ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Qelect: Lattice-based Single Secret Leader Election Made Practical

Yunhao Wang
Yale University

Fan Zhang
Yale University

Abstract

In a single secret leader election (SSLE) protocol, all parties collectively and obliviously elect one leader. No one else should learn its identity unless it reveals itself as the leader. The problem is first formalized by Boneh *et al.* (AFT'20), which proposes an efficient construction based on the Decision Diffie-Hellman (DDH) assumption. Considering the potential risk of quantum computers, several follow-ups focus on designing a post-quantum secure SSLE protocol based on pure lattices or fully homomorphic encryption. However, no concrete benchmarks demonstrate the feasibility of deploying such heavy cryptographic primitives.

In this work, we present Qelect, the first practical constant-round post-quantum secure SSLE protocol. We first adapt the commitment scheme in Boneh *et al.* (AFT'23) into a *multi-party randomizable commitment* scheme, and propose our novel construction based on an adapted version of ring learning with errors (RLWE) problem. We then use it as a building block and construct a *constant-round* single secret leader election (crSSLE) scheme. We utilize the single instruction multiple data (SIMD) property of a specific threshold fully homomorphic encryption (tFHE) scheme to evaluate our election circuit efficiently. Finally, we built Qelect from the crSSLE scheme, with performance optimizations including a preprocessing phase to amortize the local computation runtime and a retroactive detection phase to avoid the heavy zero-knowledge proofs during the election phase. Qelect achieves asymptotic improvements and is concretely practical. We implemented a prototype of Qelect and evaluated its performance in a WAN. Qelect is at least two orders of magnitude faster than the state-of-the-art.

1 Introduction

Permissionless consensus [26, 31] is a novel primitive that powers blockchains and decentralized applications. A key building block is to elect leaders among a group of nodes without fixed membership in a fair, unique, and secret way. Fairness [6] is crucial for Sybil-proofness, ensuring that each node is elected with the same probability, even if some parties are malicious.[1] Uniqueness ensures that a single leader is elected at a time to avoid ambiguity. Unpredictability (or secrecy) refers to a privacy property stating that the leader's identity is unknown to the public until the leader announces itself. Achieving all these properties is challenging, and most existing systems fail to achieve secrecy or uniqueness. For instance, VRF-based cryptographic sortition [20] does not guarantee uniqueness, whereas Ethereum's randomness-beacon-based scheme does not guarantee secrecy.

Recent works tackled this challenge with a primitive called *Single Secret Leader Election* (SSLE) [3, 4, 6, 11, 12, 16, 18, 21, 28]. However, most practical schemes in the literature [4, 11, 12, 21] are based on cryptographic assumptions (such as DDH) vulnerable to quantum adversaries, who can break these SSLE by recovering secret keys from public tokens. Considering that SSLE is to be run in a highly adversarial environment, the demand for a post-quantum version naturally arises, especially following the recent success of standardization by NIST. While prior works [6, 18] and [4, section 5] have presented post-quantum schemes using lattices and Fully Homomorphic Encryption (FHE), they are theoretical and provide no concrete implementation. It remains open whether using such heavy cryptographic machinery to build practical SSLE schemes is feasible.

In this paper, we present Qelect, a novel and highly optimized FHE-based scheme, taking a substantial step towards practical post-quantum blockchain protocols.

1.1 Existing Works and Challenges

Several works [4, 6, 18] proposed post-quantum SSLE constructions, but none is practical. We briefly recall representative schemes to show their challenges and provide background for our solution. We use G to denote the number of leader candidates, which is typically less than a few thousands.

[1]We assume that each node has the same amount of stake as in prior works [4, 6]. We note that this does not trivialize the problem.

The original post-quantum scheme by Boneh *et al.* [4, Section 5] uses threshold FHE (tFHE) to draw a random leader from a list of registered users. Specifically, users register by uploading a tFHE ciphertext encrypting a secret k_i. To elect a leader, users evaluate an optimized circuit to compute G ciphertexts with a random one encrypting 1 and others encrypting 0. Then, they take the inner product of the ciphertext vector and the said binary vector and reveal the decryption. The user whose secret is revealed knows they have been elected, but nobody else knows whose the revealed secret is. While the scheme described so far is efficient (the circuit is optimized to have only $O(\log\log G)$ multiplicative depth), it is vulnerable to what they called the *modification attack* due to the malleability of tFHE ciphertext (a form of chosen-ciphertext attacks). To prevent this, users need to attach proofs for the well-formedness of FHE ciphertexts, making the protocol much less practical (according to [13], for a ring with dimension 32768, a single such proof takes minutes.) Moreover, the tFHE ciphertext for registration needs a sufficient noise budget, which further harms the efficiency of such proofs [7].

Freitas *et al.* [18] proposed a scheme that prevents the modification attack by hashing the secrets in FHE so that the secrets cannot be recovered from the output. The drawback is that the FHE circuit they construct requires extensive homomorphic multiplications ($O(G^2)$).

Recently, Boneh *et al.* [6] introduced a post-quantum secure solution based on the ring learning with errors (RLWE) assumption, following the paradigm of the original DDH solution in [4]. The core primitive is a post-quantum *re-randomizable commitment* (RRC) scheme, which uses shuffling and randomization to break the linkage between scheme outputs and inputs. Unfortunately, their scheme involves users *sequentially* shuffling and re-randomizing a list of G commitments, thus incurring $O(G)$ round complexity. Moreover, RLWE-based randomization suffers from the buildup of noises, so users need to re-register after a certain number of randomization operations. These factors greatly limit their performance in real-world networks.

Technical challenges. Observe that modification attacks are possible for two reasons: first, the tFHE ciphertext encrypting user secrets is inherently malleable (allows homomorphic operations); second, the scheme reveals the leader's secret directly. Together, what is akin to a chosen-ciphertext attack is possible. The scheme in [4] fixes the malleability by attaching proofs of well-formedness, which is expensive. The scheme in [18] avoids this by hashing the secret in the FHE circuit, which leads to an inefficient circuit. The scheme in [6] achieves a similar effect by randomizing commitments to secrets in FHE, requiring *sequential* computation by each user, leading to linear round complexity and super-linear noise build-up.

One lesson learned from existing works is that although tFHE is the most promising direction in constructing a post-quantum secure SSLE protocol, there are several technical challenges in designing a practical one. First, it is essential to keep the depth of the FHE circuit low so that the local computation is light. Second, we must avoid zero-knowledge proofs on the critical path of the protocol, as they can be prohibitively expensive. Third, since SSLE will be run in WAN, the communication rounds and the bandwidth consumption should be minimized.

In this work, we address these challenges and propose Qelect. Compared to prior works, Qelect achieves asymptotic improvements as summarized in Table 1 with constant rounds of communications, linear local homomorphic operations, and an FHE circuit of only logarithmic multiplicative depth. More importantly, Qelect is the first protocol with concretely practical implementation.

1.2 Our Solution

Qelect is constructed in a modular fashion. First, we present the new Multi-party randomizable commitment (MPRC) scheme that features low (constant) round complexity and small noise build-up. Then, using MPRC as a building block, we present a constant-round SSLE scheme (crSSLE) in the honest-but-curious model. crSSLE features optimized circuits that take full advantage of the SIMD (Single Instruction Multiple Data) operations of our tFHE scheme and have a low multiplication depth. Finally, we augment crSSLE with malicious security and obtain Qelect while avoiding heavy proofs for FHE evaluations.

In this section, we present an overview of each component and its improvement over existing works; we will present a step-by-step walk-through in § 4.

System model. We assume a synchronous network with G parties connected by pairwise authenticated communication channels. In principle, Qelect can be built using a τ-ouf-of-G tFHE scheme, but such schemes are not practical for large G currently (c.f. § 6.2.3). We focus on G-out-of-G tFHE instead. Note that this model implies that all G parties remain online, or progress (liveness) is lost trivially.

Multi-party randomizable commitment. We present a new primitive we call *multi-party randomizable commitment* (MPRC) that can randomize commitments without communication. MPRC allows users to compute a commitment c to a message m; anyone can add randomness to c via a *local* procedure to get randomized commitment c'; multiple such c' can be combined to get a single commitment c'' to m. In addition to satisfying binding and hiding, MPRC guarantees *unlinkability*, stating that an aggregated commitment c'' cannot be linked to the original version c as long as one of the users adds enough entropy.

MPRC can be viewed as a generalization of the re-randomizable commitment (RRC) in [4, 6], but with several advantages: first, MPRC has constant round complexity as opposed to $O(G)$ thanks to the local randomization procedure; second, the final noise grows only logarithmically with the number of parties as opposed to superlinear; third, our

Table 1: Asymptotic cost comparison across SSLE protocols that offer post-quantum security. G is the number of parties, $q \geq G$ is the plaintext modulus, N is the ring dimension of underlying tFHE scheme. For the RLWE based RRC scheme in [6], the local computation work only involves normal multiplications in plaintext. For the *multi-election* column, we consider whether a scheme could be easily extended to elect multiple leaders in one round. *Notice that for larger party size ($G \geq 128$), we could reduce the communication round to 3. **The constructions in [4, 6] could not be trivially generated to multi-election, while the work [18] that implements a sorting algorithm in FHE circuit involves a scaling circuit to make it suitable for multi-election, of which the authors leave the concrete arithmetic circuit design to future works. ***The work in [4] suffers from modification attack, and thus they need each party to generate a heavy proof of knowledge of its initial input.

	Commu. Round	Commu. Cost	FHE Multiplication Depth		Homomorphic Multiplication Operations	Randomness Beacon	Modification Attack***
			Single Election	Multi-Election**			
tFHE-based in [4]	2	$O(G)$	$O(\log\log G)$		$O(\log G)$	Yes	Yes
tFHE-based in [18]	2	$O(G)$	16	$\Omega(\log G)$	$O(G^2)$	Yes	No
RLWE-based in [6]	G	$O(G^2)$	0		0	Yes	No
Our work (§ 6.2.2)	**6***	$\mathbf{O(G)}$	$\log q + 2$		$\log q + N + G$	**No**	**No**

MPRC primitive allows the commitment to convey a message, as opposed to just all-zeros, which could be of independent interest in other applications.

Constant-round single secret leader election. The parallel randomization feature of MPRC enables a constant-round single secret election scheme, which we call crSSLE.

In crSSLE, each party commits to its secret via MPRC and uses the commitment as its identity in an election. To elect a leader from a set of identities, all parties derive a common ciphertext encrypting a random value $u \in \mathbb{Z}_G$ and expand it into G "indicator ciphertexts" with the u-th one encrypting one, while all others encrypting zero. Next, each party locally randomizes the commitments from all parties (using MPRC) and homomorphically evaluates an inner product between the indicator ciphertexts and the randomized commitments, selecting the u-th randomized commitment while hiding u from all. Finally, all parties combine their local results as the final output. Users then use MPRC to verify whether the revealed commitment corresponds to their secret, and if so, they reveal the secret to claim the leadership. This step is standard in SSLE schemes.

crSSLE satisfies all three properties (*uniqueness*, *fairness*, and *unpredictability*) under the honest-but-curious setting where users follow the protocol honestly but try to glean more information. Unpredictability is achieved, as the final output cannot be linked to inputs due to randomization, and that u is hidden by FHE. Fairness is satisfied because crSSLE guarantees that u is uniformly random. Uniqueness is satisfied because exactly one commitment will be chosen.

The performance optimization of crSSLE involves taking full advantage of the SIMD operations supported by the tFHE scheme we use. Specifically, we adapt the Brakerski/Fan-Vercauteran (BFV) scheme [8, 17] into a threshold fashion [5, 25]. By leveraging the SIMD property of BFV, we can efficiently evaluate the election circuit on multiple data encrypted in one ciphertext and amortize our computation time across multiple rounds. Asymptotically, the local computation only takes $O(G)$ homomorphic operations. With $2^{15} = 32768$ parties, the multiplication depth of our FHE circuit could be as low as 18.

Qelect: our SSLE protocol with malicious security. Finally, we augment crSSLE to deal with malicious users and obtain our final construction Qelect. Two types of deviation are possible: first, an attacker can send maliciously crafted messages, potentially derived from received messages, to manipulate the election results; second, an attacker can send malformed messages (e.g., random noise) to prevent progress. They are handled differently in Qelect.

First, we present a novel technique to prevent the adaptive crafting of inputs based on received messages. While the standard commit-and-reveal can work (each party first publishes the hash value of their messages before releasing the actual messages), they add communication rounds. We use a *local* random sampling method that avoids additional communication. The idea is that each party hashes the received messages to sample a subset of them to aggregate. Under proper parameters, the probability where the attacker can guess the correct subset before crafting its message is negligible. This trick applies to the first and second broadcast, and saves two rounds of broadcasts compared to standard commit-and-reveal.

Since we operate in G-out-of-G setting, the model inherently assumes the participation of all parties, but a malicious user can send arbitrarily malformed messages. For instance, they can send random noise so that the final output cannot be opened by any user—no leader will be elected. While sending malformed messages is hard to prevent (without heavy cryptographic proofs), we observe that SSLE is typically used in blockchain protocols with the ability to penalize misbehavior when irrefutable proof is presented. E.g., with Proof of Stake such as Ethereum, parties deposit collateral to join the protocol, and their collateral can be programmatically taken away upon detection of misbehavior. This ability is known as slashing [9] and is widely supported by production systems. Qelect is designed to catch the senders of malformed messages after the fact, so they can slashed.

Qelect thus executes on an efficient optimistic path by default; A pessimistic path is invoked when the protocol fails to elect a leader. The pessimistic path identifies the misbehaving nodes, removes their right to participate in future elections, and confiscates their collateral. Because stalling the election is the best thing an adversary can do, and such misbehavior can be detected and punished, there is a strong disincentive to cheating in the first place.

Implementation and evaluation. We are the first to present an implementation of a post-quantum secure SSLE protocol, which is implemented in $\sim$3K lines of C++. Since there is currently no efficient Distributed Key Generation (DKG) for the tFHE protocol we use, our implementation and benchmark assume an efficient centralized trusted setup.

With party size $G \leq 2048$, the local evaluation of our FHE circuit could finish within seconds. Since none of the prior post-quantum [4, 6, 18] SSLE constructions offer concrete benchmarks, we compare our protocol with them in Table 1 regarding the asymptotic cost and other security aspects. We assert that our approach achieves a near-optimal round complexity with lightweight local computations while preserving the same level of security guarantees as previous works. To highlight, with the rigorous implementation and concrete benchmarks, the asymptotic cost of our protocol is free from large hidden constants.

Compared with the DDH-based schemes [4, 21], our local computation is two to three orders of magnitude slower than the estimation given in [21], which could be seen as the overhead introduced by post-quantum security. To compare with the state-of-the-art post-quantum scheme [6] (which does not have an implementation), we made a generous estimation of its performance by only taking the time of propagating messages into account (i.e., assuming computation does not take time) with parameters set to *minimize* their message size. Under the same network setting, our scheme is *at least two orders of magnitude faster* than [6]. Hence, we claim that our implementation is the first practical SSLE protocol that offers post-quantum security.

Contributions

- Qelect is the first practical post-quantum SSLE scheme. We reported on its construction and implementation. We plan to open-source the code.
- We evaluated the performance of Qelect in LAN and WAN settings. Qelect is at least *two orders of magnitude faster* than the state-of-the-art [6].
- Our asymptotically and concretely efficient construction of *multi-party randomizable commitment* (MPRC) is of its own interest, with applications where shuffled output should remain hidden until opened by the sender, such as anonymous sealed-biding and multi-party shuffling.

2 Related Work

In the original work of [4], three different approaches are proposed to solve the SSLE problem: the first one is to use indistinguishable obfuscation to hide the election problem that takes in all public keys of the users and outputs a random token encrypted via the elected leader's public key. The second approach leverages threshold fully homomorphic encryption to evaluate a block cipher to get a randomness and uses that randomness to perform an inner product with all winner tokens. The final approach, the most practical and well-studied one, denoted directly as BEGH, relies on the DDH assumption: every user generates a Diffie-Hellman commitment, attaches it to a public list containing commitments of all other users previously registered, and shuffles the whole list before broadcasting. A random commitment is selected via a randomness beacon and the leader opens the commitment as the proof of winning.

Later on, two works strengthen the security definition of SSLE: [11] forwards the universally composable definition based on the original game-based model and [12] formalizes the attack model in the presence of adaptive corruptions. The former work proposes a construction based on public key encryption with keyword search which is realized via pairing under symmetric external Diffie-Hellman assumption and achieves better on-chain efficiency. The later work mainly follows the path of BEGH but keeps track of two different lists of commitments, such that one is used for shuffling similar to BEGH while the other is used for secret-updating so that the adversary who adaptively corrupts the user gains no information for other rounds.

Functionality-wise, both works [3, 18] consider the non-uniform stake distribution. [3] realizes the SSLE protocol via oblivious selection in the generic MPC model, which consumes $O(\log G)$ rounds of communication, where G is the total number of users in the committee, and brings only $O(\log S)$ overhead to the protocol, where S is the total amount of stake. [18] constructs SSLE protocol with nearly no overhead for non-uniform stake distribution, but relies on an expensive FHE circuit that needs to evaluate comparison, selection, and domain transformation of a uniform random number.

One of the latest works [6] formally adapts the BEGH method in a post-quantum setting, and abstracts the scheme into re-randomizable commitment (RRC) primitive plus a shuffling protocol. They realize the RRC primitive from lattices based on the (ring) learning-with-errors ((R)LWE) assumption. However, compared to the original BEGH scheme which takes a single round of shuffling performed by the previous leader during the election, its post-quantum adaption needs $O(G)$ sequential shuffling for every round of election due to the inherent noise issue of (R)LWE.

A recent blog post [28] proposes a potential analog of Whisk based on commutative super-singular isogenies [10], which is believed to be post-quantum secure.

3 Preliminary

Notations. Let $\mathcal{D}$ denote an arbitrary distribution, $\mathcal{B}$ denote the binary distribution (i.e., uniformly random over $\{0,1\}$). An element u is uniformly sampled from space $\mathbb{Z}$ when $u \xleftarrow{\$} \mathbb{Z}$, and drawn from the distribution $\mathcal{D}$ when $u \leftarrow \mathcal{D}$ or $u \leftarrow_r \mathcal{D}$ with random coin r. When invoking an interface f, we say $y \xleftarrow{\$} f(x)$ is the output of the randomized algorithm f given input x, and $y' \leftarrow f(x)$ is the output of the deterministic algorithm. For a vector $\vec{x}$ (or simply x), we use $\vec{x}[i]$ to represent the i-th element of this vector. We use $\vec{e}_i$ to present a one-hot vector with value one on index i. Let $\mathcal{R} := \mathbb{Z}[X]/(X^N+1)$ denote the $2N$-th cyclotomic ring where the ring dimension N is a power-of-two, and $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$, for some prime $q \in \mathbb{Z}$ s.t. $q \mod 2N = 1$. We also use the vector of the coefficients to denote a ring element, i.e., $a \in \mathcal{R}_q = \sum_{i\in[N]} a_i X^i, a_i \in \mathbb{Z}_q$, can be specified by the vector $\vec{a}$ (or simply a) where $\vec{a}[i] := a_i$.

We use a few syntax sugar to simplify notation. We write a vector of N elements as a string of N symbols, and use $\|$ (concatenation) to concatenate vectors. E.g., the vector of N zeroes is denoted as 0^N; the N-dimension vector $(u,0,\cdots,0)$ is denoted $u\|0^{N-1}$. We use $\boxed{x}$ to represent a ciphertext encrypting x (element or vector).

For simplicity, when we say "broadcast", we mean that some message is sent to all via *pairwise* authenticated channel, i.e., we are not invoking any broadcasting protocol or assuming a broadcast channel, unless otherwise specified.

The product of two ring elements $a,x \in \mathcal{R}_q$ is defined by polynomial multiplication. Using the vector form, ax can be written as:

$$\begin{pmatrix} a[0] & -a[N-1] & -a[N-2] & \dots & -a[1] \\ a[1] & a[0] & -a[N-1] & \dots & -a[2] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a[N-1] & a[N-2] & a[N-3] & \dots & a[0] \end{pmatrix} \begin{pmatrix} x[0] \\ x[1] \\ \vdots \\ x[N-1] \end{pmatrix}.$$

3.1 Ring Learning With Errors

RLWE assumption. We recall the standard decision ring learning with error (RLWE) assumption [24]. Let $n,q,\mathcal{D},\chi$ be parameters dependent on λ and n is a power of two. Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n+1)$. The ring learning with error (RLWE) assumption $\mathsf{RLWE}_{n,q,\mathcal{D},\chi}$ states that it is computationally infeasible to distinguish $(a, a\cdot s+e)$ and (a,b), where $a \xleftarrow{\$} \mathcal{R}_q, s \leftarrow \mathcal{D}, e \leftarrow \chi$ and $b \xleftarrow{\$} \mathcal{R}_q$.

RLWE-based public-key encryption. In our construction, we use the RLWE-based short-key public-key encryption, denoted sRLWE [22, 24]. The "short-key" stands for the *ternary* secret key of this scheme, which also has a fixed small hamming weight h. This feature allows the scheme to have a smaller public key size than LWE-based schemes. Looking ahead, the sRLWE public key encryption scheme will be used to instantiate our multi-party randomizable commitment scheme. At a high level, the sRLWE contains four algorithms 1) $\mathsf{sRLWE.GenParams}(1^\lambda, \ell, q, \sigma, h)$ which returns the public parameter $\mathsf{pp_{rlwe}}$ which includes the ring dimension n of sRLWE, the secret key distribution $\mathcal{D}$ and the error range γ, 2) $\mathsf{sRLWE.KeyGen}(\mathsf{pp_{rlwe}}; r)$ which outputs a key pair $(\mathsf{sk}, \mathsf{pk})$ based on the random coin r, 3) $\mathsf{sRLWE.Enc}(\mathsf{pp_{rlwe}}, \mathsf{pk}, \vec{m})$ which outputs a ciphertext $\mathsf{ct} := (a,b)$ encrypting the plaintext message $\vec{m}$ under the public key pk, and 4) $\mathsf{sRLWE.Dec}(\mathsf{pp}, \mathsf{sk}, \mathsf{ct})$ which decrypts ct into $\vec{m}$ with secret key sk. A more detailed constructions is given in Appendix A.

In prior works [22,24], with a plaintext space $\mathbb{Z}_q, q = \{0,1\}$, the original RLWE scheme in [22] mainly considers applications that treat plaintexts decrypted into 0 as valid, and 1 as invalid. Thus, with a small error range β, they define the *zero-plaintext wrong-key decryption* property stating that for any honest secret key, it should be hard to decrypt a random ciphertext into 0^ℓ. However, our error bound γ is specially set, s.t. $(\frac{4\gamma+2}{q})^\ell = \mathsf{negl}(\lambda)$, where q is the plaintext modulus and ℓ can be treated as a small constant used as probability amplifier. In this way, we generalize the definition to *wrong-key decryption*. I.e., with overwhelming probability, there exists some slot $i \in [\ell]$ of the decrypted message from a random ciphertext by any honest secret key that *falls out of* the error. This property is crucial in deriving the "binding" property of our commitment construction introduced in § 5 and we formalize it as follows. A formal definition of sRLWE and its proof are deferred to Appendix A.

3.2 Fully Homomorphic Encryption

First constructed by [19], fully homomorphic encryption enables users to perform circuit evaluations on encrypted data, and lots of progress has been made to improve the efficiency. To accommodate large data sets that go through the same circuit, we use the Brakerski/Fan-Vercauteran (BFV) scheme [8,17], and to fit in the setting of multi-party, we adapt the threshold encryption to BFV [5,25]. We briefly recall the definitions of BFV and threshold FHE as follows.

BFV. The BFV scheme consists of algorithms $(\mathsf{GenParams}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$, which is essentially the same as in § 3.1. Different from a normal RLWE scheme, BFV supports Single Instruction Multiple Data (SIMD) operations to be performed on the input vector, i.e., all the following operations are performed homomorphically on each element of the input vector being encrypted.

- $\mathsf{ct} \leftarrow \mathsf{BFV.Eval}(+, \{\mathsf{ct}_i\}_{i\in[m]})$: given a list of ciphertexts, outputs a single ciphertext ct, s.t., $\mathsf{BFV.Dec}(\mathsf{ct}) = \sum_{i\in[m]} \mathsf{BFV.Dec}(\mathsf{ct}_i)$.
- $\mathsf{ct} \leftarrow \mathsf{BFV.Eval}(\cdot, \mathsf{ct}_1, \mathsf{ct}_2)$: given two input ciphertexts, outputs a single ciphertext ct, s.t., $\forall i \in [N], \mathsf{BFV.Dec}(\mathsf{ct})[i] = \mathsf{BFV.Dec}(\mathsf{ct}_1)[i] \cdot \mathsf{BFV.Dec}(\mathsf{ct}_2)[i]$.
- $\mathsf{ct}' \leftarrow \mathsf{BFV.Rotate}(\mathsf{ct}, k)$: given an input ciphertexts, outputs ciphertext ct', s.t., $\forall i \in [N], \mathsf{BFV.Dec}(\mathsf{ct}')[i] = \mathsf{BFV.Dec}(\mathsf{ct})[i+k \mod N]$.

Threshold FHE. A threshold FHE scheme [5, 15] not only allows computation on encrypted data but also threshold encryption. We consider a scheme with the threshold denoted as τ, s.t., for G parties, each party i holding a secret key share $[\![\mathsf{msk}]\!]_i$ and the common master public key mpk (generated from tFHE.Setup), given a ciphertext, only when τ parties gathered together and generate the partial decryptions via tFHE.ParDec w.r.t. their secret key shares, could they recover the plaintext message encrypted under that ciphertext via tFHE.FinDec. We refer readers to the full version [30] for algorithm specification.

A threshold FHE scheme normally needs a trusted setup to distribute the secret key shares, which is also the approach we take in our benchmark. We use tBFV to denote the threshold BFV scheme used in our construction. tBFV consists of all the interfaces above (public-key encryption, BFV SIMD operations, and threshold decryption.)

Helpers algorithms. To facilitate our construction, we introduce the following helper algorithms for tBFV based on homomorphic addition and multiplication operations: 1) $\mathsf{ct}' \leftarrow \mathsf{tBFV.Extract}(\mathsf{mpk}, \mathsf{ct}, i)$ that extracts the i-th slot of ciphertext ct, 2) $\mathsf{ct}' \leftarrow \mathsf{tBFV.Fill}(\mathsf{mpk}, \mathsf{ct})$ that transform a ciphertext of form $\boxed{x\|0^{N-1}}$ into $\boxed{x^N}$, 3) $\mathsf{ct}' \leftarrow \mathsf{tBFV.OblSel}(\mathsf{mpk}, \boxed{x^N}, \{\mathsf{ct}_i\}_{i\in[G]}, q)$ that outputs a new ciphertext $\mathsf{ct}' := \boxed{\mathsf{ct}_x}$, and 4) $\{\boxed{(\vec{x}[i])^N}\}_{i\in[N]} \leftarrow \mathsf{tBFV.OblExp}(\boxed{\vec{x}})$ that expands a ciphertext encrypting a vector $\vec{x}$ of length N into N ciphertexts each encrypting a single entry of $\vec{x}$. The formal specification is given the full version [30] and the last operation BFV.OblExp is given in [1, figure 3].

4 Technical Overview

In this section, we present a bottom-up technical overview, starting with MPRC, then crSSLE in the honest-but-curious setting, and finally Qelect.

Multi-party randomizable commitment (MPRC). For the ease of exposition, we will abstract away the construction of our MPRC scheme Π_{MPRC} and use it in a black-box way for this overview. Recall that MPRC allows users to compute a commitment c to a witness w; anyone can add randomness to c via a *local* procedure to get a randomized commitment $c' = \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(c)$; multiple such c' can be combined into a commitment $c'' = \Pi_{\mathsf{MPRC}}.\mathsf{Combine}(\{c'_i\})$ to the same witness w.

The definition and construction of MPRC will be presented in § 5; we simplified the interface slightly to reduce clutter.

Constant-round single secret leader election. With MPRC as a building block, we now present a constant-round SSLE scheme that achieves *uniquess*, *fairness* and *unpredictability* in an honest-and-curious setting.

Each party k computes a commitment x_k to its witness y_k. The election begins with broadcasting their commitments $\{x_i\}_{i\in[G]}$. Received commitments are ordered lexicographically to form an input list $X = (x_0, \ldots, x_{G-1})$. To draw a random leader, each party k samples and broadcasts $u_k \xleftarrow{\$} \mathbb{Z}_G$. Users would like to use $u := \sum_{i\in[G]} u_i \mod G$ to draw a leader from X. However, they cannot simply output x_u, which immediately leaks the leader identity (whoever broadcasts x_u at the beginning) and breaks *unpredictability*. Therefore, Π_{MPRC} is needed to randomize the commitments in the input list X to get $(x'_0, \cdots, x'_{G-1})$. Second, randomization alone is insufficient because the ordering of commitments remains unchanged. That is, outputting x'_u still immediately reveals the leader's identity. Hence, we build a tFHE circuit to hide u while selecting the u-th commitment from X.

In more detail, each party k proceeds as follows:

- First, party k samples u_k and broadcasts $(\boxed{u_k}, x_k)$ where x_k is an MPRC commitment to its secret witness y_k.
- Then, party k locally evaluates a tFHE circuit that takes as input a list of ciphertexts $\{\boxed{u_1}, \ldots, \boxed{u_G}\}$, and outputs a vector of G "indicator ciphertexts":
$$\mathbb{1}_u := (\boxed{0^N}, \ldots, \underbrace{\boxed{1^N}}_{u\text{-th element}}, \ldots, \boxed{0^N}),$$
where $u = \sum_i u_i \mod G$. Note that u is hidden from parties.
- Next, party k locally randomizes the input list into $X'_k = (x'_{0,k}, \cdots, x'_{G-1,k})$ where $x'_{i,k} = \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(x_i)$. It computes the inner product between $\mathbb{1}_u$ and X'_k and gets $\boxed{x'_{u,k}}$. It broadcasts $\boxed{x'_{u,k}}$.
- After receiving $\{\boxed{x'_{u,i}}\}_{i\in[G]}$ from all parties, party k homomorphically evaluate the $\Pi_{\mathsf{MPRC}}.\mathsf{Combine}$ procedure, which essentially adds them up to get $\boxed{\bar{x}_u}$ where $\bar{x}_u = \sum_{i\in[G]} x'_{u,i}$. That is, $\bar{x}_u$ is a randomized commitment of party u, with randomness from all parties.
- Party k then releases a partial decryption of $\boxed{\bar{x}_u}$, denoted $[\![\bar{x}_u]\!]_k$. After receiving all partial encryptions, it decrypts and gets $\bar{x}_u$. If $\bar{x}_u$ commits to y_k, the party k is elected.
- If elected, party k broadcasts y_k to claim the leadership; All other parties go to step 1 to start the next election (party k will use a new commitment).

Figure 1 depicts the workflow using an example of $G = 4$ parties. To summarize, our scheme consists of three main

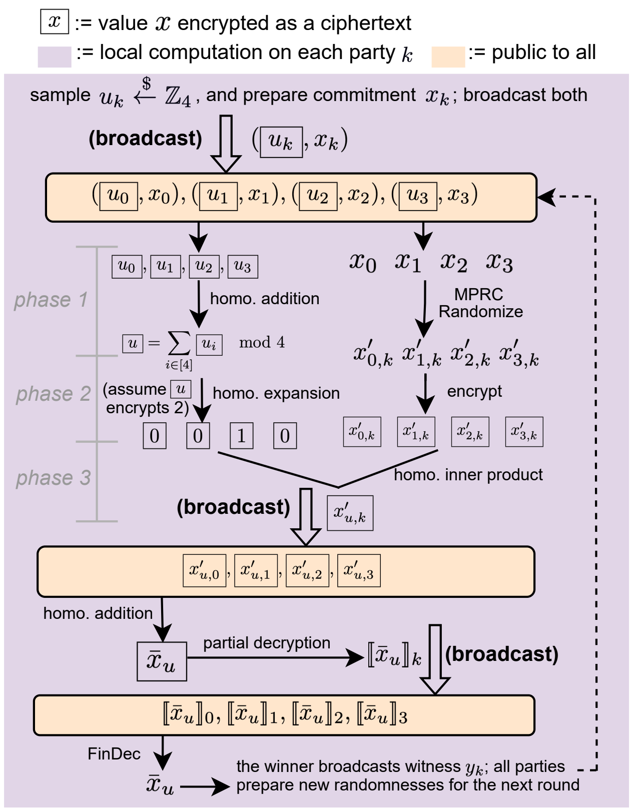
$\boxed{x}$:= value x encrypted as a ciphertext
:= local computation on each party k
:= public to all
sample $u_k \xleftarrow{\$} \mathbb{Z}_4$, and prepare commitment x_k; broadcast both
(broadcast)
$(\boxed{u_k}, x_k)$
$(\boxed{u_0}, x_0), (\boxed{u_1}, x_1), (\boxed{u_2}, x_2), (\boxed{u_3}, x_3)$
phase 1
phase 2
phase 3
homo. addition
$\boxed{u} = \sum_{i\in[4]} \boxed{u_i}$ mod 4
MPRC Randomize
(assume $\boxed{u}$ encrypts 2)
homo. expansion
encrypt
homo. inner product
(broadcast)
homo. addition
partial decryption
(broadcast)
FinDec
the winner broadcasts witness y_k; all parties prepare new randomnesses for the next round

Figure 1: The workflow of crSSLE for party k assuming $G = 4$. Variables with a yellow background are broadcast outputs.

phases: 1) homomorphically generates a ciphertext encrypting the randomness $u \in \mathbb{Z}_G$ that is hidden from all; 2) homomorphically expand a ciphertext encrypting $u \in \mathbb{Z}_G$ into G indicator ciphertexts $\mathbb{1}_u$. 3) homomorphically compute the inner product between the randomized input lists and $\mathbb{1}_u$. We present details on how these three phases are efficiently done with SIMD operations in BFV in § 6.1.

Malicious security and Qelect. A malicious user can break crSSLE in two ways: 1) they can craft messages so the adversary is more likely to win; 2) they can broadcast garbage to disrupt the election. We handle them differently as follows.

In the first case, an adversary can break *fairness* and *unpredictability*. For instance, a malicious party k can wait to receive others' $\boxed{u_i}$ and send $\boxed{k} - \sum_{i\neq k} \boxed{u_i}$; Similar attacks are possible in the second and third broadcasts. A standard fix is *commit-and-reveal*: add a round of broadcast of the hash of $\boxed{u_i}$ before broadcasting $\boxed{u_i}$. This does not increase the round complexity (still constant), but it adds three round trips.

We present an optimization that avoids two of the three round trips, assuming G is reasonably large (e.g., $G \geq 128$) and there is an honest majority among users. The idea is to use local randomness to choose a subset of received broadcast messages for aggregation. We illustrate this idea on the aggregation of $\boxed{u_i}$, and the same trick could also be applied to $\boxed{x'_{u,i}}$. As in crSSLE, all parties broadcast $\{\boxed{u_i}\}_{i\in[G]}$. Instead of aggregating all of them to get u, each party locally computes $r_s \leftarrow H(\{\boxed{u_i}\}_{i\in[G]})$ and use r_s to choose a random subset $\mathcal{S}$ for aggregation, i.e., $u = \sum_{i\in\mathcal{S}} u_i$. This defeats the above attack because the adversary cannot predict the subset with non-negligible probability under proper choice of parameters. However, this method does not apply to the last broadcast since we need all G partial decryptions to recover the plaintext. One additional round of commit-and-reveal is necessary.

Another type of attack involves broadcasting invalid (garbage) ciphertexts to contaminate the final output. To prevent this, a naive approach is to let all parties generate ZKP of 1) well-formedness of the initial ciphertext $\boxed{u_i}$, 2) honest evaluation of intermediary commitment $\boxed{x'_{u,i}}$ and 3) honest partial decryption $[\![\bar{x}_u]\!]$. Previous post-quantum secure SSLE works [4, 6, 18] all follow this approach, which leads to impractical performance.

In Qelect, we observe that the blockchain protocols that will use SSLE typically have built-in ways to penalize misbehavior after the fact (i.e., slashing [9], which underpins the security of many production systems, including Ethereum), so we present a protocol to identify senders of malformed messages. Specifically, if no user claims the leadership, the protocol launches the following *retroactive detection phase*: 1) parties are asked to broadcast the randomnesses used to compute $\boxed{u_i}$ and $\boxed{x'_{u,i}}$ in the previous election; everyone can verify the correctness of received $\boxed{u_i}$ and $\boxed{x'_{u,i}}$ by reconstructing them from other messages and the randomness. The senders of inconsistent messages are slashed. 2) parties are asked to generate a relatively lightweight zero-knowledge proof for the correctness of their partial decryption $[\![\bar{x}_u]\!]_i$ and broadcast it to all. Parties who cannot generate a valid proof are slashed. Note that this protocol is only revoked when no leader is elected (i.e., no one is able to open the final commitment $\bar{x}_u$), so it is not on the critical path of Qelect.

Summary. The modifications to crSSLE can be summarized as follows, assuming a reasonably large party size ($G \geq 128$) and a malicious adversary corrupting less than $G/2$ parties:

- After the first broadcast, all parties use $H(\{\boxed{u_i}\})$ to select a subset to aggregate and get $\boxed{u}$.
- Similarly, after the second broadcast, parties apply the subset-sampling technique using $H\{\boxed{x'_{u,i}}\}$ to compute the final commitment $\boxed{\bar{x}_u}$.
- Before broadcasting partial decryption, every party publishes $H([\![\bar{x}_u]\!]_k)$.
- If the election gets aborted due to no one can claim the leadership, invoke the retroactive detection phase.

A complete description of the Qelect protocol and discussions on practical concerns will be presented in § 6.2.

5 Multi-party Randomizable Commitment

In this section, we present a primitive called multi-party randomizable commitment (MPRC), which serves as the key building block in our post-quantum constant-round single-leader election schemes.

At a high level, the goal is to allow a set of parties to randomize a given commitment *in parallel* without network communication. Specifically, upon receiving a commitment c from a sender, each other party k can randomize c independently and produce a randomized commitment c'_k, all of which can then be aggregated to form a final output c'' that can be opened with the original witness of the sender. At the same time, given two original commitments c_0, c_1 and a final commitment c'' of c_b, $b \xleftarrow{\$} \{0,1\}$, a PPT adversary cannot decide b, as long as one of the parties is honest.

We formalize our goal with the following definition and present a post-quantum secure construction based on sRLWE defined in § 3.1. Notice that in [4], the authors propose a DDH-based commitment scheme, which is later abstracted as a primitive called *re-randomization commitment* (RRC) with a post-quantum secure construction. We provide a detailed comparison between our primitive and RRC in Remark 5.1.

Definition 5.1 (Multi-party Randomizable Commitment (MPRC))**.** For G parties, an MPRC scheme Π_{MPRC} consisting of algorithms: $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Randomize}, \mathsf{Combine}, \mathsf{Verify})$ is defined as follows.

- $\mathsf{pp}_C \xleftarrow{\$} \mathsf{Gen}(1^\lambda, \cdot))$: generate the public parameter for the commitment scheme based on the security parameter.
- $(c, w) \xleftarrow{\$} \mathsf{Commit}(\mathsf{pp}_C, m)$: given the public parameter pp_C and a message m, output a commitment c together with its corresponding witness w.
- $c' \xleftarrow{\$} \mathsf{Randomize}(\mathsf{pp}_C, c)$: given a commitment c, provide a randomized new commitment c'.
- $c'' \leftarrow \mathsf{Combine}(\mathsf{pp}_C, \{c'_i\})$: given a list of $\leq G$ randomized commitments, deterministically aggregate them into a new randomized commitment c''.
- $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{pp}_C, c'', w, m)$: given a randomized commitment, a secret witness and a message m, output 1 if c'' is indeed a commitment of m w.r.t. witness w, and 0 otherwise.

Denote the message space as $\mathcal{M}$, and the witness space as $\mathcal{W}$, an MPRC scheme also needs to satisfy the following properties:

- *Multi-party Binding:* for any PPT algorithm $\mathcal{A}$, let $\mathsf{pp}_C \xleftarrow{\$} \mathsf{Gen}(1^\lambda, \cdot)$, for any message $m \in \mathcal{M}$, for any $(c, w) \xleftarrow{\$} \mathsf{Commit}(\mathsf{pp}_C, m)$, denote the set $\mathcal{C} = \{c' \mid c' \xleftarrow{\$} \mathsf{Randomize}(\mathsf{pp}_C, c)\}$. For any set $\mathcal{C}_G \subseteq \mathcal{C}$ s.t. $|\mathcal{C}_G| \leq G$, let $c'' \leftarrow \mathsf{Combine}(\mathsf{pp}_C, \mathcal{C}_G)$, let $(w', m' \neq \perp) \leftarrow \mathcal{A}(c, m, c'')$, $\Pr[\mathsf{Verify}(\mathsf{pp}_C, c'', w, m) = \mathsf{Verify}(\mathsf{pp}_C, c'', w', m')] = \mathsf{negl}(\lambda)$.
- *Unlinkability:* for any PPT adversary $\mathcal{A}_0, \mathcal{A}_1$, let $\mathsf{pp}_C \xleftarrow{\$} \mathsf{Gen}(1^\lambda, \cdot)$, for any message $m_0, m_1 \in \mathcal{M}$, for any $(c_0, w_0) \xleftarrow{\$} \mathsf{Commit}(\mathsf{pp}_C, m_0), (c_1, w_1) \xleftarrow{\$} \mathsf{Commit}(\mathsf{pp}_C, m_1)$, denote the set $\mathcal{C}_0 = \{c'_0 \mid c'_0 \xleftarrow{\$} \mathsf{Randomize}(\mathsf{pp}_C, c_0)\}$ and $\mathcal{C}_1 = \{c'_1 \mid c'_1 \xleftarrow{\$} \mathsf{Randomize}(\mathsf{pp}_C, c_1)\}$. For any $\mathcal{C}'_{0,G} \subseteq \mathcal{C}_{0,G} \subseteq \mathcal{C}_0, \mathcal{C}'_{1,G} \subseteq \mathcal{C}_{1,G} \subseteq \mathcal{C}_1$, s.t. $|\mathcal{C}_{0,G}|, |\mathcal{C}_{1,G}| \leq G$, let $(\mathsf{st}, \mathcal{C}''_{0,G}, \mathcal{C}''_{1,G}) \xleftarrow{\$} \mathcal{A}_0(c_0, c_1, \mathcal{C}'_{0,G}, \mathcal{C}'_{1,G})$, where $|\mathcal{C}''_{0,G}| = |\mathcal{C}'_{0,G}|, |\mathcal{C}''_{1,G}| = |\mathcal{C}'_{1,G}|$ serving as the maliciously crafted randomized commitments outputted by the corrupted parties replacing the subset $\mathcal{C}'_{0,G}, \mathcal{C}'_{1,G}$. Let $b \xleftarrow{\$} \{0,1\}$, $c''_b \leftarrow \mathsf{Combine}(\mathsf{pp}_C, \mathcal{C}''_{b,G} \cup \mathcal{C}_{b,G} \backslash \mathcal{C}'_{b,G})$, $\Pr[\mathcal{A}(\mathsf{st}, c''_b) = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

Construction. We first observe that with a public-key encryption scheme based on sRLWE (§ 3.1), for a key pair $(\mathsf{pk}, \mathsf{sk})$, encrypting some message under pk "commits" to the corresponding sk since only the correct secret key can decrypt to a valid message based on the *wrong-key decryption* property. One caveat is that encryption of $m = \vec{0}$ is not a binding commitment to the secret key. Given public key $\mathsf{pk} := (\alpha, \beta = \alpha\mathsf{sk} + e)$ with some noise e and ciphertext encryption m, one could easily craft different sk' such that $\alpha\mathsf{sk}' - \beta \approx 0$, breaking the *multi-party binding* property. This is easily fixed, by using the random coin r as the witness, which is used to generate the secret key sk in $\mathsf{sRLWE.KeyGen}$.

MPRC is built on this idea. We present a construction based on sRLWE in Algorithm 1. To commit to a message, $\Pi_{\mathsf{MPRC}}.\mathsf{Commit}$ outputs a tuple $(\mathsf{pk}, \mathsf{ct})$ where $\mathsf{ct} \xleftarrow{\$} \mathsf{sRLWE.Enc}(\cdot, \mathsf{pk}, \vec{m})$. To randomize a commitment $(\mathsf{pk}, \mathsf{ct})$, $\Pi_{\mathsf{MPRC}}.\mathsf{Randomize}$ constructs a new ciphertext $\mathsf{ct}' \xleftarrow{\$} \mathsf{sRLWE.Enc}(\cdot, \mathsf{pk}, \vec{0})$ encrypting zeros, and return $\mathsf{ct}'' = \mathsf{ct} + \mathsf{ct}'$ where ct' would act as a random mask for ct. Therefore, since ct' is generated based on local private random coin and is pseudorandom, any PPT adversary would not be able to link ct'' to ct or pk based on the key privacy property of sRLWE, thus achieves the *unlinkability* property of MPRC. To verify, one simply re-generates its secret key sk with the witness $w := r$ and decrypts the ciphertext ct'' with sk.

Theorem 5.1. The scheme Π_{MPRC} specified in Algorithm 1 is a multi-party randomizable commitment scheme defined in Definition 5.1.

Parameter choice. Concretely, for sRLWE scheme with the ciphertext space $q = 65537$, $\ell = 256$, $n = 1024$, we could set $\gamma = 10000$ and thus $p = \{0,1\}$, i.e., we could allow decryption results to be either 0 or 1, while anything out of the error range will be output as $\perp$. Since a sRLWE ciphertext essentially packs ℓ bits of message, each party would be able to convey a message from $\mathbb{Z}_{2^\ell}$.

Remark 5.1. In [6], the authors introduce a primitive denoted as *re-randomizable commitment* (RRC), generalized from the

Algorithm 1 Multi-party Randomizable Commitment from RLWE

```
1:  procedure $\Pi_{\mathsf{MPRC}}.\mathsf{Gen}(\mathsf{pp} = (1^\lambda, G, \ell, q, \sigma, h))$
2:      $\mathsf{pp}_{\mathsf{rlwe}} \leftarrow \mathsf{sRLWE.GenParams}(1^\lambda, \ell, q, \sigma, h)$
3:      if $\ell(1 - \mathsf{erf}(\frac{\gamma}{\sqrt{4G(2h+1)\sigma}})) = \mathsf{negl}(\lambda)$ then
4:          return $\mathsf{pp}_C := \mathsf{pp}_{\mathsf{rlwe}}$
5:      else
6:          return $\perp$    $\triangleright$ all algorithms return $\perp$ if $\mathsf{pp}_C = \perp$.
7:  procedure $\Pi_{\mathsf{MPRC}}.\mathsf{Commit}(\mathsf{pp}_C, \vec{m})$
8:      Choose a random coin $r$
9:      $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{sRLWE.KeyGen}(\mathsf{pp}_C; r)$
10:     $\mathsf{ct} \xleftarrow{\$} \mathsf{sRLWE.Enc}(\mathsf{pp}_C, \mathsf{pk}, \vec{m})$
11:     return $(c := (\mathsf{pk}, \mathsf{ct}), w := r)$
12: procedure $\Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}_C, c = (\mathsf{pk}, \mathsf{ct}))$
13:     $\mathsf{ct}' \xleftarrow{\$} \mathsf{sRLWE.Enc}(\mathsf{pp}_C, \mathsf{pk}, \vec{0}) \in \mathcal{R}_q \times \mathcal{R}_q$
14:     return $\mathsf{ct}'' := \mathsf{ct} + \mathsf{ct}'$
15: procedure $\Pi_{\mathsf{MPRC}}.\mathsf{Combine}(\mathsf{pp}_C, \{c'_i\}_{i \in [G]})$
16:     return $c'' := \sum_{i \in [G]} c'_i \in \mathcal{R}_q \times \mathcal{R}_q$ (normal addition between ring elements)
17: procedure $\Pi_{\mathsf{MPRC}}.\mathsf{Verify}(\mathsf{pp}_C, c'', w, \vec{m})$
18:     $\mathsf{sk} \leftarrow_w \mathcal{D}$ (same as generated in sRLWE.KeyGen)
19:     Let $\vec{m}''[i] = \vec{m}'[i] \cdot G \mod 2$, where $\vec{m}' \leftarrow \mathsf{sRLWE.Dec}(\mathsf{pp}_C, \mathsf{sk}, c'')$
20:     return 1 if $\forall i \in [\ell], \vec{m}''[i] = \vec{m}[i]$ , and 0 o.w.
```

DDH-based commitment scheme proposed in [4]. Briefly, our commitment scheme differs from them in the following aspects. 1) For G parties to collectively randomize a commitment, our protocol takes constant communication rounds. In contrast, the protocol in [6] needs $O(G)$ rounds, which is impractical in the WAN setting when G is large. 2) Our commitment size is at least $\Omega(\log q)$ smaller than the one in [6], where q is the underlying plaintext modulus. 3) The noise of the final randomized commitment grows logarithmically with the party size G, while in [6], it grows exponentially with G.

6 Practical Post-Quantum SSLE from MPRC

With MPRC, we can build a *constant-round single secret leader election* (crSSLE). We first present a definition for crSSLE based on [4] and all its follow-ups [3, 6, 11, 12, 16, 18] with explicit interfaces that restricts the communication round to be constant. We then provide a construction for crSSLE in § 6.1. Finally, in § 6.2.2, we present Qelect, our SSLE protocol built on crSSLE with optimizations and retroactive detection steps to handle malicious adversaries.

Definition 6.1 (Constant-round Single Secret Election (crSSLE))**.** Suppose there are G nodes with up to τ of them being malicious. An crSSLE protocol $\Pi = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{ParElect}, \mathsf{Elect}, \mathsf{Combine}, \mathsf{Verify})$ is defined as follows:

- $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i \in [G]}) \leftarrow \mathsf{crSSLE.Setup}(\mathsf{pp} = (G, \cdot), 1^\lambda)$: take as input the public parameter pp and security parameter 1^λ; output a master public key mpk, and distribute $[\![\mathsf{msk}]\!]_i$ to each single user i secretly.
- $(x_k, y_k, \mathsf{pl}_k) \xleftarrow{\$} \mathsf{crSSLE.Gen}(\mathsf{pp}, \mathsf{mpk}, 1^\lambda, m_k)$: for each party k, take in public parameter pp, the master public key mpk, the security parameter 1^λ, and a user-specific message m_k; generate the commitment x_k serving as the input to the election protocol, the witness y_k, and pl_k that serves as the the election randomness.
- $\mathsf{ct}_{x'_k} \xleftarrow{\$} \mathsf{crSSLE.ParElect}(\mathsf{mpk}, \{(x_i, \mathsf{pl}_i)\}_{i \in [G]})$: for each party k, take in a list of variables $\{(x_i, \mathsf{pl}_i)\}$; output a single share $\mathsf{ct}_{x'_k}$ which is elected based on public randomnesses $\{\mathsf{pl}_i\}$ and locally randomized based on some private coins.
- $[\![\bar{x}]\!]_k \xleftarrow{\$} \mathsf{crSSLE.Elect}(\{\mathsf{ct}_{x'_i}\}_{i \in [G]}, [\![\mathsf{msk}]\!]_k)$: for each party k, take in a list of shares $\{\mathsf{ct}_{x'_i}\}$ and a secret key share $[\![\mathsf{msk}]\!]_k$; output a single share $[\![\bar{x}]\!]_k$.
- $\bar{x} \leftarrow \mathsf{crSSLE.Combine}(\{[\![\bar{x}]\!]_i\}_{i \in [G]})$: take in a list of shares $\{[\![\bar{x}]\!]_i\}$; output the final randomized election result $\bar{x}$.
- $\{0, 1\} \leftarrow \mathsf{crSSLE.Verify}(\mathsf{pp}, \bar{x}, y, m)$: take in public parameter pp, a randomized commitment $\bar{x}$, a witness y and a message m; output 1 if y opens $\bar{x}$ into message m; and 0 otherwise.

A crSSLE also needs to satisfy the following properties. For any PPT adversary $\mathcal{A}$, let $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i \in [G]}) \leftarrow \mathsf{crSSLE.Setup}(\mathsf{pp} = (G, \cdot), 1^\lambda)$. The adversary $\mathcal{A}$ chooses a corrupted set $\mathcal{W}_{\mathsf{cor}}, |\mathcal{W}_{\mathsf{cor}}| \leq \tau$, and then plays the role of the parties inside $\mathcal{W}_{\mathsf{cor}}$ while holding all secret values of parties in $\mathcal{W}_{\mathsf{cor}}$. Let $\{(x_i, y_i, \mathsf{pl}_i)\}_{i \in \mathcal{W}_{\mathsf{cor}}}$ be the public and private outputs generated by $\mathcal{A}$, and $(x_i, y_i, \mathsf{pl}_i) \xleftarrow{\$} \mathsf{crSSLE.Gen}(\mathsf{pp}, \mathsf{mpk}, 1^\lambda, m_i)$, for $i \in [G] \backslash \mathcal{W}_{\mathsf{cor}}$ honestly generated by other parties. For parties in $[G] \backslash \mathcal{W}_{\mathsf{cor}}$, execute crSSLE.ParElect honestly to get $\{\mathsf{ct}_{x'_i}\}_{i \in [G] \backslash \mathcal{W}_{\mathsf{cor}}}$. For parties in $\mathcal{W}_{\mathsf{cor}}$, let $\mathcal{A}$ publish $\{\mathsf{ct}_{x'_i}\}_{i \in \mathcal{W}_{\mathsf{cor}}}$. With all $\{\mathsf{ct}_{x'_i}\}_{i \in [G]}$, parties in $[G] \backslash \mathcal{W}_{\mathsf{cor}}$ execute crSSLE.Elect to get $[\![\bar{x}]\!]_{i \in [G] \backslash \mathcal{W}_{\mathsf{cor}}}$, and $\mathcal{A}$ publishes $[\![\bar{x}]\!]_{i \in \mathcal{W}_{\mathsf{cor}}}$ for the corrupted parties. With all $[\![\bar{x}]\!]_{i \in [G]}$ published, denote $\bar{x} \leftarrow \mathsf{crSSLE.Combine}(\{[\![\bar{x}]\!]_i\}_{i \in [G]})$ be the final election output. Denote all transcripts as st. A crSSLE scheme also needs to satisfy the following properties:

- *Uniqueness:* There exists some j, such that: $\Pr[\mathsf{crSSLE.}(\mathsf{Verify}(\mathsf{pp}, \bar{x}, y_j, m_j) = 1] \geq 1 - \mathsf{negl}(\lambda)$, and for $j' \neq j$, $\Pr[\mathsf{crSSLE.Verify}(\mathsf{pp}, \bar{x}, y_{j'}, m_{j'} = 0)] \geq 1 - \mathsf{negl}(\lambda)$.
- *Unpredictability:* Given st and all secrets holding by $\mathcal{W}_{\mathsf{cor}}$, $\mathcal{A}$ outputs an index $k \in [G] \backslash \mathcal{W}_{\mathsf{cor}}$, it holds that $\Pr[\mathsf{crSSLE.Verify}(\mathsf{pp}, \bar{x}, y_k, m_k) = 1] \leq \frac{1}{G - |\mathcal{W}_{\mathsf{cor}}|} + \mathsf{negl}(\lambda)$.
- *Fairness:* For every $i \in [G] \backslash \mathcal{W}_{\mathsf{cor}}$, it holds that $\Pr[\mathsf{crSSLE.Verify}(\mathsf{pp}, \bar{x}, y_i, m_i) = 1] \leq \frac{1}{G - |\mathcal{W}_{\mathsf{cor}}|} + \mathsf{negl}(\lambda)$, where the randomness is taken over $\{\mathsf{pl}_i\}_{i \in [G]}$.

In § 6.1, we show a plain scheme achieving the above properties under an honest-but-curious model. The scheme could easily achieve the same security guarantee under a fully malicious model by attaching general ZKP to all transcripts. To avoid the overhead of ZKP, we patch our scheme in a much lighter way to guarantee *unpredictability* and *fairness* with a malicious adversary in § 6.2.1. And then in § 6.2.2, we present our main SSLE protocol Qelect with optimizations, which achieves a relaxed version of the *uniqueness* property under a malicious setting. Looking ahead, we claim that an adversary would either break *uniqueness* by preventing progress or be slashed in the next election round.

6.1 crSSLE Construction

This section dives into the details of the three phases in Fig. 1. Since many steps of the protocol perform the same set of operations on multiple different values, a key performance optimization is to pack a whole vector in one ciphertext so we enjoy the speedup from SIMD operations available in BFV.

Homomorphic randomness generation. The task of this phase is to generate a ciphertext encrypting randomness u that is hidden from all. Specifically, a single tBFV ciphertext with ring dimension N can fit N values in $\mathbb{Z}_G$, thus the ciphertext to generate encrypts $u\|0^{N-1}$. The most intuitive way to do so is to let each party k sample $u_k \xleftarrow{\$} \mathbb{Z}_G$ and broadcast $\boxed{u_k\|0^{N-1}}$ together with their initial commitments x_k. (For simplicity, let each party k commit to $m_k := 0^N$ under witness y_k.) All parties can then add up all $\{\boxed{u_i\|0^{N-1}}\}_{i\in[G]}$ and get a single ciphertext $\boxed{u\|0^{N-1}}$, where $u := \sum_{i\in[G]} u_i \mod G$. [2]

To optimize, we use all N slots to generate randomness for many rounds in batches. Each party k samples $\vec{\mathbf{u}}_k \xleftarrow{\$} \mathbb{Z}_G^N$ and generates $\boxed{\vec{\mathbf{u}}_k}$. After aggregation, the resultant ciphertext is $\boxed{\vec{\mathbf{u}}}$ where $\vec{\mathbf{u}}[i] = \sum_{j\in[G]} \vec{\mathbf{u}}_j[i] \mod G$. During the l-th round of the protocol, the parties extract the l-th slot of $\vec{\mathbf{u}}$ into $\boxed{\vec{\mathbf{u}}[l]\|0^{N-1}}$ via tBFV.Extract defined in § 3.2. The communication cost of publishing $\boxed{\vec{\mathbf{u}}_k}$ and the local computation of aggregating them could be amortized across multiple (at most N) rounds. The only overhead introduced is by invoking tBFV.Extract, which is relatively negligible.

Homomorphic expansion. Given $\boxed{u\|0^{N-1}}$ the output from the previous phase, this step expand it into $\mathbb{1}_u := (\boxed{0^N},\ldots,\underbrace{\boxed{1^N}}_{u\text{-th element}},\ldots,\boxed{0^N})$. We first transform $\boxed{u\|0^{N-1}}$ into a ciphertext $\boxed{\vec{e_u}}$ encrypting the one-hot vector $\vec{e_u} := (0,\ldots,\underbrace{1}_{u\text{-th elem.}},\ldots,0)$, then expand $\boxed{\vec{e_u}}$ into $\mathbb{1}_u$. The latter transformation can be done with tBFV.OblExp given in [1, figure 3]. We thus show how to achieve the former.

For simplicity, we assume that the ring dimension N is greater than G, which is ideally the largest possible value for u. We first use tBFV.Fill defined in § 3.2 to "fill" $\boxed{u\|0^{N-1}}$ and get $\boxed{u^N}$. We then compute a ciphertext ct_v encrypting the vector $\{0,1,\ldots,N\}$ and compute $\mathsf{ct}' \leftarrow \mathsf{tBFV.Eval}(+,\mathsf{ct}_v,-\boxed{u^N})$. ct' encrypts $i-u$ in slot i; in particular, it encrypts 0 in slot u. With the plaintext modulus for tBFV scheme to be of a prime q, based on Fermat's little theorem, raising ct' to the power of $q-1$ would yield a ciphertext $\boxed{(1,\ldots,\underbrace{0}_{u\text{-th elem.}},\ldots,1)}$. To get what we want (i.e., $\boxed{\vec{e_u}}$), we homomorphically subtract 1 from all slots from the previous ciphertext. For $G \geq N$, we repeat the aforementioned steps $\lceil\frac{G}{N}\rceil$ times, each with $\mathsf{ct}_{v,i}$ encrypting $(iN, iN+1,\ldots,(i+1)N-1)$, for $i \in [\lceil\frac{G}{N}\rceil]$.

Asymptotically, this construction takes $\log N$ rotation and addition operations to get the ciphertext $\boxed{u^N}$ via tBFV.Fill, and $\log(q-1)$ levels to raise it slot-wisely to the power of $q-1$. Once again, by evaluating tBFV.OblExp adapted from [1, figure 3], we can expand $\boxed{\vec{e_u}}$ into $\mathbb{1}_u$, concluding this phase.

Homomorphic randomized commitments aggregation. After executing the above two steps, each party should hold $\vec{\mathbf{u}}_b$. By invoking $\Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\cdot)$ with private coins on the initial input list $\{x_i\}_{i\in[G]}$, each party k would derive its own randomized commitments $\{x'_{i,k}\}_{i\in[G]}$, which are essentially sRLWE ciphertext $\in \mathcal{R}_q \times \mathcal{R}_q$ that can be represented as vectors of its coefficients. It then encrypts commitments into G corresponding tBFV ciphertexts $\{\boxed{x'_{1,k}},\ldots,\boxed{x'_{G,k}}\}$ and perform inner product with $\mathbb{1}_u$ to derive a fresh ciphertext $\boxed{x'_{u,k}}$, and broadcasts it. After receiving the randomizations from all parties $\{\boxed{x'_{u,1}},\ldots,\boxed{x'_{u,G}}\}$, all parties sum them up into a single ciphertext $\boxed{\bar{x}_u}$, where $\bar{x}_u = \sum_{i\in[G]} x'_{u,i}$, and release the partial decryption as $[\![\bar{x}_u]\!]_k$. After combining all partial decryptions, each party k recovers the underlying $\bar{x}_u$ and learns if itself is elected by checking $\Pi_{\mathsf{MPRC}}.\mathsf{Verify}(\cdot,\bar{x}_u,y_k,m_k) = 1$.

Summary. To recap, the l-th instance of election involves three rounds of broadcasting. In the first round, each party k publishes its initial commitments x_k and a ciphertext $\boxed{\vec{\mathbf{u}}_k}$, followed by the local computation to first homomorphically derive the randomness $\vec{\mathbf{u}} = \sum_{i\in[G]} \vec{\mathbf{u}}_i$ and then obliviously select the encrypted randomized commitment for x_u where $u = \vec{\mathbf{u}}[l]$, denoted as $\boxed{x'_{u,k}}$. In the second round, each party k publishes $\boxed{x'_{u,k}}$ and homomorphically adds up $\{\boxed{x'_{u,i}}\}_{i\in[G]}$ into $\boxed{\bar{x}_u}$. In the last round, each party uses its secret key share for the tBFV scheme to release a partial decryption of $\boxed{\bar{x}_u}$, denoted as $[\![\bar{x}_u]\!]_k$, which allows them to recover the final randomized commitments of party u.

Under an honest-but-curious setting, the *uniqueness* property mainly relies on the *multi-party binding* property of the underlying MPRC scheme. I.e., as long as the incurred noise by homomorphically aggregating all randomized commitment shares $\{[\![\bar{x}_u]\!]_i\}_{i\in[G]}$ does not overflow, the final randomized commitment could still be opened by its corresponding wit-

[2]We assume that the plaintext modulus $q = G$ which is generally not the case since q is a prime and we normally consider G to be a power-of-two in our benchmarks. We discuss this issue in Appendix D.

Algorithm 2 crSSLE Construction

```
1: procedure crSSLE.Setup($\mathsf{pp} = (G, \tau, q, \sigma, h), 1^\lambda$)
2:     $\mathsf{pp}_C \leftarrow \Pi_{\mathsf{MPRC}}.\mathsf{Gen}(1^\lambda, G, \ell, q, \sigma, h)$
3:     $(\mathsf{pp}_{\mathsf{tBFV}}, \mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}) \leftarrow \mathsf{tBFV.GenParams}(1^\lambda, G, \tau)$    ▷ parse $\mathsf{pp}_{\mathsf{tBFV}} = (t, N, \cdot)$, which contains all required parameters for threshold BFV FHE scheme
4:     Let $\mathsf{rm} := q \mod G$, choose $\bar{t}$ s.t. $(\frac{\mathsf{rm}}{q})^{\bar{t}} = \mathsf{negl}(\lambda)$
5:     Append $(\mathsf{pp}_C, \mathsf{pp}_{\mathsf{tBFV}})$ to pp
6:     return $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]})$
7: procedure crSSLE.Gen($\mathsf{pp} = (\mathsf{pp}_C, \mathsf{pp}_{\mathsf{tBFV}}, \cdot), \mathsf{mpk}, 1^\lambda, m_k$)    ▷ we treat $m_k$ as all zeros for crSSLE scheme.
8:     $\vec{\mathbf{u}}_k \xleftarrow{\$} \mathbb{Z}_q^N$
9:     $\boxed{\vec{\mathbf{u}}_k} \xleftarrow{\$} \mathsf{tBFV.Enc}(\mathsf{mpk}, \vec{\mathbf{u}}_k)$
10:    $(x_k, y_k) \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Commit}(\mathsf{pp}_C, m_k)$
11:    return $(x_k, y_k, \boxed{\vec{\mathbf{u}}_k})$
12: procedure crSSLE.ParElect($\mathsf{mpk}, \{(x_i, \boxed{\vec{\mathbf{u}}_i})\}_{i\in[G]}$)
13:    $\mathsf{ct}_{\vec{\mathbf{u}}}, \leftarrow \mathsf{tBFV.Eval}(+, \{\boxed{\vec{\mathbf{u}}_i}\}_{i\in[G]})$
14:    $\mathsf{ct}_u \leftarrow \mathsf{tBFV.Extract}(\mathsf{mpk}, \mathsf{ct}_{\vec{\mathbf{u}}})$    ▷ extract the $j$-th slot out of all $N$ slots in the $j$-th round, if amortized across multiple rounds.
15:    $\boxed{u^N} \leftarrow \mathsf{tBFV.Fill}(\mathsf{mpk}, \mathsf{ct}_u)$    ▷ where $u = \sum_{i\in[G]} \vec{\mathbf{u}}[j]$
16:    $x'_i \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}_C, x_i), i \in [G]$
17:    $\boxed{x'_i} \xleftarrow{\$} \mathsf{tBFV.Enc}(\mathsf{mpk}, x'_i), i \in [G]$
18:    $\boxed{x'_{u,k}} \leftarrow \mathsf{tBFV.OblSel}(\mathsf{mpk}, \boxed{u^N}, \{\boxed{x'_i}\}_{i\in[G]}, q)$
19:    return $\boxed{x'_{u,k}}$
20: procedure crSSLE.Elect($\{\boxed{x'_{u,i}}\}_{i\in[G]}, [\![\mathsf{msk}]\!]_k$)
21:    $\boxed{\bar{x}_u} \leftarrow \mathsf{tBFV.Eval}(+, \{\boxed{x'_{u,i}}\}_{i\in[G]})$
22:    $[\![\bar{x}_u]\!]_k := \mathsf{tBFV.ParDec}(\boxed{\bar{x}_u}, [\![\mathsf{msk}]\!]_k)$
23:    return $[\![\bar{x}_u]\!]_k$
24: procedure crSSLE.Combine($\{[\![\bar{x}_u]\!]_i\}_{i\in[G]}$)
25:    $\bar{x}_u \leftarrow \mathsf{tBFV.FinDec}(\{[\![\bar{x}_u]\!]_i\}_{i\in[G]})$,
26:    return $\bar{x}_u$
27: procedure crSSLE.Verify($\mathsf{pp}, \bar{x}_u, y, m$)
28:    return $\Pi_{\mathsf{MPRC}}.\mathsf{Verify}(\mathsf{pp}, \bar{x}_u, y, m)$
```

ness. The *unpredictability* property could be shown in a hybrid manner based on the semantic security of tFHE and the *unlinkability* of the embedded MPRC scheme. The *fairness* property is simply achieved as long as there exists one honest party i that supplies a random election randomness pl_i. A formal proof is deferred to Appendix C.1.

Theorem 6.1. Given G parties, for any PPT *honest-but-curious* adversary corrupting $\leq G-1$ parties, the crSSLE construction given in Algorithm 2 is a constant-round single secret leader election defined in Definition 6.1.

6.2 Qelect

Qelect augments crSSLE with malicious security. This section describes the changes made to crSSLE to obtain Qelect.

A malicious adversary could deviate in two ways. First, it could craft messages (after seeing all others' messages) so that the final aggregated result favors the adversary. We call this kind of attack *aggressive attack* and will present solutions in § 6.2.1. Second, the adversary can send garbage or equivocate to prevent progress. We call this kind of attack *passive attack* and address it in § 6.2.2 with a retroactive detection phase.

6.2.1 Preventing Aggressive Attacks

Recall that in Algorithm 2, there are three rounds of broadcast: 1) the ciphertexts $\{\boxed{\vec{\mathbf{u}}_i}\}_{i\in[G]}$ encrypting some randomnesses, 2) the ciphertexts $\{\boxed{x'_{u,i}}\}_{i\in[G]}$ encrypting each party's randomized commitment for the leader, 3) the partial decryption shares $\{[\![\bar{x}_u]\!]_i\}_{i\in[G]}$. After receiving those messages, the local computation is to only homomorphic add them up correspondingly on line 13, 21, and 25. Since those aggregations are all deterministic, the adversary could fix the aggregation result in favor of itself and craft its broadcast message by reversing the procedure after seeing all others' outputs. For instance, a malicious party k can wait to receive others' $\boxed{\vec{\mathbf{u}}_i}$ and send $\boxed{k^N} - \sum_{i\neq k} \boxed{\vec{\mathbf{u}}_i}$; Similar attacks are possible in the second and third broadcasts. This kind of misbehavior breaks the *fairness* and *unpredictability* of SSLE.

A standard solution would be *commit-and-reveal*: Before publishing a message, each party first broadcasts the hash of it as a "commitment" to the message. They will not broadcast the actual message before receiving all commitments. This commitment can be built from collision-resistant hash functions. In this way, the adversary is forced to generate its messages independently without seeing others'.

Specifically, given a collision resistance hash function H, in the first round of broadcast of crSSLE, each party publishes $H(\boxed{\vec{\mathbf{u}}_\mathbf{i}})$ before releasing $\boxed{\vec{\mathbf{u}}_\mathbf{i}}$ to guarantee that the aggregated result on line 13 in Algorithm 2 is still random. Similarly, in the second round of broadcast, each party publishes $H(\boxed{x'_{u,i}})$ before releasing $\boxed{x'_{u,i}}$ so that the final commitment ciphertexts on line 21 are random. Finally, before broadcasting the partial decryption result $[\![\bar{x}_u]\!]$, each party also publishes its hash value $H([\![\bar{x}_u]\!])$ as the "commitment". Hence, the distribution of the decryption results $\bar{x}_u$ (line 25) would be uniformly random from the plaintext space, where the randomness is taken over $\{\boxed{\vec{\mathbf{u}}}\}_{i\in[G]}$. Together, this guarantees the *fairness* and *unpredictability* of SSLE.

This approach, however, doubles the communication round, which dominates our runtime since local computation is lightweight. Now, we present a solution to minimize the overhead. The idea is to bring in *new randomness* after the adversary has chosen its message. Let H be a hash function; after receiving the messages from all parties, they use the hash value of received messages as randomness to sample a subset $\mathcal{S} \subset [G]$, s.t., $|\mathcal{S}| = G/2$. The aggregation procedure only takes the messages inside the subset $\mathcal{S}$ as inputs. As a result, the adversary only gets to learn which messages will take effect in the aggregation after it crafts the message. With $\mathsf{poly}(\lambda)$ number of queries to H but $O(2^G)$ possible subsets $\mathcal{S}$, as long as G is large enough, we claim that the aggregation result is still "random" given the potentially malicious messages sent by the adversary. In the following claim, we detail the above high-level intuition to accommodate the first and second rounds of broadcast of crSSLE (i.e., the outputs of crSSLE.Gen and crSSLE.ParElect).

Claim 6.2 (*Random-subset*)**.** Given a collision-resistant hash function H and a threshold FHE scheme tFHE. Denote the plaintext space as $\mathcal{P}$, closed under addition. For any PPT adversary $\mathcal{A}$, let G be the total number of parties. Denote the corrupted set as $\mathcal{W}_{\mathsf{cor}}$, $|\mathcal{W}_{\mathsf{cor}}| < G/2$. Let $\mathcal{A}$ play the role of parties in $\mathcal{W}_{\mathsf{cor}}$ in the following procedure. All honest parties $i \in [G]\backslash\mathcal{W}_{\mathsf{cor}}$ samples $x_i \xleftarrow{\$} \mathcal{P}$ and generate the ciphertext $\mathsf{ct}_i := \boxed{x_i}$. The adversary crafts $\{\mathsf{ct}_i\}_{i\in\mathcal{W}_{\mathsf{cor}}}$ and broadcasts to all. Let $h_s = H(\{\mathsf{ct}_i\}_{i\in[G]})$ be the encoding of a subset $\mathcal{S} \subset [G]$, $|\mathcal{S}| = G/2$. Let $\mathsf{ct} \leftarrow \mathsf{TFHE.Eval}(+, \{\mathsf{ct}_i\}_{i\in\mathcal{S}})$. For any $\mathcal{A}$ making at most $\mathsf{poly}(\lambda)$ queries, for any $x, x' \in \mathcal{P}$, we have $|\Pr[\mathsf{TFHE.Dec}(\mathsf{msk},\mathsf{ct}) = x] - \Pr[\mathsf{TFHE.Dec}(\mathsf{msk},\mathsf{ct}) = x']| = \mathsf{negl}(\lambda)$, where the randomness is taken over $\{x_i\}_{i\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$.

With the above claim, the probability of any PPT adversary being able to tamper with the distribution of $\vec{\mathbf{u}}$ and the leader's randomized commitment in favor of $\mathcal{A}$ is negligible. Notice that this "random-subset" approach does not work for the third round of broadcast (i.e., releasing the partial decryptions as the outputs of crSSLE.Elect) because we assume a G-out-of-G scheme and all partial decryption shares are required to recover the underlying plaintext. Thus, the last broadcast will always adapt the commit-and-reveal approach.

To conclude, the above changes augment crSSLE with *fairness* and *unpredictability* against malicious users, i.e., the adversary can, at best, mount DoS attacks and break *uniqueness*. For small G, this is achieved in 6 round of communication and 4 for large G with honest majority.

6.2.2 Retroactive Detection for Passive Attacks

Now that we have ruled out the possibility of biasing elections to the attacker's advantage, we consider attacks with the sole goal of disrupting (i.e., aborting) elections. Since the G-out-of-G model inherently assumes participation from all parties (or DoS is trivial), the attacker must participate but can send arbitrary messages. However, the attacker's hands are tied: if they equivocate (sending different messages to different honest parties) when they are supposed to broadcast, their signatures on conflicting messages are irrefutable evidence of malice. Thus the only concern left is the attacker broadcasting malformed messages. For instance, if the attacker broadcasts a random noise or a noise-overflowed ciphertext, all operations performed on this message will produce overflowed noise and contaminate the underlying plaintext. As a result, the protocol output would not be a valid commitment to any party. This will break the *uniqueness* guarantee in the sense that no one can claim the leadership.

If any party observes no valid leader is elected, they can complain and cause the protocol to enter a *retroactive detection phase* to identify senders of malformed messages. If no misbehavior is detected, the party who complains is slashed.

A naive attempt is asking all parties to present proofs when someone complains: a proof for the well-formedness of $\boxed{u_k}$, the correct evaluation of randomized commitments $\boxed{x'_{u,k}}$, and a proof for the correct partial decryption $[\![\bar{x}_u]\!]_k$. We note that this is already an improvement over schemes (e.g., [6]) where similar proofs are *always* required.

We present a method to avoid the first two proofs, which are the heaviest ones, as follows. We observe that FHE evaluation is deterministic if the random coins are known. Thus, parties are asked to reveal the randomnesses used for tBFV.Enc on line 9 and 17, and the randomness used for Π_{MPRC}.Randomize on line 16. Anyone could deterministically reconstruct $\boxed{\vec{\mathbf{u}}_k}$ and $\boxed{x'_{u,k}}$ to verify if they are consistent with the messages received during the election phase. Since all messages are signed, any inconsistency immediately constitutes evidence of malice. As a further optimization, parties only need to reveal the seed that generates those randomness. Revealing the randomness will not affect secrecy since new randomness is generated independently of the long-term secrets for each election.

This optimization does not apply to the last proof (the proof for partial decryption) since it would involve leaking the long-term secret key shares for the tFHE scheme. With extensive works on efficient proof systems for (t)FHE encryption and decryption [2, 7, 13, 23, 27], this proof is relatively efficient. We stress that the retroactive detection is only invoked when no leader is elected, thus it is not on the critical path of Qelect.

After detecting misbehavior with irrefutable evidence, the offender can be slashed in the same way as existing consensus protocol [9]: Confiscate its collateral and remove it from future participation.

Putting everything together. With the augmentation to crSSLE presented in §§ 6.2.1 and 6.2.2, we obtain Qelect. A formal specification of Qelect and the security proof of the following theorem are deferred to Appendix C.3 in the interest of space.

Theorem 6.3. Given G parties, for any fully malicious PPT adversary corrupting less than $G/2$ parties, Qelect is an SSLE protocol defined in Definition 6.1.

6.2.3 Practical Considerations

We discuss some practical concerns specific to our protocol. For general issues, such as node churn, adding new parties and weighted elections, we claim the same as in [4].

Trusted setup and the threshold. tFHE schemes require a setup phase to generate decryption key shares. We adopt the tFHE scheme in [5] that relies on the 0,1-linear-secret-sharing scheme (LSSS). We favor LSSS over the standard Shamir secret-sharing scheme for three main reasons. First, performance-wise, LSSS has a lightweight decryption algorithm that only involves additions (without Lagrange interpolation). Second, since only additions are performed in the final decryption, the noise bound is linear to the party size instead of exponential in Shamir secret sharing. Importantly, a recent work [14] shows some security limitations of Shamir-secret-sharing-based tFHE. However, there is currently no efficient Distributed Key Generation (DKG) for the fFHE protocol we use. Our implementation assumes a centralized trusted setup.

We use performance numbers from [15] to guide our choice of τ, the threshold. One consideration is that the key share size grows exponentially in $\binom{G}{\tau}$ in LSSS. For large G such as 2^{15}, the key share size for $\tau = G-2$ is around 66TB, but only 100KB if $\tau = G$. For small G such as $G \leq 16$, we could have $\tau = G/2$ with key share size ≤ 0.2GB. A similar trend applies to the runtime of the trusted setup as shown in [15, section 5.2]. The key distribution process takes several hours for $G-3 \leq \tau \leq G$ and large $G \leq 2^{15}$. For $\tau = G$, the setup time is mainly quadratic in G: less than an hour for $G = 2^{15}$ and less than 5 minutes with $G \leq 8192$. On the other hand, as claimed in [15], the decryption time is dominated by the partial decryption, which does not grow with $\binom{G}{\tau}$ and only takes ~ 0.01 second.

Therefore, we focus on the G-out-of-G setting that scales to a larger number of parties. For small-scale settings ($G \leq 32$), a more aggressive threshold (e.g., $\tau = G/2$) can be used.

Preprocessing phase. We present an optimization where the first broadcast message (i.e., u) is generated during the key generation. The trusted dealer who is responsible for distributing the $\{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}$ could generate κ ciphertexts $\{\boxed{\vec{\mathbf{u}}^{(j)}}\}_{j\in[\kappa]}$ during the setup, replacing the ciphertext aggregated from all $\{\boxed{\vec{\mathbf{u}}_i}\}_{i\in[G]}$. In this way, the parties neither need to broadcast and aggregate the initial encrypted randomnesses nor need to prove the well-formedness of $\boxed{\vec{\mathbf{u}}_i}$ by sending its underlying randomness in detection phase.

Concretely, with $N = 32768, \kappa = 16$, the trust setup could prepare initial randomness for 2^{19} rounds with little overhead. As they are consumed, parties could keep generating $\boxed{\vec{\mathbf{u}}}$ without relying on the trusted key dealer. For the parameter we choose, it takes around 30 minutes for all parties to locally aggregate $\{\boxed{\vec{\mathbf{u}}_i}\}_{i\in[G]}$ into $\boxed{\vec{\mathbf{u}}}$, which provides enough randomnesses for N rounds, where N is the ring dimension. Thus, all parties could prepare $\boxed{\vec{\mathbf{u}}}$ for the next N election instances while participating in the current N instances in parallel.

Performance estimation of the pessimistic path. The runtime of retroactive detection is dominated by the proof generation and verification time for partial decryptions. Based on [13], it takes approximately minutes to generate such a proof and a similar amount of time to verify. Thus, for $G = 128$, it takes more than 4 hours to finish for a party to verify all others' proofs. Details on the performance estimation are relegated to Appendix C.4. We also point out that if a party gets slashed, the remaining parties must perform a trusted setup again.

To avoid disrupting the blockchain consensus protocol, the protocol continues with a new set of G parties while the current set finishes retroactive detection. This is consistent with the deployment model of Whisk [21]: In Ethereum, the number of validators is huge (over one million at the time of writing in January 2025), so SSLE is executed among a small subset of validators rotated on a schedule (e.g., daily). A similar strategy can be taken in Qelect, where the next scheduled set of G parties enter the early protocol while the current set finishes the retroactive detection protocol.

7 Benchmark

We implement Qelect protocol (modulo the pessimistic path in § 6.2.2) in C++ (will be released as open source). Our implementation uses the SEAL [29] library for basic BFV FHE operations. We benchmark these schemes on AWS EC2 c6i.2xlarge instances with 8 vCPU, 16GB RAM and Intel Xeon Scalable processors.

Parameters and setup. We run experiment with a party size from $\{4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\}$, with the following parameters for the underlying tBFV scheme: ring dimension $N = 32768$, plaintext modulus $q = 65537$, ciphertext modulus Q that $\log Q = 720$. The parameters for the sRLWE scheme used in Π_{MPRC} are: ring dimension $n = 1024$, plaintext modulus $p = 2$, ciphertext modulus $q' = 65537$, error distribution with $\sigma = 0.5$, hamming weight of secret keys $h = 32$. For all the schemes $\Pi_{\mathsf{MPRC}}, \mathsf{crSSLE}, \mathsf{Qelect}$, we have the security parameter $\lambda \geq 80$. The size of the final ciphertext and its partial decryption is approximately 983 KB. For $G = 128$, each party needs to send 126.83MB data in each broadcast. The choice of τ is application-based and tuneable. In our benchmark, we set $\tau = G$, which means that every party needs to receive partial decryption from all other parties before it can decrypt.

For the local area network (LAN) setting, we put all in-

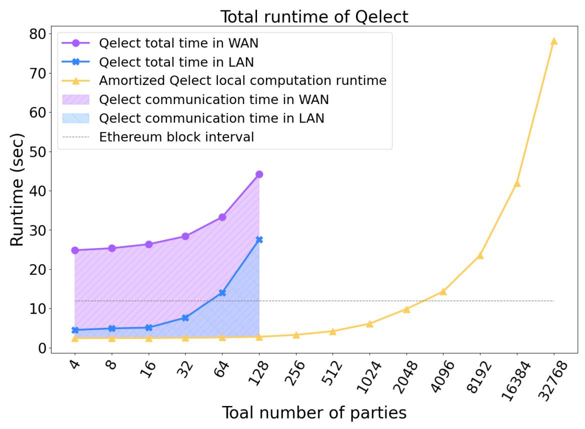

Figure 2: Runtime of Qelect adapted from crSSLE under LAN and WAN setting. It is important to note that the x-axis grows exponentially with each tick, and thus the underlying trend between runtime (both local computation time and communication time) and group size is linear, which matches our asymptotic estimates. We also plot the Ethereum block interval as a reference to illustrate the practicality of our protocol.

stances under the AWS us-east-2 region (Ohio). For the wide area network (WAN) setting, we distribute all instances uniformly under AWS us-east-2, us-west-1 (California), eu-west-1 (Ireland), and ap-southeast-1 (Singapore). The available network bandwidth in LAN varies from $\sim$4.95 Gbps to $\sim$9.53 Gbps, and the available network bandwidth in WAN varies from $\sim$160 Mbps to $\sim$530 Mbps.

Computation time of Qelect**.** Fig. 2 demonstrates our local computation time for each party in the Qelect protocol, the communication time in a LAN setting (shown as the blue shaded area), and the communication time in a WAN setting (shown as the purple shaded area). The runtime grows linearly with the group size, and the communication time dominates the total runtime. Moreover, the local computation with $G \leq 2048$ could be finished within 10 seconds.

Comparison with non-post-quantum secure works. Post-quantum secure protocols are usually slower than classical ones. We first compare to existing classical schemes to offer a perspective on the cost of post-quantum security in Qelect.

The most practical DDH-based SSLE protocol is introduced in [4] and a variant is implemented in Whisk [21]. Based on the estimation given in [21], the local computation is dominated by the generation of shuffling proof, around 880ms and at least two orders of magnitude faster than ours. And as in [4], they only need a single round of one-to-all broadcast of message size around 16KB based on the implementation detail in [21], while our protocol requires three rounds of all-to-all broadcast during the election phase.

Another line of SSLE uses MPC [3]. The runtime is dominated by $O(\log G)$ rounds of communications. The main advantage of [3] is that it can handle unbalanced weight elections efficiently: With the total weight of all parties to be S, the total cost grows logarithmically in S, instead of linearly as in DDH-based protocols [4, 6]. Based on their benchmark [3, Table 2], for 128 parties, the runtime under LAN setting is $>$ 80s and $>$ 317s under WAN, which is 3x-7x higher than ours. However, similar to the DDH-based protocol, the construction in [3] can tolerate a fully malicious adversary corrupting up to $G-1$ parties and can guarantee liveness as long as there exists an honest majority staying in the protocol.

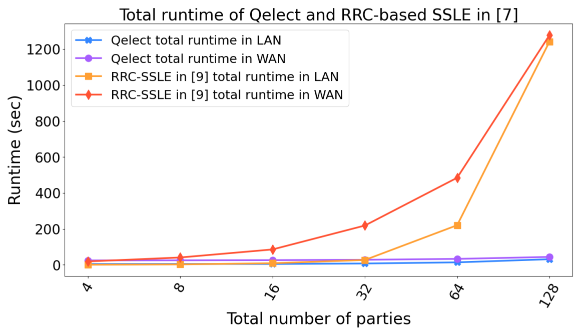

Figure 3: The total runtime Qelect based on crSSLE and the SSLE protocol in [6]. Referring to Fig. 2, we could see that both works have the runtime grow linearly with the group size, while [6] has a more rapid trend.

Comparison with the latest post-quantum SSLE [6]. In the line of post-quantum secure SSLE, we take the SOTA work [6] as our baseline for comparison. In Fig. 3, we first benchmark our Qelect protocol with $G \leq 128$ under both LAN and WAN settings. And for the RRC-based protocol in [6], as previously mentioned, the local computation in their protocol only involves some matrix multiplication and shuffles in plain, and we generously omit their local computation time. Since their RRC scheme is based on RLWE, we set the corresponding parameter as the same in our sRLWE defined in § 3.1, with $n = 1024, q' = 65537, \ell = 256$. As claimed [6, section 5], they have $m = \Omega(\log q)$, so we set $m = \log q = 16$ to give them an additional comparative advantage. With $G = 128$, each party then needs to broadcast 8.38 MB to the other parties in their protocol. While in Qelect, each party only broadcasts 983 KB to each other. Moreover, they need the broadcast to be taken sequentially. Therefore, in Fig. 3 we see that the total runtime of their protocol grows much more promptly than ours, and we are at least two orders of magnitude faster.

Acknowledgments

We thank Katerina Sotiraki for the insightful comments on our malicious proofs and Zeyu Liu for the discussion on the overall FHE circuit. We also thank the valuable suggestions from the anonymous reviewers. This work is partly supported by an award from Yale Roberts Innovation Fund.

8 Ethics Considerations

Our work implements a practical solution based on cryptographic assumptions for a long-existing primitive and strictly follows the ethics guidelines. All experiments performed are simulations with no real-world participants. Our results do not impose harm or danger on the larger community. This also does not involve disclosing any unknown vulnerabilities of the existing protocols or systems.

9 Open Science

To support the Open Science policy, our source code is permanently published at [32].

References

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. Cryptology ePrint Archive, Paper 2017/1142, 2017.

[2] Thomas Attema, Ronald Cramer, and Lisa Kohl. A Compressed Σ-Protocol Theory for Lattices. Cryptology ePrint Archive, Paper 2021/307, 2021.

[3] Michael Backes, Pascal Berrang, Lucjan Hanzlik, and Ivan Pryvalov. A framework for constructing Single Secret Leader Election from MPC. Cryptology ePrint Archive, Paper 2022/1040, 2022.

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

[5] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *CRYPTO 2018*, 2018.

[6] Dan Boneh, Aditi Partap, and Lior Rotem. Post-Quantum Single Secret Leader Election (SSLE) From Publicly Re-randomizable Commitments. Cryptology ePrint Archive, Paper 2023/1241, 2023.

[7] Enrico Bottazzi. Greco: Fast Zero-Knowledge Proofs for Valid FHE RLWE Ciphertexts Formation. Cryptology ePrint Archive, Paper 2024/594, 2024.

[8] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO 2012*, 2012.

[9] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X. Zhang. Combining GHOST and Casper. *CoRR*, abs/2003.03052, 2020.

[10] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An Efficient Post-Quantum Commutative Group Action. Cryptology ePrint Archive, Paper 2018/383, 2018.

[11] Dario Catalano, Dario Fiore, and Emanuele Giunta. Efficient and Universally Composable Single Secret Leader Election from Pairings. Cryptology ePrint Archive, Paper 2021/344, 2021.

[12] Dario Catalano, Dario Fiore, and Emanuele Giunta. Adaptively Secure Single Secret Leader Election from DDH. Cryptology ePrint Archive, Paper 2022/687, 2022.

[13] Sylvain Chatel, Christian Mouchet, Ali Utkan Sahin, Apostolos Pyrgelis, Carmela Troncoso, and Jean-Pierre Hubaux. PELTA – Shielding Multiparty-FHE against Malicious Adversaries. Cryptology ePrint Archive, Paper 2023/642, 2023.

[14] Wonhee Cho, Jiseung Kim, and Changmin Lee. (In)Security of Threshold Fully Homomorphic Encryption based on Shamir Secret Sharing. Cryptology ePrint Archive, Paper 2024/1858, 2024.

[15] Siddhartha Chowdhury, Sayani Sinha, Animesh Singh, Shubham Mishra, Chandan Chaudhary, Sikhar Patranabis, Pratyay Mukherjee, Ayantika Chatterjee, and Debdeep Mukhopadhyay. Efficient Threshold FHE for Privacy-Preserving Applications. Cryptology ePrint Archive, Paper 2022/1625, 2022.

[16] Miranda Christ, Valeria Nikolaenko, and Joseph Bonneau. Leader Election from Randomness Beacons and Other Strategies, Nov 2022. https://a16zcrypto.com/posts/article/leader-election-from-randomness-beacons-and-other-strategies.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

[18] Luciano Freitas, Andrei Tonkikh, Adda-Akram Bendoukha, Sara Tucci-Piergiovanni, Renaud Sirdey, Oana Stan, and Petr Kuznetsov. Homomorphic sortition – single secret leader election for PoS blockchains. Cryptology ePrint Archive, Paper 2023/113, 2023.

[19] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. Stanford university, 2009.

[20] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *SOSP*, 2017.

[21] G. Kadianakis. Whisk: A practical shuffle-based SSLE protocol for Ethereum, Jan 2022. https://ethresear.ch/t/whisk-a-practical-shuffle-based-ssle-protocol-for-ethereum/11763.

[22] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: Oblivious Message Retrieval with Reduced Communication and Computation. *IACR Cryptol. ePrint Arch.*, 2024:204, 2024.

[23] Vadim Lyubashevsky, Ngoc Khanh Nguyen, and Maxime Plançon. Lattice-based zero-knowledge proofs and applications: shorter, simpler, and more general. In *CRYPTO*, 2022.

[24] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*. Springer, 2010.

[25] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(4):291–311, 2021.

[26] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Satoshi Nakamoto*, 2008.

[27] Ngoc Khanh Nguyen and Gregor Seiler. Practical Sublinear Proofs for R1CS from Lattices. Cryptology ePrint Archive, Paper 2022/1048, 2022.

[28] Antonio Sanso. Towards practical post quantum Single Secret Leader Election (SSLE) - Part 1, Aug 2022. https://crypto.ethereum.org/blog/pq-ssle.

[29] Microsoft SEAL (release 3.0). http://sealcrypto.org, October 2018. Microsoft Research, Redmond, WA.

[30] Yunhao Wang and Fan Zhang. Qelect: Lattice-based single secret leader election made practical. Cryptology ePrint Archive, Paper 2025/122, 2025.

[31] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.

[32] Yunhao Wang. Zenodo Source Code for Qelect, 2025. https://doi.org/10.5281/zenodo.14757398.

A Definitions and Proofs for sRLWE

We present the changes we made for the sRLWE construction we use and refer readers to [22, 24] for other details.

- $\mathsf{pp}_{\mathsf{rlwe}} = (n, \ell, q, \sigma, \mathcal{D}, \gamma) \leftarrow \mathsf{sRLWE.GenParams}(1^\lambda, \ell, q, \sigma, h)$: all parameters set accordingly as in [22] and set the error range γ s.t. $(\frac{4\gamma+2}{q})^\ell = \mathsf{negl}(\lambda)$ and $\ell \cdot (1 - \mathsf{erf}(\frac{\gamma}{\sqrt{2(2h+1)}\sigma})) = \mathsf{negl}(\lambda)$; output $\mathsf{pp}_{\mathsf{rlwe}} := (n, \ell, q, \sigma, \mathcal{D}, \gamma)$ as the public parameter for sRLWE scheme.
- $(\mathsf{sk}, \mathsf{pk}) \xleftarrow{\$} \mathsf{sRLWE.KeyGen}(\mathsf{pp}_{\mathsf{rlwe}}; r)$: same as in [22]
- $\mathsf{ct} = (a, b) \xleftarrow{\$} \mathsf{sRLWE.Enc}(\mathsf{pp}_{\mathsf{rlwe}}, \mathsf{pk}, \vec{m})$: same as in [22]
- $\vec{m} \leftarrow \mathsf{sRLWE.Dec}(\mathsf{pp}_{\mathsf{rlwe}}, \mathsf{sk}, \mathsf{ct} = (a, b))$: Compute $a' = a\mathsf{sk} \in \mathcal{R}_q$. Let $\vec{d}[i] = b_i - a'_i$ for $i \in [\ell]$, decrypt $\vec{m}[i] = \begin{cases} 0 & \text{if } \vec{d}[i] \in [0, \gamma] \cup [q - \gamma, q] \\ 1 & \text{if } \vec{d}[i] \in [q/2 - \gamma, q/2 + \gamma] \\ \perp & o.w. \end{cases}$, for $i \in [\ell]$.

Due to space constraints, we defer the formal definitions and proofs for the *correctness*, *CPA security*, *key privacy* properties of our scheme sRLWE to the full version [30].

B Security Proof of MPRC

Proof of Theorem 5.1. **Multi-party Binding**: the high-level intuition is that the final randomized commitment is basically the sum of (at most) $2G$ sRLWE ciphertexts. Thus, as long as the noise aggregated does not mask out the underlying message, the corresponding secret key would still be able to decrypt it. More formally, notice that for one single ciphertext, sRLWE.Dec would be $e_1\mathsf{sk} - ex - e_2$, where $e_1, e, e_2 \leftarrow \chi_\sigma$, and $\mathsf{sk}, x \leftarrow \mathcal{D}$ are two ternary vectors with fixed hamming weight h. This can thus be seen as drawing $2h+1$ Gaussian noises, and summing up (at most) $2G$ such ciphertexts (each randomized commitment would be a sum-up between two ciphertexts, and $\Pi_{\mathsf{MPRC}}.\mathsf{Combine}$ sums up all G randomized commitments) would be equivalent to drawing $\leq 2G(2h+1)$ Gaussian noises. Therefore, given a final commitment c'' with its corresponding witness w (which is the random coin used to draw the original secret key sk), the probability of decrypting into garbage instead of $\{0, 1\}$ is $\leq 1 - \mathsf{erf}(\frac{\gamma}{\sqrt{4G(2h+1)}\sigma})$. For all ℓ elements, we union bound it to be $\ell(1 - \mathsf{erf}(\frac{\gamma}{\sqrt{4G(2h+1)}\sigma}))$ which is negligible w.r.t. λ based on line 3.

On the other hand, for a random secret key generated by another random coin $r' \neq r$, the probability of one element in the decrypted vector falling into the error range for valid decryption is $\frac{4\gamma+2}{q}$. And thus the probability for the decryption result to contain no $\perp$ is $\leq (\frac{4\gamma+2}{q})^\ell = \mathsf{negl}(\lambda)$ based on sRLWE.GenParams in § 3.1. Hence, $\Pr[\mathsf{Verify}(\mathsf{pp}, c'', w, m) = \mathsf{Verify}(\mathsf{pp}, c'', w', m)] = \Pr[\mathsf{Verify}(\mathsf{pp}, c'', w, m) = \mathsf{Verify}(\mathsf{pp}, c'', w', m) = 0] + \Pr[\mathsf{Verify}(\mathsf{pp}, c'', w, m) = \mathsf{Verify}(\mathsf{pp}, c'', w', m) = 1] \leq \Pr[\mathsf{Verify}(\mathsf{pp}, c'', w, m) = 0] + \Pr[\mathsf{Verify}(\mathsf{pp}, c'', w', m) = 1] \leq \mathsf{negl}(\lambda) + \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$.

Unlinkability: given an adversary $\mathcal{A}$ that could break the unlinkability of Algorithm 1 with probability $\frac{1}{2} + \mathsf{pr}$, i.e., it has non-negligible advantage pr, we then construct an adversary

$\mathcal{A}'$ that breaks the key privacy of sRLWE as follows:

- The challenger first sends $(\mathsf{pk}_0, \mathsf{pk}_1)$ to $\mathcal{A}'$.
- $\mathcal{A}'$ receives the $(m_0, m_1, \mathcal{W}_{\mathsf{cor}})$ from $\mathcal{A}$, where $\mathcal{W}_{\mathsf{cor}}$ is the indices of corrupted parties, and passes (m_0, m_1) to the challenger.
- The challenger samples $b \xleftarrow{\$} \{0,1\}$, evaluates $\mathsf{ct}_b \xleftarrow{\$} \mathsf{sRLWE.Enc}(\mathsf{pk}_b, m_b)$, and sends ct_b to $\mathcal{A}'$.
- $\mathcal{A}'$ samples $b' \xleftarrow{\$} \{0,1\}$ and directly sets $c_{b'} := (\mathsf{pk}_{b'}, \mathsf{ct}_b)$. $\mathcal{A}'$ then computes $c_{1-b'} \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Commit}(\mathsf{pp}, m_{1-b'})$.

  For $i \in [G]$, generate $c'_{0,i} \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}, c_0)$, $c'_{1,i} \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}, c_1)$, and denote $\mathcal{C}_{0,G} := \{c'_{0,i}\}_{i\in[G]}, \mathcal{C}_{1,G} := \{c'_{1,i}\}_{i\in[G]}$.

  $\mathcal{A}'$ then sends $(c_0, c_1, \mathcal{C}'_{0,G} := \{c'_{0,i}\}_{i\in\mathcal{W}_{\mathsf{cor}}}, \mathcal{C}'_{1,G} := \{c'_{1,i}\}_{i\in\mathcal{W}_{\mathsf{cor}}})$ to $\mathcal{A}$.
- After receiving $\mathcal{C}''_{0,G}, \mathcal{C}''_{1,G}$ from $\mathcal{A}$, $\mathcal{A}'$ generates $c'' \leftarrow \Pi_{\mathsf{MPRC}}.\mathsf{Combine}(\mathsf{pp}, \mathcal{C}''_{b',G} \cup \mathcal{C}_{b',G} \backslash \mathcal{C}'_{b',G})$, and sends c'' to $\mathcal{A}$.
- If $\mathcal{A}$ outputs 0, $\mathcal{A}'$ also outputs 0; otherwise, $\mathcal{A}$ outputs $\{0,1\}$ uniformly random.

If $\mathcal{A}'$ with probability $1/2$ chooses $b' = b$, $\mathcal{A}'$ simulates the game of *unlinkability* honestly and thus $\mathcal{A}$ should output 0 with probability $1/2 + \mathsf{pr}$. Otherwise, $\mathcal{A}'$ has probability of $1/2$ to guess b correctly. Thus, we have $\mathcal{A}'$ break key privacy with probability $\frac{1}{2}(\frac{1}{2} + \mathsf{pr}) + \frac{1}{4} = \frac{1}{2} + \frac{1}{2}\mathsf{pr}$, which has a non-negligible advantage.

□

C Complementary Materials for § 6

C.1 crSSLE: Honest-but-Curious Setting

Proof of Theorem 6.1. The *uniqueness* proof of Algorithm 2 could be broken into the following conditions:

1. Out of $\bar{t}$ random indices being extracted on line 14, there exists at least one index that will be mapped in $\mathbb{Z}_G$, which is achieved by the parameter generation process in line 4.
2. After summing up all randomized shares for the leader i from G parties, the final commitment encrypted under $\mathsf{ct}_{l\in[\bar{t}]}$ derived on line 21 is still a valid commitment of m_i w.r.t. y_i, which is based on the *multi-party binding* property of the underlying Π_{MPRC} scheme defined in Algorithm 1.
3. The decryption of $\mathsf{ct}_{l\in[\bar{t}]}$ succeeds, which is simply satisfied under a honest-but-curious setting so that there would be at least τ parties supplying their partial decryptions.

The *unpredictability* could be proved in a hybrid manner. Take the unpredictability game of crSSLE (defined in Definition 6.1) as H_0, we define H_1 to let *Challenger*, instead of outputing $\{[\![\vec{x}']\!]_j\}_{j\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$ by honestly evaluating crSSLE.ParElect, sample random $\mathcal{R}_Q \times \mathcal{R}_Q$ tuple as a tBFV ciphertext, where Q is the ciphertext modulus of tBFV. H_1 is computationally indistinguishable to H_0 based on semantic security of tBFV. In H_2, we let *Challenger*, instead of outputing $\{[\![x']\!]_j\}_{j\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$ by honestly evaluating crSSLE.Elect, output random $\mathcal{R}_Q \times \mathcal{R}_Q$ tuple as the partial decryptions. The hybrid H_2 is computationally indistinguishable to H_1 since for a random ciphertext $(a,b) \in \mathcal{R}_Q \times \mathcal{R}_Q$, we would have the b part acting as a random mask to the decryption and thus the partial decryption is pseudorandom.

It is clear to see that in H_2, we have the combine result x' to be pseudorandom, which is unrelated to any original $x_{i\in[G]}$, and this hybrid is computationally indistinguishable to the security game H_0. Therefore, the view of a malicious adversary leaks no information about the elected leader and the adversary should not have any non-negligible advantage.

For the *fairness*, we first observe that as long as the aggregated ciphertext $\mathsf{ct}_{\mathsf{pl}}$ obtained on line 13 encrypts a uniformly random index $u \in \mathbb{Z}_G$, then the *fairness* property is achieved. It is clear to see that even if the adversary could craft some $\mathsf{pl}_{k\in\mathcal{W}_{\mathsf{cor}}}$ in favor of itself, where $\mathcal{W}_{\mathsf{cor}}$ is the corrupted set, as long as there exists one honest party i who supplies pl_i, the aggregated result would be random.

□

C.2 Related Materials for Malicious Security

Claim C.1 (*Commit-and-reveal*)**.** Given a hash function H and a threshold FHE scheme TFHE. Denote the plaintext as $\mathcal{P}$, closed under addition. For any PPT adversary $\mathcal{A}$, let G be the total number of parties. Denote the corrupted set as $\mathcal{W}_{\mathsf{cor}}$, $|\mathcal{W}_{\mathsf{cor}}| < G$. Let $\mathcal{A}$ play the role of parties in $\mathcal{W}_{\mathsf{cor}}$ in the following procedure. All honest parties $i \in [G]\backslash\mathcal{W}_{\mathsf{cor}}$ sample $x_i \xleftarrow{\$} \mathcal{P}$ and generate ciphertexts $\mathsf{ct}_i := \boxed{x_i}$ with the hashed value $h_i \leftarrow H(\mathsf{ct}_i)$. The parties $\in [G]\backslash\mathcal{W}_{\mathsf{cor}}$ first publish $\{h_i\}_{i\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$, and the adversary publishes $\{h_i\}_{i\in\mathcal{W}_{\mathsf{cor}}}$. After seeing $\{h_i\}_{i\in[G]}$, all honest parties publish $\{\mathsf{ct}_i\}_{i\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$, and the adversary publishes $\{\mathsf{ct}_i\}_{i\in\mathcal{W}_{\mathsf{cor}}}$ s.t. $H(\mathsf{ct}_i) = h_i$ for $i \in \mathcal{W}_{\mathsf{cor}}$. For any $\mathcal{A}$ making at most $\mathsf{poly}(\lambda)$ queries to H, let $\mathsf{ct}' \leftarrow \mathsf{TFHE.Eval}(+, \{\mathsf{ct}_i\}_{i\in[G]})$, for any $x, x' \in \mathcal{P}$, we have $|\Pr[\mathsf{TFHE.Dec}(\mathsf{msk}, \mathsf{ct}) = x] - \Pr[\mathsf{TFHE.Dec}(\mathsf{msk}, \mathsf{ct}) = x']| = \mathsf{negl}(\lambda)$, where the randomness is taken over $\{x_i\}_{i\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$.

The proof is deferred to the full version [30].

Proof of Claim 6.2. There will be only two cases regarding the composition of the subset $\mathcal{S}$, with $|\mathcal{S}| = G/2$ and $|\mathcal{W}_{\mathsf{cor}}| < G/2$.

1. If $\mathcal{S}$ contains ciphertexts generated all from honest parties, then trivially we have $\Pr[\mathsf{TFHE.Dec}(\mathsf{msk}, \mathsf{ct}) = x] = \Pr[\mathsf{TFHE.Dec}(\mathsf{msk}, \mathsf{ct}) = x'] = 1/|\mathcal{P}|$ for any $x, x' \in \mathcal{P}$, since x_i are all chosen uniformly at random.

2. If $\mathcal{S}$ contains both ciphertexts from the honest and corrupted parties, then observe that the best thing the adversary can do is to craft $\tilde{\mathsf{ct}}_1, \ldots, \tilde{\mathsf{ct}}_{|\mathcal{W}_{\mathsf{cor}}|}$ in place of the ciphertexts generated by the corrupted parties w.r.t. $|\mathcal{W}_{\mathsf{cor}}|$ different subsets $S_1, \ldots, S_{\mathcal{W}_{\mathsf{cor}}}$, s.t. the summation results $\mathsf{TFHE.Eval}(+, \{\mathsf{ct}_i\}_{i \in S_j}), j \in |\mathcal{W}_{\mathsf{cor}}|$ are ciphertexts encrypting some specific values pre-determined by the adversary. Since $|\mathcal{W}_{\mathsf{cor}}| < G/2$, the adversary can only target at $O(G)$ specific subsets, with $O(1/2^G)$ many possible subsets, the probability of the hitting the subsets that the adversary bets on is $\mathsf{negl}(G)$.

   On the other hand, if the subset with mixing ciphertexts from both honest and corrupted parties is not one of the subsets $\mathcal{S}_{j \in [\mathcal{W}_{\mathsf{cor}}]}$ that the adversary targets to, then we could treat the outputs of the adversary as independently chosen of the honest ciphertexts and are some random bytes with $\mathsf{negl}(\lambda)$ probability to be valid ciphertexts. Therefore, the added up results will also be some random garbage with $1 - \mathsf{negl}(\lambda)$ probability, and we have $\Pr[\mathsf{TFHE.Dec}(\mathsf{msk}, \mathsf{ct}) = x] = \Pr[\mathsf{TFHE.Dec}(\mathsf{msk}, \mathsf{ct}) = x'] = \mathsf{negl}(\lambda)$ for any $x, x' \in \mathcal{P}$.

$\square$

The formal algorithm for the patched crSSLE is in Algorithm 3, with differences highlighted in blue.

Theorem C.2. Given G parties, for any fully malicious PPT adversary corrupting $< G/2$ parties, the crSSLE construction given in Algorithm 3 satisfies the *fairness* and *unpredictability* defined in Definition 6.1.

Proof of Theorem C.2. We use a hybrid argument to prove the *fairness* and *unpredictability* under the case of large G. The case of mall G follows the same proof. Let $\mathcal{A}$ play the role of all corrupted parties and the challenger plays the role of the environment.

- Hyb_0 is the initial game defined in Definition 6.1, where the challenger honestly generates the transcripts on behalf of all honest parties.
- In Hyb_1, change the added up ciphertext $\mathsf{ct}_{\vec{\mathbf{u}}}$ on line 16 or 18 into a ciphertext $\boxed{\vec{\mathbf{u}}^*}$, where $\vec{\mathbf{u}}^* \xleftarrow{\$} \mathbb{Z}_G^N$. Based on Claim 6.2 and RLWE assumption, Hyb_1 is computationally indistinguishable from Hyb_0. (For small G, we construct the same hybrid but argue the indistinguishability based on Claim C.1 instead. Same for the following point.)
- In Hyb_2, change the added up ciphertexts $\boxed{\bar{x}_u}$ on line 32 or 34 into ciphertexts $\boxed{\bar{x}_{u^*}}$, s.t. $\bar{x}_{u^*}, \leftarrow \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}_C, x_{\vec{\mathbf{u}}^*[j]})$, where j is the number of the current election round. Notice that the commitment space of Π_{MPRC} would be the plaintext space $\mathcal{P}$ in Claim 6.2, which is indeed close under addition. Thus, based on Claim 6.2 and RLWE assumption, Hyb_2 is computationally indistinguishable from Hyb_1.
- In Hyb_3, change the combined partial decryption into the result of combining all shares by partially decrypting $\boxed{x'_{u^*}}$, where $x'_{u^*} \leftarrow \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}_C, x_{u^*}), u^* \xleftarrow{\$} [G]$. Same as above, we have the commitment space of Π_{MPRC} to be the plaintext space $\mathcal{P}$ in Claim C.1. Based on the collision resistance property of hash functions and RLWE assumption, Hyb_3 is computationally indistinguishable from Hyb_2.

Observe that in Hyb_3, the environment faced by the adversary will elect a leader with uniform probability, which guarantees the adversary is not able to predict the leader better than random guessing. Since it is indistinguishable from the original game, we conclude that the patched crSSLE in Algorithm 3 achieve *fairness* and *unpredictability* under a malicious setting. $\square$

C.3 Formal Protocol of Qelect

Preprocessing Phase of Qelect:

1. Let G parties be involved in a trusted setup. Let the trusted authority first invokes $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i \in [G]}) \xleftarrow{\$} \mathsf{crSSLE.Setup}(\mathsf{pp}, 1^\lambda)$. W.l.o.g., we assume that all evaluation keys are packed in mpk. The trusted authorities also generates G commitments C_i w.r.t. each $[\![\mathsf{msk}]\!]_i$ based on ring-SIS: $C_i := a \cdot [\![\mathsf{msk}]\!]_i$, where $a \in \mathcal{R}_q^m, m = \log q$. This serves as the public parameter when generating and verifying the ZKP of partial decryptions.

   It then generates κ ciphertext $\boxed{\vec{\mathbf{u}}_\rho} \xleftarrow{\$} \mathsf{tBFV.Enc}(\mathsf{mpk}, \vec{\mathbf{u}}_\rho), \rho \in [\kappa]$, with $\vec{\mathbf{u}}_\rho \xleftarrow{\$} \mathbb{Z}_G^N$. κ could be treated as a tuneable parameter.

   All parties get involved in the setup phase of the zero-knowledge proof system ZKP: $(\mathsf{PK}, \mathsf{VK}) \xleftarrow{\$} \mathsf{ZKP.KeyGen}(\mathsf{pp}, 1^\lambda)$, with $\{C_i\}$ attached to both PK, VK. Each party k receives $(\mathsf{PK}, \mathsf{VK}, \mathsf{mpk}, [\![\mathsf{msk}]\!]_k, \{\boxed{\vec{\mathbf{u}}_\rho}\}_{\rho \in [\kappa]})$.
2. (**Broadcast**) For each party k, invoke $(x_{k,\mathsf{ep}_0}, y_{k,\mathsf{ep}_0}, \cdot) \xleftarrow{\$} \mathsf{crSSLE.Gen}(\mathsf{pp}, \mathsf{mpk}, 1^\lambda, 0^N)$.

   Broadcast x_{k,ep_0} to all other parties.

Election Phase at instance ep_t of Qelect**:**

1. (**Broadcast**) For each party k, initializes three random number generators $\mathsf{rng}_{0,k,\mathsf{ep}_t}, \mathsf{rng}_{1,k,\mathsf{ep}_t}, \mathsf{rng}_{2,k,\mathsf{ep}_t}$.

   Invoke $\boxed{x'_{u,k}} \leftarrow \mathsf{crSSLE.ParElect}(\mathsf{mpk}, \{(x_{i,\mathsf{ep}_t}, \cdot\}_{i \in [G]})$ (and potentially also $H(\boxed{x'_{u,k}})$ if the party size is small) from line 19 in Algorithm 3.

   Broadcast $\boxed{x'_{u,k}}$ (and potentially $H_2(\boxed{x'_{u,k}})$ if the party

Algorithm 3 crSSLE Construction with security patches

```
1:  procedure crSSLE.Setup(pp = $(G,\tau,q,\sigma,h), 1^\lambda$)
2:      same as in Algorithm 3, but Sample a hash function $H$ and append it to pp.
3:      return $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]})$
4:  procedure crSSLE.Gen(pp = $(\mathsf{pp}_C, \mathsf{pp}_{\mathsf{tBFV}}, \cdot), \mathsf{mpk}, 1^\lambda, m_k$)   ▷ we treat $m_k$ as all zeros for crSSLE scheme.
5:      $\vec{\mathbf{u}}_k \xleftarrow{\$} \mathbb{Z}_q^N$
6:      $\boxed{\vec{\mathbf{u}}_k} \xleftarrow{\$} \mathsf{tBFV.Enc}(\mathsf{mpk}, \vec{\mathbf{u}}_k)$
7:      $(x_k, y_k) \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Commit}(\mathsf{pp}_C, m_k)$
8:      if $G \geq 128$ and $|\mathcal{W}_{\mathsf{cor}}| < G/2$ then
9:          return $(x_k, y_k, \boxed{\vec{\mathbf{u}}_k})$
10:     else
11:         $h_k = H(\boxed{\vec{\mathbf{u}}_k})$ and publish $(x_k, y_k, h_k)$
12:         After receiving $\{h_i\}_{i\in[G]}$, publish $\boxed{\vec{\mathbf{u}}_k}$
13: procedure crSSLE.ParElect(mpk, $\{(x_i, \boxed{\vec{\mathbf{u}}_i})\}_{i\in[G]}$)
14:     if $G \geq 128$ and $|\mathcal{W}_{\mathsf{cor}}| < G/2$ then
15:         $x_s = H(\{\boxed{\vec{\mathbf{u}}_i}\}_{i\in[G]})$
16:         $\mathsf{ct}_{\vec{\mathbf{u}}}, \leftarrow \mathsf{tBFV.Eval}(+, \{\boxed{\vec{\mathbf{u}}_i}\}_{i\in\mathcal{S}})$ ▷ $\mathcal{S}$ is encoded by $x_s$
17:     else
18:         $\mathsf{ct}_{\vec{\mathbf{u}}}, \leftarrow \mathsf{tBFV.Eval}(+, \{\boxed{\vec{\mathbf{u}}_i}\}_{i\in[G]})$
19:     $\mathsf{ct}_u \leftarrow \mathsf{tBFV.Extract}(\mathsf{mpk}, \mathsf{ct}_{\vec{\mathbf{u}}})$   ▷ extract the $j$-th slot out of all $N$ slots in the $j$-th round, if amortized across multiple rounds
20:     $\boxed{u^N} \leftarrow \mathsf{tBFV.Fill}(\mathsf{mpk}, \mathsf{ct}_u)$   ▷ where $u = \sum_{i\in[G]} \vec{\mathbf{u}}[j]$
21:     $x'_i \xleftarrow{\$} \Pi_{\mathsf{MPRC}}.\mathsf{Randomize}(\mathsf{pp}_C, x_i), i \in [G]$
22:     $\boxed{x'_i} \xleftarrow{\$} \mathsf{tBFV.Enc}(\mathsf{mpk}, x'_i), i \in [G]$
23:     $\boxed{x'_{u,k}} \leftarrow \mathsf{tBFV.OblSel}(\mathsf{mpk}, \boxed{u^N}, \{\boxed{x'_i}\}_{i\in[G]}, q)$
24:     if $G \geq 128$ and $|\mathcal{W}_{\mathsf{cor}}| < G/2$ then
25:         return $\boxed{x'_{u,k}}$
26:     else
27:         $h_k = H(\boxed{x'_{u,k}})$ and publish $h_k$
28:         After receiving $\{h_i\}_{i\in[G]}$, publish $\boxed{x'_{u,k}}$
29: procedure crSSLE.Elect($\{\boxed{x'_{u,i}}\}_{i\in[G]}, [\![\mathsf{msk}]\!]_k$)
30:     if $G \geq 128$ and $|\mathcal{W}_{\mathsf{cor}}| < G/2$ then
31:         $x_s = H(\{\boxed{x'_{u,i}}\}_{i\in[G]})$
32:         $\boxed{\bar{x}_u} \leftarrow \mathsf{tBFV.Eval}(+, \{\boxed{x'_{u,i}}\}_{i\in\mathcal{S}})$   ▷ $\mathcal{S}$ is encoded by $x_s$
33:     else
34:         $\boxed{\bar{x}_u} \leftarrow \mathsf{tBFV.Eval}(+, \{\boxed{x'_{u,i}}\}_{i\in[G]})$
35:     $[\![\bar{x}_u]\!]_k := \mathsf{tBFV.ParDec}(\boxed{\bar{x}_u}, [\![\mathsf{msk}]\!]_k)$
36:     $h_k = H([\![\bar{x}_u]\!]_k)$ and publish $h_k$
37:     After receiving $\{h_i\}_{i\in[G]}$, publish $[\![\bar{x}_u]\!]_k$
38: procedure crSSLE.Combine($\{[\![\bar{x}_u]\!]_i\}_{i\in[G]}$)
39:     $\bar{x}_u \leftarrow \mathsf{tBFV.FinDec}(\{[\![\bar{x}_u]\!]_i\}_{i\in[G]})$
40:     return $\bar{x}_u$
41: procedure crSSLE.Verify(pp, $\bar{x}_u, y, m$)
42:     return $\Pi_{\mathsf{MPRC}}.\mathsf{Verify}(\mathsf{pp}, \bar{x}_u, y, m)$
```

size is small).

2. (**Broadcast**) For each party k, invoke $[\![\bar{x}_u]\!]_{k,\mathsf{ep}_t} \xleftarrow{\$}$ $\mathsf{crSSLE.Elect}(\{\boxed{x'_{u,i}}\}_{i\in[G]}, [\![\mathsf{msk}]\!]_k)$.

   Broadcast $[\![\bar{x}_u]\!]_{k,\mathsf{ep}_t}$ (and potentially $H_3([\![\bar{x}_u]\!]_{k,\mathsf{ep}_t})$ if the party size is small).

3. For each party k, invoke $\bar{x}_{u,\mathsf{ep}_t} \leftarrow \mathsf{crSSLE.Combine}($ $\{[\![\bar{x}_u]\!]_{i,\mathsf{ep}_t}\}_{i\in[G]})$.

4. For each party k, evaluate $\mathsf{crSSLE.Verify}(\mathsf{pp}, \bar{x}_{u,\mathsf{ep}_t}, y_{k,\mathsf{ep}_t}, 0^N)$ and carry out the later steps if result equals to 1. For the elected leader, generate $(x_{k,\mathsf{ep}_{t+1}}, y_{k,\mathsf{ep}_{t+1}}, \cdot) \xleftarrow{\$} \mathsf{crSSLE.Gen}(\mathsf{pp}, \mathsf{mpk}, 1^\lambda, 0^N)$. It later broadcasts y_{k,ep_t} as the proof of the elected leader and $x_{k,\mathsf{ep}_{t+1}}$ as its own new identity.

   For all others, $x_{i,\mathsf{ep}_{t+1}} := x_{i,\mathsf{ep}_t}$.

Retroactive Detection Phase for instance ep_t **of** Qelect**:** (note that this phase will be executed only when no one claims itself as the elected leader in epoch ep_t)

1. For each party k, *broadcast* $\{\mathsf{rng}_{0,i,\mathsf{ep}_t}, \mathsf{rng}_{1,i,\mathsf{ep}_t}, \mathsf{rng}_{2,i,\mathsf{ep}_t}, y_{i,\mathsf{ep}_t}\}_{i\in[G]}$ to all others. Re-execute the *election phase* step 1 to reconstruct $\{\boxed{x'_{u,i}}\}_{i\in[G]}$ and compare them with the transcripts received from the pair-wise authenticated channel. Slash the corresponding party if any inconsistency is detected.

2. Based on $\{\boxed{x'_{u,i}}\}_{i\in[G]}$ reconstructed from all broadcast randomnesses in the previous step, reconstruct the ciphertext $\boxed{\bar{x}_u}$ on line 32 (or line 34, depending on the subset size). For each party k, generate the proof of the partial decryption in election instance ep_t as $\pi_{k,\mathsf{ep}_t} \leftarrow \mathsf{ZKP.Prove}(\mathsf{pp}, \mathsf{PK}, \boxed{\bar{x}_u},$ $[\![\bar{x}_u]\!]_{k,\mathsf{ep}_t}, [\![\mathsf{msk}]\!]_k)$ and broadcast π_{k,ep_t} to all via BC. Slash the corresponding party if a proof is invalid.

3. If all no inconsistency is identified and all proofs are valid, slash the party i if $\mathsf{crSSLE.Verify}(\mathsf{pp}, \bar{x}_u, y_{i,\mathsf{ep}_t}, 0^N) = 1$.

Proof of Theorem 6.3. Notice that the differences between our final protocol Qelect and the patched version depicted in Algorithm 3 are that: 1) the generation of $\boxed{\vec{\mathbf{u}}}$ are extracted into the preprocessing phase by leveraging a trusted key dealer, and 2) we consider elections taken place in multiple instances, and invoke a retroactive detection phase during the $(j+1)$-th election instance if the j-th election instance fails.

Both changes do not affect our security argument and thus Qelect achieves *fairness* and *unpredictability* by following the same argument as in Theorem C.2. We just prove that with the retroactive detection phase, the protocol either satisfy *uniqueness*, i.e., there will be one and only one party that successfully claims the leadership at the end, or identify a misbehavior and slashes the corresponding party.

Observer that in the retroactive detection phase, all parties first broadcast (via some reliable broadcast protocol BC) the randomnesses used in crSSLE.ParElect together with their commited witness y_i. This guarantees that $\boxed{x'_{u,i}}$ is honestly generated (i.e., not a malformed ciphertext with noise overflowed) and the adversary could not equivocate. In the second broadcast, all parties broadcast (via BC) the proofs of the partial decryptions and guarantees that all $[\![\bar{x}_u]\!]_{i\in[G]}$ are honestly generated and the adversary could not equivocate. Together, the final commitment $\bar{x}_u$ recovered locally from each party would either be consistent and valid (i.e., binding to some party's witness) across all parties, or a misbehavior (sending malformed ciphertext instead of $\boxed{x'_{u,i}}$ or incorrect partial decryption share to undermine the plaintext) should be identified with the corresponding party being slashed.

Thus, based on the wrong-key decryption property of MPRC, with a valid $\bar{x}_u$, we would have one and only one party being able to claim the leadership, while the probability of the adversary being able to come up with another witness that also passes the Π_{MPRC}.Verify is negligible.

To summarize, with the security patches and Claim 6.2, we achieve *fairness* and *unpredictability*, and with the retroactive detection phase, Qelect either achieve *uniqueness* or misbehaved parties would be punished by having their stake slashed. $\square$

C.4 Estimating Partial Decryption Runtime

In the formal protocol depicted in Appendix C.3, each party needs to generate a proof of the honest partial decryption in the retroactive detection phase if no leader is elected and the adversary could not be tracked by other ways. Based on our implementation details in § 7, the relation of partial decryption is defined as $R([\![\mathsf{msk}]\!], \mathsf{ct}, p) : p \leftarrow$ tBFV.ParDec(ct, $[\![\mathsf{msk}]\!]$). Concretely, we have $\mathsf{ct} := (a,b) \in \mathcal{R}^2_{Q'}, [\![\mathsf{msk}]\!] \in \{-1,0,1\}^N, p \in \mathcal{R}_{Q'}$, and the proof involves proving that $p = a[\![\mathsf{msk}]\!] + b + e + e_{\mathbf{sm}}$, where e is some small noise and e_{sm} is the large smudging noise which is around 40 bits under our parameter set [5], and $Q' \approx 120$ bits.

For a rough estimation, we refer to the work [13] that greatly improves the efficiency of the range proof of e_{sm}. For ring dimension $N = 2^{13} = 8192$, $Q' = 218$, each party needs 17 seconds to generate the proof and 18 seconds to verify [13, Table 2]. Based on the trend shown in [13, Table 4], for ring dimension $N = 2^{15} = 32768$, the proof generation and verification time would both blow up to > 160s. Therefore, for $G = 128$, naively each party needs 5.7 hours to finish the partial decryption check by verifying all others' proofs. Assuming honest majority, we could let each honest party randomly choose $3G/4$ and only verify their proofs, which gives us the probability of omitting the proof of that adversary who misbehaves to be $\frac{G/2}{(1/4)^{G/2}} = 1/2^{122}$. In this way, the runtime reduces to 4.3 hours.

D Plaintext Mismatch Issue for Homomorphic Randomness Generation

When plaintext modulus $q \neq G$, addition modulo G is not trivial, because the additions performing on tBFV ciphertexts are modulo q: if all parties sample $\vec{\mathbf{u}} \xleftarrow{\$} \mathbb{Z}_q^N$, the aggregated random value u would be in $\mathbb{Z}_q$.

The high-level idea of mapping it back to $\mathbb{Z}_G$ is to treat the plaintext modulus $q = K \cdot G + \mathsf{rm}$ as K chunks with rm leftover values. And we instantiate a vector $\{0,1,\ldots,K\cdot G\}$, encrypts it into ct_v and use ct_v to perform the homomorphic expansion procedure. The resulted ciphertexts $\{\mathsf{ct}_{b,i}\}_{i\in[G]}$ would have $\mathsf{ct}_{b,u}$ encrypt 1^N if $u \in [K\cdot G]$ and 0^N for the others; if $u \in [K\cdot G+1, q]$, then all of them would encrypt 0^N. After homomorphic randomized commitments aggregation, the final output would encrypt a valid commitment with probability $\frac{q-\mathsf{rm}}{q}$. Fortunately, for party size G to be also a power-of-two, this probability is $\frac{q-1}{q}$ and thus the failure probability of no one being elected is $\frac{1}{q} < \frac{1}{2^{16}}$ for $q = 65537$.

To ensure that there would exist at least one randomness mapped to $[G]$ with overwhelming probability ($\geq 1 - \mathsf{negl}(\lambda)$), instead of homomorphically expanding only the first slot of $\mathsf{ct}_{\mathsf{pl}}$, we let all parties expand the first $\bar{t}$ slots, s.t. $1/q^{\bar{t}} = \mathsf{negl}(\lambda)$. Thus, eventually there would be $\bar{t}$ final ciphertexts being published, out of which at least one would be a valid randomized commitment with overwhelming probability and others might be all zeros if the corresponding randomness is not within $[KG] \subseteq \mathbb{Z}_q$. Since the evaluation process of FHE circuit is deterministic, all parties would agree on the same set of the final $\bar{t}$ ciphertexts, they could then easily pick a single commitment based on some pre-determined ordering.

Regarding the SSLE protocol Qelect defined in § 6.2.2, if we apply the idea of letting the trusted authority directly prepare κ ciphertexts each encrypting a vector $\vec{\mathbf{u}} \in \mathbb{Z}_G^N$, we could avoid this $\bar{t}$ blow-up. Alternatively, with $q = 65537$ and $G < q$ being a power-of-two, the probability of having the aggregated u not able to map back to $\mathbb{Z}_G$ is $\frac{1}{2^{16}}$, which is already sufficient for most real-world applications. And we could simply accommodate the overhead of preparing a new u' if, with probability $1/2^{16}$, no one gets elected.