it comes to comparing two integers, only the last carry bit is of interest so we do not need to compute all the prefixes. In this case, a much simpler algorithm exists and allows to compute the comparison with $m-1$ calls to $\circ$ but a communication cost which remains of the order of $\log(m)$. We call this algorithm Chained Lesser-Than (see Algorithm 15).

Note that this algorithm returns an additional bit R which tells whether the two inputs are equal. If this bit is not needed, some computations can be saved (remove lines 11 and 20).

Algorithm 15: $\texttt{CLT}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1}), (Y_0, \cdots, Y_{m-1})$ bit-wise encryption of x and y.
Ensure: Z, R such that $Z = \texttt{Enc}(x < y)$ and $R = \texttt{Enc}(x = y)$

1 Let $(m_j)_{j=0}^{l-1}$ be the binary representation of m, such that $m = \sum_{j=0}^{l-1} m_j 2^j$
2 **for** $i = 0$ *to* $m - 1$ *(in parallel)* **do**
3 $\quad A_i = \texttt{CSZ}(X_i, Y_i)$
4 $\quad P_i = X_i Y_i / A_i^2$
5 $\quad G_i = Y_i / A_i$
6 $\quad B_{i,0} = (P_i, G_i)$, $A_{i,0} = P_i$.
7 $r = 0$ (* A boolean which tells whether there is a remainder *)
8 $R_B = (E_1, E_0)$, $R_A = E_1$ (* initialize the remainders as neutral element *)
9 **for** $j = 1$ *to* l **do**
10 $\quad$ **for** $i = 0$ *to* $\lfloor l/2^j \rfloor - 1$ *(in parallel)* **do**
11 $\quad\quad A_{i,j} = \texttt{CSZ}(A_{2i,j-1}, A_{2i+1,j-1})$
12 $\quad\quad (P, G) = B_{2i,j-1}$
13 $\quad\quad (P', G') = B_{2i+1,j-1}$
14 $\quad\quad T = \texttt{CSZ}(P', G)$
15 $\quad\quad B_{i,j} = (\texttt{CSZ}(P, P'), TG'/\texttt{CSZ}(T, G'))$
16 $\quad$ **if** $m_{j-1} \wedge \neg r$ *(in parallel)* **then**
17 $\quad\quad R_B = B_{2\lfloor l/2^j \rfloor, j-1}$, $R_A = A_{2\lfloor l/2^j \rfloor, j-1}$
18 $\quad\quad r = 1$
19 $\quad$ **if** $m_{j-1} \wedge r$ *(in parallel)* **then**
20 $\quad\quad A_{2\lfloor l/2^j \rfloor, j-1} = \texttt{CSZ}(A_{2\lfloor l/2^j \rfloor, j-1}, R_A)$
21 $\quad\quad (P, G) = B_{2\lfloor l/2^j \rfloor, j-1}$
22 $\quad\quad (P', G') = R_B$
23 $\quad\quad T = \texttt{CSZ}(P', G)$
24 $\quad\quad B_{2\lfloor l/2^j \rfloor, j} = (\texttt{CSZ}(P, P'), TG'/\texttt{CSZ}(T, G'))$
25 $\quad\quad r = 0$, $R_B = (\texttt{Enc}(1), \texttt{Enc}(0))$, $R_A = \texttt{Enc}(1)$
26 $(\_, G) = B_{0,l}$
27 Return $G, A_{0,l}$

A.8. An example: Searching the minimum and the maximum

As an illustration of the previous functionalities, we describe the `MinMax` algorithm which takes as input a list of bit-wise encrypted integers ${X_1}^{\text{bits}}, \cdots, {X_n}^{\text{bits}}$ and returns two bit-wise encrypted integers $Z^{\text{bits}}, T^{\text{bits}}$ which correspond to the minimum and the maximum of the list, as well as bit-wise encryptions of their indexes in the list ${X_1}^{\text{bits}}, \cdots, {X_n}^{\text{bits}}$. A straightforward implementation would be to linearly scan the list, using a comparison algorithm. However, the min and max operators are associative and as such, allows tree-based parallelization. This gives Algorithm 16, in which $\texttt{LT}^{\text{bits}}$ is either $\texttt{CLT}^{\text{bits}}$ or the second output of $\texttt{SubLT}^{\text{bits}}$, depending on the focus in optimization (use $\texttt{CLT}^{\text{bits}}$ for a better communication cost, and $\texttt{SubLT}^{\text{bits}}$ for a better computation cost). We give the two corresponding costs in Figure 9.

A.9. Paillier-specific algorithms

Conversion from natural to bit-encoding (Paillier only)

We recall here the work from [31] which allows to get the bit-encoding representation from a Paillier-encrypted integer. While the homomorphic property of the Paillier cryptosystem allows extremely efficient solutions for the addition and the subtraction, the comparison is not so easy to perform. Therefore, it is important to provide an algorithm which converts to the bit-encoding.

The idea is to use the mask-and-decrypt paradigm, which consists of applying a random mask r to the encrypted value x, which gives an encryption of $x - r$, to decrypt $y = x - r$ then to perform the relevant operation (here, an addition with

Algorithm 16: $\texttt{MinMax}^{\text{bits}}$

Require: $({X_1}^{\text{bits}}, \cdots, {X_n}^{\text{bits}})$ bit-wise encryptions of $x_1, \cdots, x_n$ and y
Ensure: $Z^{\text{bits}}, T^{\text{bits}}, I^{\text{bits}}, J^{\text{bits}}$, bitwise encryptions of $\min_{i=1}^{n}(x_i)$ and $\max_{i=1}^{n}(y_i)$, as well as their indexes in the input vector

Let $(n_j)_{j=0}^{l-1}$ be the binary representation of n, such that $n = \sum_{j=0}^{l-1} n_j 2^j$
${Z_{i,0}}^{\text{bits}} = {T_{i,0}}^{\text{bits}} = {X_i}^{\text{bits}}$ for all i
${I_{i,0}}^{\text{bits}} = {J_{i,0}}^{\text{bits}} = i^{\text{bits}}$ for all i (* trivial bit-wise encryption of i *)
$r = 0$ (* A boolean which tells whether there is a remainder *)
for $j = 1$ *to* l **do**
  for $i = 1$ *to* $\lfloor l/2^j \rfloor$ *(in parallel)* **do**
    (* Both operations are performed in parallel *)
    $B_Z = \texttt{LT}^{\text{bits}}({Z_{2i-1,j-1}}^{\text{bits}}, {Z_{2i,j-1}}^{\text{bits}})$
    $B_T = \texttt{LT}^{\text{bits}}({T_{2i,j-1}}^{\text{bits}}, {T_{2i-1,j-1}}^{\text{bits}})$
    (* All four operations are performed in parallel *)
    ${Z_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({Z_{2i,j-1}}^{\text{bits}}, {Z_{2i-1,j-1}}^{\text{bits}}, B_Z)$
    ${T_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({T_{2i,j-1}}^{\text{bits}}, {T_{2i-1,j-1}}^{\text{bits}}, B_T)$
    ${I_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({I_{2i,j-1}}^{\text{bits}}, {I_{2i-1,j-1}}^{\text{bits}}, B_Z)$
    ${J_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({J_{2i,j-1}}^{\text{bits}}, {J_{2i-1,j-1}}^{\text{bits}}, B_T)$
  if $m_{j-1} \wedge \neg r$ **then**
    $R_Z = Z_{2\lfloor l/2^j \rfloor, j-1}$, $R_T = T_{2\lfloor l/2^j \rfloor, j-1}$
    $R_I = I_{2\lfloor l/2^j \rfloor, j-1}$, $R_J = J_{2\lfloor l/2^j \rfloor, j-1}$
    $r = 1$
  if $m_{j-1} \wedge r$ **then**
    $i = \lfloor l/2^j \rfloor + 1$
    $B_Z = \texttt{LT}^{\text{bits}}({Z_{2i-1,j-1}}^{\text{bits}}, {R_Z}^{\text{bits}})$
    $B_T = \texttt{LT}^{\text{bits}}(R_T, {T_{2i-1,j-1}}^{\text{bits}})$
    ${Z_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({R_Z}^{\text{bits}}, {Z_{2i-1,j-1}}^{\text{bits}}, B_Z)$
    ${T_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({R_T}^{\text{bits}}, {T_{2i-1,j-1}}^{\text{bits}}, B_T)$
    ${I_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({R_I}^{\text{bits}}, {I_{2i-1,j-1}}^{\text{bits}}, B_Z)$
    ${J_{i,j}}^{\text{bits}} = \texttt{Select}^{\text{bits}}({R_J}^{\text{bits}}, {J_{2i-1,j-1}}^{\text{bits}}, B_T)$
    $r = 0$
Return ${Z_{1,l}}^{\text{bits}}, {T_{1,l}}^{\text{bits}}, {I_{1,l}}^{\text{bits}}, {J_{1,l}}^{\text{bits}}$

r) to deduce the (encrypted) result. The overall process that we call `BinExpand` is described by Algorithm 20. To create a mask, we use Algorithm 18 from [31] which requires Zero Knowledge Ranged proofs, such as the ones from [23] or more recently [9]. We do not dig too deep into the details as the security, correction and complexity is fully discussed in [31]. We emphasize that these Paillier-specific algorithms use a `RandBit` function given by Algorithm 17 which is also available in ElGamal. The same holds for the $\texttt{AddKnown}^{\text{bits}}$ function of Algorithm 19 which is a variant of $\texttt{Add}^{\text{bits}}$ where one operand is a cleartext. While we did not need them for the tally function that we studied, they might prove useful in other contexts.

Algorithm 17: `RandBit`

Ensure: Z, an encryption of $b \in_r \{0, 1\}$
$Z_0 = \texttt{Enc}(1)$
for $i = 1$ *to* a **do**
  Authority i chooses $s_i \in_r \{-1, 1\}$ and $r \in_r \mathbb{Z}_n$
  She reveals $Z_i = Z_{i-1}^{s_i}\texttt{Enc}(0, r)$, as well as a Zero Knowledge proof of well-formedness
The authorities check each others' proofs
Return $(E_1 Z_a)^{\frac{1}{2}}$

Integer comparison with precomputation and sublinear online complexity (Paillier only)

Algorithm 18: `RandBits` (Paillier only)

Require: m, a number of bits
Ensure: $R, (R_0, \cdots, R_{m-1})$ such that R is an encryption of $r \in_r \mathbb{Z}_n$, while $(R_0, \cdots, R_{m-1})$ are encryptions of the m first (least significant) bits of r

for $i = 0$ *to* $m-1$ **do**
  $R_i = \texttt{RandBit}()$
Each authority i chooses $r_{*,i} \in_r [0, 2^{m+\kappa-1} - 1]$ and publishes $R_{*,i} = \texttt{Enc}(r_{*,i})$, along with a Zero Knowledge Ranged Proof
$R = \prod_{i=1}^{a} R_{*,i}$
for $i = m-1$ *to* 0 **do**
  $R = R^2$
  $R = RR_i$
Return R, $(R_0, \cdots, R_{m-1})$

Algorithm 19: $\texttt{AddKnown}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1})$ bit-wise encryptions of x and bits $(y_0, \cdots, y_{m-1})$
Ensure: $Z_0, \cdots, Z_{m-1}$, bitwise encryption of $x + y$ modulo 2^m

$R = X_0^{y_0}$
$Z_0 = X_0 \texttt{Enc}(y_0, 1)/R^2$ (* $x_0 \oplus y_0$ *)
for $i = 1$ *to* $m-1$ **do**
  $A = X_i \texttt{Enc}(y_i, 1)/(X_i^{y_i})^2$ (* $x_i \oplus y_i$ *)
  $Z_i = AR/\texttt{CSZ}(A, R)^2$ (* $x_i \oplus y_i \oplus r$ *)
  $R = (X_i \texttt{Enc}(y_i, 1) R/Z_i)^{\frac{1}{2}}$
Return $Z_0, \cdots, Z_{m-1}$

Algorithm 20: `BinExpand` (Paillier only)

Require: X, an encryption of $x < 2^m$
Ensure: $X_0, \cdots, X_{m-1}$, the bit-wise encryption of x

$R, (R_0, \cdots, R_{m-1}) = \texttt{RandBits}(m)$
$Y = X/R$
$y' = \texttt{Dec}(Y)$ (* $y' = x - r$ modulo n *)
Let $y = y' - n$ modulo 2^m and $(y_0, \cdots, y_{m-1})$ the bits of y
Return $\texttt{AddKnown}^{\text{bits}}((R_0, \cdots, R_{m-1}), (y_0, \cdots, y_{m-1}))$

Algorithm 21: `RandInv` (Paillier only)

Ensure: R, R', encryptions of $r \in_r \mathbb{Z}_n^x$ and $r' \in \mathbb{Z}_n$ such that $r' = r^{-1}$

The authorities (simultaneously) display two ciphertexts A_i, B_i
$A = \prod_i A_i$, $B = \prod_i B_i$, $C = \texttt{Mul}(A, B)$
$c = \texttt{Dec}(C)$
$R = A$, $R' = B^{c^{-1}}$.
Return R, R'.

Algorithm 22: `Prefixes` (Paillier only)

Require: $M_1, \cdots, M_m$ encryptions of $m_1, \cdots, m_m$, each coprime with n
Ensure: $Z_1, \cdots, Z_m$, encryptions of $m_1, m_1 m_2, \cdots, \prod_i m_i$

for $i = 1$ *to* m *(in parallel)* **do**
  $R_i, R'_i = \texttt{RandInv}()$
  $S_i = \texttt{Mul}(R_{i-1}, M_i)$ (* with $R_0 = 1$ *)
  $S_i = \texttt{Mul}(S_i, R'_i)$ (* $s_i = r_{i-1} m_i r_i^{-1}$ *)
  The authority decrypt S_i to get s_i.
for $i = 2$ *to* m *(in parallel)* **do**
  $a_i = \prod_{j=1}^{i} s_j$
  $Z_i = R_i^{a_i}$
Return $Z_1, \cdots, Z_m$ (* with $Z_1 = M_1$ *)

Algorithm 23: `EQH` (Paillier only)

Require: X, m, P_m, where X is an encryption of an integer $x << n$ and P_m the unique polynomial of degree m such that $P_m(1) = 1$ and $P_m(k) = 0$ for $k \in \{2, \cdots, m+1\}$
Ensure: Z, an encryption of 1 if $x = 0 \bmod 2^m$, of 0 otherwise

$R, R_{m-1}, \cdots, R_0 = \texttt{RandBits}(m)$
$M, M' = \texttt{RandInv}()$
$M_1, \cdots, M_m = \texttt{Prefixes}(M, \cdots, M)$
$A = X/R$
$a = \texttt{Dec}(A)$
Let $a_0, \cdots, a_{m-1}$ be the bit representation of $a - n$ modulo 2^m
$H = \texttt{Enc}(1) \prod_{i=0}^{m-1} \texttt{Enc}(a_i) R_i^{1-2a_i}$ (* $h = 1 + \sum_{i=0}^{m-1} a_i \oplus r_i$ *)
$M_H = \texttt{Mul}(M', H)$
$m_H = \texttt{Dec}(M_H)$
for $i = 0$ *to* m *(in parallel)* **do**
  $H_i = M_i^{(m_H)^i}$
Return $Z = \prod_{i=0}^{m} H_i^{\alpha_i}$ (* where the α_i are the coefficients of P_m *)

Algorithm 24: `GTH` (Paillier only)

Require: X, Y, l, two encryptions of l-bit integers x and y
Ensure: Z, T, encryptions of $(x \geq y)$ and $(x = y)$

if $l = 1$ **then**
  $A = \texttt{Mul}(X, Y)$
  $T = E_1 A^2 / (XY)$ (* $\neg(x \oplus y)$ *)
  $A\texttt{Not}(Y)$ (* $\neg(y \wedge \neg x)$ *)
$R, R_\top, R_\bot = \texttt{BinExpand}(l)$
$W = \texttt{Enc}(2^l) X / Y$
$M = WR$
$m = \texttt{Dec}(M)$
$m_\top = m \bmod 2^{l/2}$, $m_\bot = \lfloor m / 2^{l/2} \rfloor \bmod 2^{l/2}$
$B = \texttt{EQH}(\texttt{Enc}(m_\top), R_\top)$ (* $x_\top = y_\top$ *)
$C = \texttt{Mul}(B, \texttt{Enc}(m_\bot - m_\top))\texttt{Enc}(m_\top)$ (* $m_\bot$ if $b = 1$, $m_\top$ otherwise *)
$D = \texttt{Mul}(B, R_\bot / R\top) R_\top$ (* $r_\bot$ if $b = 1$, $r_\top$ otherwise *)
$F = \texttt{Enc}(1) / \texttt{GTH}(C, D)$
$W' = F^{2^l} \texttt{Enc}(m \bmod 2^l) / (R_\top^{l/2} R\bot)$ (* $w \bmod 2^l$ *)
Return $Z = (W/W')^{1/2^l}$, T

We mention here a work from [24], which is only exploitable in the Paillier setting. They present some algorithms for the equality test and the comparison which are mostly precomputable. We do not go into all the details here and refer to [24] for a more complete description. To compare two m bits integers x and y, Lipmaa and Toft suggest to create the unique polynomial P_m such that $P_m(1) = 1$ and $P_m(k) = 0$ for $k \in \{2, \cdots, m+1\}$. Their strategy is to first compute the Hamming weight h of $x - y$, then to evaluate P_m on $1 + h$, from which they derive the result. To do so, they use some classical primitives in MPC (Algorithms 17, 21 and 22). The overall process is presented in Algorithm 23. The advantage of this approach is that the procedure can be precomputed so that only a negligible part has to be done *online*, after the operands are known. Compared to Algorithm 12, Algorithm 23 does not require the costly binary expansion as the inputs do not have to be bit-wise encrypted. The complexity of the procedure is less dependent in m, the bit size of the integers to compare, but is of the same order. When m is small, which might be the case in an e-voting setting, the complexity is much higher due to some constant overloads. However, since most of the procedure can be precomputed, the approach is of interest even when m is small.

The `RandInv` Algorithm 21 (adapted from [6]) allows to (collectively) generate two ciphertexts R, R', which encrypt respectively r and r'. The plaintext r is a random invertible integer (modulo n, the Paillier public key), while $r' = r^{-1}$.

The `Prefixes` Algorithm 22 (adapted from [6]) takes as input m ciphertexts $M_1, \cdots, M_m$ which are encryptions of $m_1, \cdots, m_m$. This algorithm returns ciphertexts $Z_1, \cdots, Z_m$ such that Z_i is an encryption of $\prod_{1 \leq j \leq i} m_j$.

From these, in [24], the authors present two algorithms for the inequality test in the Paillier setting, but we will only present one of them. The idea is to use a recursive algorithm which first tests the equality of the most significant halves of x and y, using Algorithm 23. If they are equals, we recursively compare the integers represented by the other halves. If not, we recursively compare the integers represented by the most significant halves. The main process is given in Algorithm 24. Note that at line 5, we took the liberty to denote $R_\top, R_\bot$ the result of `BinExpand` while `BinExpand` returns encryptions of the form $R, (R_0, \cdots, R_{l-1})$. We can derive $R_\bot$ as $\prod_{i<l/2}(R_i)^{2^i}$ and $R_\top$ in a similar manner.

A.10. Advanced arithmetic: aggregation, multiplication and division

In the Paillier setting, integers can be represented in the natural encoding, and we already gave a multiplication algorithm `Mul` as a way to implement the `CSZ` functionality. We now come to the more difficult question of doing multiplication and other arithmetic operations in the bit-encoding, in order to have them available in the ElGamal setting.

Aggregation of several encrypted bits

This $\texttt{Aggreg}^{\text{bits}}$ operation is ubiquitous in e-voting. More often than not, the ballots of the voters are encoded as a sequence of encrypted bits and the first step of the tally is to aggregate some of them, *i.e.* counting all the bits that are set at a given position. The resulting encrypted integers should be in the bit-encoding format, so as to be able to perform comparisons (for instance). The algorithm for that is pretty simple: we just use repeatedly the addition algorithm 9, each time with the minimal value of m, the bit length of the operands.

For simplicity in Algorithm 25, we give the process when the number of bits to aggregate (denoted n) is a power of 2. In this algorithm, we took the liberty to denote $\texttt{Add}^{\text{bits}}$ an addition algorithm which returns $m + 1$ encrypted bits when the operands' bitsize is m (the last bit is the carry bit, so we just add $Z_m = R$ in Algorithm 9). Note that at line 4, the $n/2^i$ calls to $\texttt{Add}^{\text{bits}}$ are made with inputs of length i, so that the cost is exactly $(2i-1)\texttt{CSZ}$. Therefore the cost of the procedure is

$$\begin{aligned}\sum_{i=1}^{\log n} \frac{n}{2^i}(2i-1)\texttt{CSZ} &\leq \sum_{i=1}^{\infty} \frac{2i-1}{2^i} n\texttt{CSZ} \\ &\leq 3n\texttt{CSZ}\end{aligned}$$

As for the communication cost, the process can be parallelized with a classical tree-based approach, since the addition is an associative operation.

Multiplication of integers in the bit-encoding.

In Algorithm 26, we detail the schoolbook algorithm for multiplication. This procedure is quite costly, as it requires about $3m^2\texttt{CSZ}$ for the computation cost and transcript size, and $2m^2\texttt{CSZ}$ for the communication cost, where m is the bitsize of the input integers.

Schoolbook division algorithm.

For the Single Transferable Vote (see Section 6), we chose to represent fractions with a fixed number of binary places so that a fraction is encoded and encrypted as an integer. This allows to re-use most of the primitives from this section, while providing a certain degree of precision and generality. From the schoolbook division algorithm, we derive Algorithm 27, which takes as inputs bit-wise encryptions of x and y with $y > x$ and return the r first binary places of x/y. This algorithm could be generalised for any pair (x, y) (*i.e.* the condition $y > x$ is not necessary), but the restriction is useful in the special case of STV, and gives a simpler description.

Algorithm 25: $\texttt{Aggreg}^{\text{bits}}$

Require: $B_1, \cdots, B_n$ such that for all i, $B_i = E(b_i)$ with $b_i \in \{0,1\}$.
Ensure: $S_0, \cdots, S_{\log n - 1}$ such that for all i, $S_i = E(s_i)$ with $s_i \in \{0,1\}$ and $\sum_{i=0}^{\log n - 1} s_i 2^i = \sum_i b_i$

$B_{0,1}, \cdots, B_{0,n} = B_1, \cdots, B_n$
for $i = 1$ *to* $\log n$ **do**
  for $k = 1$ *to* $n/2^i$ *(in parallel)* **do**
    $B_{i,k} = \texttt{Add}^{\text{bits}}(B_{i-1,2k-1}, B_{i-1,2k})$
Return $B_{\log n,1}$

Algorithm 26: $\texttt{Mul}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m_x-1}), (Y_0, \cdots, Y_{m_y-1})$, bitwise encryptions of x and y
Ensure: $Z_0, \cdots, Z_{m_x+m_y-1}$, bitwise encryption of xy

for $i \in [0, m_x - 1], j \in [0, m_y - 1]$ *(in parallel)* **do**
  $A_{i,j} = \texttt{CSZ}(X_i, Y_j)$
$Z_0 = A_{0,0}$
$(T_0, \cdots, T_{m_y-2}) = (A_{0,1}, \cdots, A_{0,m_y-1})$
for $i = 1$ *to* $m_x - 1$ **do**
  $(T_0, \cdots, T_{m_y-1}) = \texttt{Add}^{\text{bits}}((T_0, \cdots, T_{m_y-2}), (A_{i,0}, \cdots, A_{i,m_y-2}))$
  $T_{m_y} = \texttt{CSZ}(T_{m_y-1}, A_{i,m_y-1})$
  $T_{m_y-1} = T_{m_y-1} A_{i,m_y-1} / T_{m_y}^2$
  $Z_i = T_0$
  for $j = 0$ *to* $m_y - 1$ **do**
    $T_j = T_{j+1}$
for $i = m_x$ *to* $m_x + m_y - 1$ **do**
  $Z_i = T_{i-m_x}$
Return $Z_0, \cdots, Z_{m_x+m_y-1}$

Appendix B. Single choice voting: algorithms for finding the s largest values

In Section 3, we explained that the basic single choice voting or the more advanced list-voting à la D'Hondt reduce both to finding efficiently the s highest value in an array of encrypted integers.

Before giving more details on how this operation can be done, we recall with a bit more details how we deal with the problem of equalities.

B.1. Breaking ties

The idea is simply to force the scores of the candidates to be distinct, even if two candidates received the same number of voices. To do so, the election administrators must agree on an arbitrary ordering of the candidates, which allows to break ties. For instance, it can be decided that, in case of a tie, candidate i wins over candidates j if $i > j$. To put this into act,

Algorithm 27: $\texttt{Div}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m-1}), (Y_0, \cdots, Y_{m-1}), r$, bit-wise encryptions of integers $0 \leq x < y$, and a precision r
Ensure: $Z_0, \cdots, Z_{r-1}$, encryptions of the first r binary places of x/y (in reverse order: z_0 is the least significant bit)

$A^{\text{bits}} = X^{\text{bits}}$
for $i = 0$ *to* $r - 1$ **do**
  $B^{\text{bits}}, R_i = \texttt{SubLT}(A^{\text{bits}}, Y^{\text{bits}})$
  $A^{\text{bits}} = \texttt{Select}^{\text{bits}}(B^{\text{bits}}, A^{\text{bits}}, R_i)$
  $Z_i = \texttt{Not}(R_i)$
Return $Z_0, \cdots, Z_{r-1}$

Algorithm	Exp per trustee	Comm. cost	Transcript size
NaiveSMax	$\frac{K(K+1)}{2}\texttt{LT} + K\texttt{Dec}$	$2\texttt{LT} + \texttt{Dec}$	$\frac{K(K+1)}{2}\texttt{LT} + K\texttt{Dec}$
MergeSort	$K \log K(\texttt{LT} + \texttt{Dec})$	$K(\texttt{LT} + \texttt{Dec})$	$K \log K(\texttt{LT} + \texttt{Dec})$
S-Insert	$K \log s(\texttt{LT} + \texttt{Dec})$ $+\texttt{Mixnet}(2K) + s\texttt{Dec}$	$(K-s) \log s(\texttt{LT} + \texttt{Dec})$ $+s(\texttt{LT} + \texttt{Dec})$ $+R + B$	$K \log s(\texttt{LT} + \texttt{Dec})$ $+\texttt{Mixnet}(2K) + s\texttt{Dec}$
S-Merge (comp.)	$K(1 + \log s)(\texttt{LT} + \texttt{Dec})$ $+\texttt{Mixnet}(2K) + s\texttt{Dec}$	$(1 + \log(K/s))s(\texttt{LT} + \texttt{Dec})$ $+R + B$	$K(1 + \log s)(\texttt{LT} + \texttt{Dec})$ $+\texttt{Mixnet}(2K) + s\texttt{Dec}$
S-Merge (comm.)	$K(2s+1)\texttt{LT}$ $+\texttt{Mixnet}(2K) + (2K+s)\texttt{Dec}$	$\log(K/s)(\texttt{LT} + \texttt{Dec})$ $+R + B$	$K(2s+1)\texttt{LT}$ $+\texttt{Mixnet}(2K) + (2K+s)\texttt{Dec}$

Algorithm	Option	Exp per trustee	Comm. cost	Transcript size
S-Insert	EG+CLT	$80maK \log s$	$2 \log m(K-s) \log sR$	$72maK \log s$
S-Insert	P+GTH	$(27m + 146 \log m)aK \log s$	$(K-s) \log s \log m(2R + 13B)$	$(28m + 50 \log m)aK \log s$
S-Merge (comp.)	EG+CLT	$80amK \log s$	$2s \log(K/s) \log mR$	$72maK \log s$
S-Merge (comp.)	P+GTH	$(27m + 146 \log m)aK \log s$	$s \log(K/s) \log m(2R + 13B)$	$(28m + 50 \log m)aK \log s$
S-Merge (comm.)	EG+CLT	$160masK$	$2 \log(K/s) \log mR$	$144msaK$
S-Merge (comm.)	P+GTH	$(54m + 292 \log m)asK$	$\log(K/s) \log m(2R + 13B)$	$(56m + 100 \log m)asK$

Figure 10. Cost of various s-max algorithms. In the table at the top, we express the costs in terms of Dec, Mixnet and LT. The latter denotes the cost of any implementation of $\texttt{LT}^{\text{bits}}$ or GT (see Figure 9). In the table at the bottom, we propose a few instantiations, for various choices of LT. Everywhere, K is the number of operands.

we simply modify the scores s_i of each candidate to turn it into $s'_i = 2^l s_i + i$, where $l = \lceil \log(k+1) \rceil$ and assuming the candidates are labelled from 0 to $k-1$. This can be done with a very little extra cost, which comes from the fact that the number of bits increased in every procedure. Therefore, if m is the number of bits required to encode the scores (typically, m is logarithmic in the number of voters), the overall process is $\frac{m+\log k}{m}$ more expensive in terms of computations, while the impact on communication is about a factor $\frac{\log(m+\log k)}{\log m}$. In general, the impact is smaller than a factor 2 as the number of candidates is smaller than the number of voters. Note that the modification of the scores itself is basically free, and can be performed as follows.

- In natural encoding, $S'_i = (S_i)^{2^l} E_i$.
- In bit-encoding, $S'_i = i^{\text{bits}} || S_i$, where i^{bits} is a bit-encoding of i and $||$ stands for the concatenation.

Note that if the agreed ordering of the candidates has to be kept secret, the authorities can first run a reencryption mixnet to shuffle $E_0, \cdots, E_{k-1}$ (in either bit- or natural encoding) to obtain $(p_0, \cdots, p_{k-1})$. Afterwards, p_i can be used instead of E_i in natural encoding (resp. i^{bits} in bit-encoding).

Therefore, from now on we can assume that all the scores are distinct.

B.2. Various algorithm for the s largest values.

The naive approach.

Following [22], it is possible to return the indexes of the s best candidates (or list of candidates) using a quadratic number of comparisons. We call it the naive approach, but it could be very efficient in practice due to the low communication cost. We recall it for completeness in Algorithm 28 which is a slightly modified version of the one in [22].

The s-insertion approach.

A first compromise is to use a dichotomous insertion algorithm. In the clear, we would sort the s first values, then sequentially insert the $K-s$ next values in the right position, which we would find using a dichotomous search. Each insertion would therefore cost $\log(s)$ comparison, leading to a total cost of $K \log s$ comparisons (instead of $K \log K$ for sorting all the K values). The main obstacle before we can apply this strategy on the ciphertexts is efficiency. Indeed, efficiently sorting without revealing anything about the cleartexts is not that easy, and the same goes for inserting a ciphertext at some position. To obliterate those difficulties, a simple solution is to first use a reencryption mixnet to hide the initial order. Afterwards, the results of the comparisons as well as the insertion indexes can be revealed, as they will not reveal anything about the initial order, but only about the shuffled order.

Algorithm 28: `NaiveSMax`

Require: $X_0, \cdots, X_{n-1}$, encryptions of integers $x_0, \cdots, x_{n-1}$
Ensure: $i_0, \cdots, i_{s-1}$, indexes of the encryptions of the s largest integers

for $i = 0$ *to* $n-1$ *(in parallel)* **do**
  for $j = i+1$ *to* $n-1$ *(in parallel)* **do**
    $B_{i,j} = \texttt{LT}(X_i, X_j)$
  $B_{i,i} = \texttt{Enc}(0)$
  for $j = 0$ *to* $i-1$ **do**
    $B_{i,j} = \texttt{Not}(B_{j,i})$
for $i = 0$ *to* $n-1$ *(in parallel)* **do**
  $S_i = \prod_{j=0}^{n-1} B_{i,j}$
  $g_i = \texttt{Dec}(\texttt{LT}(S_i, s))$
Return $\{i \mid g_i = 1\}$

We first give our MPC version of the well-known merge-sort algorithm (see Algorithm 30), which we choose because of its potential for parallelization. This version leaks information about the ordering of the cleartexts and must not be used without a preliminary mixing. Afterwards, we give Algorithm 31 which sum up the whole process.

Algorithm 29: `Merge` (leaks partial information on the ordering; don't use without mixing input first)

Require: $(X_0, \cdots, X_{n-1}), m$, encryptions of integers $(x_1, \cdots, x_n)$ and $0 < m < n$ such that $X_0, \cdots, X_{m-1}$ and $X_m, \cdots, X_{n-1}$ are sorted in decreasing order of their respective plaintexts
Ensure: $Y_0, \cdots, Y_{n-1}$, a permutation of the inputs, sorted in decreasing order of the plaintexts.

$k_0 = 0$; $k_1 = m$; $i = 0$
while $k_0 < m \wedge k_1 < n$ **do**
  $r = \texttt{Dec}(\texttt{LT}(X_{k_0}, X_{k_1}))$
  $Y_i = X_{k_r}$
  $k_r += 1$; $i += 1$
while $k_0 < m$ **do**
  $Y_i = X_{k_0}$
  $k_0 += 1$; $i += 1$
while $k_1 < n$ **do**
  $Y_i = X_{k_1}$
  $k_1 += 1$; $i += 1$
Return $Y_0, \cdots, Y_{n-1}$

Algorithm 30: `MergeSort` (leaks partial information on the ordering; don't use without mixing input first)

Require: $X_0, \cdots, X_{n-1}$, encryptions of integers (* bit- or natural encoding *)
Ensure: $Y_0, \cdots, Y_{n-1}$, a permutation of the inputs in decreasing order of the plaintexts

if $n = 1$ **then**
  Return X_0
$m = \lfloor n/2 \rfloor$
(* The two following recursive calls are made in parallel *)
$Y_0, \cdots, Y_{m-1} = \texttt{MergeSort}(X_0, \cdots, X_{m-1})$
$Y_m, \cdots, Y_{n-1} = \texttt{MergeSort}(X_m, \cdots, X_{n-1})$
Return $\texttt{Merge}((Y_0, \cdots, Y_{n-1}), m)$

The s-merge approach.

The s-insertion approach allows a way better complexity than the naive approach ($O(K \log s)$ compared to $O(K^2)$). However, it comes with an extra communication cost ($O(K \log s)$ instead of $O(1)$). An alternative is to use a merge-sort-like approach which allows more parallelization. The idea is to simply divide the K inputs into (K/s) lists of size s, then to

Algorithm 31: `S-Insert`

Require: $(X_0, \cdots, X_{n-1}), s$, encryptions of integers $x_0, \cdots, x_{n-1}$ and $0 < s \leq n$
Ensure: $i_0, \cdots, i_{s-1}$, indexes of the encryptions of the s largest integers among $x_0, \cdots, x_{n-1}$

1 $(Y_0, \cdots, Y_{n-1}) = ((X_0, \texttt{Enc}(0)), \cdots, (X_{n-1}, \texttt{Enc}(n-1)))$
2 $(Y_0, \cdots, Y_{n-1}) = \texttt{Mixnet}(Y_0, \cdots, Y_{n-1})$
3 $D_0, \cdots, D_{s-1} = \texttt{MergeSort}(Y_0, \cdots, Y_{s-1})$
4 **for** $i = s$ *to* $n-1$ **do**
5 $k_0 = 0$; $k_1 = s-1$
6 **while** $k_0 \leq k_1$ **do**
7 $m = \lfloor (k_0 + k_1)/2 \rfloor$
8 $r = \texttt{Dec}(\texttt{LT}(D_m[0], Y_i[0]))$
9 $k_r = m - 2r + 1$
10 **for** $j = s-1$ *downto* $k_0 + 1$ **do**
11 $D_j = D_{j-1}$
12 $D_{k_0} = Y_i$
13 Return $\texttt{Dec}(D_0[1]), \cdots, \texttt{Dec}(D_{s-1}[1])$

use a tree-based algorithm to merge them into a single list of s elements. Even if the idea is simple, there is a lot of trade-offs depending on the subfunction we use to merge two sorted lists of size s. As we cannot explore all the possibilities, we will only list two opposite approaches, one which focuses on communications and the other on computations. The computation-focussed one is a variant of merge-sorting Algorithm 29, which allows to merge two lists (see Algorithm 32). The communication-focussed one is a variant of the quadratic method Algorithm 28, which allows to give the s largest elements from any given list. In the first case, we need to pre-sort the list of size s so that the merging process can be efficient, while in the second case, this is not required. As for the s-insertion algorithms, to prevent the sorting and merging processes to leak any information, we use a mixnet to shuffle the data. The computation-focussed approach is given in Algorithm 33. For the communication-focussed approach, this algorithm can be modified by calling `NaiveSMax` instead of `Merge-`s and removing the pre-sorting at line 3.

Algorithm 32: `Merge-`s (leaks partial information on the ordering; don't use without mixing input first)

Require: $(X_0, \cdots, X_{s-1}), (Y_0, \cdots, Y_{s-1})$, encryptions of integers $\boldsymbol{x}$ and $\boldsymbol{y}$ such that $\boldsymbol{X}$ and $\boldsymbol{Y}$ are sorted in decreasing order of their respective plaintexts
Ensure: $Z_0, \cdots, Z_{s-1}$, such that the respective plaintexts are the s largest in $\boldsymbol{x} \bigcup \boldsymbol{y}$, and sorted in decreasing order.

1 $k_0 = 0$; $k_1 = 0$; $i = 0$
2 **while** $k_0 < s \wedge k_1 < s \wedge i < s$ **do**
3 $r = \texttt{Dec}(\texttt{LT}(X_{k_0}, Y_{k_1}))$
4 **if** r **then**
5 $Z_i = Y_{k_1}$
6 $k_1 += 1$; $i += 1$
7 **else**
8 $Z_i = X_{k_0}$
9 $k_0 += 1$; $i += 1$
10 Return $Z_0, \cdots, Z_{s-1}$

Discussion.

While the naive approach is extremely effective in terms of communication cost, its quadratic number of comparisons can be too expensive if the number of candidates is large. The s-merge approach provides two interesting compromises: a computation-focussed one and a communication-focussed one. The computation-focussed has the same computation complexity as s-insertion, which we designed to optimize computation with no care about communication. As such, it is quite efficient. However, it is way more efficient in terms of communications, since K is larger than s. The communication-focused compromise is even more interesting since it allows to win an s factor in communications, while only losing a factor $\frac{2s}{\log s}$ in computations. For this reason, we choose to use this approach as a basis in Figure 2. Note, however, that the algorithms presented in this section require already encrypted integers. This means that the procedure `Aggreg` should be called in ElGamal in order to obtain the bit-encoding encrypted integers from the voters' ballots.

Algorithm 33: `S-Merge`

Require: $(X_0, \cdots, X_{n-1}), s$, encryptions of integers $x_0, \cdots, x_{n-1}$ and $0 < s \leq n$
Ensure: $i_0, \cdots, i_{s-1}$, indexes of the encryptions of the s largest integers among $x_0, \cdots, x_{n-1}$

$(Y_0, \cdots, Y_{n-1}) = ((X_0, \texttt{Enc}(0)), \cdots, (X_{n-1}, \texttt{Enc}(n-1)))$
$(Y_0, \cdots, Y_{n-1}) = \texttt{Mixnet}(Y_0, \cdots, Y_{n-1})$
$(\boldsymbol{D}_{0,0}, \cdots, \boldsymbol{D}_{0,n/s-1}) = \texttt{MergeSort}(Y_0, \cdots, Y_{s-1}), \cdots, \texttt{MergeSort}(Y_{n-s}, \cdots, Y_{n-1})$
for $i = 1$ *to* $\lceil \log(n/s) \rceil$ **do**
  for $j = 0$ *to* $(n/s)/2^i - 1$ *(in parallel)* **do**
    $\boldsymbol{D}_{i,j} = \texttt{Merge-}s\ (\boldsymbol{D}_{i-1,2j}, \boldsymbol{D}_{i-1,2j+1})$
Return $\texttt{Dec}(D_{\lceil \log(n/s) \rceil, 0}[1]), \cdots, \texttt{Dec}(D_{\lceil \log(n/s) \rceil, s-1}[1])$

B.3. Summing-up the costs

We finally explain how to obtain the costs given in Figure 2 in Section 3. The first line is taken from [22] and is for the basic single choice setting. The second and third lines are our versions of the same tally function, except that we always return exactly s winners. This explain the change from m to m_1 for the bit-length of the integers. The second line, in the Paillier setting, relies on s-merge, with the `GTH` option, in natural encoding. The third line, in the ElGamal setting, hence bit-encoding, is the sum of the costs of the aggregation of the ballots with `Aggreg`, and of the s-merge algorithm with the `CLT` option.

The other two lines are for the D'Hondt tally function. Following the technique we explained in Section 3, we can replace divisions by multiplications by the weights. This increases again the bit-length m_2 of the integers we consider. The fourth line of Figure 2 is for the Paillier setting, and here, we followed the naive quadratic approach. The last line is in the ElGamal setting and is obtained with the s-merge method with the `CLT` option. Here, however we need to multiply by the LCM of all the weights, thus increasing again the bit-length m_3. The multiplication by the weights can be optimized using a specialized $\texttt{MulKnown}^{\text{bits}}$ algorithm where one of the operand is not encrypted (see Algorithm 34). Adding the cost of aggregation yields the claimed complexity.

Algorithm 34: $\texttt{MulKnown}^{\text{bits}}$

Require: $(X_0, \cdots, X_{m_x-1}), (y_0, \cdots, y_{m_y-1})$, bitwise encryptions of x, and a (public) bitwise representation of y
Ensure: $Z_0, \cdots, Z_{m_x+m_y-1}$, bitwise encryption of xy

for $i \in [0, m_x - 1], j \in [0, m_y - 1]$ *(in parallel)* **do**
  $A_{i,j} = X_i^{y_j}$
$Z_0 = A_{0,0}$
$(T_0, \cdots, T_{m_y-2}) = (A_{0,1}, \cdots, A_{0,m_y-1})$
for $i = 1$ *to* $m_x - 1$ **do**
  $(T_0, \cdots, T_{m_y-1}) = \texttt{Add}^{\text{bits}}((T_0, \cdots, T_{m_y-2}), (A_{i,0}, \cdots, A_{i,m_y-2}))$
  $T_{m_y} = \texttt{CSZ}(T_{m_y-1}, A_{i,m_y-1})$
  $T_{m_y-1} = T_{m_y-1} A_{i,m_y-1} / T_{m_y}^2$
  $Z_i = T_0$ **for** $j = 0$ *to* $m_y - 1$ **do**
    $T_j = T_{j+1}$
for $i = m_x$ *to* $m_x + m_y - 1$ **do**
  $Z_i = T_{i-m_x}$
Return $Z_0, \cdots, Z_{m_x+m_y-1}$

Appendix C. Majority Judgement

In this section, we will give the details of what is sketched in Section 4: we start with a precise definition of how Majority Judgement (MJ) is defined, then we recall our algorithm for computing the winners and give a complete proof of its correctness. Finally we explain how to adapt it for MPC for both Paillier and ElGamal settings.

C.1. Definition

In a MJ protocol, there are k candidates and a set of d grades, which is totally ordered. For instance, the set could be {Excellent,Good,Medium,Bad,Reject}. For the computations, we represent grades with integers and the tradition in MJ is to use a reversed ordering (*i.e.* 1 is a better / higher grade than 2). Each voter has to grade each candidate with a single grade. Hence, if n is the number of voters (who did not abstain or vote blank), each candidate has a list of n grades. For simplicity, we assume that the lists are sorted in decreasing order (highest grades first). Thus, we consider that each candidate has a sorted n-tuple. Note that two n-tuples are equal if and only if the candidates received exactly the same number of each grade. Given a sorted n-tuple $u_1, \cdots, u_n$ the *median* of u is simply $\mathsf{med}(u) = u_{\lceil n/2 \rceil}$. We denote $\hat{u}$ the $(n-1)$-tuple $u_1, \cdots, u_{\lceil n/2 \rceil - 1}, u_{\lceil n/2 \rceil + 1}, \cdots, u_n$; that is, the tuple u in which the median element has been removed. Finally, we define the $\leq_{maj}$ relation as follows, where $<$ stands for the grade-wise comparison (which is the opposite of the natural comparison of integers).

Definition 1 (The relation $\leq_{maj}$)**.** *Let u and v be grade n-tuples sorted in decreasing order. If $n = 1$, $u <_{maj} v$ if $u_1 < v_1$. Else, $u <_{maj} v$ if one of the following conditions holds:*

- $\mathsf{med}(u) < \mathsf{med}(v)$,
- $\mathsf{med}(u) = \mathsf{med}(v)$ *and* $\hat{u} <_{maj} \hat{v}$.

Finally, $u \leq_{maj} v$ if $u = v$ or $u <_{maj} v$.

It is straightforward to show that $\leq_{maj}$ is a total order. The majority judgement declares as winner any candidate whose grades form a maximal n-tuple (once sorted) according to $\leq_{maj}$.

C.2. Proof of our algorithm for MJ

For convenience, we recall in Algorithm 35 our simplified algorithm already presented in Algorithm 1 of Section 4. To prove its correctness, we first give Definition 2. From this definition and Definition 1, it is straightforward to show that $\leq_{maj}$ is the lexicographic order for the median sequences. Hence, it is important to describe the behavior of the median sequence, which is done in Lemma 1.

Definition 2 (The median sequence)**.** *The median sequence of a sorted n-tuple u, denoted $m(u)$ is the sequence formed by* $\mathsf{med}(u)$ *followed by $m(\hat{u})$.*

Lemma 1. *Let u a sorted n-tuple. The k^{th} element of the median sequence of u is the element of index $m + (-1)^{k+n} \lfloor k/2 \rfloor$, where $m = \lceil \frac{n}{2} \rceil$.*

Proof. We distinguish the cases where n is even or odd and give a recurrence in k.

Case 1: n is even. The first element of the median sequence is u_m by definition. Let $k \geq 1$. Suppose that for $i \in [1, k]$, the i^{th} element of the median sequence is $u_{m+(-1)^i \lfloor i/2 \rfloor}$. By definition, the $(k+1)^{th}$ element of the median sequence is the element of index $\lceil \frac{n-k}{2} \rceil$ of some $(n-k)$-tuple, obtained by removing the first k elements of the median sequence of u.

If k is even, by recurrence hypothesis, the removed elements have indexes $m, m+1, m-1, \cdots, m-(k/2-1), m+k/2$ thus the remaining elements are

$$(u_1, \cdots, u_{m-k/2}, u_{m+k/2+1}, \cdots, u_n).$$

As n and k are even, $\lceil \frac{n-k}{2} \rceil = m - k/2$. Therefore, the $(k+1)^{th}$ element of the median sequence is $u_{m-k/2}$, and since k is even, $m - k/2 = m + (-1)^{k+1} \lfloor \frac{k+1}{2} \rfloor$.

If k is odd, by recurrence hypothesis, the removed elements have indexes $m, m+1, m-1, \cdots, m+(k-1)/2, m-(k-1)/2$ so the remaining elements are

$$(u_1, \cdots, u_{m-(k+1)/2}, u_{m+(k+1)/2}, \cdots, u_n).$$

Since n is even while k odd, $\lceil \frac{n-k}{2} \rceil = m-(k-1)/2$, so the $(k+1)^{th}$ element of the median sequence is the one following $u_{m-(k+1)/2}$ in the above list, namely $u_{m+(k+1)/2}$, with $m + (k+1)/2 = m + (-1)^{k+1} \lfloor \frac{k+1}{2} \rfloor$.

Case 2: n is odd. The first element of the median sequence is u_m by definition. Let $k \geq 1$. Suppose that for $i \in [1, k]$, the i^{th} element of the median sequence is $u_{m-(-1)^i \lfloor i/2 \rfloor}$. By definition, the $(k+1)^{th}$ element of the median sequence is the element of index $\lceil \frac{n-k}{2} \rceil$ of some $(n-k)$-tuple, obtained by removing the first k elements of the median sequence of u.

If k is even, by recurrence hypothesis, the removed elements have indexes $m, m-1, m+1, \cdots, m+(k/2-1), m-k/2$ so the remaining elements are

$$(u_1, \cdots, u_{m-k/2-1}, u_{m+k/2}, \cdots, u_n).$$

As n is odd and k even, $\lceil \frac{n-k}{2} \rceil = m - k/2$. Therefore the $(k+1)^{th}$ element of the median sequence is the one following $u_{m-k/2-1}$ in the above list, namely $u_{m+k/2}$ with $m + k/2 = m - (-1)^{k+1} \lfloor \frac{k+1}{2} \rfloor$.

Algorithm 35: Majority Judgement

Require: a the aggregated matrix, d the number of grades, n the number of voters
Ensure: C the set of MJ winner(s)

1 Let $m = \max\{m_i \mid m_i \text{ is the median of candidate } i\}$
2 Let C be the set of candidates with m as median grade.
3 Let $I^- = 1$ and $I^+ = 1$ be counters.
4 Let $s = 1$.
5 **for** $i \in C$ **do**
6 $\quad p_i = \sum_{j=1}^{m-1} a_{i,j}$, $q_i = \sum_{j=m+1}^{d} a_{i,j}$,
7 $\quad m_i^- = \lfloor \frac{n}{2} \rfloor - p_i$, $m_i^+ = \lfloor \frac{n}{2} \rfloor - q_i$
8 **while** $(\mid C \mid > 1) \wedge (s \neq 0)$ **do**
9 $\quad$ **for** $i \in C$ **do**
10 $\quad\quad$ **if** $m_i^- \leq m_i^+$ **then**
11 $\quad\quad\quad s_i = p_i$
12 $\quad\quad$ **else**
13 $\quad\quad\quad s_i = -q_i$
14 $\quad s = \max\{s_i \mid i \in C\}$
15 $\quad C = \{i \in C \mid s_i = s\}$.
16 $\quad$ **if** $s \geq 0$ **then**
17 $\quad\quad$ **for** $i \in C$ **do**
18 $\quad\quad\quad m_i^+ = m_i^+ - m_i^-$, $m_i^- = a_{i,m-I^-}$
19 $\quad\quad\quad p_i = p_i - a_{i,m-I^-}$
20 $\quad\quad I^- = I^- + 1$
21 $\quad$ **else**
22 $\quad\quad$ **for** $i \in C$ **do**
23 $\quad\quad\quad m_i^- = m_i^- - m_i^+$, $m_i^+ = a_{i,m+I^+}$
24 $\quad\quad\quad q_i = q_i - a_{i,m+I^+}$
25 $\quad\quad I^+ = I^+ + 1$
26 Return C.

If k is odd, by recurrence hypothesis, the removed elements have indexes $m, m-1, m+1, \cdots, m-(k-1)/2, m+(k-1)/2$ so the remaining elements are

$$(u_1, \cdots, u_{m-(k+1)/2}, u_{m+(k+1)/2}, \cdots, u_n).$$

As n and k are odds, $\lceil \frac{n-k}{2} \rceil = m - (k+1)/2$. Hence the $(k+1)^{th}$ element of the median sequence is $u_{m-(k+1)/2}$, with $m - (k+1)/2 = m - (-1)^{k+1} \lfloor \frac{k+1}{2} \rfloor$.

□

In order to prove the correctness of Algorithm 35, we exhibit the following loop invariants, where a sum indexed with the empty set is 0 and $g_{i,1}, \cdots, g_{i,n}$ denote the list of grades received by candidate i, sorted in decreasing order. Note that m is used to denote the best median (line 1), and not $\lceil \frac{n}{2} \rceil$ as in the previous lemma.

Lemma 2. *In Algorithm 35, the following loop invariants hold at the beginning of the loop (line 8) and at the end of the loop (line 25).*

1) *For all $i \in C$, $p_i + m_i^- = m_i^+ + q_i$, and this value is the same for all i.*
2) *For all $i \in C$, $m_i^+ \geq 0$ and $m_i^- \geq 0$.*
3) *For all $i \in C$, $p_i = \sum_{j=1}^{m-I^-} a_{i,j}$. Hence $p_i \geq 0$.*
4) *For all $i \in C$, $q_i = \sum_{j=m+I^+}^{d} a_{i,j}$. Hence, $q_i \geq 0$.*
5) *Let $L + p_i + m_i^- + m_i^+ + q_i$. The $n - L$ first elements of the median sequence are identical for all $i \in C$.*

6) For all $i \in C$, for all $j \in [1, m_i^-]$, $g_{i,p_i+j} = m - I^- + 1$ and, for all $j \in [1, m_i^+]$, $g_{i,n-q_i-j+1} = m + I^+ - 1$.
7) C contains all the MJ winners.

Proof. **Initialization.** First of all, we verify that the loop invariants are true after line 7.

Invariants 1 to 4:

We have $p_i + m_i^- = \lfloor n/2 \rfloor = m_i^+ + q_i$.

Moreover p_i is the number of grades strictly greater than the median, so by definition of the median, $p_i \leq \lfloor n/2 \rfloor$ hence $m_i^- = \lfloor n/2 \rfloor - p_i \geq 0$. Similarly, q_i is the number of grades strictly worse than the median, so by definition of the median, $q_i \leq \lfloor n/2 \rfloor$ hence $m_i^+ = \lfloor n/2 \rfloor - q_i \geq 0$.

Finally, Equalities 3 and 4 are true with $I^- = I^+ = 1$.

Invariant 5:

Initially, $L = p_i + m_i^- + m_i^+ + q_i = 2\lfloor n/2 \rfloor$ so if n is even, $n - L = 0$. Else, $n - L = 1$. As the first element of the median sequence is the median, the $n - L$ first elements are the same for all candidates in C after line 7.

Invariant 6:

After line 7, p_i is the number of grades strictly greater than the median for candidate i so, for all $j \geq 1$, $g_{i,p_i+j} \geq m$. Moreover m_i^- is lower than the number of grades equal to the median received by i. So for all $j \leq m_i^-$, $g_{i,p_i+j} \leq m$. Hence, for all $j \in [1, m_i^-]$, $g_{i,p_i+j} = m$. Similarly, for all $j \in [1, m_i^+]$, $g_{i,n-q_i-j+1} = m$.

Invariant 7:

After line 7, C contains the candidates who have the best median, thus contains the winners.

Heredity. Assume that the loop invariants are verified at the beginning of the loop, we show that they are preserved at the end of the loop.

We first show the following result, which is a consequence of loop invariants 1 to 4.

Sub-lemma. For all candidates i, $s_i \geq 0$ if and only if $m_i^- \leq m_i^+$.

Let i be a candidate. Suppose $s_i \geq 0$ and $m_i^- > m_i^+$. Then $0 \leq s_i = -q_i \leq 0$ so $q_i = 0$ and as $p_i + m_i^- = m_i^+ + q_i$, we have $p_i + m_i^- = m_i^+$, which contradicts $p_i \geq 0$. Conversely, if $m_i^- \leq m_i^+$, $s_i = p_i \geq 0$.

To show that the loop invariants are preserved, we denote C_1 the set C at the beginning of the loop and C_2 the set C at the end of the loop. Let $i \in C_2$. Let $i \in C_2$, then $i \in C_1$ so the loop invariants hold at the beginning of the loop, for all $i \in C_2$. We denote p_1 the value of p_i at the beginning of the loop and p_2 at the end, and the same for all other variable m_i^-, m_i^+, q_i, I^-, I^+ and L.

Invariants 1 to 4: Let $s = \max\{s_i \mid i \in C\}$. $C_2 = \{i \mid s_i = s\}$.

If $s \geq 0$, then $s_i = s \geq 0$ so $m_1^- \leq m_1^+$ by the sub-lemma. Hence $m_2^+ = m_1^+ - m_1^- \geq 0$, $m_2^- = a_{i,m-I_1^-} \geq 0$.

In addition, $p_2 = p_1 - a_{i,m-I_1^-}$ and $q_2 = q_1$. Therefore $p_2 + m_2^- = p_1 = s_i = s$, which is the same for all i. Moreover $m_2^+ + q_2 = m_1^+ - m_1^- + q_1 = p_1 + m_1^- - m_1^- = p_1 = S$.

Finally, line 20 together with line 21 and loop invariant 3 give $p_2 = \sum_{j=1}^{m-I_2^-} a_{i,j}$, which shows that invariant 3 is preserved. (Invariant 4 is also preserved because $q_2 = q_1$ and $I_2^+ = I_1^+$.)

If $s < 0$, then $s_i = s < 0$ so $m_1^- > m_1^+$ by the sub-lemma. Hence $m_2^- = m_1^- - m_1^+ \geq 0$, $m_2^+ = a_{i,m+I_1^+} \geq 0$, $q_2 = q_1 - a_{i,m+I_1^+}$ et $p_2 = p_1$. So $m_2^+ + q_2 = q_1 = -S_i = -S$, which is the same for all i. In addition $p_2 + m_2^- = p_1 + m_1^- - m_1^+ = m_1^+ + q_1 - m_1^+ = q_1 = -S$. Finally line 26 together with line 27 and loop invariant 4 give $q_2 = \sum_{j=m+I_2^+}^{c} a_{i,j}$, so that invariant 4 is preserved. (Invariant 3 is also preserved because $p_2 = p_1$ and $I_2^- = I_1^-$.)

Invariant 5:

If $s \geq 0$, $m_1^- \leq m_1^+$. Consequently, $p_1 = s_i = s$ and since $p_1 + m_1^-$ is the same for all i, we deduce that m_1^- is the same for all i. In addition we have $p_2 + m_2^- = p_1$ (lines 19 and 20), $m_2^+ = m_1^+ - m_1^-$ (line 18) and $q_2 = q_1$, so

$$\begin{aligned} L_2 &= p_2 + m_2^- + m_2^+ + q_2 \\ &= p_1 + m_1^+ - m_1^- + q_1 \\ &= p_1 + m_1^- + m_1^+ + q_1 - 2m_1^- = L_1 - 2m_1^-, \end{aligned}$$

and since the $n - L_1$ first elements of the median sequence are the same for all candidates in C_1, we only have to show that the $2m_1^-$ next elements are the same for all candidates in C_2. For this purpose, we remark that loop invariant 1 implies that L_1 is even and we suppose $m_1^- > 0$. (If $m_1^- = 0$, our job is already done.)

By Lemma 1, the elements of indexes $n - L_1 + 1, \cdots, n - L_1 + 2m_1^-$ of the median sequence are the elements

$$g_{i,\lceil n/2 \rceil + (-1)^{2n-L_1+1}\lfloor (n-L_1+1)/2 \rfloor}, g_{i,\lceil n/2 \rceil + (-1)^{2n-L_1+2}\lfloor (n-L_1+2)/2 \rfloor}, \cdots, g_{i,\lceil n/2 \rceil + (-1)^{2n-L_1+2m_1^-}\lfloor (n-L_1+2m_1^-)/2 \rfloor} ;$$

which are also

$$g_{i,\lceil n/2\rceil-\lfloor(n+1)/2\rfloor+L_1/2}, g_{i,\lceil n/2\rceil+\lfloor n/2\rfloor-L_1/2+1}, \cdots, g_{i,\lceil n/2\rceil-\lfloor(n-1)/2\rfloor+L_1/2-m_1^-}, g_{i,\lceil n/2\rceil+\lfloor n/2\rfloor-L_1/2+m_1^-}.$$

But $L_1 = p_1 + m_1^- + m_1^+ + q_1$ so, by invariant 1, $L_1/2 = p_1 + m_1^- = m_1^+ + q_1$. Since $\lceil n/2\rceil = \lfloor(n+1)/2\rfloor$ and $\lceil n/2\rceil + \lfloor n/2\rfloor = n$ for all n, we can rewrite them as

$$g_{i,p_1+m_1^-}, g_{i,n-q_1-m_1^++1}, \cdots, g_{i,p_1+1}, g_{i,n-q_1-m_1^++m_1^-}.$$

In what follow, we prove that for all $j \in [1, m_i^-]$, $g_{i,n-q_1-m_1^++j} = m + I_1^+ - 1$. Indeed, $n - q_1 - m_1^+ + j = n - q_1 - (m_1^+ - j + 1) + 1$ and since $m_1^+ \geq m_1^- > 0$, $m_1^+ - j + 1 \in [1, m_1^+]$ for all $j \in [1, m_1^-]$, which allows to prove our claim by invariant 6.

In addition, $g_{i,p_1+j} = m - I_1^- + 1$ for all $j \in [1, m_1^-]$ by invariant 6, so the elements listed above are equal to $m - I_1^- + 1, m + I_1^+ - 1, \cdots, m - I_1^- + 1, m + I_1^+ - 1$ and therefore are the same for all $i \in C_2$, which shows that invariant 5 is preserved.

If $s < 0$, $m_1^- > m_1^+$. Consequently, $q_1 = -s_i = -s$ and since $m_1^+ + q_1$ is the same for all i, so is m_1^+. Moreover $m_2^+ + q_2 = q_1$ (lines 25 and 26), $m_2^- = m_1^- - m_1^+$ (line 24) and $p_2 = p_1$ so

$$\begin{aligned} L_2 &= p_2 + m_2^- + m_2^+ + q_2 \\ &= p_1 + m_1^- - m_1^+ + q_1 \\ &= p_1 + m_1^- + m_1^+ + q_1 - 2m_1^+ = L_1 - 2m_1^+, \end{aligned}$$

and since the $n - L_1$ first elements of the median sequence are the same for all candidates in C_1, we only have to show that the $2m_1^+$ next elements are the same for all candidates in C_2. For this purpose, we remark that invariant 1 implies that L_1 is even and we suppose that $m_1^+ > 0$. (If $m_1^+ = 0$, our job is done.)

By Lemma 1, the elements of indexes $n - L_1 + 1, \cdots, n - L_1 + 2m_1^+$ of the median sequence are

$$g_{i,\lceil n/2\rceil+(-1)^{2n-L_1+1}\lfloor(n-L_1+1)/2\rfloor}, g_{i,\lceil n/2\rceil+(-1)^{2n-L_1+2}\lfloor(n-L_1+2)/2\rfloor}, \cdots, g_{i,\lceil n/2\rceil+(-1)^{2n-L_1+2m_1^+}\lfloor(n-L_1+2m_1^+)/2\rfloor};$$

which are also

$$g_{i,\lceil n/2\rceil-\lfloor(n+1)/2\rfloor+L_1/2}, g_{i,\lceil n/2\rceil+\lfloor n/2\rfloor-L_1/2+1}, \cdots, g_{i,\lceil n/2\rceil-\lfloor(n-1)/2\rfloor+L_1/2-m_1^+}, g_{i,\lceil n/2\rceil+\lfloor n/2\rfloor-L_1/2+m_1^+}.$$

But $L_1 = p_1 + m_1^- + m_1^+ + q_1$ so, by invariant 1, $L_1/2 = p_1 + m_1^- = m_1^+ + q_1$. Since $\lceil n/2\rceil = \lfloor(n+1)/2\rfloor$ et $\lceil n/2\rceil + \lfloor n/2\rfloor = n$ for all n, we can rewrite them as

$$g_{i,p_1+m_1^-}, g_{n-q_1-m_1^++1}, \cdots, g_{i,p_1+m_1^--m_1^++1}, g_{i,n-q_1}.$$

We now show that for all $j \in [1, m_i^+]$, $g_{i,p_1+m_1^--j+1} = m - I_1^- + 1$. Indeed, $p_1 + m_1^- - j + 1 = p_1 + (m_1^- - j + 1)$ and since $m_1^- > m_1^+ > 0$, $(m_1^- - j + 1) \in [1, m_1^-]$ for all $j \in [1, m_1^+]$, which allows to prove our claim by invariant 6.

In addition, $g_{i,n-q_1-j+1} = m + I_1^+ - 1$ for all $j \in [1, m_1^+]$ by invariant 6, so the elements listed above are equal to $m - I_1^- + 1, m + I_1^+ - 1, \cdots, m - I_1^- + 1, m + I_1^+ - 1$ and therefore are the same for all $i \in C_2$, which shows that invariant 5 is preserved.

Invariant 6:

If $s \geq 0$, $m_1^- \leq m_1^+$ so $p_2 = p_1 - a_{i,m-I_1^-}$ and $m_2^- = a_{i,m-I_1^-}$. But $p_1 = \sum_{j=1}^{m-I_1^-} a_{i,j}$, which is exactly the number of grades strictly greater than $m - I_1^- + 1$ received by i so by definition of $a_{i,m-I_1^-}$, p_2 is the number of grades strictly greater than $m - I_1^-$. Therefore g_{i,p_2+1} is lower than $m - I_1^-$ and as there are $a_{i,m-I_1^-} = m_2^-$ grades equal to $m - I_1^-$, we deduce that $g_{i,p_2+j} = m - I_1^- = m - (I^- + 1) + 1 = m - I_2^- + 1$ for all $j \in [1, m_2^-]$. In addition, for all $j \in [1, m_1^+]$, $g_{i,n-q_1-j+1} = m + I_1^+ - 1$ so, *a fortiori*, for all $j \in [1, m_1^+ - m_1^-]$, $g_{i,n-q_1-j+1} = m + I_2^+ - 1$.

If $s < 0$, $m_1^- > m_1^+$ so $q_2 = q_1 - a_{i,m+I_1^+}$ and $m_2^+ = a_{i,m+I_1^+}$. But $q_1 = \sum_{j=m+I_1^+}^{c} a_{i,j}$, which is exactly the number of grades strictly worse than $m + I_1^+ - 1$ so by definition of $a_{i,m+I_1^+}$, q_2 is the number of grades strictly worse than $m + I_1^+$. Therefore $g_{i,n-q_2}$ is greater than $m + I_1^+$ and as there are $a_{i,m+I_1^+} = m_2^+$ grades equal to $m + I_1^+$, we deduce that $g_{i,n-q_2-j+1} = m + I_1^+ = m + (I^+ + 1) - 1 = m + I_2^+ - 1$ for all $j \in [1, m_2^+]$. In addition, for all $j \in [1, m_1^-]$, $g_{i,p_1+j} = m - I_1^- + 1$ so, *a fortiori*, for all $j \in [1, m_1^+ - m_1^-]$, $g_{i,p_1+j} = m - I_2^- + 1$.

Invariant 7:

Let $b \in C_2$, (namely $b \in C_1$ such that $s_b = s$). We show that for all $a \in C_1 \backslash C_2$, (namely for all $a \in C_1$ such that $s_a < s$), $a <_{maj} b$.

Positive case. Suppose that $s \geq 0$. Let $a \in C_1$ such that $s_a < s$.

Positive-negative case. We first assume that $s_a < 0$. Therefore $s_a < 0 \leq s = s_b$. By the sub-lemma, we have $m_a^- > m_a^+$ and $m_b^- \leq m_b^+$.

Suppose that $m_a^+ < m_b^-$. With the same reasoning as in the proof of invariant 6, we show that the elements of indexes 1 to $n - L + 2m_a^+$ of the median sequence of a and b are the same. Since $m_a^- > m_a^+$, by Lemma 1 and loop invariant 1 and 6, the $n - L + 2m_a^+ + 1$ th elements of the median sequence of a and b are respectively

$$g_{a,p_a+m_a^- -m_a^+} = m - I^- + 1 \text{ and}$$
$$g_{b,p_a+m_a^- -m_a^+} = g_{b,p_b+m_b^- -m_a^+} = m - I^- + 1.$$

However, the $n - L + 2m_a^+ + 2$ th element of the median sequence of a is

$$g_{a,n-q_a+1} < g_{a,n-q_a} = m + I^+ - 1,$$

while b's is

$$g_{b,n-q_a-m_a^++m_a^++1} = g_{b,n-q_b-(m_b^+-m_a^+)+1} = m + I^+ - 1.$$

Therefore $b >_{maj} a$.

Now suppose that $m_a^+ \geq m_b^-$. As above, the $n - L + 2m_b^-$ first elements of the median sequence of a and b are the same. The elements of index $n - L + 2m_b^- + 1$ are respectively

$$g_{a,p_a+m_a^- -m_b^-} = g_{a,p_a+(m_a^- -m_a^+)+(m_a^+ -m_b^-)} = m - I^- + 1^- \text{ and}$$
$$g_{b,p_a+m_a^- -m_b^-} = g_{b,p_b} > m - I^- + 1.$$

Therefore $b >_{maj} a$.

Positive-positive case. Now suppose that $0 \leq s_a$. By the sub-lemma, $m_b^- \leq m_b^+$, $m_a^- \leq m_a^+$. Consequently $s_a = p_a$ and $s_b = p_b$ and since $s_a < s_b$, by invariant 1, we have $m_a^- > m_b^-$. Then again, we deduce that the $n - L + 2m_b^-$ first elements of the median sequence are the same and that b wins over a thanks to the next element.

Negative case. Finally, suppose that $s < 0$. Then $s_a < s_b = s < 0$ so, by the sub-lemma, $m_a^- > m_a^+$ and $m_b^- > m_b^+$. Consequently $s_a = -q_a$ and $s_b = -q_b$ and since $s_a < s_b$, by invariant 1, we have $m_b^+ > m_a^+$. Then again, we deduce that the $n - L + 2m_a^+$ first elements of the median sequence are the same. In addition $m_a^- > m_a^+$, so by Lemma 1 and invariants 1 and 6, the $n - L + 2m_a^+ + 1$ th elements of the median sequence of a and b are

$$g_{a,p_a+m_a^- -m_a^+} = m - I^- + 1 \text{ and}$$
$$g_{b,p_a+m_a^- -m_a^+} = g_{b,p_b+m_b^- -m_a^+} = m - I^- + 1.$$

However, the $n - L + 2m_a^+ + 2$ th element for a is

$$g_{a,n-q_a+1} > g_{a,n-q_a} = m + I^+ - 1,$$

while b's is

$$g_{b,n-q_a-m_a^++m_a^++1} = g_{b,n-q_b-(m_b^+-m_a^+)+1} = m + I^+ - 1.$$

Therefore $b >_{maj} a$. □

Once the loop invariants are established, it is straightforward to show the correctness of our algorithm (Theorem 1).

Theorem 1. *Algorithm 35 returns the set of maxima according to $\leq_{maj}$ in $O(kd)$ comparisons between grades.*

Proof. **Complexity.** By Lemma 2, $p_i = \sum_{j=1}^{m-I^-} a_{i,j}$ and $q_i = \sum_{j=m+I^+}^{c} a_{i,j}$. But at each iteration, we subtract $a_{i,m-I^-}$ to p_i or $a_{i,m+I^+}$ to q_i so there cannot be more than d iterations before both are equal to 0. When $p_i = q_i = 0$ for all i, $s = 0$, which terminates the loop. Hence the Algorithm terminates en $O(kd)$ comparisons.

Correctness. If the algorithm terminates because $\mid C \mid = 1$, C contains only one element and since C contains the winners, C is the set of winners. Otherwise, $s = 0$. Recall that s is the maximum of s_i and let i such that $s_i = s$. If $m_i^- > m_i^+$, we have $s_i = -q_i$ thus $q_i = 0$, which contradicts $p_i + m_i^- = m_i^+ + q_i$ and $p_i \geq 0$ so $m_i^- \leq m_i^+$ and $p_i = s_i = s = 0$. But $m_i^- \leq m_i^+$ and $p_i + m_i^- = m_i^+ + q_i$. Since $q_i \geq 0$, $q_i = 0$ thus $m_i^- = m_i^+$. Hence, by invariants 6 and 7, each candidate in C are equal with respect to $\leq_{maj}$. Since C contains the winners, C is the set of winners. □

C.3. An adaptation in MPC in the Paillier setting

In this section, we show how to adapt Algorithm 35 in MPC in the Paillier setting. Since we only focus on the tallying phase and since obtaining (an element-wise encryption of) the aggregated matrix from the ballots is easy in the Paillier setting, we consider that (an element-wise encryption of) the aggregated matrix is available. We first rewrite the algorithm into Algorithm 38 and prove that the new algorithm is equivalent to Algorithm 35. Using the building blocks from Section 2.1, it is easy to implement Algorithm 38 in MPC (see Algorithm 42).

We first provide Algorithm 36 which returns the grade vector as defined in [10]. The grade vector is a (term-by-term) encryption of g such that $g_j = 1$ if j is strictly greater than the best median m, and $g_j = 0$ otherwise. It will be useful to initialize p_i, m_i^-, m_i^+, q_i, $m - I^-$ and $m + I^+$.

Algorithm 36: Grade (Paillier setting)

Require: A such that, for all (i, j), $A_{i,j}$ is an encryption of the number of grades j given to candidate i
Ensure: G, such that for all j, G_j is an encryption of 1 if j is strictly greater than the best median, of 0 otherwise.

$V = \prod_{j=1}^{d} A_{1j}$
for $i = 1$ *to* k *(in parallel)* **do**
  for $j = 1$ *to* d *(in parallel)* **do**
    $B = \left(\prod_{l=1}^{j} A_{il}\right)^2$
    $C_{ij} = \texttt{Not}(\texttt{GTH}(B, V))$
for $j = 1$ *to* d *(in parallel)* **do**
  $G_j = C_{1j}$
  for $i = 2$ *to* k *(tree-based parallelisation is possible)* **do**
    $G_j = \texttt{Mul}(G_j, C_{ij})$
Return G

The idea of this algorithm is that, for all candidate i and grade j, j is strictly greater than the best median if and only if the number of grades greater than j is strictly lower than half the number of grades. This translates into the formula $2\sum_{l=1}^{j} a_{i,l} < n = \sum_{l=1}^{d} a_{i,l}$, which allows to compute $c_{i,j}$ for all (i, j), where $c_{i,j} = 1$ if j is strictly greater than i's median. To deduce the grade vector, we compute the logical conjunction column by column.

Once the grade vector is computed, we can initialize p_i, m_i^-, m_i^+ and q_i with Algorithm 37, which is adapted from [10].

Algorithm 37: InitD (Paillier setting)

Require: $(A_{ij}), G, n$ such that $A_{i,j}$ is an encryption of the number of j grades given to candidate i, while G is the grade vector and n the number of voters.
Ensure: P, M^-, M^+, Q where, for all i,
- P_i is an encryption of p_i, the number of grades received by i which are strictly greater than the best median,
- M_i^- is an encryption of $\lfloor n/2 \rfloor - p_i$,
- Q_i is an encryption of the number q_i of grades received by i which are strictly worse than the best median,
- M_i^+ is an encryption of $\lfloor n/2 \rfloor - q_i$.

for $i = 1$ *to* k **do**
  $P_i = \prod_{j=1}^{d} \texttt{Mul}(A_{ij}, Gj)$
  $M_i^- = \texttt{Enc}(\lfloor n/2 \rfloor)/P_i$
  $Q_i = \prod_{j=2}^{d} \texttt{Mul}(A_{ij}, \texttt{Not}(G_{j-1}))$
  $M_i^+ = \texttt{Enc}(\lfloor n/2 \rfloor)/Q_i$

The idea is that p_i can be obtained from G thanks to $p_i = \sum_{j=1}^{d} a_{i,j} g_j$ while q_i can be obtained similarly with a right shift of G's negation. Indeed, $\texttt{Not}(G)$ is the vector of encryptions of 1 if j is worse than the best median, of 0 otherwise. Its right shift is therefore encryptions of 1 if j is strictly worse than the best median, of 0 otherwise.

At this point, we remark that we can replace C as defined in line 2 of Algorithm 35 by the whole set of candidates, this without affecting the result, (see Lemma 3). In what follows, we call Algorithm 35.3 the Algorithm 35 in which this transformation has been done.

Lemma 3. *In Algorithm 35, replacing line 2 by "Let C be the set of all candidates" will not alter the output.*

Algorithm 38: MJ; version with a fixed number of loops, and an array of bits (indicator) instead of a set.

Require: a, the aggregated matrix.
Ensure: c, the indicator of the set of MJ winners.
1 Let m be the best median among all candidates
2 Let c such that for $i \in [1,k]$, $c_i = 1$
3 Let $I^- = 1$ and $I^+ = 1$ be counters
4 **for** $i = 1$ *to* k **do**
5 $\quad p_i = \sum_{j=1}^{m-1} a_{i,j}$, $q_i = \sum_{j=m+1}^{d} a_{i,j}$
6 $\quad m_i^- = \lfloor \frac{n}{2} \rfloor - p_i$, $m_i^+ = \lfloor \frac{n}{2} \rfloor - q_i$
7 **for** $j = 1$ *to* d **do**
8 $\quad$ **for** $i = 1$ *to* k **do**
9 $\quad\quad$ **if** $m_i^- \leq m_i^+$ **then**
10 $\quad\quad\quad s_i = p_i$
11 $\quad\quad$ **else**
12 $\quad\quad\quad s_i = -q_i$
13 $\quad$ **if** $c_i = 0$ **then**
14 $\quad\quad s_i = -n$ (* Already eliminated candidates are given a fake score *)
15 $\quad$ Let $s = \max\{s_i \mid i \in [1,k]\}$
16 $\quad$ **for** $i = 1$ *to* k **do**
17 $\quad\quad c_i = c_i \wedge (s_i == s)$
18 $\quad$ **if** $s \geq 0$ **then**
19 $\quad\quad$ **for** $i = 1$ *to* k **do**
20 $\quad\quad\quad m_i^+ = m_i^+ - m_i^-$
21 $\quad\quad\quad m_i^- = a_{i,m-I^-}$
22 $\quad\quad\quad p_i = p_i - a_{i,m-I^-}$
23 $\quad\quad I^- = I^- + 1$
24 $\quad$ **else**
25 $\quad\quad$ **for** $i = 1$ *to* k **do**
26 $\quad\quad\quad m_i^- = m_i^- - m_i^+$
27 $\quad\quad\quad m_i^+ = a_{i,m+I^+}$
28 $\quad\quad\quad q_i = q_i - a_{i,m+I^+}$
29 $\quad\quad I^+ = I^+ + 1$
30 Return c.

Proof. We show that after the first iteration of the loop, the C sets of both algorithms are the same, which shows that invariants from Lemma 2 are verified at the beginning of the second iteration of the loop, if any (if not the output is correct as well since the sets are the same).

Let m be the best median, and a and b be two candidates such that $\mathsf{med}(b) < \mathsf{med}(a) = m$. For all i, after line 7 in both algorithms, p_i is the number of grades strictly better than m received by candidate i while q_i is the number of grades strictly worse than m received by candidate i. By definition of the median, we have $q_a \leq \lfloor n/2 \rfloor$. On the other hand, $p_b \leq \lfloor n/2 \rfloor < q_b$. But after line 7, we have $m_i^- + p_i = m_i^+ + q_i = \lfloor n/2 \rfloor$ for all i so $m_b^- > m_b^+$ and $S_b = -q_b$ after line 13. As $S_a \in \{p_a, -q_a\}$ with $p_a \geq -q_a \geq -\lfloor n/2 \rfloor > -q_b$, we have $S_b < S_a$. Therefore b is discarded from C at line 15. □

Lemma 3 allows to initialize p_i, m_i^-, m_i^+ and q_i for all candidate i with no care of whether i's median is m or not. Now we explain how to run the while loop in MPC without revealing the number of iterations, nor the number of candidates which remain at any given point (see Lemma 4).

Lemma 4. *In Algorithm 35.3, we can replace line* 8 *by a for loop on* d *iterations, without affecting the result. Moreover, invariants from Lemma 2 are still preserved.*

Proof. Following the proof of Lemma 2, we remark that the proof does not depend on the number of iterations, so the loop invariants are preserved even if additional iterations are performed. Since the number of iterations is at most d as explained in the proof of Theorem 1, this concludes the proof. □

In what follows, we denote Algorithm 35.4 the Algorithm 35.3 in which line 8 is replaced by "for $j = 1$ to d do".

To encode C, we use its indicator (which we also denote C). To show the implied modification, we explicitly give Algorithm 38, where the transformations induced by Lemmas 3 and 4 have been made. To prove its correctness, we give the following lemma.

Lemma 5. *In Algorithm 38,* c *is the indicator of* C *from Algorithm 35.4.*

Proof. We verify that this property holds as a loop invariant.

Initialisation. Before the first loop iteration, we have $c_i = 1$ for all $i \in [1, k]$ and $C = [1, k]$ so c is C's indicator.

Heredity. Suppose that before the j^{th} iteration in Algorithm 38, c is the indicator of the set C such as before the j^{th} iteration in Algorithm 35.4. Then for $i \in C$, $c_i = 1$ so s_i is the same in both algorithms. On the other hand, for $i \notin C$, $c_i = 0$ so $s_i = -n$ in Algorithm 38. By Lemma 3, after the first loop iteration in Algorithm 35.3, C only contains candidates of median m. They therefore have at least a grade equal to m, so for all $i \in C$, $q_i \leq n - 1 < n$ after the first iteration. Since q_i can only decrease, we always have $p_i \geq -q_i > -n$ for $i \in C$, hence $s_i > -n$. Therefore, for $i \in C$ and $j \notin C$, $s_i > s_j$. This is also true in Algorithm 35.4, so s is the same in both algorithms after line 15. □

Now we explain how to get $a_{i,m-I^-}$ and $a_{i,m+I^+}$ without revealing $m - I^-$ et $m + I^+$. We use two vectors L and R of size d such that L_j is an encryption of 1 if $j = m - I^-$, of 0 otherwise, while R_j is an encryption of 1 if $j = m + I^+$, of 0 otherwise. This way $a_{i,m-I^-}$ and $a_{i,m+I^+}$ can be obtained with `SelectInd`. To initialize L and R, we use Algorithm 39 which uses the grade matrix g such that $g_j = 1$ if $j < m$, where m is the best median, and $g_j = 0$ otherwise. The idea is that $m - 1$ is the last index for which $g_j = 1$, so that $l_j = g_j - g_{j+1}$. Note that an initialization of R is obtained from L, with two right shifts. The only difficulty is when the best median is equal to the best possible grade, in which case g and l are null, while $r_2 = 1$. In any other case, $g_0 = 1$ and $r_2 = 0$, so we have $r_2 = 1 - g_0$.

Algorithm 39: InitP (Paillier setting)

Require: G, the grade matrix
Ensure: L, R, two vectors such that, for all i,
- L_i is an encryption of $i == m - 1$,
- R_i is an encryption of $i == m + 1$.

1 **for** $i = 1$ *to* $d - 1$ **do**
2 $\quad L_i = G_i / G_{i+1}$
3 $L_d = \texttt{Enc}(0)$
4 **for** $i = 3$ *to* d **do**
5 $\quad R_i = L_{i-2}$
6 $R_1 = \texttt{Enc}(0)$, $R_2 = \texttt{Not}(G_0)$
7 Return L, R

Algorithm 40: ConditionalLeftShift (`CLS`)

Require: V, B where V is a vector of $n - 1$ ciphertexts and B an encryption of a bit b.
Ensure: Return a (reencrypted) left shift of V if $b = 1$, a reencryption of V otherwise.

1 **for** $j = 1$ *to* $n - 1$ *(in parallel)* **do**
2 $\quad V'_j = \texttt{Select}(V_j, V_{j+1}, B)$ (* $V_n = \texttt{Enc}(0)$ *)
3 Return V'

In order to increment I^- and I^+, we use the simple Algorithms 40 and 41. Note that we always have $L_d = \texttt{Enc}(0)$ while $R_d = \texttt{Enc}(0)$, so L and R can be processed as vectors of $d - 1$ ciphertexts.

The complete procedure is given in Algorithm 42, whose correctness is the claim of Theorem 2. In this Algorithm, we add the constant n (the number of voters) to the candidates' scores at line 15, so that each integers to be compared are non-negative. The comparison requires therefore one additional bit but only for the first loop iteration. In the remaining

Algorithm 41: ConditionalRightShift (CRS)

Require: V, B where V is a vector of $n-1$ ciphertexts and B an encryption of a bit b.
Ensure: Return a (reencrypted) right shift of V if $b = 1$, a reencryption of V otherwise.

for $j = 2$ *to* $n-1$ *(in parallel)* **do**
  $V_j = \texttt{Select}(V_j, V_{j-1}, B)$ (* $V_1 = \texttt{Enc}(0)$ *)
Return V

Algorithm 42: MJ: MPC version (Paillier setting)

Require: A, n, the (encrypted) aggregated matrix and the number of voters.
Ensure: c, the indicator of the set of winners.

for $i = 1$ *to* k **do**
  $C_i = \texttt{Enc}(1)$
$G = \texttt{Grade}(A)$
$P, M^-, M^+, Q = \texttt{InitD}(A, G, n)$
$L, R = \texttt{InitP}(G)$
for $j = 1$ *to* d **do**
  (* scores computation *)
  for $i = 1$ *to* k *(in parallel)* **do**
    $B_1 = \texttt{GTH}(P_i, Q_i)$ (* $p_i \geq q_i$ *)
    $S_i = \texttt{Select}(1/Q_i, P_i, B_1)$ (* p_i if $p_i \geq q_i$, $-q_i$ otherwise *)
    $S_i = \texttt{Select}(\texttt{Enc}(-n), S_i, C_i)$ (* eliminated candidates get the fake $-n$ score*)
    $S_i = \texttt{Enc}(n)S_i$ (* $s_i = s_i + n$ *)
  $S = S_1$ (* research of the best score *)
  for $i = 2$ *to* k *(tree-based parallelisation is possible)* **do**
    $B_2 = \texttt{GTH}(S_i, S)$
    $S = \texttt{Select}(S, S_i, B_2)$ (* s_i is $s_i \geq s$, s otherwise *)
  for $i = 1$ *to* k *(in parallel)* **do**
    $B_3 = \texttt{EQH}(S, S_i)$
    $C_i = \texttt{Mul}(C_i, B_3)$ (* elimination of candidates who do not have the best score*)
  $B_4 = \texttt{GTH}(S, \texttt{Enc}(n))$
  for $i = 1$ *to* k *(in parallel)* **do**
    $A'_{i,m-I^-} = \texttt{SelectInd}((A_{i,1}, \cdots, A_{i,d-1}), L)$
    $A'_{i,m+I^+} = \texttt{SelectInd}((A_{i,2}, \cdots, A_{i,d}), R)$
    $T^+ = \texttt{Select}(A'_{i,m+I^+}, M_i^+/M_i^-, B_4)$ (* $m_i^+ - m_i^-$ if $b_4 = 1$, $a_{i,m+I^+}$ otherwise *)
    $T^- = \texttt{Select}(M_i^-/M_i^+, A'_{i,m-I^-}, B_4)$ (* $a_{i,m+-I^-}$ if $b_4 = 1$, $m_i^- - m_i^+$ otherwise *)
    $P_i = \texttt{Select}(P_i, P_i/A'_{i,m-I^-}, B_4)$ (* $p_i - a_{i,m-I^-}$ if $b_4 = 1$, p_i otherwise *)
    $M_i^- = T^-$
    $M_i^+ = T^+$
    $Q_i = \texttt{Select}(Q_i/A'_{i,m+I^+}, Q_i, B_4)$ (* q_i if $b_4 = 1$, $q_i - a_{i,m+I^+}$ otherwise *)
  $L = \texttt{CLS}(L, B_4)$, $R = \texttt{CRS}(R, \texttt{Not}(B_4))$
$c = \texttt{Dec}(C)$ (* bit-wise decryption *)
Return c

iterations, we have $q_i \leq \lfloor n/2 \rfloor$ so that we can add $\lfloor n/2 \rfloor$ instead of n. Since $p_i \leq \lfloor n/2 \rfloor$, we no longer need an extra bit. For simplicity, we did not explicitly write this optimization in Algorithm 42. Another notable difference compared to Algorithm 38 is that instead of computing $m_i^- \leq m_i^+$, we compute $p_i \geq q_i$ (which is equivalent by invariant 1 from Lemma 2) since p_i and q_i are non-negative, while m_i^+ and m_i^- could be negative during the first loop iteration.

Theorem 2. *Algorithm 42 is correct.*

Proof. See Lemmas 3, 4, 5 and 2, as well as Lemma 6 below. □

Lemma 6. *In Algorithm 42, after the i^{th} loop iteration, L and R are such that L_j is an encryption of $j == m - I^-$, while R_j is an encryption $j == m + I^+$.*

Algorithm 43: InitALL (ElGamal setting)

Require: $\boldsymbol{A}^{\text{bits}}$, such that, for all (i,j), ${A_{i,j}}^{\text{bits}}$ is a bit-encdoded encryption of $a_{i,j}$ from the aggregated matrix.

Ensure: $\boldsymbol{P}^{\text{bits}}, \boldsymbol{M^-}^{\text{bits}}, \boldsymbol{M^+}^{\text{bits}}, \boldsymbol{Q}^{\text{bits}}, L, R, C$ where, for all $i \in [1,k]$,

- ${P_i}^{\text{bits}}$ is a bit-wise encryption of p_i, the number of grades received by candidate i which are strictly greater than the best median,
- ${M_i^-}^{\text{bits}}$ is a bit-wise encryption of $\lfloor n/2 \rfloor - p_i$,
- ${Q_i}^{\text{bits}}$ is a bit-wise encryption of q_i, the number of grades received by candidate i which are strictly worse than the best median,
- ${M_i^+}^{\text{bits}}$ is a bit-wise encryption of $\lfloor n/2 \rfloor - q_i$,
- L_j is an encryption of $j == N - 1$ for all j, where N is the best median,
- R_j is an encryption of $j == N + 1$ for all j, where N is the best median,
- C_i is an encryption of 1 if i's median is N, of 0 otherwise.

for $i = 1$ *to* k *(in parallel)* **do**
  ${S_{i,1}}^{\text{bits}} = {A_{i,1}}^{\text{bits}}$
  for $j = 1$ *to* $d - 2$ **do**
    $D_{i,j} = \texttt{LT}({S_{i,j}}^{\text{bits}}, \lceil n/2 \rceil)$
    ${S_{i,j+1}}^{\text{bits}} = \texttt{Add}^{\text{bits}}({S_{i,j}}^{\text{bits}}, {A_{i,j+1}}^{\text{bits}})$ (* $s_{i,j} = \sum_{k=1}^{j} a_{i,j}$ *)
  $D_{i,d-1} = \texttt{LT}({S_{i,d-1}}^{\text{bits}}, \lceil n/2 \rceil)$
  $S_i, d^{\text{bits}} = n$
for $j = 1$ *to* $d - 1$*(in parallel)* **do**
  $G_j = D_{1,j}$
  for $i = 2$ *to* k *(tree-based parallelisation is possible)* **do**
    $G_j = \texttt{CGate}(G_j, D_{i,j})$
for $i = 1$ *to* k *(in parallel)* **do**
  $X = G_1 D_{i,1} / \texttt{CGate}(G_1, D_{i,1})^2$ (* $g_1 \oplus d_{i,1}$ *)
  $C_i = \texttt{Not}(X)$ (* $g_1 == d_{i,1}$ *)
  for $j = 2$ *to* $d - 1$ *(tree-based parallelisation is possible)* **do**
    $X = G_j D_{i,j} / \texttt{CGate}(G_j, D_{i,j})^2$, $C_i = \texttt{CGate}(C_i, \texttt{Not}(X))$ (* $g_j = d_{i,j}$ for all j *)
$L, R = \texttt{InitP}(G)$
for $i = 1$ *to* k *(in parallel)* **do**
  ${P_i}^{\text{bits}} = \prod_{j=1}^{d-1} \texttt{CGate}({S_{i,j}}^{\text{bits}}, L_j)$ (* Bit-wise product and $\texttt{CGate}$, as in $\texttt{SelectInd}^{\text{bits}}$ *)
  ${Q_i}^{\text{bits}} = \prod_{j=2}^{d} \texttt{CGate}({S_{i,j}}^{\text{bits}}, L_{j-1})$ (* same as above *)
  ${Q_i}^{\text{bits}} = \texttt{Sub}^{\text{bits}}(n, {Q_i}^{\text{bits}})$
  ${M_i^-}^{\text{bits}} = \texttt{Sub}^{\text{bits}}(\lfloor n/2 \rfloor, {P_i}^{\text{bits}})$, ${M_i^+}^{\text{bits}} = \texttt{Sub}^{\text{bits}}(\lfloor n/2 \rfloor, {Q_i}^{\text{bits}})$
Return $(\boldsymbol{P}^{\text{bits}}, \boldsymbol{M^-}^{\text{bits}}, \boldsymbol{M^+}^{\text{bits}}, \boldsymbol{Q}^{\text{bits}}, L, R, C)$

C.4. An adaptation in MPC in the ElGamal setting

In the previous section, we gave an adaptation in MPC of the MJ tally function in the Paillier setting. As explained in Section 2.2, it is interesting to consider ElGamal encryptions to obtain a better computation complexity, especially at the voter-side. Note that most of Algorithm 42 is easy to adapt in the ElGamal setting thanks to the toolbox we provide. In this setting, the (encrypted) aggregated matrix must be encrypted in bit-encoding, so that obtaining the aggregated matrix from the list of encrypted ballots is no longer straightforward, but requires kd parallel calls to $\texttt{Aggreg}^{\text{bits}}$, which is the main drawback of this approach. Even if those computations can be made on the fly while the voters submit their ballot, if nkd is too large, the Paillier setting might be preferable as this phase would be too expensive.

Algorithm 44: MJ: MPC version (ElGamal setting)

Require: $\boldsymbol{B}$, the n encrypted ballots (indexed by v)
Ensure: c, the indicator of the set of winners.

for $i = 1$ *to* k *(in parallel)* **do**
  for $j = 1$ *to* d *(in parallel)* **do**
    ${A_{i,j}}^{\text{bits}} = \texttt{Aggreg}^{\text{bits}}(B_{i,j,1}, \cdots, B_{i,j,n})$
$\boldsymbol{P}^{\text{bits}}, \boldsymbol{M}^{-\text{bits}}, \boldsymbol{M}^{+\text{bits}}, \boldsymbol{Q}^{\text{bits}}, L, R, C = \texttt{InitALL}(\boldsymbol{A}^{\text{bits}})$
for $j = 1$ *to* d **do**
  for $i = 1$ *to* k *(in parallel)* **do**
    $B_1 = \texttt{Not}(\texttt{LT}({P_i}^{\text{bits}}, {Q_i}^{\text{bits}}))$
    ${P^+}^{\text{bits}} = P_{i,0}, \cdots, P_{i,m-2}, E(1)$ (* $2^{m-1} + p_i$ *)
    ${Q^+}^{\text{bits}} = \texttt{Neg}({Q_i}^{\text{bits}})$ (* $2^{m-1} - q_i$ *)
    ${S_i}^{\text{bits}} = \texttt{Select}^{\text{bits}}({Q^+}^{\text{bits}}, {P^+}^{\text{bits}}, B_1)$
    ${S_i}^{\text{bits}} = \texttt{CGate}(S_{i,0}, C_i), \cdots, \texttt{CGate}(S_{i,m-1}, C_i)$ (* give the fake score 0 to already eliminated candidates *)
  ${S}^{\text{bits}} = {S_1}^{\text{bits}}$
  for $i = 2$ *to* k *(tree-base parallelisation is possible)* **do**
    $B_2 = \texttt{LT}({S}^{\text{bits}}, {S_i}^{\text{bits}})$
    ${S}^{\text{bits}} = \texttt{Select}^{\text{bits}}({S}^{\text{bits}}, {S_i}^{\text{bits}})$
  for $i = 1$ *to* k *(in parallel)* **do**
    $B_3 = \texttt{EQ}^{\text{bits}}({S}^{\text{bits}}, {S_i}^{\text{bits}})$
    $C_i = \texttt{CGate}(C_i, B_3)$
  $B_4 = S_{m-1}$ (* the most significant bit of s tells whether $s \geq 2^{m-1}$ *)
  for $i = 1$ *to* k *(in parallel)* **do**
    ${A'_{i,m-I^-}}^{\text{bits}} = \prod_{j=1}^{d-1} \texttt{CGate}({A_{i,j}}^{\text{bits}}, L_j)$ (* bit-wise product and CGate *)
    ${A'_{i,m+I^+}}^{\text{bits}} = \prod_{j=2}^{d} \texttt{CGate}({A_{i,j}}^{\text{bits}}, R_j)$ (* same as above *)
    ${M_{+-}}^{\text{bits}} = \texttt{Sub}^{\text{bits}}({M_i^+}^{\text{bits}}, {M_i^-}^{\text{bits}})$
    ${M_{-+}}^{\text{bits}} = \texttt{Neg}({M_{+-}}^{\text{bits}})$
    ${T^+}^{\text{bits}} = \texttt{Select}^{\text{bits}}({A'_{i,m+I^+}}^{\text{bits}}, {M_{+-}}^{\text{bits}}, B_4)$
    ${T^-}^{\text{bits}} = \texttt{Select}^{\text{bits}}({M_{-+}}^{\text{bits}}, {A'_{i,m-I^-}}^{\text{bits}}, B_4)$
    ${P_i}^{\text{bits}} = \texttt{Select}^{\text{bits}}({P_i}^{\text{bits}}, \texttt{Sub}^{\text{bits}}({P_i}^{\text{bits}}, {A'_{i,m-I^-}}^{\text{bits}^{\text{bits}}}, B_4))$
    ${M_i^-}^{\text{bits}} = {T^-}^{\text{bits}}$
    ${M_i^+}^{\text{bits}} = {T^+}^{\text{bits}}$
    ${Q_i}^{\text{bits}} = \texttt{Select}^{\text{bits}}(\texttt{Sub}^{\text{bits}}({Q_i}^{\text{bits}}, {A'_{i,m+I^+}}^{\text{bits}}), {Q_i}^{\text{bits}}, B_4)$
  $\texttt{CLS}(L, B_4)$, $\texttt{CRS}(R, \texttt{Not}(B_4))$
$c = \texttt{Dec}(C)$ (* bit-wise decryption *) Return c

Another difference is that in the Paillier setting, some procedures were performed thanks to the homomorphic property while they need the $\texttt{Add}^{\text{bits}}$ algorithm in the ElGamal setting. As replacing each multiplication of two ciphertexts in

Algorithm 42 by a call to Algorithm 9 might deteriorate the complexity too much, we made a few modifications listed below.

First, we give Algorithm 43 which allows to initialize p_i, m_i^-, m_i^+ and q_i, just as Algorithm 37, but also initialize L and R as in Algorithm 39. Finally Algorithm 43 also initializes C as the indicator of the candidates whose median is the best median. In what follows, we use bold characters to denote a matrix of elements. For instance, $\boldsymbol{A}^{\text{bits}}$ stands for a matrix of size kd, whose elements are bit-encoded encrypted integers. By abuse of notation, we use $\lfloor n/2 \rfloor$ or n instead of bit-encoded encryption of the said integer.

Algorithm 43 is a merger of Algorithms 36, 37 and 39. Merging all three algorithms together allows to exploit common intermediate computations. Note that at line 4, we compute $\lceil n/2 \rceil > s_{i,j}$ instead of $n > 2s_{i,j}$, so as to use one bit less. (See Lemma 7 which states that the two comparisons are equivalent.)

Lemma 7. *For all $n, s \in \mathbb{Z}$, we have $n > 2s$ if and only if $\lceil n/2 \rceil > s$.*

Proof. Let n, s be integers. If $n > 2s$, $\lceil n/2 \rceil \geq n/2 > s$. Conversely, suppose that $\lceil n/2 \rceil > s$. We first consider the case where n is even. Then $n/2 = \lceil n/2 \rceil$ so $n = 2\lceil n/2 \rceil > 2s$. If n is odd, we have $\lceil n/2 \rceil = (n+1)/2$ so $n+1 > 2s$, therefore $n + 1 \geq 2s + 1$, hence $n \geq 2s$. Since n is odd, $n \neq 2s$, thus $n > 2s$. □

In Algorithm 42, we did not have to initialize C (see Lemma 3). However, as the variables could be negative, we decided to add a constant. This would not be that easy in the ElGamal setting since adding a constant to a bit-encoded encrypted integers would require a non-trivial operations. In this case, eliminating the candidates who do not have the best median right away so as to initialize C consistently with Algorithm 35 has approximately the same computational cost. Afterwards, for all i, we have $\mid s_i \mid \leq \lfloor n/2 \rfloor$ so we can add the constant 2^{m-1} instead, where m is the bit length of the integers. Indeed, $2^{m-1} > \lfloor n/2 \rfloor \geq q_i$ and $2^{m-1} + p_i \leq 2^{m-1} + \lfloor n/2 \rfloor < 2^m$. This is of interest because computing $2^{m-1} + p_i$ is completely free (just add $\texttt{Enc}(1)$ as the most significant bit); so we just have to call $\texttt{Neg}$ once (to compute $2^{m-1} - q_i$) instead of calling twice $\texttt{Add}^{\text{bits}}$.

Finally, we obtain Algorithm 44 for our ElGamal version of a fully-hiding tallying of MJ.

Appendix D. Condorcet method: Schulze and ranked-pairs variants

In this Section, we give details about our approach to handle the Condorcet tally function that was only sketched in Section 5. While only the Condorcet-Schulze variant is mentioned in the main body of the article, we cover here also the ranked-pairs method. We refer to [1] for a discussion and a comparison of the many Condorcet variants, Schulze and ranked pairs being only two of them.

After recalling the notion of adjacency matrix, we will define with more details Schulze and ranked pairs and explain how they can be implemented once this matrix is known. We will then focus on how to compute this matrix from the encrypted ballots. We mostly consider the context of ElGamal bit-encoding encrypted integers, as it is the most complex. If one is ready to let the adjacency matrix leak, then things become easier and we can use the natural encoding.

D.1. Schulze and ranked pairs from the adjacency matrix

In the Condorcet methods, voters are asked to rank each candidate, potentially with ties (several candidates may have the same rank). The Condorcet winner is the candidate which is preferred to every other candidate by a majority of voters. Schulze and ranked pairs differ when there is no Condorcet winner. Like in many versions of Condorcet, only the adjacency matrix is needed to compute the winners, which is defined in Definition 3. In all what follows, we denote $d_{i,j}$ the number of voters who prefers (strictly) candidate i over candidate j.

Definition 3 (Adjacency matrix)**.** *The adjacency matrix is the matrix $(a_{i,j})$ defined by*

$$a_{i,j} = \begin{cases} d_{i,j} - d_{j,i} & \text{if } d_{i,j} \geq d_{j,i} \\ 0 & \text{otherwise.} \end{cases}$$

The Schulze variant.

The Schulze variant consists of several steps. First, compute $d_{i,j}$ for all (i, j). Second, compute $b_{i,j} = d_{i,j} - d_{j,i}$ for all (i, j). For all pair of candidates (u, v), a path p of length n from u to v is a finite sequence of $n + 1$ candidates such that $u = p_0$ and $v = p_l$. We say that $(i, j) \in p$ if there exists an index $0 \leq k < n$ such that $i = p_k$ and $j = p_{k+1}$. The strength of a path p is defined as $s(p) = \min_{(i,j)\in p} b_{i,j}$. The third step of the Schulze method is to compute $f_{i,j} = \max_{\sigma \in [i \rightsquigarrow j]} s(\sigma)$, where $[i \rightsquigarrow j]$ denotes the set of all paths from i to j. Finally, i is a winner by the Schulze method if $f_{i,j} \geq f_{j,i}$ for all j.

If a is the adjacency matrix, a Schulze tally can be derived from a (see Lemma 8). When a is seen as the adjacency matrix of a graph, the Schulze method is well known to be equivalent to the shortest path problem [28], that can be solved with standard algorithms [15], [34].

Lemma 8. *A Schulze tally can be performed from the adjacency matrix, by using $a_{i,j} = \max\{0, b_{i,j}\}$ instead of $b_{i,j} = d_{i,j} - d_{j,i}$, where $d_{i,j}$ is the number of voters who prefers i over j.*

Proof. For all path p, we denote $s(p) = \min_{(i,j)\in p} b_{i,j}$ and $s'(p) = \min_{(i,j)\in p} a_{i,j}$. For all (i,j), we denote

$$f_{i,j} = \max_{\sigma\in[i\rightsquigarrow j]} \min_{(u,v)\in\sigma} b_{i,j}$$
$$f'_{i,j} = \max_{\sigma\in[i\rightsquigarrow j]} \min_{(u,v)\in\sigma} a_{i,j}.$$

With these notations, the statement of the lemma becomes

$$\forall i, (\forall j, f_{i,j} \geq f_{j,i}) \iff (\forall j, f'_{i,j} \geq f'_{j,i}).$$

Let i be a candidate, suppose that for all j, $f_{i,j} \geq f_{j,i}$ (*i.e.* i is a Schulze winner). Let j be any candidate. If $j = i$, clearly $f'_{i,j} \geq f'_{j,i}$, so we assume that $j \neq i$. Since $j \neq i$, there is no path from i to j (nor from j to i) of length 0. As $f_{i,j} \geq f_{j,i}$, there exists a path p from i to j (of length $n > 0$) such that for all path p' from j to i (of length $n' > 0$), there exists $k' < n'$ such that for all $k < n$, $b_{p'_{k'},p'_{k'+1}} \leq b_{p_k,p_{k+1}}$. We consider two cases.

First, if $b_{p_k,p_{k+1}} < 0$ for some k, then for all p', $b_{p'_{k'},p'_{k'+1}} < 0$ for all k', hence $a_{p'_{k'},p'_{k'+1}} = 0$ for all k', thus $s'(p') = 0 \leq s'(p)$. Since this is holds for all p', $f'_{j,i} = 0 \leq f'_{i,j}$.

Second, if $b_{p_k,p_{k+1}} \geq 0$ for all k, then for all k, $a_{p_k,p_{k+1}} = b_{p_k,p_{k+1}}$. Now consider any path p' (of length $n' > 0$) from j to i. If $b_{p'_{k'},p'_{k'+1}} \geq 0$ for all k', then $s'(p') = s(p') \leq f_{j,i} \leq f_{i,j} = s(p) = s'(p) \leq f'_{i,j}$. If there exists k' such that $b_{p'_{k'},p'_{k'+1}} < 0$, then $s'(p') = 0 \leq f'_{i,j}$. Therefore $f'_{j,i} \leq f'_{i,j}$.

Conversely, let i such that $f'_{j,i} \leq f'_{i,j}$ for all j. Let j be any candidate (as above, w.l.o.g. we assume that $i \neq j$). We consider three cases.

First, suppose that $f_{i,j} < 0$. Then for all path p from i to j, there exists $(u,v) \in p$ such that $b_{u,v} < 0$ (we call this proposition $*$). In particular, $b_{i,j} < 0$, so $b_{j,i} = -b_{i,j} > 0$, hence $b_{j,i} = a_{j,i}$ and $f'_{j,i} \geq s'_{j,i} = a_{j,i} = b_{j,i}$. In addition, $s_{j,i} = b_{j,i} > 0$, so $f'_{j,i} \geq s_{j,i} > 0$. On the other hand, by $*$ we have $f'_{i,j} = 0$, which contradicts $f'_{j,i} \leq f'_{i,j} < 0$. Therefore $f_{i,j} \geq 0$.

Second, suppose that $f_{i,j} = 0$. Then for all path p from i to j, there exists $(u,v) \in p$ such that $b_{u,v} \leq 0$, hence $f'_{i,j} = 0$. Let p' be a path from j to i (of length $n' > 0$). Suppose that for all $(u,v) \in p'$, $b_{u,v} > 0$. Then $0 < s'(p) \leq f'_{j,i}$, which contradicts $f'_{j,i} \leq f'_{i,j}$. Consequently, there exists $(u,v) \in p'$ such that $b_{u,v} \leq 0$, therefore $s(p') \leq 0 = f_{i,j}$. This holds for all p' so $fj, i \leq f_{i,j}$.

Finally, suppose that $f_{i,j} > 0$. Let p' be a path from j to i. If there exists $(u,v) \in p'$ such that $b_{u,v} \leq 0$, then $s(p') \leq 0 < f_{i,j}$. Otherwise, for all $(u,v) \in p'$, $b_{u,v} > 0$ so $s(p') = s'(p') \leq f'_{j,i} \leq f'_{i,j}$, so we just have to show that $f_{i,j} \geq f'_{i,j}$.

Let p be a path from i to j. If there exists $(u,v) \in p$ such that $b_{u,v} \leq 0$, $s'(p) = 0 < f_{i,j}$. Otherwise, for all $(u,v) \in p$, $b_{u,v} > 0$ so $s'(p) = s(p) \leq f_{i,j}$, which concludes the proof. □

From this lemma, the Schulze tally can be derived by a simple Floyd-Warshall algorithm and we give it in Algorithm 46 for completeness. This has a cost that is cubic in the number of candidates n.

Algorithm 45: `FW` (Floyd-Warshall algorithm)

Require: P, the encrypted adjacency matrix
Ensure: S, such that $S_{i,j}$ is an encryption of the strength of the strongest path from i to j
(* n is the number of candidates *)

```
S = P
for k = 1 to n do
    for i = 1 to n (in parallel) do
        for j = 1 to n (in parallel) do
            (* proceed only if (i ≠ j) *)
            A_{i,j} = Select(S_{k,j}, S_{i,k}, LT(S_{i,k}, S_{k,j}))
            B_{i,j} = Select(S_{i,j}, A_{i,j}, LT(S_{i,j}, A_{i,j}))
    S_{i,j} = B_{i,j} for all (i ≠ j)
Return S
```

The ranked pairs variant.

Algorithm 46: Schulze (from adjacency matrix)

Require: A, the encrypted adjacency matrix
Ensure: c, the indicator of the Schulze winners
(* n is the number of candidates *)
$S = \texttt{FW}(A)$
for $i = 1$ *to* n *(in parallel)* **do**
  for $j \neq i$ *(in parallel)* **do**
    $b_j = \texttt{Not}(\texttt{LT}(S_{i,j}, S_{j,i}))$
  $C_i = \texttt{CSZ}_{j \neq i}(b_j)$ (* use tree-based parallelization to compute the conjunction of all b_j *)
Return $c = \texttt{Dec}(C)$

The ranked pairs is another algorithm which allows to break ties when there is no Condorcet winner. In this method, the adjacency matrix is seen as the adjacency matrix of a graph G. The Ranked Pairs protocol consists of three steps. First, sort the edges of G in decreasing order of weights. Let G' be the graph which consists of k vertices (where k is the number of candidates) and no edge. Second, for all edge of G taken in decreasing order, if this edge does not create a cycle in G', add this edge in G'. Finally, as G' is an oriented graph without cycle, G' is the graph of a partial order over the candidates. The sources of the graph are the winners according to the Ranked Pairs protocol.

Assuming the adjacency matrix is known, an MPC version of the ranked pairs method goes as follows. First, to shuffle the edges, we can use Algorithm 30 for `MergeSort`, not forgetting to run a mixnet first. The edges can be encoded with three ciphertexts, one for the source, one for the destination and one for the weight. Then, the main procedure is to update a matrix $B_{i,j} = \texttt{Enc}(b_{i,j})$, where $b_{i,j} = 1$ if there is a path from i to j, and 0 otherwise. Initially, B is simply an encryption of the identity matrix. To add the edge (i, j) simply compute $b'_{s,t}$ for all (s, t), as follows:

$$b'_{s,t} = b_{s,t} \vee (b_{s,i} \wedge b_{j,t}).$$

The edge will create a cycle if and only $b'_{s,t} = b'_{t,s} = 1$ for some (s, t), hence we compute the encryption of the boolean

$$c = \vee_{s \neq t}(b'_{s,t} \wedge b'_{t,s}).$$

Finally, we can update $b_{i,j}$ using `Select` and c.

The problem that remains is that (i, j) is unknown, since the edges are encrypted. A simple solution is to perform the test $u == i$ and $v == j$ for all (u, v), using the known (u, v) and the encryptions of (i, j), and to update each $b_{u,v}$ using `Select`, so as to hide the results of both tests (only one $b_{u,v}$ will be modified, while the others will be re-encrypted). This leads to an additional $O(k^2 \log k)\texttt{CGate}$, as `EQ` requires $O(\log k)\texttt{CSZ}$. Finally, finding the source of the graph can be done by exhaustive search on the final B, which cost $O(k^2\texttt{CSZ})$. The whole process can be performed in $O(k^4 \log k)\texttt{CSZ}$ in terms of computation and transcript size, and $O(k^2 \log k\texttt{CSZ})$ in terms communications.

D.2. How to obtain the adjacency matrix from the voters' ballots

The preference matrices.

The choice of a voter can be modelled by a preference matrix. We consider two types of such matrices (see Figure 11). The m_a preference matrix format is antisymmetric, therefore only $k(k-1)/2$ elements need to be considered. The m_p preference matrix has only non-negative integers, which can also be an advantage.

$$m_a[i,j] = \begin{cases} 1 & \text{if } i \text{ is preferred over } j \\ -1 & \text{if } j \text{ is preferred over } i \\ 0 & \text{otherwise.} \end{cases} \qquad m_p[i,j] = \begin{cases} 1 & \text{if } i \text{ is preferred over } j \\ 0 & \text{otherwise.} \end{cases}$$

Figure 11. Two types of preference matrix

Deducing the adjacency matrix from the set of preference matrices of each voter boils down to aggregating with a bit more details as explained below. Using the homomorphic property, this is straightforward from the m_a type, but this requires either to be in the Paillier setting or to reveal the adjacency matrix. Otherwise, the bit-encoding is required, and then the m_p matrix format is better suited.

Ballots encoded as list of integers.

We assume here that each ballot consists of $k\lceil \log(k+1) \rceil$ ciphertexts, along with zero knowledge proofs that they are encryptions of 0 or 1. Those ciphertexts are interpreted as k bit-encoded integers, which encrypt integers in $[0, 2^L - 1]$,

where 2^L is the first power of 2 greater than k. This way the voter can give each candidate a rank (which is not necessarily between 0 and $k-1$), and can give the same rank to several candidates without any restriction.

First, we consider the easy case where only the m_a preference matrix in the natural encoding is needed, because the adjacency matrix will be revealed. In that case, we simply use a variant of LT which returns an additional bit for the equality test (see Section A.6). Let C_i^{bits} and C_j^{bits} be the bitwise encrypted rank of candidates i and j for some ballot. Let $Z, T = \texttt{LT}(C_i^{\text{bits}}, C_j^{\text{bits}})$. Then $M_a[i,j] = Z^2T/\texttt{Enc}(-1)$, and $M_a[j,i] = 1/M_a[i,j]$. Therefore the preference matrix m_a can be obtained in $k(k-1)/2$ calls of LT, which accounts for $\frac{3}{2}k(k-1)\log k\,\texttt{CSZ}$ in computation and transcript size, and $2\log k\,\texttt{CSZ}$ in communication since all $m_a[i,j]$ can be computed in parallel.

For a full tally-hiding procedure, we need the result to be in bit-encoding and the m_p preference matrix is better suited. Similarly, we use a variant of LT which returns an additional bit. This additional bit allows to derive $m_p[j,i]$ from $m_p[i,j]$ using Not and CSZ. Hence the preference matrix is obtained with $\frac{3}{2}k(k-1)\log k\,\texttt{CSZ}$ in computation and transcript size, and $2\log k\,\texttt{CSZ}$ in communication just as in the previous case. The aggregation requires to call $\texttt{Aggreg}^{\text{bits}}$ to obtain a matrix D. By construction, $D_{i,j}$ is a bit-wise encryption of the number $d_{i,j}$ of voters who prefers i over j. For all $i < j$, we can then use SubLT to compute (bit-encoded encryptions of) $b_{i,j} = d_{i,j} - d_{j,i}$, as well as an additional bit $(b_{i,j} < 0)$. This bit allows to derive the adjacency matrix by setting all negative values to zero using CSZ, and by computing $b_{j,i}$ from $b_{i,j}$ using Neg and CSZ.

Ballots encoded as preference matrices (quadratic algorithm).

We explain now how the voters can directly encode their choice as a preference matrix of the m_a type. The difficulty is for the voter to prove in zero-knowledge that the matrix encoded in their ballot is indeed a preference matrix, *i.e.* that it corresponds to an ordering of the candidates. This is of great interest if one is ready to leak the adjacency matrix, because then the tally can be done by the authorities without any MPC protocol apart from the decryption.

We start by explaining our method in the cleartexts. Suppose that Alice wants to vote the ordering $(1, \cdots, k)$ (*i.e.* the candidate number i is ranked i^{th}). Then her preference matrix would be as follows.

$$m_{\mathsf{init}}[i,j] = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } i < j \\ -1 & \text{otherwise.} \end{cases}$$

Now assume that Alice wants to rank $\sigma(i)^{th}$ the candidate number i, for some permutation σ that encodes her choice. If the candidate number i were numbered $\sigma(i)$ instead, Alice could have voted with m_{init} as above. This means that the preference matrix of Alice m_a is such that $m_a[\sigma^{-1}(i), \sigma^{-1}(j)] = m_{\mathsf{init}}[i,j]$ for all (i,j). Therefore m_a can be obtained by using the permutation σ to shuffle m_{init} (using the permutation on the rows, then on the columns, with the ShuffleMatrix function).

So far, Alice can only choose a strict ordering of the candidates. Assume that she wants to give the same rank to several candidates and let r_i be the rank of candidate i according to her. Alice first sorts the candidates according to their rank, in increasing order. Let σ be the permutation used for sorting. At this point, σ is an arbitrary permutation such that $\sigma(i) < \sigma(j) \implies r_i \leq r_j$. To obtain her preference matrix m_a from m_{init}, Alice will transform m_{init} into m_σ, such that $m_\sigma[i,j] = m_a[\sigma^{-1}(i), \sigma^{-1}(j)]$. For this purpose, she computes a vector b of size $k-1$ such that for all i, $b_i = 1$ if $r_{\sigma^{-1}(i)} = r_{\sigma^{-1}(i+1)}$, and 0 otherwise. Afterwards, Alice modifies m_{init} diagonal by diagonal, so as to incorporate this change. For the first diagonal, we have $m_{\mathsf{init}}[i, i+1] = 1$ while we would like $m_\sigma[i, i+1] = 1 - b_i$. This can be done easily using the homomorphic property.

For the $(j+1)^{th}$ diagonal $(i, i+j+1)_i$, assume that the previous diagonal is correct. Then, as the candidates are sorted in order of preference, we have

$$m_\sigma[i, i+j+1] = \begin{cases} 0 & \text{if } (m_\sigma[i, i+j] = 0) \wedge (m_\sigma[i+1, i+j+1] = 0), \\ 1 & \text{otherwise.} \end{cases}$$

Therefore, Alice can apply an iterative algorithm, using the following formula:

$$\begin{aligned} m_\sigma[i, i+j+1] &= 1 - (1 - m_\sigma[i, i+j])(1 - m_\sigma[i+1, i+j+1]) \\ &= m_\sigma[i, i+j] + m_\sigma[i+1, i+j+1] - m_\sigma[i, i+j] m_\sigma[i+1, i+j+1]. \end{aligned}$$

Once m_σ is obtained, Alice can finally derive m_a by shuffling the rows and the columns, using the permutation σ and the ShuffleMatrix function.

The algorithm that we sketched above is interesting because it requires only a quadratic number of steps and it only uses transformations for which there is a standard zero knowledge proof. Indeed, a public and canonical encryption of m_{init} is available so Alice does not have to prove that m_{init} is well-formed. For the first diagonal, Alice simply has to provide $(k-1)$ ciphertexts and $0/1$ zero knowledge proofs. For the remaining diagonals, Alice has to provide an encryption

Z of $m_\sigma[i, i+j]m_\sigma[i+1, i+j+1]$, as well as zero knowledge proof of well-formedness. For this purpose, Alice uses Algorithm 47 which produces a transcript π_{mul} of the form $(e_1, e_2, a_1, a_2, a_3)$. To verify the proof, one computes $d = \texttt{hash}(X||Y||Z||e_1||e_2)$ where X is the encryption of $m_\sigma[i, i+j]$ and Y the encryption of $m_\sigma[i+1, i+j+1]$, and checks that the following equations are verified:

$$Y^{a_3}\texttt{Enc}(0, a_1)Z^{-d} = e_1$$
$$\texttt{Enc}(a_3, a_2)X^{-d} = e_2.$$

Finally, the shuffle can be performed with a standard proof of a shuffle.

Algorithm 47: ZKmult

Require: $\texttt{hash}, X, Y, x, r_x$, such that $X = \texttt{Enc}(x, r_x)$ and Y is any ciphertext
Ensure: Z, π_{mul}, such that $Z = \texttt{ReEnc}(Y^x)$ and π_{mul} is a zero knowledge proof of well-formedness

$\alpha, r_1, r_2, w \in_r \mathbb{Z}_q$, $Z = Y^x\texttt{Enc}(0, \alpha)$
$e_1 = Y^w\texttt{Enc}(0, r_1)$, $e_2 = \texttt{Enc}(w, r_2)$
$d = \texttt{hash}(X||Y||Z||e_1||e_2)$
$a_1 = r_1 + \alpha d$, $a_2 = r_2 + r_x d$, $a_3 = w + xd$
$\pi_{mul} = (e_1, e_2, a_1, a_2, a_3)$
Return Z, π_{mul}

Ballots encoded as preference matrices (cubic algorithm).

For completeness, we propose another way to prove that an encrypted m_a preference matrix is well-formed, where the voter provides two proofs:

- A proof that each element is an encryption of either 0, 1 or -1,
- A proof of transitivity.

The proof of transitivity must prove the following statements, for all (i, j, k) and $u \in \{-1, 0, 1\}$

$$(m_a[i, k] = u) \wedge (m_a[k, j] = u) \implies m_a[i, j] = u.$$

Since the voter also provides a proof that each $m_a[i, j]$ is indeed in $\{-1, 0, 1\}$, this is equivalent to proving that, for all (i, j, k), $m_a[i, k] = m_a[k, j] \implies m_a[i, j] = m_a[i, k]$, which is equivalent to proving that the following statement is true:

$$(m_a[i, j] \neq m_a[i, k]) \vee (m_a[i, k] = m_a[k, j]).$$

To prove that $m_a[i, j] \neq m_a[i, k]$, one can prove that $m_a[i, j] - m_a[i, j] \in \{-2, -1, 1, 2\}$ and to prove that $m_a[i, k] = m_a[k, j]$, we prove that the difference is 0. Overall, the voter has to prove that, for all (i, j, k),

$$(m_a[i,j]-m_a[i,k] = -2)\vee(m_a[i,j]-m_a[i,k] = -1)\vee(m_a[i,j]-m_a[i,k] = 1)\vee(m_a[i,j]-m_a[i,k] = 2)\vee(m_a[i,k]-m_a[k,j] = 0).$$

The proof of the disjunction can be obtained with the process of [13] (see algorithm below). To verify such a proof, simply compute $d = \texttt{hash}(A_1||\cdots||A_5||e_1||\cdots||e_5)$ and check that $e_j = \texttt{Enc}(0, \rho_j)(A_j/\texttt{Enc}(b_j, 0))^{-\sigma_j}$ for all j. Overall, the zero knowledge proof requires about $18k^3$ for the prover and $20k^3$ for the verifier.

Algorithm 48: ZKP of a 5-disjunction

Require: $\texttt{hash}, A_1, \cdots, A_5, a_1, \cdots, a_5, r_1, \cdots, r_5, b_1, \cdots, b_5$, such that
- for all i, $A_i = \texttt{Enc}(a_i, r_i)$
- there exists i such that $a_i = b_i$

Ensure: $(e_1, \cdots, e_5, \sigma_1, \cdots, \sigma_5, \rho_1, \cdots, \rho_5)$, a Zero Knowledge proof that there exists i such that $a_i = b_i$.

Let i such that $a_i = b_i$
$w \in_r \mathbb{Z}_q$, $e_i = \texttt{Enc}(0, w)$
for $j \neq i$ **do**
    $\sigma_j, \rho_j \in_r \mathbb{Z}_q$
    $e_j = \texttt{Enc}(0, \rho_j)(A_j/\texttt{Enc}(b_j, 0))^{-\sigma_j}$
$d = \texttt{hash}(A_1||\cdots||A_5||e_1||\cdots||e_5)$
$\sigma_i = d - \sum_{j\neq i} \sigma_i$
$\rho_i = w + \sigma_i r_i$
Return $(e_1, \cdots, e_5, \sigma_1, \cdots, \sigma_5, \rho_1, \cdots, \rho_5)$

Appendix E. Single Transferable Vote

Section 6 contains a sketch of our results on Single Transferable Vote (STV). We give here more material about this: we recall the general idea of STV and some variants, then explain in details the algorithms to use for each step of an MPC implementation, and finally explain how the costs given in table 6 were obtained.

Overview of STV

STV consists of the following algorithm. As usual, we denote by s the number of seats to be attributed. First, each voter chooses a subset of candidates (any other candidate is not deemed of interest by the voter) and rank them in a strict order. For instance, if there are four candidates, Alice can vote $(1, 3)$ while Bob can vote $(4, 1, 2)$. Each ballot is attributed a weight, which is initially 1. Once all the ballots are cast, the tallying process consists of several rounds. During each round, each ballot grants a number of votes (equal to the ballot's weight) to the first candidate mentioned in the ballot. If some candidates meet a certain quota q (which is fixed during the whole process), the one with the greatest number of votes is selected. The selected candidates keep q votes for themselves and transfer each of their ballot to the next candidate on the ballot, with a transfer coefficient $t = (v - q)/v$, where v is the number of votes of the selected candidate (note that v might not be an integer). In other words, the name of the selected candidate is removed from the ballot and the weight is multiplied by t. The eliminated candidates transfer their ballot to the next candidate in the ballot, but with the same weight. The process terminates when s candidates are elected, or when the number of candidates that remain is equal to the number of (still) available seats.

There are several versions of STV. In the version that we chose to consider, the tallying process consists of several rounds, and in each round, exactly one candidate is either selected or eliminated. In some other versions, several candidates can be selected or eliminated simultaneously, if some conditions are met. This comes with two problems. First, for an MPC tally, revealing no more than the result also means not to reveal the number of candidates which were selected or eliminated in any round, so having a non-constant number of eliminations or selections is quite difficult. Second, if several candidates are selected simultaneously, the exact way in which the transfer should occur is not clear. Indeed, suppose that candidates a and b are selected. For each ballot possessed by a, a has to transfer a certain proportion t of the ballot to the second candidate mentioned in the ballot, but t depends on the number of votes possessed by a. So what if a must transfer some votes to b while b must transfer some votes to a? Which transfer coefficient should be used? While some variants of STV choose to ignore the selected candidates in the transfer process (don't transfer to b but to the next candidate that is not already selected), some other variants require to solve a system of c equations of degree c, where c is the number of candidates selected simultaneously [25].

A tally-hiding algorithm for STV.

In what follows, we will only consider the ElGamal setting with bit-encoding, but a similar approach could be used in the Paillier setting as well (some procedures would become easier). Recall from Section 6 that each ballot consists of $(k+1)$ bit-wise encrypted integers, which are obtained by shuffling a public vector which contains bit-encoded encryptions of $(0, \cdots, k)$, where k is the number of candidates. The candidate 0 is an artificial candidate: any candidate ranked after 0 should be ignored. Finally, recall that we represent rational numbers with an approximation in the first r binary places, where r is fixed by the election administrator. First, we initialize a data structure as follows (recall that q is the quota, s the number of seats, k the number of candidates and n the number of voters).

- H is the *hopeful* vector. It contains k encryptions of bits (initially E_1, the public encryption of 1).
- W is the *winner* vector. It contains k encryptions of bits (initially E_0, the public encryption of 0).
- S is the *score* vector. It contains k bit-encoded encrypted integers of size $m + r$, where $m = \lceil \log(n + 1) \rceil$.
- B is the *ballots* matrix. For all $i \in [1, n]$, B_i consists of a weight V_i (a bit-encoded encrypted integer of size $r + 1$, initially $(E_0, \cdots, E_1)$, which stands for the bit-encoded encryption of 1; the r less significant bits represent the r binary places) and $k + 1$ candidates $B_i[0], \cdots, B_i[k]$ (candidates are represented as bit-encoded encrypted integers, of size $\lceil \log(k + 1) \rceil$ bits).

In what follows, if P is a MPC procedure that requires two (bit-encoded) inputs, we denote P_k the procedure P in which the second input is known in the clear. If m is the bitsize of the inputs, P_k costs generally $m\,\mathtt{CSZ}$ less than P, which often leads to a good improvement (a third or a half of the computations is saved, see Algorithm 19 for an example). Our algorithm consists of $k - 1$ rounds, which themselves consists of the following procedures (in this order, we wrote the procedures so as to match as much as possible with what is explained in Section 6). At the very end, the vector W is decrypted into w, and the elected candidates i are such that $w_i = 1$.

In STV, the procedure should stop when s candidates have been selected or when the number of candidates that remain is equal to the number of seats that remain. If s candidates are selected, adding some additional rounds will not modify the result as it is not possible for $(s + 1)$ or more candidates to reach the quota (*i.e.* no subsequent selection would occur, therefore W will no longer be modified). However, if the number of candidates that remain is equal to the number of seats that remain, adding an additional round may lead to an elimination if no candidate reach the quota, so it is important to select all candidates right away. Since a candidate is either selected or eliminated each round, the round index t is such

Algorithm 49: Finished

Require: s, t, H, W, where t the round index (initially 0)
Ensure: Modify W
$N^{\text{bits}} = \texttt{Aggreg}^{\text{bits}}(W_1, \cdots, W_k)$ (* bit-wise encryption of the number of selected candidates *)
$F = \texttt{EQ}_k(N, s - k + t)$ (* when one of the operand is known in the clear, the procedure is cheaper *)
for $i = 1$ *to* k *(in parallel)* **do**
    $H_i = \texttt{CSZ}(H_i, F)$
    $W_i = \texttt{Select}(W_i, W_i H_i, F)$

that the number of candidates that remain is equal to $k - t$. Moreover, the number of seats that remain is simply s minus the number of selected candidates. So we compute the latter (say n') and we test if $n' = s - k + t$, which is equivalent to $s - n' = k - t$.

Note that we do not want to reveal *when* the procedure stop so, in MPC, the procedure should actually continue. In what follows, we explain why the result (the decryption of W) will not be modified if subsequent iterations are run. First, once this test returns true, n' becomes s and since $t < k$ (there are $k - 1$ rounds), the test can no longer return true, so this modification will occur only once. Afterwards, only selection and elimination would occur and since selecting a candidate which is already selected does not change anything, the outcome is not altered by the subsequent rounds.

Algorithm 50: CountVotes

Require: B, S
Ensure: Modify S
for $i = 1$ *to* n *(in parallel)* **do**
    for $j = 1$ *to* k *(in parallel)* **do**
        $C_{i,j} = \texttt{EQ}_k(B_i[0], j)$
for $j = 1$ *to* k *(in parallel)* **do**
    ${S_j}^{\text{bits}} = 0$
    for $i = 1$ *to* n *(tree-based parallelisation is possible)* **do**
        ${S_j}^{\text{bits}} = \texttt{Select}^{\text{bits}}({S_j}^{\text{bits}}, \texttt{Add}^{\text{bits}}({S_j}^{\text{bits}}, V_i), C_{i,j})$

In the procedure CountVotes, we mention that tree-based parallelisation is possible. Indeed, it is possible to compute all $\texttt{Select}^{\text{bits}}(0, V_i, C_{i,j})$ in parallel, then to add them together using a tree-based algorithm. Hence the communication cost of this step is $O(\log(n)\texttt{Add})$, where $\texttt{Add}$ is the communication cost of an addition.

The last two procedures, namely SearchMinMax, and SelectDeleteTransfer are self-explanatory.

Algorithm 51: SearchMinMax

Require: S, q
Ensure: $D, C^{\text{bits}}, T^{\text{bits}}$, where
- D is an encryption of a bit d ($d = 1$ for a selection, 0 for an elimination)
- C^{bits} is a bit-wise encryption of the index of some candidate (with $\lceil \log(k+1) \rceil$ bits)
- T^{bits}, is a bit-wise encryption of the transfer coefficient (with $r + 1$ bits)

$\_, M^{\text{bits}}, I^{\text{bits}}, J^{\text{bits}} = \texttt{MinMax}^{\text{bits}}({S_1}^{\text{bits}}, \cdots, {S_k}^{\text{bits}})$
$\Delta^{\text{bits}}, D = \texttt{SubLT}_k(M^{\text{bits}}, q)$, D=Not (D)
$T^{\text{bits}} = \texttt{Div}(\Delta^{\text{bits}}, M^{\text{bits}})$
$T^{\text{bits}} = \texttt{Select}^{\text{bits}}(1, T^{\text{bits}}, D)$ (* use a bit-wise encryption of 1 *)
$C^{\text{bits}} = \texttt{Select}^{\text{bits}}(I^{\text{bits}}, J^{\text{bits}}, D)$
Return $D, C^{\text{bits}}, T^{\text{bits}}$

Complexity analysis (naive approach).

Recall that k is the number of candidates, n the number of voters, s the number of seats, $m = \lceil \log(m+1) \rceil$ and r the precision in terms of binary places. First, assume that we use the naive version for each algorithm.

The complexity of $\texttt{Finished}$ can be derived directly from Figure 9. Since we use $\texttt{Aggreg}^{\text{bits}}$ with k operands, one $\texttt{EQ}$ for two operands of size $\log k$ and $2k\texttt{CSZ}$, the complexity of this step is $(5k + \log k)\texttt{CSZ}$ in terms of computation and

Algorithm 52: SelectDeleteTransfer

Require: $D, C^{\text{bits}}, T^{\text{bits}}, W, H, B$
Ensure: Modify W, H, B

for $i = 1$ *to* k *(in parallel)* **do**
  $Z = \texttt{EQ}_k(C^{\text{bits}}, i)$
  $H_k = \texttt{CSZ}(H_k, Z)$
  $W_k = \texttt{Select}(W_k, \texttt{Enc}(1), Z)$
for $i = 1$ *to* n *(in parallel)* **do**
  $A = \texttt{EQ}(B_i[0], C^{\text{bits}})$
  $F = A$ **for** $j = 0$ *to* $k - 1$ **do**
    $B_i[j] = \texttt{Select}^{\text{bits}}(B_i[j], B_i[j+1], F)$
    $Z = \texttt{EQ}(B_i[j+1], C^{\text{bits}})$
    $F = FZ/\texttt{CSZ}(F, Z)$ (* $f = 1$ iff the candidate c has been found in the list *)
  $B_i[k] = \texttt{Select}^{\text{bits}}(B_i[k], 0, F)$ (use a bit-wise encryption of 0)
  $V_i = \texttt{Select}^{\text{bits}}(W_i, \texttt{Mul}^{\text{bits}}(W_i, T^{\text{bits}}), A)$

transcript size, and $((\log k)^2 + \log\log k + 1)$ rounds in terms of communications. (For simplicity we will only keep the leading terms, here $5k$ and $(\log k)^2$.)

The complexity of `CountVotes` can also be derived from Figure 9. There are nk calls to `EQ` for inputs of size $\log k$ and nk calls of $\texttt{Add}^{\text{bits}}$ and $\texttt{Select}^{\text{bits}}$ for inputs of size $(m+r)$. Therefore the cost is $nk(\log k + 3(m+r))\texttt{CSZ}$ in terms of computation and transcript size. However, as a tree-based parallelisation is possible, the communication cost is about $2(m+r)m$ rounds, as $m \approx \log n$.

The complexity of `SearchMinMax` is also obtained from Figure 9. As there are k operands of size $m+r$, `MinMax` costs $2k(3(m+r) + 2\log k)\texttt{CSZ}$ in terms of computation and transcript size, and $2(m+r)\log k$ rounds of communication. The remaining of the procedure consists (mainly) of a call to `Div` and `SubLT` (the two `Select` cost $O(\log k)$ and $O(r)$ in computations, and 1 round each). Overall, the cost of this step is about $(3(m+r)(r+2k) + 2k\log k)\texttt{CSZ}$ in terms of computation and transcript size and $2(m+r)(r+\log k)$ rounds of communication.

In `SelectDeleteTransfer`, there are k calls to $\texttt{EQ}_k$ which costs $\log k\texttt{CSZ}$ each (the subsequent `CSZ` and `Select` use $1\texttt{CSZ}$ each. This part is negligible in terms of both computations and communications ($O(\log\log k)$ rounds.) Afterwards, there are nk calls to `EQ` and $\texttt{Select}^{\text{bits}}$ for inputs of size $\log k$, which accounts to $3nk\log k\texttt{CSZ}$ in terms of computation, and $k\log\log k$ rounds of communication. Finally, we multiply two inputs of size r and $m+r$ and use $\texttt{Select}^{\text{bits}}$ for inputs of size $(m+r)$, which accounts to $3nr(m+r)\texttt{CSZ}$ in terms of computation and transcript size and $2r(m+r)$ rounds. Overall, the complexity is about $3n(k\log k + r(m+r))\texttt{CSZ}$ in terms of computation and transcript size, and $(k\log\log k + 2r(m+r))$ rounds.

Overall, since there are $k-1$ rounds, the leading terms of the complexity are

- $nk(4k\log k + 3(m+r)(k+r))\texttt{CSZ}$ in terms of computation and transcript size,
- $2k((m+r)(m+2r+\log k) + \frac{1}{2}k\log\log k)$ rounds of communications.

Complexity analysis (advanced approach).

The complexity of our algorithm is satisfying in terms of computations: recall from Section 6 that we aim for $O(nk^2)$ operations; and the $\log k$ and $(m+r)$ terms seems unavoidable as they are the bitsize of some operands. However, the number of rounds is quadratic in both m, r and k. While m, as the logarithm of n, is not expected to grow too much, the strong dependency in k and r can be problematic. In what follows, we use the arithmetic of Section A.7 to explain how to avoid this quadratic number of rounds. For this purpose, it is crucial to identify which processes need a quadratic number of rounds. From the analysis above, we identify four terms which contribute to this.

- In `CountVotes`, we use the associativity of the addition to sketch a tree-based parallelisation of the loop which leads to $2(m+r)m$ rounds of communications. To mitigate this quadratic cost, we can use Algorithm 14 for the addition instead of Algorithm 9. This allows to perform the same step in $2m\log(m+r)$ rounds instead, but requires $\frac{3}{2}nk^2(m+r)\log(m+r)\texttt{CSZ}$ instead of $2nk^2(m+r)\texttt{CSZ}$.
- In `SearchMinMax`, the computation of the transfer coefficient implies a division, which leads to a quadratic number of rounds $2(m+r)r$. By replacing `SubLT` calls by the equivalent Unbounded Fan-in composition (the subtraction can be obtained similarly with the same complexity), the division can be performed in $2r\log(m+r)$ rounds, but the complexity increases slightly (it becomes $\frac{3}{2}r(m+r)\log(m+r)\texttt{CSZ}$ instead of $3r(m+r)\texttt{CSZ}$). Note that the complexity of this phase in terms of computation is still negligible compared to the rest of the algorithm.

- In `SelectDeleteTransfer`, the multiplications can all be computed in parallel, but each still requires a quadratic number $2r(m+r)$ of rounds. Just as above, using Algorithm 14 instead of the naive $\texttt{Add}^{\text{bits}}$ allows to reduce the number of rounds to $2r\log(m+r)$, but the computation costs increases from $3nr(m+r)\texttt{CSZ}$ to $\frac{3}{2}nr(m+r)\log(m+r)\texttt{CSZ}$.
- In `SelectDeleteTransfer` again, there is a for loop in k which imposes a round complexity of $O(k\log\log k)$ (testing the equality of two integers of $\log k$ bits takes $\log\log k$ rounds). As the procedure is repeated $k-1$ times, this leads to a quadratic number of rounds in k. Once again, we can use the strategy from Section A.7 to solve this problem. First, compute all equality tests in parallel (denote the result $b_0, b_1, \cdots, b_k$. Then use an Unbounded Fan-in circuit to compute all the prefixes $b_0, b_0 \vee b_1, \cdots, b_0 \vee \cdots \vee b_k$. (Since the operation $\vee$ is associative, the same technique can be applied.) Finally for all i in parallel, compute the updated $B_i[j]$ as $\texttt{Select}^{\text{bits}}(B_i[j], B_i[j+1], b_0 \vee \cdots \vee b_j)$, where $B_i[k+1]$ is a bit-wise encryption of 0 for all i. This time the number of rounds decreases to $k\log k\log\log k$ (from $k^2\log\log k$), while the computation cost increases slightly (from $3nk^2\log k\texttt{CSZ}$ to $\frac{7}{2}nk^2\log k\texttt{CSZ}$). Note that interestingly, the communication cost of this step becomes negligible before the aggregation process in the `Finished` procedure, which was negligible in the naive approach.

Using the modifications sketched above, we arrive to a good communication / computation trade-off: the impact on the computation is minimal, but the number of rounds is no longer quadratic in any variable.