Relect: Single Secret Leader Election via FHE with Reduced Computation and Communication and Transparent Setup

Haofei Liang
Shanghai Jiao Tong University
lianghaofei@sjtu.edu.cn

Zeyu Liu
Yale University
zeyu.liu@yale.edu

Yunhao Wang
Yale University, IC3
yunhao.wang@yale.edu

Xiang Xie
East China Normal University
xiexiangiscas@gmail.com

Yu Yu
Shanghai Jiao Tong University
yyuu@sjtu.edu.cn

Fan Zhang
Yale University, IC3
f.zhang@yale.edu

October 15, 2026

Abstract

In a single secret leader election (SSLE) protocol, all parties collectively and obliviously elect one leader. Parties other than the selected leader should not be able to learn the identity of the leader unless it is revealed by the leader itself. The problem is first formalized by Boneh *et al.* (AFT 2020), and the first concretely feasible lattice-based SSLE with proof-of-concept implementations, Qelect, was recently introduced by Wang and Zhang (USENIX 2025).

In this work, we present Relect, an efficient SSLE protocol, based on the Ring Learning with Error assumption. We build it by leveraging the algebraic structure of the underlying threshold Fully Homomorphic Encryption (FHE) and by designing tailored homomorphic circuits. Compared to prior works, Relect (1) achieves substantially higher efficiency and (2) removes the strong environment assumption in Qelect (a trusted setup), while allowing dynamic leader selection for each round.

Concretely, for $32 - 2048$ parties, our local FHE computation runtime (a major efficiency bottleneck for SSLE) achieves $7.15 - 42.4\times$ faster than Qelect for a single thread and $7.10 - 48\times$ faster for 16 threads. Furthermore, we show that for the same parameters, our communication cost is also $1.14 - 2\times$ smaller. As mentioned, this is achieved while removing the trusted setup.

In terms of end-to-end runtime, following Qelect, we tested $2 - 128$ parties. We show that under the LAN setting, Relect is $2.77 - 345\times$ faster than Qelect per round. Under the WAN setting, Relect is 1.94 to $17.2\times$ faster than Qelect. Note that these performance gains are all achieved while removing the trusted assumption and achieving dynamic leader selection for each round.

1 Introduction

Selecting leaders among participating nodes is a core component of blockchain systems [49, 59]. The leader selection process should satisfy three fundamental properties: *fairness*, *uniqueness*, and *secrecy*.

Fairness [7] ensures that every node has an equal chance of being elected, even in the presence of malicious parties. Uniqueness guarantees that exactly one leader is elected per round, preventing ambiguity. Secrecy (or unpredictability) ensures that the leader's identity remains unknown to the public until the leader voluntarily reveals it.

Achieving all three properties simultaneously is challenging. Existing approaches often compromise at least one: for instance, VRF-based sortition [29] may violate uniqueness, while Ethereum's randomness beacon exposes the leader in advance, compromising secrecy. This lack of secrecy can easily enable denial-of-service (DoS) attacks, as adversaries who know the leader in advance can prevent them from performing their duties.

These challenges motivate the study of *Single Secret Leader Election* (SSLE) protocols [26, 4, 29], which allow participants to collectively produce an output where only the elected leader can determine their role, thus mitigating preemptive attacks. Recent SSLE constructions exploit diverse cryptographic primitives [5, 16, 11, 24, 3, 7, 10, 36, 53].

However, the most practical SSLE schemes [5, 11, 10, 36] rely on classical assumptions (e.g., DDH) that are vulnerable to quantum attacks. Since SSLE is deployed in highly adversarial environments, post-quantum security is essential, particularly in light of NIST's standardization efforts [50].

To achieve post-quantum security, prior work has leveraged fully homomorphic encryption (FHE), a powerful primitive introduced in [52] and realized in [28]. FHE has already been widely applied in practice-oriented settings, such as privacy-preserving machine learning [37, 34, 38, 35, 18], secure data processing [27], private information retrieval [2, 47, 22, 63, 32], private set intersection [13, 12, 17], and oblivious message retrieval [43, 44, 45, 39, 42, 40, 23].

Although lattice-based and FHE-based post-quantum SSLE schemes have been proposed [7, 24] and in [5, Section 5], they remain largely theoretical and lack practical implementations. The first practical post-quantum SSLE scheme, Qelect [58], was introduced only recently. It relies on a heavy trusted setup and employs the BFV homomorphic encryption scheme [21, 8], leading to substantial computation and communication overhead per participant.

In this paper, we present Relect, a highly optimized lattice-based SSLE scheme. Our construction eliminates the need for a heavy trusted setup and significantly reduces both computation and communication costs compared to Qelect. Notably, Relect does not rely on black-box FHE constructions such as BFV [21, 8] or TFHE [14]. Instead, we directly exploit the algebraic structure of RLWE [46] and RGSW [48], enabling novel design techniques that avoid the expensive homomorphic operations inherent in black-box approaches.

1.1 Related Works

The SSLE problem has been approached from several directions. Boneh et al. [5] describe three possibilities. One relies on indistinguishability obfuscation to hide the election functionality, essentially producing a random token under the chosen leader's public key. Another employs threshold FHE: a block cipher is homomorphically evaluated to generate randomness, which is then combined with all candidate tokens via an inner product. The third, known as BEGH and by far the most influential, is built on the DDH assumption. In BEGH, each participant posts a Diffie-Hellman commitment, the set of commitments is shuffled, and a randomness beacon points to the winner, who then opens the corresponding commitment. Among the three, only the last one can potentially be employed in practice in terms of concrete efficiency.

Later work sharpened the security landscape of SSLE. The universally composable treatment of [10] reinterprets the original game-based framework and achieves improved on-chain efficiency through PKE with keyword search, instantiated under the symmetric external Diffie-Hellman assumption. In parallel, the adaptive-corruption model of [11] revisits BEGH with two lists of commitments: one for shuffling, the other for secret updates, preventing an adversary from learning across rounds even after corruption.

A different line of research examines non-uniform stake distributions. The MPC-based approach of [3] realizes SSLE through oblivious selection, requiring only logarithmic rounds in the committee size and incurring logarithmic overhead in the total stake. Homomorphic sorting [24] provides almost no additional overhead, but at the expense of heavy FHE circuits that must support comparison, selection, and domain transformation.

The post-quantum adaptation of BEGH proposed in [7] reframes the scheme as a re-randomizable commitment (RRC) combined with a shuffling protocol. Their instantiation builds RRC from lattices under the (R)LWE assumption. A key distinction, however, is efficiency: whereas BEGH requires just one shuffle per round, the lattice-based variant demands $O(G)$ sequential shuffles, reflecting the noise growth intrinsic to (R)LWE.

There are exploratory directions outside lattices. A recent blog post [53] suggests an analogue of Whisk based on commutative supersingular isogenies [9], which are considered strong candidates for post-quantum security.

The state-of-the-art lattice-based SSLE construction is Qelect: It realizes SSLE via threshold FHE and the so-called *multi-party randomizable commitment* (MPRC). MPRC enables users to compute a commitment c to message m; others can locally randomize c into c' non-interactively; multiple such c' can be combined into a single commitment c'' to m. Besides, c'' is computationally unlinkable to the original c if sufficient entropy is introduced during randomization. This MPRC is realized using the RLWE encryption system, which we recall in § 3.1. While this work makes a step towards making lattice-based SSLE practical, there are several limitations:

- The local runtime of Qelect is quite large. With 2048 parties, it takes > 18 seconds for each party to perform homomorphic computation.
- It takes $\sim$4 GB of online communication.
- It requires a trusted setup during public parameter generation. This setup phase by itself raises a few concerns:
  - It can be a privacy concern for real-world deployment, since the key is generated via a centralized trusted party.
  - Additionally, in Qelect, the trusted setup determines the leader selections *in advance* for a batch of rounds. Their identities remain hidden until the respective leaders reveal them [58, Section 6.2.3].

In this work, we greatly reduce these costs and remove such a trusted setup.[1]

1.2 Our Contribution

We propose Relect, a Ring Learning with Error (RLWE)-based secret single leader election protocol. Relect utilizes FHE in a non-black-box way and exploits the algebraic structures of the underlying rings, such that the homomorphic evaluation of the leader election circuit can be performed efficiently. We have implemented Relect (publicly available [41]) and show its concrete efficiency. In particular,

- Relect achieves 7.15× to 42.4× faster local FHE computation time for a single thread and 7.10 to 48.0× faster for 16 threads than the state-of-the-art Qelect [58].
- The communication cost of Relect is smaller (1.14 to 2× smaller) than Qelect [58].
- In terms of end-to-end time,
  - (LAN setting) For 2 to 128 parties, Relect is 2.77 to 345 times faster than Qelect. Moreover, in Qelect, FHE computation is the major efficiency bottleneck, accounting for $70\% - 99\%$ of the total runtime. However, for Relect, communication becomes the new bottleneck, accounting for up to 84% of the total runtime. Furthermore, we show that by replacing the pairwise communication with recursive-doubling communication, the total runtime can be further reduced by 30%.
  - (WAN setting) For 2 to 128 parties, Relect is 1.94 to 17.2 times faster than Qelect. In this setting, for both schemes, communication becomes the main bottleneck. However, since we have reduced the communication cost and the FHE computation still takes a reasonable portion of the total time, we achieve such speedup. Note that for WAN settings with better latency, throughput, and/or communication algorithms, Relect can show advantages similar to the LAN setting, due to our significant improvement in terms of the FHE computation costs.
- We remove the need for a trusted setup in Qelect, greatly reducing the environment assumption, and achieve the flexibility of selecting leaders dynamically in each round, while maintaining our efficiency advantages stated above.

[1] A recent independent and concurrent work [19] explores the so-called *state uniqueness* property of SSLE and is orthogonal to our focus.

2 Technical Overview

This section provides a high-level overview of Relect.

Threat model. We follow the same threat model as [58]: there are G parties and the computationally bounded adversary statically corrupts up to f of them, where $f < \frac{1}{2}G$ and $f < \tau$ for decryption threshold τ. The adversary can view all the public information, including the keys and communication among parties. Note that our main construction considers semi-honest adversaries, same as Qelect [58]. In [58], the authors discuss general ways to achieve malicious security and we recall them in § 4.6. Since these are applicable to our Relect as well, we focus on the semi-honest case.

Overall paradigm. The overall paradigm of our SSLE construction works as follows:

- Setup: The generation of public parameters, the master public key and the master secret key shares.
- Instance generation: Each party k generates a commitment com_k representing its identity, together with the corresponding key pair $(\mathsf{pk}_k, \mathsf{sk}_k)$, the random seed s_k for generating sk_k, and its *random* selection r_k.
- The main election procedure
  - **Partial Election**: Each party, using the public component of the instance generated by all the parties, locally generates the selection of the leader.
  - **Election:** Each party, using each locally generated selection, generates a final selection share.
  - **Combine:** Each party combines the shares to recover the final commitment com''_r representing the selected leader (but remains unknown to parties without the correct key).
- Verify: Given the final commitment com''_r, any party can verify whether a party is selected, given the random seed of the corresponding secret key.

These steps follow the interface for SSLE (defined in [58, Definition 6.1]), designed to guarantee that the whole process has constant round communication. Note that, while following the interfaces, these high-level steps in our paradigm *already* differ from [58]: specifically, in Qelect [58], the trusted setup prepares a batch of encrypted random selections, fixing the corresponding leader identities in advance. However, in our paradigm, we allow each party to generate r_k itself during instance generation. In [58], they work over the BFV homomorphic encryption scheme with a prime plaintext modulus. Since this modulus differs from G when G is a power of two, homomorphic addition alone does not directly implement addition modulo G. Looking ahead, we circumvent this issue by generating the random selection in the exponent in a power-of-two ring. This is a core step towards removing the trusted setup: With this change, as long as the underlying threshold FHE (ThFHE) scheme we use does not require a trusted setup, we can remove the trusted setup requirement. We discuss in more detail below how we instantiate this paradigm.

Fig. 1 shows the workflow of each party $k \in [G]$ for Relect when the parties count $G = 4$. We will clarify each step in more detail below.

2.1 Setup

During the setup phase of Relect, parties jointly run the setup phase for threshold fully homomorphic encryption (ThFHE), generating:

- A master public key mpk
- Secret key shares $[\![\mathsf{msk}]\!]_k$ held individually by each party k (which guarantees that only if at least τ (threshold) of all the parties participate, a ciphertext under mpk can be decrypted)

The mpk is subsequently utilized to perform all necessary computations for the SSLE scheme, while $[\![\mathsf{msk}]\!]_k$ enables distributed decryption of the final ciphertext. A comprehensive description of the ThFHE construction is provided in § 3.3.

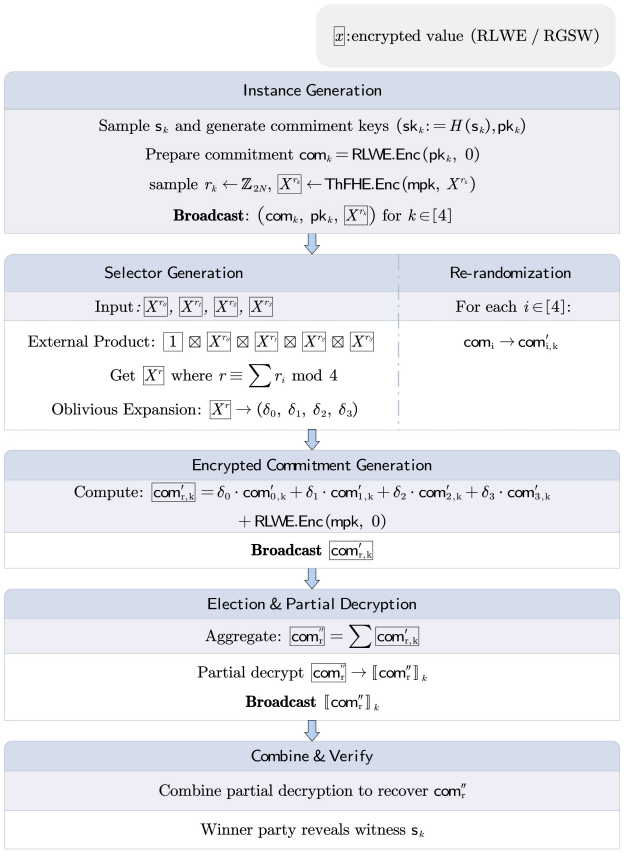


Figure 1: The (simplified) workflow of Relect for party k assuming $G = 4$. The zero-encryption term denotes two independent encryptions of zero, one added to each output ciphertext.

2.2 Instance Generation

First, each party k generates a commitment com_k which is an encryption of 0 under party k's public key pk_k. The corresponding secret key sk_k (not its master key share $[\![\mathsf{msk}]\!]_k$) is generated with a random seed s_k. Looking ahead, this commitment is used to decide who is elected as the leader, and is *opened* by revealing the corresponding random seed: any other (randomly sampled) secret key generated via some other seed cannot be used to decrypt such a public key encryption (PKE) ciphertext to 0 except for negligible probability (guaranteed by the wrong-key-decryption property of the PKE scheme § 3.1). Then, com_k is published together with its corresponding public key pk_k for subsequent steps in Relect. Lastly, party k should also publish a ciphertext ct_k under mpk, used to select the leader later.

More specifically, it should generate some randomness $r_k \xleftarrow{\$} [G]$ and $(\sum_k r_k) \bmod G \in [G]$ represents who is selected as the leader in this round. Naively, to publish this randomness privately (while facilitating future computation), each party k could simply encrypt it, i.e., computing $\mathsf{ct}_k \leftarrow \mathsf{ThFHE.Enc}(\mathsf{mpk}, r_k)$. However, to enable our efficient FHE circuit design, we discuss how we generate ct_k differently below.

2.3 The Main Election Procedure

This phase is the most technically challenging phase: we need to design this phase to be as efficient as possible in terms of computation and communication.

2.3.1 Partial Election

Each party's local randomness. The very first step is to allow all parties to jointly generate some randomness $r \in [G]$ deciding who is elected as the leader (secretly).

To do this in a distributed way, naively, as mentioned above, each party k could simply generate a random value r_k and encrypt it $\mathsf{ct}_k \leftarrow \mathsf{ThFHE.Enc}(\mathsf{mpk}, r_k)$. Then all parties share all $\{\mathsf{ct}_k\}_{k\in[G]}$ and accumulated to obtain $\mathsf{ct} \leftarrow \sum_{k\in[G]} \mathsf{ct}_k$, which can be decrypted to $r \equiv \sum_{k\in[G]} r_k \mod G$.

However, this intuitive approach poses additional challenges to the following steps: Looking ahead, we need to use r to select com_r (and then re-randomize it such that no one knows who is selected), representing that party r is selected (secretly). In that step, we need the "selector" ciphertexts $\mathsf{sel}_0, \ldots, \mathsf{sel}_{G-1}$ to be such that sel_r encrypts 1 (under $\mathsf{ThFHE.msk}$) while the other ciphertexts encrypt 0, such that later we can use this vector to "select" com_r by a simple inner-product.

Thus, instead of using ThFHE as a black-box, we leverage the underlying algebraic property. To our knowledge, all the existing ThFHE constructions are based on RLWE. Specifically, all of the operations work over ring $\mathcal{R} := \mathbb{Z}[X]/\Phi(X)$ where $\Phi(X)$ is usually a cyclotomic polynomial of degree N being a power of two (i.e., $\Phi(X) := X^N + 1$ for some power-of-two N). All the ThFHE ciphertexts encrypt some message $m \in \mathcal{R}_t$ where t is the plaintext modulus (where $\mathcal{R}_t$ means that all the elements in $\mathcal{R}$ are represented as polynomials with coefficients being $\mathbb{Z}_t$).

For simplicity, first assume $G = N$. Then, instead of generating $\mathsf{ct}_k \leftarrow \mathsf{ThFHE.Enc}(\mathsf{mpk}, r_k)$, each party k generates $\mathsf{ct}_k \leftarrow \mathsf{ThFHE.Enc}(\mathsf{mpk}, X^{r_k})$ where $r_k \in [G] = [N]$ (i.e., encrypting the monomial $X^{r_k} \in \mathcal{R}_t$). Instead of computing $\mathsf{ct} \leftarrow \sum_{k\in[G]} \mathsf{ct}_k$, we compute $\mathsf{ct} \leftarrow \prod_{k\in[G]} \mathsf{ct}_k$. Now, ct encrypts X^r by the nature of operations in $\mathcal{R}$ (note that it is also possible to obtain $-X^r$ instead, but to keep the overview simple, assume that we always get X^r and we handle $-X^r$ in § 4,[2] since this negative sign may leak information).

The partial selection. The next step is to select the correct com_r and homomorphically re-randomize it. We start with the re-randomization: for each com_j for $j \in [G]$, each party k generates $\mathsf{com}_{j,k} \leftarrow \mathsf{com}_j + \mathsf{Enc}(\mathsf{pk}_j, 0)$, which is again a ciphertext encrypting 0 under sk_j.

Now, after obtaining all the $\mathsf{com}_{j,k}$, each client needs to obtain the "selector" ciphertexts, $\mathsf{sel}_0, \ldots, \mathsf{sel}_{G-1}$, where sel_r encrypts 1 while others encrypt 0. Luckily, with $\mathsf{Enc}(X^r)$, using *oblivious expansion*, we can obtain $\mathsf{sel}_0, \ldots, \mathsf{sel}_{G-1}$ easily (more details in § 3.2.3). Now, with the selectors, each party k can simply compute $\mathsf{ct}'_k \leftarrow \sum_{j\in[G]} \mathsf{sel}_j \cdot \mathsf{com}_{j,k}$ (i.e., an inner-product between $(\mathsf{sel}_0, \ldots, \mathsf{sel}_{G-1})$ and $(\mathsf{com}_{0,k}, \ldots, \mathsf{com}_{G-1,k})$), which encrypts $\mathsf{com}_{r,k}$. Of course, here, we need the commitment to be composed of $\mathcal{R}_t$ elements to allow such homomorphic operations to be possible. This is simple since we can simply match the RLWE ciphertext space with $\mathcal{R}_t$.

Here, ct'_k consists of two RLWE ciphertexts, one for each polynomial of $\mathsf{com}_{r,k}$. To hide the local commitment contributions in the published outputs, party k also randomizes these two ciphertexts by adding a fresh, independent encryption of 0 under mpk to each before broadcasting. This does not change their encrypted messages. The result ct'_k is broadcast, serving as the input to the next step.

2.3.2 Election

Upon receiving ct'_i from other parties $i \in [G] \setminus \{k\}$, each party k aggregates all $\{\mathsf{ct}'_i\}_{i\in[G]}$ componentwise to get $\mathsf{ct}'' \leftarrow \sum_{i\in[G]} \mathsf{ct}'_i$.

Then, each party partially decrypts the two ciphertexts separately via ThFHE (using their master secret key share of this ThFHE) and broadcasts their partial decryptions. Note that every party obtains the same ct'', so the partial decryptions are with respect to the same ciphertext pair.

2.3.3 Combine

Combining is simple. Upon receiving the partial decryptions, each party locally combines these partial decryptions componentwise to recover $\mathsf{com}'' \leftarrow \mathsf{ThFHE.Dec}(\mathsf{ct}'')$.

[2] In short, we sample in $[2N]$ instead of $[N]$

2.4 Verify

The resulting ciphertext com'' possesses the following properties:

- Decryption with party r's secret key yields 0
- Decryption attempts by other parties produce 0 with negligible probability.

Now, the party r knows it is the secret leader, and other parties know that they are not leaders but do not know who the leader is. To declare the leadership, party r simply publishes the random seed s_r for generating sk_r and all the other parties can verify whether this declaration is valid by decrypting com'' themselves with the secret key generated with seed s_r. If the decryption is 0, the declaration is valid.

This conceptually concludes our Relect constructions.

Since the secret randomness of the leader's commitment is revealed in this round, the leader needs a fresh instance for later rounds. In the construction in § 4, every party refreshes its instance by resampling its witness and generating a new key pair, commitment, and election ciphertext; these instances may be generated in advance.

2.5 Core Optimizations for Concrete Efficiency

For $G \ll N$. The discussion above assumes $G = N$. However, if $G \ll N$, the efficiency of the scheme above is more limited. In practice, this case is possible, this N is usually a quite large number (e.g., 2^{11} or 2^{12}), while G may be small (e.g., ≤ 128 as benchmarked in [58]). Thus, we now discuss an optimization for $G \ll N$ (but $G|N$, which are the mainstream parameters used in practice [59] and in prior works [58]).

The inefficiency mainly comes from the oblivious expansion process (which involves $\Omega(N)$ ring operations, taking $\omega(N^2)$ time in total) done naively above. To avoid this inefficiency, first, instead of computing $\prod_{i\in[G]} \mathsf{ct}_i$ to obtain an encryption of X^r, we compute $\mathsf{ct}_0 \times \cdots \times \mathsf{ct}_{G-1} \times u(X)$ where $u(X) = (1 + X^G + X^{2G} + \cdots + X^{N-G})$.[3] This result in an encryption of $(X^r + X^{r+G} + \cdots + X^{r+N-G})$, instead of X^r (again, technically, the result is $\pm X^r \pm X^{r+G} \pm \cdots \pm X^{r+N-G}$, but we ignore the negative signs in the overview for simplicity). In this case, when we perform oblivious expansion, we only need to perform it over the first G terms, and the selectors sel_r encrypts $u(X)$ instead of 1. This makes the oblivious expansion only cost $\Omega(G)$ ring operations instead of $\Omega(N)$ operations.

However, this introduces a new issue. Naively, one could only encode $\log(t) \cdot G$ bits instead of $\log(t) \cdot N$ bits (where t is the plaintext modulus), which means that the it takes $|\mathsf{com}_j|/(\log(t) \cdot G)$ ciphertexts to encode a single commitment. To avoid this inefficiency, we observe that such $u(X)$ is invertible, which means that one could treat an encryption of $u(X)$ as an encryption of 1. The final decryption output is $u(X) \cdot \mathsf{com}'_r$ instead of com'_r, but since $u(X)$ is invertible, every party can recover com'_r locally. We discuss this in more detail in § 4.5.

For $G \gg \lambda$. We introduce an additional optimization for $G \gg \lambda$ for security parameter λ (e.g., 128). We observe that since Qelect requires an honest majority requirement in general [58], we can perform a subset sampling. Specifically, instead of computing $\mathsf{ct} \leftarrow \prod_{k\in[G]} \mathsf{ct}_k$ as in § 2.3.1, we uniformly select a set $S \subset [G]$ without replacement such that $|S| = \lambda$, where the randomness is obtained by hashing all the ct_k's for $k \in [G]$ (and viewing this hash as a random oracle). Then, by honest majority, at least one selected party is honest with probability $> 1 - 2^{-\lambda}$. Thus the selected inputs include an honest party's random choice except with negligible probability; all G parties remain candidates. Note that the subset has to be chosen by hashing all the ciphertexts sent by all the parties, since, otherwise, it is unclear how all the parties can agree on the same randomness during each round instead of in advance. Note that this optimization does not directly apply to [58] since they do not have this product step.

[3] Note that $u(X)$ can be encoded as an RLWE ciphertext, and ct_i can all be RGSW ciphertexts, so the product optimization above remains functional.

3 Preliminary

Notations. The column vectors over $\mathbb{Z}$ is denoted with bold lower-case letters such as $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{x}$. For a vector $\boldsymbol{x}$, we use $\boldsymbol{x}[i]$ to represent the i-th element of this vector. For $q \in \mathbb{Z}$, we identify $\mathbb{Z}_q$ with the set $[-q/2, q/2) \cap \mathbb{Z}$ by default. We use $\mathbb{Z}_q^n$ to define the set of the vectors with all elements in $\mathbb{Z}_q$ and length n. Let $[n]$ denote the set $\{0, \ldots, n-1\}$, $[a, b]$ denote the set $\{a, a+1, \ldots, b-1, b\}$, and $[a, b)$ denote the set $\{a, a+1, \ldots, b-1\}$. 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 $\boldsymbol{a}$ where $\boldsymbol{a}[i] := a_i$. We use $\|$ (concatenation) to concatenate vectors, i.e. $\boldsymbol{a} \| \boldsymbol{b}$.

Let $\mathcal{D}$ denote an arbitrary distribution and χ_σ denote the discrete Gaussian distribution with mean being 0 and standard deviation being σ. 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.

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.

3.1 Ring Learning With Errors

RLWE assumption. We recall the standard decision ring learning with error (RLWE) assumption [46]. 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. RLWE-based public-key encryption [46], denoted RLWE, contains four algorithms 1) $\mathsf{RLWE.GenParams}(1^\lambda, t)$ which returns the public parameter $\mathsf{pp}_{\mathsf{RLWE}}$ which includes the ring dimension n of RLWE, the ciphertext modulus q, the secret key distribution $\mathcal{D}$ and the error gaussian distribution χ_σ, 2) $\mathsf{RLWE.KeyGen}(\mathsf{pp}_{\mathsf{RLWE}})$ which outputs a key pair $(\mathsf{sk}, \mathsf{pk})$, 3) $\mathsf{RLWE.Enc}(\mathsf{pp}_{\mathsf{RLWE}}, \mathsf{pk}, m)$ which outputs a ciphertext $\mathsf{ct} := (a, b)$ encrypting the plaintext message m under the public key pk, and 4) $\mathsf{RLWE.Dec}(\mathsf{pp}_{\mathsf{RLWE}}, \mathsf{sk}, \mathsf{ct})$ which decrypts ct into m with secret key sk.

The RLWE encryption of the message $m \in \mathcal{R}_t$ is of the form: $\mathsf{RLWE}_s^{t/q}(m) = (a,\ b) = (a,\ as + \frac{q}{t}m + e)$, where $s \in \mathcal{R}$ is the RLWE secret key, $a \xleftarrow{\$} \mathcal{R}_q$ and $e \leftarrow \chi_\sigma^N$.

Two properties. There are two additional properties of RLWE we leverage.

- Rerandomizability: Given a public key $\mathsf{pk} = (c, d = cs + e')$ for $c \xleftarrow{\$} \mathcal{R}_q, e \leftarrow \chi_\sigma^N$, and a ciphertext ct as above, $\mathsf{ct}' \leftarrow \mathsf{ct} + \mathsf{RLWE.Enc}(\mathsf{pk}, 0)$ encrypts the same message as ct. However, $(\mathsf{ct}', \mathsf{ct}) \approx_c (\mathsf{ct}', \mathsf{ct}'')$ where ct'' is a random ciphertext.
- Wrong-key-decryption: Given a ciphertext ct generated honestly with secret key sk, decrypting under an independently generated sk' results in 0 with probability at most $p_0 + \mathsf{negl}(\lambda)$ under the corresponding RLWE assumptions. Here, $p_0 = (|D_0|/q)^n$, where D_0 is the set of residues decoded to zero in one coefficient; for odd q and nearest-integer decoding, $|D_0| = 2\lfloor q/(2t) \rfloor + 1$. For a uniform ciphertext, $b - a\mathsf{sk}'$ is uniform for any fixed sk', so p_0 is the probability that all n coefficients decode to zero. The parameters are required to make p_0 negligible in λ; the additional $\mathsf{negl}(\lambda)$ term accounts for the RLWE replacement.

3.2 Homomorphic Operations

This section surveys the homomorphic operations needed by Relect.

3.2.1 Ciphertext Multiplications via RGSW

RGSW is a lattice-based FHE scheme operating over polynomial rings. It supports efficient homomorphic operations over both addition and multiplication. RGSW natively implements the **external product** (denoted $\boxtimes$):

$$\mathsf{RGSW}(m_1) \boxtimes \mathsf{RLWE}(m_2) \to \mathsf{RLWE}(m_1 \cdot m_2)$$

allowing multiplicative homomorphism between RGSW and RLWE ciphertexts. For the signed monomial multipliers used in this work, this operation achieves **quasi-additive noise growth**, critical for deep circuit evaluation. In this work, we leverage RGSW as a "black-box" for external products, abstracting its internal structure while utilizing its homomorphic capabilities. For more information about RGSW, please see [48].

3.2.2 Homomorphic Automorphism

Given an RLWE ciphertext $\mathsf{ct} = \mathsf{RLWE}(m(X))$, the homomorphic automorphism $\mathsf{FHE.HomAuto}(\mathsf{ct}, t)$, for odd t, outputs $\mathsf{ct}' = \mathsf{RLWE}(m(X^t))$. We omit the automorphism key for this operation for ease of reading. More information about this operation can be found in Theorem 5.3.

3.2.3 Oblivious Expansion

Given an RLWE ciphertext ct with odd ciphertext modulus Q which encrypts a polynomial m, and an integer $1 \leq c \leq N$ where $\{m_i\}_{i\in[N],i\geq c}$ are zeros, oblivious expansion $\mathsf{ObliviousExpand}(\mathsf{ct}, c)$ will output a vector elems. The elems contain c RLWE ciphertexts where $\mathsf{elems}[i]$ encrypts m_i. The full algorithm of oblivious expansion is shown in *Algorithm* 1.

Algorithm 1 (Partial) Oblivious Expansion.
ObliviousExpand/PartialObliviousExpand

Input: A ciphertext $\mathsf{ct} = \mathsf{RLWE}(m)$ with odd ciphertext modulus Q, and an integer $1 \leq c \leq N$
Output: A vector of c RLWE ciphertexts

Let d be the smallest power of two such that $d \geq c$.
$\mathsf{ct}' \leftarrow \mathsf{ct} \cdot (d^{-1} \mod Q)$
$\mathsf{elems} \leftarrow [\mathsf{ct}']$
for $i = 0$ to $\log(d) - 1$ **do**
    for $j = 0$ to $2^i - 1$ **do**
        $\mathsf{ct}_0 \leftarrow \mathsf{elems}[j]$
        $\mathsf{ct}_1 \leftarrow \mathsf{FHE.HomAuto}(\mathsf{ct}_0, N/2^i + 1)$
        $\mathsf{ct}'_j \leftarrow \mathsf{ct}_0 + \mathsf{ct}_1$
        $\mathsf{ct}'_{j+2^i} \leftarrow (\mathsf{ct}_1 - \mathsf{ct}_0) \cdot X^{N-2^i}$
    $\mathsf{elems} \leftarrow [\mathsf{ct}'_0, \ldots, \mathsf{ct}'_{2^{i+1}-1}]$
return the first c entries of elems

3.2.4 Partial Oblivious Expansion

If the condition that $\{m_i\}_{i\in[N],i\geq c}$ are zeros is not satisfied, continuing to perform the above operations will yield different outputs. We define this operation as PartialObliviousExpand.

Given an RLWE ciphertext ct which encrypts a polynomial m and an integer $1 \leq c \leq N$, where c is a power of two, partial oblivious expansion $\mathsf{PartialObliviousExpand}(\mathsf{ct}, c)$ will output a vector elems. The elems contain c RLWE ciphertexts where $\mathsf{elems}[i]$ encrypts $\sum_{j\in[N/c]} m_{i+j\cdot c} X^{j\cdot c}$. Algorithm 1 also shares the partial oblivious expansion.

3.3 Threshold Fully Homomorphic Encryption

We follow the definition of ThFHE [6], summarized in the level of details that are sufficient to understand our paper. The algorithm $\mathsf{GenParams}$ is added for completeness.

Definition 3.1. (Threshold Fully Homomorphic Encryption (ThFHE)). A $\tau-$threshold fully homomorphic encryption scheme is a tuple of PPT algorithms $\mathsf{ThFHE} = (\mathsf{GenParams}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{PartialDec}, \mathsf{Combine})$ with the following properties: Here $1 \leq \tau \leq G$ is the minimum number of shares required for decryption, and $f < \tau$ is the maximum number of corrupted parties.

1. $\mathsf{GenParams}(1^\lambda, T, G, \tau) \to \mathsf{pp}$: On input of the security parameter λ, message modulus T, the number of parties G and the threshold τ, a set of threshold FHE parameters $\mathsf{pp} = (\lambda, G, \tau, N, T, Q, \mathcal{D}_s, \chi_\sigma)$ is generated, where:

   - N denotes the lattice dimension parameter
   - Q represents the modulus defining the ciphertext space
   - $\mathcal{D}_s$ and χ_σ are the secret key and error distributions respectively

   We omit the input depth bound D of the circuit C, because C is a fixed circuit with fixed depth for Relect.

2. $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]})$: Given the threshold FHE parameters pp, this algorithm outputs:

   - A master public key mpk
   - Master secret key shares $\{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}$, where $[\![\mathsf{msk}]\!]_i$ is securely distributed to party i

   with the security guarantee that the complete master secret key msk remains hidden from any coalition of at most f parties.

3. $\mathsf{Enc}(\mathsf{mpk}, m) \to \mathsf{ct}$: Given a plaintext message $m \in \mathcal{R}_T$ as input, this algorithm encrypts m using the master public key mpk, outputting the corresponding ciphertext ct.

4. $\mathsf{Evaluate}(C, \{\mathsf{ct}_i\}_{i\in I}) \to \mathsf{ct}'$: Given a circuit C with the depth less than D, and ciphertexts $\{\mathsf{ct}_i\}_{i\in I}$ encrypting plaintexts $\{m_i\}_{i\in I}$ respectively, this algorithm homomorphically evaluates C to output a ciphertext ct' satisfying: $\mathsf{Dec}(\mathsf{msk}, \mathsf{ct}') = C(\{m_i\}_{i\in I})$ where msk denotes the master secret key corresponding to the public parameters.

5. $\mathsf{PartialDec}([\![\mathsf{msk}]\!]_i, \mathsf{ct}) \to [\![m]\!]_i$: Given a ciphertext ct which encrypts message m and the master secret key share $[\![\mathsf{msk}]\!]_i$, party i performs the partial decryption and outputs the message share $[\![m]\!]_i$.

6. $\mathsf{Combine}(\{[\![m]\!]_i\}_{i\in S}) \to m$: Given the message shares $\{[\![m]\!]_i\}_{i\in S}$, where $S \subset [G]$ and the parties count of S is at least τ (threshold), these parties will combine $\{[\![m]\!]_i\}_{i\in S}$ and outputs message m.

A ThFHE scheme also needs satisfy the following properties(see the original [6] for more details about other properties):

- *Correctness*: For $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}) \leftarrow \mathsf{ThFHE.KeyGen}(\mathsf{pp})$, for any $\{m_i\}$ and any circuit C with depth $\leq D$, let $\mathsf{ct}_i \xleftarrow{\$} \mathsf{ThFHE.Enc}(\mathsf{mpk}, m_i)$, and $\mathsf{ct}' \leftarrow \mathsf{ThFHE.Evaluate}(C, \{\mathsf{ct}_i\})$, denote $\mathcal{M} := \{[\![m']\!]_i | [\![m']\!]_i \leftarrow \mathsf{ThFHE.PartialDec}([\![\mathsf{msk}]\!]_i, \mathsf{ct}')\}$, for any $\mathcal{M}' \subset \mathcal{M}$, s.t. $|\mathcal{M}'| \geq \tau$, we have: $\Pr[\mathsf{ThFHE.Combine}(\{[\![m']\!]_i\}_{i\in\mathcal{M}'}) = C(\{m_i\})] \geq 1 - \mathsf{negl}(\lambda)$.

- *Semantic security*: For $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}) \leftarrow \mathsf{ThFHE.KeyGen}(\mathsf{pp})$, for any PPT adversary $\mathcal{A}_1, \mathcal{A}_2$ and corrupted set $\mathcal{W}_{\mathsf{cor}}$ fixed before key generation, with $|\mathcal{W}_{\mathsf{cor}}| \leq f < \tau$,
  let $(m_0, m_1, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in\mathcal{W}_{\mathsf{cor}}})$, let $b \xleftarrow{\$} \{0,1\}$ and $\mathsf{ct}' \xleftarrow{\$} \mathsf{ThFHE.Enc}(\mathsf{mpk}, m_b)$, it holds that: $\left|\Pr[\mathcal{A}_2(\mathsf{st}, \mathsf{ct}', \{[\![\mathsf{msk}]\!]_i\}_{i\in\mathcal{W}_{\mathsf{cor}}}) = b] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$.

3.4 SSLE

We first present a definition for Constant-round Single Secret Election, adapted from [58] to our static semi-honest model with an honest majority. We use this definition since it explicitly restricts the communication round to be constant.

Definition 3.2 (Constant-round Single Secret Election (crSSLE))**.** Suppose there are G nodes with up to f of them being corrupted, where $f < \tau$ and $f < G/2$. An crSSLE protocol $\Pi = (\mathsf{Setup}, \mathsf{GenInstance}, \mathsf{ParElect}, \mathsf{Elect}, \mathsf{Combine}, \mathsf{Verify})$ is defined as follows:

- $(\mathsf{pp}, \mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}) \leftarrow \mathsf{crSSLE.Setup}(1^\lambda, G)$: take as input the nodes count G and security parameter 1^λ; output public parameters pp, a master public key mpk, and distribute $[\![\mathsf{msk}]\!]_i$ to each single user i secretly.

- $(\mathsf{com}_k, \mathsf{s}_k, \rho_k) \xleftarrow{\$} \mathsf{crSSLE.GenInstance}(1^\lambda, \mathsf{pp}, \mathsf{mpk})$: for each party k, take in public parameter pp, the master public key mpk, the security parameter 1^λ; generate the commitment com_k serving as the input to the election protocol, the private witness s_k, and the public auxiliary election input ρ_k.

- $\mathsf{ct}_{\mathsf{com}'_k} \xleftarrow{\$} \mathsf{crSSLE.ParElect}(\mathsf{mpk}, \{(\mathsf{com}_i, \rho_i)\}_{i\in[G]})$: for each party k, take in a list of public inputs $\{(\mathsf{com}_i, \rho_i)\}$ (which is the public output from $\mathsf{GenInstance}$); output a partial election ciphertext $\mathsf{ct}_{\mathsf{com}'_k}$.

- $[\![\mathsf{com}'']\!]_k \xleftarrow{\$} \mathsf{crSSLE.Elect}(\{\mathsf{ct}_{\mathsf{com}'_i}\}_{i\in[G]}, [\![\mathsf{msk}]\!]_k)$: for each party k, take in a list of ciphertexts $\{\mathsf{ct}_{\mathsf{com}'_i}\}$ and a secret key share $[\![\mathsf{msk}]\!]_k$; output a single share $[\![\mathsf{com}'']\!]_k$.

- $\mathsf{com}'' \leftarrow \mathsf{crSSLE.Combine}(\{[\![\mathsf{com}'']\!]_i\}_{i\in[G]})$: take in a list of shares $\{[\![\mathsf{com}'']\!]_i\}$; output the final randomized election result com''.

- $\{0, 1\} \leftarrow \mathsf{crSSLE.Verify}(\mathsf{pp}, \mathsf{com}'', \mathsf{s}, m)$: take in public parameter pp, a randomized commitment com'', a witness s and a message m; output 1 if s opens com'' into message m; and 0 otherwise.

A crSSLE also needs to satisfy the following properties. For any (stateful) PPT static semi-honest adversary $\mathcal{A}$ (taking all the public information below as input), let $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}) \leftarrow \mathsf{crSSLE.Setup}(1^\lambda, G)$. The adversary $\mathcal{A}$ chooses a corrupted set $\mathcal{W}_{\mathsf{cor}}$, with $|\mathcal{W}_{\mathsf{cor}}| \leq f$, before setup, 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 $\{(\mathsf{com}_i, \mathsf{s}_i, \rho_i)\}_{i\in\mathcal{W}_{\mathsf{cor}}}$ be the public and private outputs generated by $\mathcal{A}$, and $(\mathsf{com}_i, \mathsf{s}_i, \rho_i) \xleftarrow{\$} \mathsf{crSSLE.GenInstance}(1^\lambda, \mathsf{pp}, \mathsf{mpk}, m_i)$, for $i \in [G]\backslash\mathcal{W}_{\mathsf{cor}}$ honestly generated by other parties. For all parties in $[G]\backslash\mathcal{W}_{\mathsf{cor}}$, execute $\mathsf{crSSLE.ParElect}$ honestly to get $\{\mathsf{ct}_{\mathsf{com}'_i}\}_{i\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$. For parties in $\mathcal{W}_{\mathsf{cor}}$, let $\mathcal{A}$ publish $\{\mathsf{ct}_{\mathsf{com}'_i}\}_{i\in\mathcal{W}_{\mathsf{cor}}}$. With all $\{\mathsf{ct}_{\mathsf{com}'_i}\}_{i\in[G]}$, parties in $[G]\backslash\mathcal{W}_{\mathsf{cor}}$ execute $\mathsf{crSSLE.Elect}$ to get $[\![\mathsf{com}'']\!]_{i\in[G]\backslash\mathcal{W}_{\mathsf{cor}}}$, and $\mathcal{A}$ publishes $[\![\mathsf{com}'']\!]_{i\in\mathcal{W}_{\mathsf{cor}}}$ for the corrupted parties. With all $[\![\mathsf{com}'']\!]_{i\in[G]}$ published, let $\mathsf{com}'' \leftarrow \mathsf{crSSLE.Combine}(\{[\![\mathsf{com}'']\!]_i\}_{i\in[G]})$ be the final election output. Denote all transcripts as st. A crSSLE scheme also needs to satisfy the following properties:

- *Uniqueness:* With probability at least $1 - \mathsf{negl}(\lambda)$, there exists some $j \in [G]$, such that: $\mathsf{crSSLE.Verify}(\mathsf{pp}, \mathsf{com}'', \mathsf{s}_j, m_j) = 1$, and for every $j' \neq j$, $\mathsf{crSSLE.Verify}(\mathsf{pp}, \mathsf{com}'', \mathsf{s}_{j'}, m_{j'}) = 0$. Let j^* denote this unique index, and set $j^* = \perp$ otherwise.

- *Unpredictability:* Given st and all secrets held by $\mathcal{W}_{\mathsf{cor}}$, before the leader reveals its identity, $\mathcal{A}$ outputs an index $k \in [G]\backslash\mathcal{W}_{\mathsf{cor}}$, it holds that $\Pr[k = j^* \mid j^* \in [G]\backslash\mathcal{W}_{\mathsf{cor}}] \leq \frac{1}{G-|\mathcal{W}_{\mathsf{cor}}|} + \mathsf{negl}(\lambda)$.

- *Fairness:* For every $i \in [G]$, it holds that $\left|\Pr[\mathsf{crSSLE.Verify}(\mathsf{pp}, \mathsf{com}'', s_i, m_i) = 1] - \frac{1}{G}\right| \leq \mathsf{negl}(\lambda)$, where the randomness is taken over all random choices in the experiment.

4 Our Construction

In this section, we will present the construction of our single secret leader election scheme Relect.

As stated in § 2, the overall paradigm of Relect comprises Setup, $\mathsf{GenInstance}$, $\mathsf{ParElect}$, Elect, $\mathsf{Combine}$ and Verify. We now proceed to elaborate in detail according to this paradigm.

4.1 Setup

First, we introduce the setup step. In this step, the public parameters pp, the master public key mpk and the master secret key shares $\{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}$ for the threshold fully homomorphic encryption (ThFHE) is generated by the procedure Setup in Algorithm 2. pp contains commitment parameters $\mathsf{pp}_{\mathsf{com}}$, ThFHE parameters $\mathsf{pp}_{\mathsf{ThFHE}}$ and a random oracle H. Here, $\mathsf{pp}_{\mathsf{com}}$ and $\mathsf{pp}_{\mathsf{ThFHE}}$ satisfy the noise constraint detailed in § 5.

The procedure Setup only takes place once. The master public key and the master secret key shares can be reused.

Besides, automorphism key is generated in this step as well. But we omit it here and treat it as the part of the mpk for easily reading.

4.2 Instance Generation

In this section, we first describe the $\mathsf{GenInstance}$ procedure: in each round of single secret leader election of Relect, every party k needs to generate an instance consisting of: 1. A commitment com_k, 2. A secret s_k, and 3. A random number $r_k \xleftarrow{\$} [2N]$ where N is a parameter of $\mathsf{pp}_{\mathsf{ThFHE}}$.

Here, party k opens com_k using secret s_k as witness; another party's independently generated secret passes this check only with negligible probability (detailed below). Thus, it is used to represent the identity of party k. After the secret election procedure, the secretly selected leader opens the (re-randomized) commitment by revealing the corresponding secret. r_k contributes to the random, to-be-selected leader (how it is used is detailed later).

Now we describe this instance generation procedure in more detail.

First, each party k generates a new secret s_k and a key pair $(\mathsf{sk}_k, \mathsf{pk}_k) \leftarrow \mathsf{RLWE.KeyGen}(\mathsf{pp}_{\mathsf{com}})$ of RLWE encryption scheme where $\mathsf{sk}_k := H(\mathsf{s}_k) \in \mathbb{Z}[X]/(X^n+1)$ and H is a random oracle from pp. Here, H includes a coefficientwise map to the prescribed ternary distribution: 0 has probability $1/2$, and ± 1 each have probability $1/4$. The seeds are sampled independently with negligible collision probability, so distinct seeds give independent keys with independent coefficients. Then each party k generates a commitment $\mathsf{com}_k := \mathsf{RLWE.Enc}(\mathsf{pk}_k, m_k) \in \mathsf{RLWE}_{\mathsf{sk}_k}(m_k)$, where m_k is a user-specific message according to $\mathsf{crSSLE.GenInstance}$ of § 3.4. One thing that's important to note is $\{m_i\}_{i\in[G]}$ are set to same message $0 \in \mathbb{Z}[X]/(X^n+1)$ in Relect.

No party other than party k initially possesses knowledge of s_k. For any other party i using its independently generated secret s_i, the probability that $\mathsf{Dec}(H(\mathsf{s}_i), \mathsf{com}_k) = 0$ is at most $p_0 + \mathsf{negl}(\lambda)$ by the wrong-key-decryption bound in § 3.1. Here, p_0 is the probability that all n coefficients decode to zero in the uniform-ciphertext experiment and is negligible under the stated parameter requirement.

Next, each party k generates some randomness $r_k \in [2N]$ and these random $\{r_i\}_{i\in[G]}$'s together decide who is elected as the leader (secretly). I.e., party $r \leftarrow \sum_k r_k \pmod G \in [G]$ will be selected as the leader. This r is then used as the "selector" to select the leader. To make sure that this r can be computed in the way we wanted, we encrypt X^{r_k} as RGSW ciphertexts with mpk, yielding $\mathsf{ct}_{r_k} := \mathsf{RGSW}(X^{r_k})$. The reason of this will become clear later. Together, $(\mathsf{pk}_k, \mathsf{ct}_{r_k})$ form the public auxiliary election input ρ_k in Definition 3.2; the index r_k remains private.

Each party k accomplishes this according to the procedure
$\mathsf{GenInstance}$ in Algorithm 2.

In practical, each party k can generate multiple instances before potential multi-round elections. Once all parties finish one round single secret leader election, they should drop the used $\{\mathsf{com}_k, \mathsf{s}_k, \mathsf{sk}_k, \mathsf{pk}_k, \mathsf{ct}_{r_k}\}$ and broadcast new instance $\{\mathsf{com}_k, \mathsf{pk}_k, \mathsf{ct}_{r_k}\}$ for next round(if needed).

4.3 Election

In this section, we describe the protocol to output a commitment com'' which is the *re-randomized* version of com_r from $\{\mathsf{com}_i\}_{i\in[G]}$, indicating that party $r \in [G]$ is selected as the leader. Due to re-randomization, any party other than party r cannot learn any information about r. This is the main technical component of Relect and we describe the entire process in more detail below.

4.3.1 Partial Election (Party Local Election)

From a high-level perspective, in this step, based on public information $\{\mathsf{com}_i, \mathsf{pk}_i, \mathsf{ct}_{r_i}\}_{i\in[G]}$, each party k outputs a ciphertext ct'_k which encrypts com'_r where $r \equiv \sum_{i\in[G]} r_i \mod G$ and com'_r is re-randomized from com_r.

First, let us analyze the sub-goals required to achieve this objective:

1. **Selector Generation**: From $\{\mathsf{ct}_{r_i}\}_{i\in[G]}$, generate "selector" ciphertexts $\mathsf{sel}_0, \ldots, \mathsf{sel}_{G-1}$ where sel_r encrypts $v \in \mathcal{R}_T$ (under ThFHE.msk, described later) while all other ciphertexts encrypt 0.
2. **Commitment Re-randomization**: For $\{\mathsf{com}_i\}_{i\in[G]}$, perform re-randomization over to obtain com'_i to ensure that it does not leak any information about who the leader is (to parties who do not hold sk_i).
3. **Encrypted Commitment Generation**: Produce a ciphertext encrypts a re-randomized com_r (i.e., com'_r) and broadcast it to all parties. This will be used for the subsequent randomized aggregation by all parties.

Now, we detail ways to achieve the three sub-goals above.

Selector Generation. After previous steps, all parties get the same ciphertexts $\{\mathsf{ct}_{r_i}\}_{i\in[G]}$ where ct_{r_i} is an RGSW ciphertext encrypts X^{r_i}.

Each party k starts with the trivial ciphertext $\mathsf{ct} := \mathsf{RLWE}(u)$ (with zero encryption noise) and computes $\mathsf{ct} \leftarrow \mathsf{ct}_{r_i} \boxtimes \mathsf{ct}$ for $i = 0, \ldots, G-1$, where u is a test polynomial (detailed below), to aggregate all random values $\{r_i\}_{i\in[G]}$. The result of this computation is $\mathsf{ct} = \mathsf{RLWE}(u')$ where $u' = u \cdot X^{w\cdot G+r}$ and $w \cdot G + r \equiv \sum_{i\in[G]} r_i \mod 2N$. We assume $G|N$ since power-of-two is the most commonly used parameter for G [59, 58]. Then the "selector" ciphertexts will be "extracted" from ct.

Let's start with a simple case $G = N$. We just set test polynomial u to be 1, then ct would be $\mathsf{RLWE}(X^{w\cdot N+r})$. It's easy to see that $w = 0$ (corresponding to $\mathsf{RLWE}(X^r)$) or 1 (corresponding to $\mathsf{RLWE}(-X^r)$). According to the "oblivious expansion" described in Algorithm 1, the operation $\mathsf{ObliviousExpand}(\mathsf{ct}, N) \to \mathsf{sel}_0, \ldots, \mathsf{sel}_{N-1}$ generate required "selector" ciphertexts where sel_r encrypts -1 or 1, which means party r is selected.

When $G < N$ (but still divides N), the situation becomes slightly more complicated.

If u is set to 1 as above, computing the above gives $\mathsf{RLWE}(X^{w'\cdot G+r})$ for some $w' \in [0, 2N/G - 1]$. To obtain the selectors, it is obvious that we cannot only obliviously expand the first G coefficients but instead need to expand all the N coefficients, such that one of these N ciphertexts encrypt 1 or -1. Then, to select one of the leaders, we could duplicate each party's commitment by N/G times, and then select among the N commitments. In other words, we simply "pad" the number of parties to $N > G$ by duplicating each party N/G times.

While this approach appears functioning, it is concretely inefficiency. Specifically, all the operations needs to be done with an overhead of N/G. Thus, instead, we propose a different way to avoid such an overhead.

Again, G divides N, and let $u = \sum_{i\in[N/G]} X^{i\cdot G}$. Here, $u' = u \cdot X^{w'\cdot G+r} \mod X^N + 1$ is a negacyclic rotation over u. Therefore, if u' is sliced into segments of G coefficients each, the r-th coefficient in each segment is ± 1 (instead of only one of these N coefficients), and all other coefficients are zero.

We can then perform $\mathsf{PartialObliviousExpand}(\mathsf{RLWE}(u'), G) \to \mathsf{sel}_0, \ldots, \mathsf{sel}_{G-1}$ according to § 3.2.4 and obtaining only G ciphertext as "selector" ciphertexts where sel_r encrypts $v = u \cdot X^{w'\cdot G}$ and other ciphertexts encrypt 0.

Commitment Re-randomization. The next step is to re-randomize the commitments $\{\mathsf{com}_i\}_{i\in[G]}$ homomorphically. This step is simple since we use an RLWE encryption (§ 3.1) as the commitment.

Specifically, for each com_i, party k simply computes $\mathsf{com}'_{i,k} \leftarrow \mathsf{com}_i + \mathsf{RLWE.Enc}(\mathsf{pk}_i, 0)$, where pk_i is the public key that party i published during the GenInstance procedure. By RLWE, $\mathsf{com}'_{i,k}$ now is indistinguishable from a random ciphertext, to any party not holding sk_i even given com_i.

Encrypted Commitment Generation. After previous steps, each party k obtains the different re-randomized commitments $\{\mathsf{com}'_{i,k}\}_{i\in[G]}$ and the same "selector" ciphertexts, where $\mathsf{sel}_r = \mathsf{RLWE}(v)$ and $v = u \cdot X^{w'\cdot G}$, while $\mathsf{sel}_0, \ldots, \mathsf{sel}_{r-1}, \mathsf{sel}_{r+1}, \ldots, \mathsf{sel}_{G-1}$ all encrypt 0.

Then, each party k computes $\sum_i \mathsf{sel}_i \cdot \mathsf{com}'_{i,k}$ by treating $\mathsf{com}'_{i,k}$ as a plaintext. For $G = N$, sel_r encrypts ± 1, and if $\mathsf{com}'_{i,k}$ fits perfectly into the plaintext space (by viewing $\mathsf{com}'_{i,k} \in \mathcal{R}_q^2$ as a vector of format $\mathbb{Z}_q^{2n}$ where q is the ciphertext modulus of the commitment, and n is its ring dimension), the result of this process gives $\mathsf{RLWE}(\pm\mathsf{com}'_{r,k})$. However, for $G < N$, sel_r encrypts $v = u \cdot X^{w'\cdot G} \neq \pm 1$. Thus, to multiply, instead, we parse $(a'_{i,k}, b'_{i,k}) \leftarrow \mathsf{com}'_{i,k}$ and perform $\mathsf{parta}_k \leftarrow \sum_i \mathsf{sel}_i \cdot a'_{i,k}$ and $\mathsf{partb}_k \leftarrow \sum_i \mathsf{sel}_i \cdot b'_{i,k}$ separately. After this, we obtain ciphertexts $\{\mathsf{parta}_k, \mathsf{partb}_k\}$, each encrypting $v \cdot a'_{r,k}$ and $v \cdot b'_{r,k}$ where $(a'_{r,k}, b'_{r,k}) \leftarrow \mathsf{com}'_{r,k}$.

Before publication, each party k randomizes both outputs using fresh RLWE encryptions of 0 under the master public key:

$$\mathsf{parta}_k \leftarrow \mathsf{parta}_k + \mathsf{RLWE.Enc}(\mathsf{mpk}, 0), \qquad \mathsf{partb}_k \leftarrow \mathsf{partb}_k + \mathsf{RLWE.Enc}(\mathsf{mpk}, 0).$$

The two encryption calls use fresh, independent randomness for each party and election; neither the zero ciphertexts nor their randomness is published separately.

The whole procedure of "Party Local Election" is detailed in procedure ParElect of Algorithm 2. After completing this procedure, each party k has effectively performed a local election (partial election), generating ciphertexts that encrypt commitments which only the secret leader can open. Subsequently, we need to aggregate all parties' outputs from this step.

4.3.2 Election

In previous step, each party k generates
$\{\mathsf{parta}_k, \mathsf{partb}_k\}$ which encrypts the re-randomized $\mathsf{com}'_{r,k}$. If a threshold decryption is performed over $\{\mathsf{parta}_k, \mathsf{partb}_k\}$, the identity of secret leader is leaked to party k. So, in this step, all
$\{\mathsf{parta}_i, \mathsf{partb}_i\}_{i\in[G]}$ are aggregated together to ensure that all parties know nothing about the identity of the secret leader (except for the selected leader).

To aggregate, each party k simply performs $\mathsf{parta} = \sum_{i\in[G]} \mathsf{parta}_i$ and $\mathsf{partb} = \sum_{i\in[G]} \mathsf{partb}_i$. After this step, each party k obtains the same $\{\mathsf{parta}, \mathsf{partb}\}$. Decrypting these 2 ciphertexts give back $(v\cdot a''_r, v\cdot b''_r)$, where a''_r, b''_r comprise com'' and $\mathsf{com}'' \leftarrow \sum_{i\in[G]} \mathsf{com}'_{r,i}$. Note that $\mathsf{com}'' = G \cdot \mathsf{com}_r + \sum_{i\in[G]} \mathsf{RLWE.Enc}(\mathsf{pk}_r, 0)$, so com'' remains a ciphertext encrypting message 0 under pk_r, subject to the aggregate noise bound in § 5.1. Moreover, com'' incorporates randomized masking ciphertexts from all parties. Consequently, no non-secret-leader party can now deduce the secret leader's identity through com''.

To finalize the election, it is necessary to decrypt $\{\mathsf{parta}, \mathsf{partb}\}$. However, we must first have each party k perform a local decryption of $\{\mathsf{parta}, \mathsf{partb}\}$ using its $[\![\mathsf{msk}]\!]_k$, thereby recovering $[\![v \cdot a''_r]\!]_k$ and $[\![v \cdot b''_r]\!]_k$. The complete com'' will be reconstructed in subsequent steps.

The algorithm of this operation is shown in procedure Elect in Algorithm 2.

4.4 Combine

Now, all parties get the ciphertexts $\{[\![v \cdot a''_r]\!]_i, [\![v \cdot b''_r]\!]_i\}_{i\in[\tau]}$. This means only a sub set of τ parties rather G parties can proceed this operation as well.

All parties participating in this step execute the operation ThFHE.Combine to reconstruct $(v \cdot a''_r, v \cdot b''_r)$ from the partial decryptions $\{[\![v \cdot a''_r]\!]_i, [\![v \cdot b''_r]\!]_i\}_{i\in[\tau]}$.

4.5 Verify

After previous operation, every party get $(v \cdot a''_r, v \cdot b''_r)$.

Recall that, $v = u \cdot X^{w' \cdot G}$ where $u = \sum_{i \in [N/G]} X^{i \cdot G}$. To obtain the intended result, we need to compute $v \cdot u^{-1}$. We observe that u^{-1} *does* exist in the ring we use. Specifically, $u^{-1} \equiv 2^{-1}(1 - X^G) \mod X^N + 1$ and 2^{-1} exists modulo the odd plaintext modulus q, which is exactly what we use.

Then, we perform $(v \cdot a''_r, v \cdot b''_r) \cdot u^{-1}$ and get $(W_a, W_b) := (X^{w' \cdot G} \cdot a''_r, X^{w' \cdot G} \cdot b''_r) \in (\mathbb{Z}_q[X]/(X^N + 1), \mathbb{Z}_q[X]/(X^N + 1))$. Here, a''_r, b''_r are represented as polynomials of degree less than n, padded with zeros when $N > n$.

Next, we recover a commitment in the smaller ring from (W_a, W_b). For $N = n$, take (W_a, W_b) directly. For $N \geq 4n$, split each polynomial into $L = N/n$ consecutive blocks of n coefficients, and let C_j be the pair of blocks at position $j \in [L]$, with coefficients reindexed from degree zero. At most two block pairs are nonzero, since both components are rotations of polynomials of degree less than n. If all block pairs are zero, return $(0, 0)$; if only one is nonzero, return that pair. If the two nonzero pairs are C_j, C_{j+1}, return $C_j - C_{j+1}$; if they are C_0, C_{L-1}, return $C_0 + C_{L-1}$. The latter addition accounts for the sign change already introduced by crossing the boundary at degree N. Denote this operation by $\mathsf{BlockRecover}_n(W_a, W_b)$.

Now, we treat the recovered pair as final commitment com''', which equals $\pm X^{w''} \cdot \mathsf{com}''$ in $(\mathbb{Z}_q[X]/(X^n + 1))^2$, where $w'' = w' \cdot G \bmod n$. The common sign and monomial multiplication preserve the maximum absolute value of the decryption error coefficients.

From the detailed description of the above interactive phase, it is known that com'' is an encryption of 0 under sk_r where r is the party id of the secret leader. So com''' is also an encryption of 0 under sk_r. For the other parties using their prescribed keys $\mathsf{sk}_i = H(\mathsf{s}_i)$, false acceptance has only negligible probability under the parameter conditions analyzed in the proof of Theorem 4.1.

Subsequently, each party k can decrypt com''' with sk_k and verify whether the decrypted result equals 0 to determine if they are the secret leader. If a party confirms itself as the secret leader (suppose it is party r), it broadcasts a claim of leadership to all other parties. And it broadcast its secret s_r to all other parties. Other parties check if the com''' is decrypted to 0 with the secret key $H(\mathsf{s}_r)$. If no issues are detected, all parties can now proceed with subsequent operations using party r as the designated leader. This concludes the description of all steps in Relect.

The complete algorithm specification is formally presented in Algorithm 2.

Theorem 4.1. Given G parties being a power-of-two, with $G \mid N$ and either $N = n$ or $N \geq 4n$, for any PPT *static honest-but-curious* adversary corrupting at most f parties, where $f < G/2$ and $f < \tau$, the crSSLE construction given in Algorithm 2 is a constant-round single secret leader election scheme defined in Definition 3.2 for a single election in the random-oracle model, under the security assumptions of the underlying RLWE-based threshold FHE scheme, assuming correct homomorphic evaluation and a correct, simulatable threshold FHE instantiation with constant-round online decryption.

Here, simulation covers setup, preprocessing, and decryption transcripts using only public key material, corrupted state, and the prescribed output, with negligible error and without the master secret key.

Proof. Uniqueness: The uniqueness of the elected leader is guaranteed by the following conditions:

1. **Selector ciphertext correctness**: The set $\{\mathsf{sel}_i\}_{i \in [G]}$ generated in Algorithm 2 satisfies that exactly one sel_r encrypts non-zero while all others encrypt 0.

2. **Noise bound preservation**: The accumulated noise in the re-randomized commitments $\{\mathsf{com}'_{r,i}\}_{i \in [G]}$ from all parties must be bounded to ensure $\mathsf{Dec}(\mathsf{sk}_r, \mathsf{com}''') = 0$ remains valid.

3. **Homomorphic evaluation correctness**: The ThFHE scheme guarantees correct homomorphic evaluation such that $\mathsf{com}'' \leftarrow \sum_{i \in [G]} \mathsf{com}'_{r,i}$ is properly computed and subsequently decrypted through threshold decryption.

4. **Independent secret keys**: All parties' secret keys sk_i are independently sampled from the prescribed ternary distribution, except for negligible seed-collision probability.

Algorithm 2 Relect: The whole procedures

procedure $\mathsf{Setup}(1^\lambda, G, \tau)$
  $\mathsf{pp}_{\mathsf{com}} \leftarrow \mathsf{RLWE.GenParams}(1^\lambda, t)$
  $\mathsf{pp}_{\mathsf{ThFHE}} \leftarrow \mathsf{ThFHE.GenParams}(1^\lambda, q, G, \tau)$ where $q \in \mathsf{pp}_{\mathsf{com}}$
  $\mathsf{pp} := (\mathsf{pp}_{\mathsf{com}}, \mathsf{pp}_{\mathsf{ThFHE}}, H)$ where H is random oracle
  $(\mathsf{mpk}, \{[\![\mathsf{msk}]\!]_i\}_{i\in[G]}) \leftarrow \mathsf{ThFHE.KeyGen}(\mathsf{pp}_{\mathsf{ThFHE}})$
  for party $i \in [G]$ **do**
    return $[\![\mathsf{msk}]\!]_i$ to party i secretly
  return pp and mpk to all parties
procedure $\mathsf{GenInstance}(1^\lambda, \mathsf{pp}, \mathsf{mpk})$
  Sample s_k randomly
  $\mathsf{sk}_k \leftarrow H(\mathsf{s}_k)$ where $H \in \mathsf{pp}$
  $\mathsf{pk}_k \leftarrow \mathsf{RLWE.KeyGen}(\mathsf{pp}_{\mathsf{com}}, \mathsf{sk}_k)$
  $\mathsf{com}_k = \mathsf{RLWE.Enc}(\mathsf{pk}_k, 0) \in \mathsf{RLWE}_{\mathsf{sk}_k}(0)$
  $r_k \xleftarrow{\$} [2N]$
  $\mathsf{ct}_{r_k} \leftarrow \mathsf{ThFHE.Enc}(\mathsf{mpk}, X^{r_k}) \in \mathsf{RGSW}(X^{r_k})$
  return com_k, pk_k and ct_{r_k}
procedure $\mathsf{ParElect}(\mathsf{mpk}, \{(\mathsf{com}_i, \mathsf{pk}_i, \mathsf{ct}_{r_i})\}_{i\in[G]})$
  $u(X) := \sum_{i\in[N/G]} X^{i\cdot G}$
  $\mathsf{ct} := \mathsf{RLWE}(u)$
  for $i \in [G]$ **do**
    $\mathsf{ct} \leftarrow \mathsf{ct}_{r_i} \boxtimes \mathsf{ct}$
  $\{\mathsf{sel}_i\}_{i\in[G]} \leftarrow \mathsf{PartialObliviousExpand}(\mathsf{ct}, G)$
  Let $\{\mathsf{parta}_k, \mathsf{partb}_k\}$ be trivial RLWE ciphertexts encrypt 0
  for $i \in [G]$ **do**
    $\mathsf{com}'_{i,k} \leftarrow \mathsf{com}_i + \mathsf{RLWE.Enc}(\mathsf{pk}_i, 0)$
    Parse $(a'_{i,k}, b'_{i,k}) \leftarrow \mathsf{com}'_{i,k}$.
    $\mathsf{parta}_k = \mathsf{parta}_k + (\mathsf{sel}_i \cdot a'_{i,k})$
    $\mathsf{partb}_k = \mathsf{partb}_k + (\mathsf{sel}_i \cdot b'_{i,k})$
  $\mathsf{parta}_k \leftarrow \mathsf{parta}_k + \mathsf{RLWE.Enc}(\mathsf{mpk}, 0)$
  $\mathsf{partb}_k \leftarrow \mathsf{partb}_k + \mathsf{RLWE.Enc}(\mathsf{mpk}, 0)$
  return $\{\mathsf{parta}_k, \mathsf{partb}_k\}$
procedure $\mathsf{Elect}(\{\mathsf{parta}_i, \mathsf{partb}_i\}_{i\in[G]})$
  $\mathsf{parta} = \sum_{i\in[G]} \mathsf{parta}_i$
  $\mathsf{partb} = \sum_{i\in[G]} \mathsf{partb}_i$
  $[\![v \cdot a''_r]\!]_k = \mathsf{ThFHE.PartialDec}([\![\mathsf{msk}]\!]_k, \mathsf{parta})$
  $[\![v \cdot b''_r]\!]_k = \mathsf{ThFHE.PartialDec}([\![\mathsf{msk}]\!]_k, \mathsf{partb})$
  return $\{[\![v \cdot a''_r]\!]_k, [\![v \cdot b''_r]\!]_k\}$
procedure $\mathsf{Combine}(\{[\![v \cdot a''_r]\!]_i, [\![v \cdot b''_r]\!]_i\}_{i\in[G]})$
  $v \cdot a''_r \leftarrow \mathsf{ThFHE.Combine}(\{[\![v \cdot a''_r]\!]_i\}_{i\in[G]})$
  $v \cdot b''_r \leftarrow \mathsf{ThFHE.Combine}(\{[\![v \cdot b''_r]\!]_i\}_{i\in[G]})$
procedure $\mathsf{Verify}(\mathsf{pp}, (v \cdot a''_r, v \cdot b''_r), \mathsf{s} = \mathsf{s}_k)$
  $(W_a, W_b) \leftarrow (v \cdot a''_r, v \cdot b''_r) \cdot u^{-1} \bmod (X^N + 1)$
  $\mathsf{com}''' \leftarrow \mathsf{BlockRecover}_n(W_a, W_b)$ ▷ See § 4.5
  if $\mathsf{RLWE.Dec}(H(\mathsf{s}_k), \mathsf{com}''')$ equals 0 **then**
    return 1
    ▷ In this case, party k can declare the leadership by broadcasting s_k.
  else
    return 0

5. **Wrong-Key decryption resistance**: For each non-selected party $i \neq r$, the probability that $\mathsf{Dec}(\mathsf{sk}_i, \mathsf{com}''') = 0$ must be negligible.

These conditions are satisfied as argued below:

- *Condition 1* is guaranteed by (1) the **partial oblivious expansion** technique, which ensures only one selector ciphertext contains non-zero, and (2) each party k generates a monomial $\pm X^{r_k}$ for selection.
- *Condition 2* is guaranteed by the correctness of RLWE according to the noise analysis in § 5.1 (such that the commitment re-randomization does not accumulate noise exceeding the noise bound for correct decryption).
- *Condition 3* is guaranteed by the correctness of ThFHE according to the noise analysis in § 5.2.
- *Condition 4* follows from applying H with the prescribed ternary output distribution to distinct seeds; seed collisions have negligible probability.
- *Condition 5* follows from a joint RLWE replacement of the selected party's $(\mathsf{pk}_r, \mathsf{com}_r)$ by independent uniform pairs of ring elements, keeping each other party's sk_i independently generated. Recompute the re-randomization contributions using the replaced public key: $\mathsf{com}'' = G \cdot \mathsf{com}_r + \sum_k \mathsf{RLWE.Enc}(\mathsf{pk}_r, 0)$ is then uniform, since com_r is uniform and independent of the sum, and G is invertible modulo the odd modulus q. Thus $b''_r - a''_r \mathsf{sk}_i$ is uniform, and all its coefficients decode to zero with probability p_0 from § 3.1. The signed monomial multiplication in § 4.5 preserves this zero check because $D_0 = -D_0$.

The selected index depends only on election inputs generated independently of the personal keys and commitments. For polynomially many parties, a union bound over the other $G-1$ parties gives false-acceptance probability at most $(G-1)p_0 + \mathsf{negl}(\lambda)$ for the correctly recovered target commitment. Together with Conditions 1–3, this yields uniqueness when this bound and the correctness-failure probability are negligible.

Fairness: We first observe that the random values obtained on line 14 of Algorithm 2 are independent uniformly random indices $r_i \in \mathbb{Z}_{2N}$. Since $G \mid N$, their sum $r = \sum_{i \in [G]} r_i \bmod G$ is a uniformly random index in $\mathbb{Z}_G$. By the preceding uniqueness analysis, under its correctness and parameter conditions, party r accepts and every other party rejects except with negligible probability. Here, correctness includes homomorphic evaluation and threshold decryption with the noise of the fresh zero ciphertexts added to parta_k and partb_k, as well as decryption of the selected commitment. The probability that any fixed party passes Verify therefore differs from $1/G$ by at most this negligible amount, so fairness holds as defined in Definition 3.2.

Unpredictability: We consider the adversary's full view before the leader reveals its identity. By the correctness and simulation assumptions in Theorem 4.1, the threshold transcripts can be simulated using the public key material, corrupted state, and intended output $(v \cdot a''_r, v \cdot b''_r)$. Under the security assumptions of the underlying RLWE-based threshold FHE scheme, fresh public-key zero ciphertexts are pseudorandom jointly with the public key material and corrupted key shares. Replacing the two independent zero ciphertexts added to each honest party's parta_k and partb_k by uniform ciphertext-space elements therefore makes these local outputs uniform, even given the public selectors. After each replacement, we recompute the aggregate ciphertexts and simulate the threshold transcripts with the same intended output; we do not decrypt the uniform replacements.

Next, joint RLWE replacements first make each honest party's $(\mathsf{pk}_i, \mathsf{com}_i)$ uniform, and then make one honest party's re-randomization contribution to each such commitment uniform. Hence each honest candidate's aggregate commitment can be replaced by a uniform pair in $(\mathbb{Z}_q[X]/(X^n+1))^2$, independently of its public key, initial commitment, and corrupted contributions. These replacements incur only negligible loss from queries to, or collisions with, honest seeds in the random-oracle model.

Finally, replace the honest RGSW inputs row by row with uniform elements using the same pseudorandomness argument, recomputing the public selectors, corrupted parties' outputs, and simulated transcripts. The original indices r_i still determine the intended output. Since $\sum_i r_i \bmod 2N = w'G + r$ is uniform, r is uniform and independent of w'. Conditioned on $r \notin \mathcal{W}_{\mathsf{cor}}$, the intended output depends only on w' and a fresh uniform commitment pair, embedded by zero padding and multiplied by $uX^{w'G}$, so its distribution

is independent of r. Thus, even after threshold decryption, the adversary's success probability is at most $1/(G - |\mathcal{W}_{\mathsf{cor}}|) + \mathsf{negl}(\lambda)$. Correctness and the honest-majority condition transfer this bound to the unique accepting party j^* in Definition 3.2, with only negligible additional loss.

□

4.6 Malicious Security

Lastly, we discuss malicious security. Similar to [58], our main construction above focuses on semi-honest adversary. Here, we discuss possible extensions to handle malicious adversaries.

We consider malicious deviations in three phases of Relect:

1. When publishing the ciphertext $\mathsf{ct}_k := \mathsf{RGSW}(X^{r_k})$, which contributes to the leader's index $r \leftarrow \sum_k r_k \bmod G$, the adversary could potentially craft its own share after seeing all others' ciphertexts. A malicious party could also submit a ciphertext that does not encrypt a valid monomial X^{r_k}.
2. Similarly, when publishing the randomized and encrypted commitment $\boxed{\mathsf{com}'_{r,k}}$, the adversary could first choose the final aggregation $\boxed{\mathsf{com}''_{r^*}}$, and then maliciously craft its own transcript $\boxed{\mathsf{com}'_{r,k}}$ given all others' shares $\{\boxed{\mathsf{com}'_{r,i}}\}_{i\neq k}$, since the aggregation procedure is a direct addition.
3. More crucially, recall that for a threshold FHE scheme, the master secret key is shared across all parties. During decryption, each party uses its own key share $[\![\mathsf{msk}]\!]_k$ to perform the partial decryption and gets the result $[\![v \cdot a''_r]\!]_k$ and $[\![v \cdot b''_r]\!]_k$. The decryption shares are then combined to recover $v \cdot a''_r$ and $v \cdot b''_r$. If decryption shares are combined linearly without validity checks, an adversary can alter the combined result by submitting an incorrect share.

As argued in the prior work Qelect [58], one could adopt the standard trick: *commit-and-reveal*, to bind each party to its message before the other messages are revealed. This does not by itself verify correct encryption, evaluation, or decryption. For large party size ($G \geq 128$), [58] proposes an optimized *random-subset* technique to eliminate a commit-and-reveal round in its ciphertext aggregation. On the other hand, if an adversary submits malformed transcripts, the retroactive detection phase introduced in [58, section 6.2.2] could provide a starting point for identifying misbehavior and penalizing the responsible party.

Adapting these techniques to Relect requires a separate malicious-security analysis, including the validity of the RGSW inputs and the threshold-decryption protocol.

Remark 4.1. Note that if a malicious party performs evaluation or decryption incorrectly, a valid result is not guaranteed; detecting such deviations requires additional checks. An alternative approach would be to require each party to provide a proof of correct computation (e.g., using [25, 61, 51]), combined with mechanisms to penalize misbehavior (e.g., see [30, 31, 62]). Exploring such extensions is an interesting direction for future work.

5 Noise analysis

This section discusses noise analysis. Note that noise analysis is essential in FHE-based solutions since we need to guarantee that the homomorphic circuit is correctly evaluated (except for negligible probability). For Relect, noise analysis is needed in two components: commitment and Threshold FHE evaluation. Commitment requires that the noise budget is sufficient for all participants to refresh their commitment and then combine. Threshold FHE component, as mentioned, needs to ensure the circuit evaluation is correct.

5.1 Commitment

The commitment scheme is based on the RLWE scheme with the parameters $(n, p, q, \mathcal{D}_{\mathsf{sk}}, \chi_{\sigma_{\mathsf{com}}})$, where n is the dimension, p is the message modulus, q is the ciphertext modulus, $\mathcal{D}_{\mathsf{sk}}$ is the secret key distribution and $\chi_{\sigma_{\mathsf{com}}}$ is the noise distribution.

The commitment com_i can be generated by calling $\mathsf{RLWE.Enc}$ to encrypt 0 with pk_i or sk_i. We use $\sigma^2_{\mathsf{com,pk}}$ and $\sigma^2_{\mathsf{com,sk}}$ to denote the noise variance of public-key and secret-key zero encryptions, respectively.

In the construction of § 4, the aggregate commitment is $\mathsf{com}'' = \sum_{i\in[G]} \mathsf{com}'_{r,i}$ $= \sum_{i\in[G]} \mathsf{com}_r + \sum_{i\in[G]} \mathsf{RLWE.Enc}(\mathsf{pk}_r, 0)$, where r denotes the secret leader's identify. In this case, com'' contains G copies of com_r, which are correlated, and G different $\mathsf{RLWE.Enc}(\mathsf{pk}_r, 0)$, which are uncorrelated. The noise variance of com'' is $\sigma^2_{\mathsf{com}''} = (G^2+G)\cdot\sigma^2_{\mathsf{com,pk}}$. Using secret-key encryption for the initial commitment reduces this to $\sigma^2_{\mathsf{com}''} = G^2\cdot\sigma^2_{\mathsf{com,sk}} + G\cdot\sigma^2_{\mathsf{com,pk}}$.

For better noise control, we additionally scale the initial commitment during re-randomization. During the commitment re-randomization phase, each party k performs $\mathsf{com}'_{i,k} \leftarrow \mathsf{com}_i \cdot G^{-1} + \mathsf{RLWE.Enc}(\mathsf{pk}_i, 0)$. Here G^{-1} is the inverse modulo the commitment ciphertext modulus q; it exists because G is a power of two and q is odd. Then

$$\begin{aligned}\mathsf{com}'' &= \sum_{i\in[G]} \mathsf{com}'_{r,i} \\ &= \sum_{i\in[G]} \mathsf{com}_r \cdot G^{-1} + \sum_{i\in[G]} \mathsf{RLWE.Enc}(\mathsf{pk}_r, 0) \\ &= \mathsf{com}_r + \sum_{i\in[G]} \mathsf{RLWE.Enc}(\mathsf{pk}_r, 0) \pmod q.\end{aligned}$$

With fresh, independent encryption randomness, $\sigma^2_{\mathsf{com}''} = \sigma^2_{\mathsf{com,sk}} + G\cdot\sigma^2_{\mathsf{com,pk}}$. The initial commitment therefore contributes only once to the aggregate noise. The individual scaled contributions are combined before commitment decryption. Our implementation, benchmarks, and concrete parameters in § A use both optimizations. The uniformity argument in the uniqueness proof of Theorem 4.1 also applies here, with com_r in place of $G\cdot\mathsf{com}_r$.

5.2 Election

The election procedure is based on the ThFHE scheme involving the operations described in § 4.

5.2.1 External Product

Lemma 5.1. Given an RLWE ciphertext $\mathsf{ct}_1 = \mathsf{RLWE}_s(u) \in \mathcal{R}_Q^2$ with the noise variance σ_1^2 and a RGSW ciphertext $\mathsf{ct}_2 = \mathsf{RGSW}_s(m) \in \mathcal{R}_Q^{2\times 2\ell_{\mathsf{gsw}}}$ with the noise variance σ_2^2, where $m = \pm X^r$ for $r \in [N]$. ℓ_{gsw} denotes the gadget length for RGSW ciphertexts and $\mathfrak{B}_{\mathsf{gsw}}$ is the gadget basis satisfying $\mathfrak{B}_{\mathsf{gsw}}^{\ell_{\mathsf{gsw}}} \le Q$. The output $\mathsf{ct} = \mathsf{ct}_2 \boxtimes \mathsf{ct}_1 = \mathsf{RLWE}_s(u\cdot m)$ has the noise with the noise variance σ^2_{out}. If the secret key s has independent centered ternary coefficients with variance $1/2$, then under the decomposition-noise model of [15, Appendix B],

$$\sigma^2_{out} = \sigma_1^2 + \underbrace{2\ell_{\mathsf{gsw}}\cdot N\cdot\frac{\mathfrak{B}_{\mathsf{gsw}}^2+2}{12}\cdot\sigma_2^2 + \frac{Q^2-\mathfrak{B}_{\mathsf{gsw}}^{2\ell_{\mathsf{gsw}}}}{12\mathfrak{B}_{\mathsf{gsw}}^{2\ell_{\mathsf{gsw}}}}(1+\frac{N}{2}) + \frac{N}{8}}_{\sigma^2_{\mathsf{ExtProd}}}$$

According to Theorem 5.1, starting from the trivial ciphertext $\mathsf{RLWE}(u)$, the noise variance after the loop containing line 21 of Algorithm 2 is estimated as $G\cdot\sigma^2_{\mathsf{ExtProd}}$ under the same model.

For more detailed proof of the noise increment of the external product between RLWE and RGSW, please refer to [15, Appendix B]. The noise analysis here is based on the derivation of [15].

5.2.2 Partial Oblivious Expand

Lemma 5.2. Given an RLWE ciphertext ct_1 with odd ciphertext modulus Q which encrypts a polynomial m with the noise variance σ_1^2 and an integer $1 \le c \le N$, c is a power of two, $\mathsf{PartialObliviousExpand}(\mathsf{ct}_1, c) \rightarrow$

elems will output a vector elems, where $\forall i \in [c], \mathsf{elems}[i] \in \mathsf{RLWE}(\sum_{j\in[N/c]} m_{i+j\cdot c}X^{j\cdot c})$. Let σ^2_{auto} be the variance of the noise increment by the homomorphic automorphism evaluation. Assume that the error coefficients introduced at different automorphism steps along an output path are uncorrelated with one another and with the input error coefficients. Then the noise variance σ^2_{poe} of $\mathsf{elems}[i]$ is given as: $\sigma^2_{\mathsf{poe}} \leq \sigma_1^2 + \frac{c^2-1}{3}\sigma^2_{\mathsf{auto}}$

The inverse scaling and subsequent expansion preserve the selected input error coefficients, giving the first term; the second term bounds the noise added by the automorphisms. For more detailed proof of the noise increment of the 'Oblivious Expand', please refer to [56, Appendix C.4]. Oblivious expand shares the same noise variance increment with the 'Homomorphic Trace'. The noise analysis here is based on the derivation of [56].

5.2.3 Homomorphic Automorphism

Lemma 5.3. Given an RLWE ciphertext $\mathsf{ct} = \mathsf{RLWE}(m(X))$ with the noise variance σ_1^2 and an automorphism key atk_t with the noise variance σ_2^2, the homomorphic automorphism $\mathsf{FHE.HomAuto}(\mathsf{ct}, t) \to \mathsf{ct}'$ for odd t outputs a ciphertext encrypts $m(X^t)$. Assume that the ℓ_{atk} denotes the gadget length for atk_t and $\mathfrak{B}_{\mathsf{atk}}$ is the gadget basis satisfying $\mathfrak{B}_{\mathsf{atk}}^{\ell_{\mathsf{atk}}} \leq Q$.

Under the same decomposition-noise model and secret-key distribution as in Theorem 5.1, the variance of the noise increment by this operation is:

$$\sigma^2_{\mathsf{auto}} = \ell_{\mathsf{atk}} \cdot N \cdot \frac{\mathfrak{B}^2_{\mathsf{atk}} + 2}{12}\sigma_2^2 + \frac{Q^2 - \mathfrak{B}^{2\ell_{\mathsf{atk}}}_{\mathsf{atk}}}{12\mathfrak{B}^{2\ell_{\mathsf{atk}}}_{\mathsf{atk}}} \cdot \frac{N}{2} + \frac{N}{8}$$

5.2.4 Aggregated Result

On line 27, there are G RLWE and plaintext multiplication aggregated together in one parta_i. And on line 33, we aggregate G parta_i into one parta, so there are G^2 RLWE and plaintext multiplication in one parta.

Let σ^2_{zero} denote the per-coefficient noise variance of a fresh outer ciphertext $\mathsf{RLWE.Enc}(\mathsf{mpk}, 0)$. Each local output includes one such ciphertext after the plaintext multiplications and additions, so each aggregate contains G zero ciphertexts. With fresh randomness independent across encryption calls and of the preceding computation, and centered ephemeral secrets and encryption errors, each added error has mean zero conditioned on the master secret key and the public-key error. Hence the added errors are uncorrelated with each other and with the existing evaluation noise. For the zero encryptions, we use the public-key error variance σ^2_{mpk} also for the fresh encryption errors, and ternary secret and ephemeral-secret coefficients of variance $1/2$. Averaging over key generation gives $\sigma^2_{\mathrm{zero}} = (N+1)\sigma^2_{\mathsf{mpk}}$.

Let the variance of the noise of the selectors sel_i be σ^2_{sel}. Under the coefficient-variance approximation for plaintext multiplication, neglecting covariances between the multiplied selector errors, the noise variance of parta and partb is estimated as

$$\begin{aligned}\sigma^2_{\mathsf{part}} &= G^2 \cdot (n \cdot \frac{q^2+2}{12} \cdot \sigma^2_{\mathsf{sel}}) + G\sigma^2_{\mathrm{zero}} \\ &\leq G^2 \cdot n \cdot \frac{q^2+2}{12} \cdot (\frac{G^2-1}{3} \cdot \sigma^2_{\mathsf{auto}} + G \cdot \sigma^2_{\mathsf{ExtProd}}) + G\sigma^2_{\mathrm{zero}}\end{aligned}$$

For the final commitment, under a centered Gaussian approximation for each coefficient, the condition $\sigma^2_{\mathsf{com}''} \leq (q/(2p \cdot z^*(p_{\mathsf{fail}})))^2$, where $z^*(p_{\mathsf{fail}}) = \sqrt{2} \cdot \mathsf{erf}^{-1}(1 - p_{\mathsf{fail}})$, gives an estimated decryption error of at most p_{fail}. The threshold-decryption error is estimated from σ^2_{part} following [33, Theorem 2].

6 Evaluation

This section presents our evaluation. We start with our methodology and then present our benchmarks based on our implementation.

6.1 Methodology

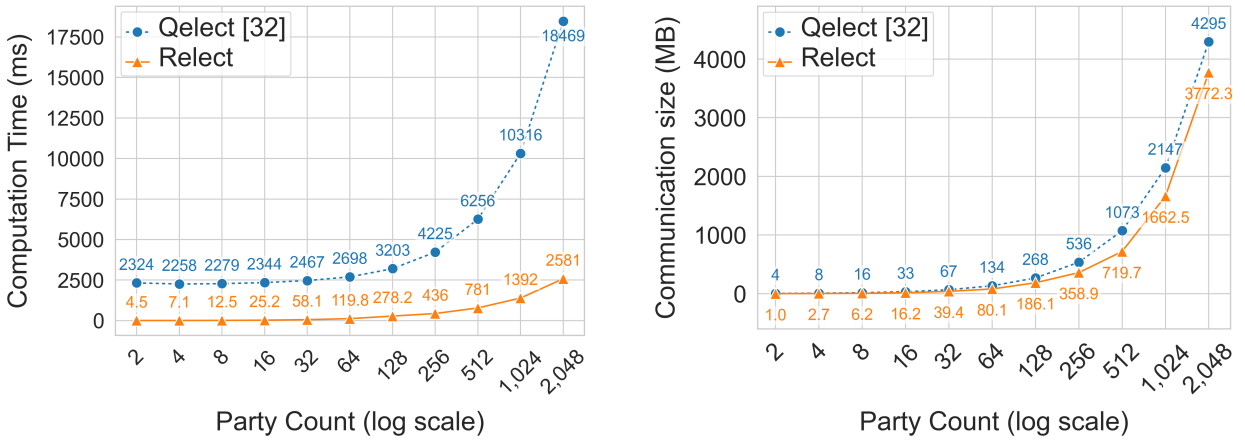
We implement the Relect in Rust. Then, we compare its performance with Qelect [58] scheme, which is the state-of-the-art constant-round lattice-based SSLE protocol implemented with SEAL [54] library.

We benchmark both Relect and Qelect with the server equipped with "Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz". We run the Qelect benchmark with its public code [57] using the same environment as Relect (both with a single thread) to ensure a fair comparison. For $G \geq 256$, the computation time of Relect is benchmarked with the optimizations introduced in § 2.5.

We benchmark $G \in [2, 4, 8, \ldots, 2048]$ for computation since these are the parameters where prior work [58] claim to have computation runtime under the block time of Ethereum (see [58, Fig. 2]) and benchmark $G \in [2, 4, 8, \ldots, 128]$ for the communication time, following prior work [58] as well. All FHE parameters of Relect for benchmark are provided in § A by using our noise analysis in § 5.

Our benchmark includes all the *online* procedures of Relect, including distributed decryption. The zero encryptions used in ParElect can be precomputed with fresh, independent randomness for each election. Our communication benchmark follows Ajax [33] for distributed decryption, and we discuss its distributed key generation for setup in § B.

6.2 Benchmark



(a) Computation runtime comparison between Qelect (1-thread) [58] and Relect (1-thread) Algorithm 2.

(b) Communication size comparison among Qelect [58] and Relect Algorithm 2.

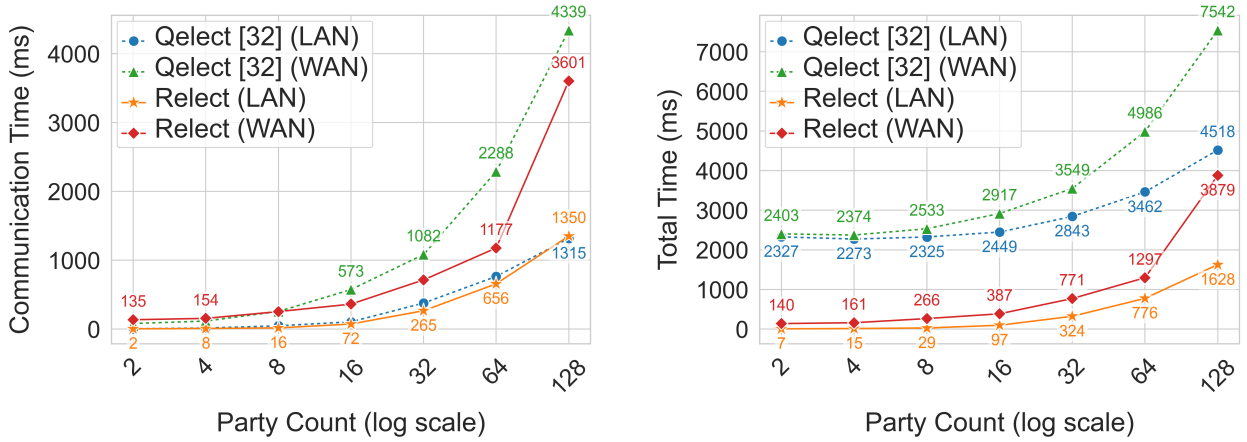
Figure 2: Performance comparison (computation runtime and communication size) between Qelect and Relect.

Computation. We start by comparing the per-party computation costs between Relect and Qelect. Fig. 2a shows the computation time of each party when executing Relect and Qelect. As shown, the computation time of Relect is much shorter compared to Qelect. Specifically, Relect achieves 7.15× to 42.4× low computation time than Qelect for 32 – 2048 parties.[4]

Communication size. As mentioned in § 3, "broadcast" means that a message is sent to all other $G-1$ parties via *pairwise* authenticated channel. Therefore, the broadcast size is the size of the message to be transferred multiplied by $G-1$. Fig. 2b shows the communication size of four rounds of communication of Relect and two rounds of communication of Qelect during the election. Relect use the distributed decryption algorithm [33, Figure 10], which needs two rounds of communication. Note that in the first communication

[4]One could alternatively use other libraries such as TFHE-rs [60], which is based on FFT instead of NTT. This can potentially yield better computation time, which we left for future work to explore.

round of distributed decryption, parties 1 to $G-1$ send their messages to party 0 (not a broadcast). It is easy to see that our communication is smaller than Qelect for all party sizes.



(a) Communication time comparison among Qelect [58] and Relect Algorithm 2.

(b) Total time comparison between Qelect [58] and Relect Algorithm 2.

Figure 3: Comparison of communication time and total time between Qelect and Relect.

Communication in LAN and WAN settings. Fig. 3a shows the communication time (via pairwise communication following Qelect [58]) of Qelect and Relect with $G \leq 128$. Here, for the local area network (LAN) setting, we put all instances under the AWS us-east-1 region (Virginia). Then, in this setting, the latency is about $0.2 - 0.4$ms and the bandwidth is less than 5 Gbit/s. For the wide area network (WAN) setting, we use the traffic control tool 'tc' to emulate network latency (30 ms) and enforce a bandwidth limit of 1 Gbit/s.

It is easy to see that Relect has a more favorable communication time than Qelect in the WAN setting. For the LAN setting, we are slightly slower than Qelect for 128 parties since we require an additional round of communication for instance generation that is not necessary for Qelect due to its *trusted setup*[5], and another for distributed decryption, which we believe is a favorable tradeoff due to the disadvantages and security concerns of trusted setup (as also discussed in § 1.1). Furthermore, we show later in § 6.4, we can optimize the communication in the LAN setting using recursive-doubling, and with this optimization, our communication is preferred in both settings. [6]

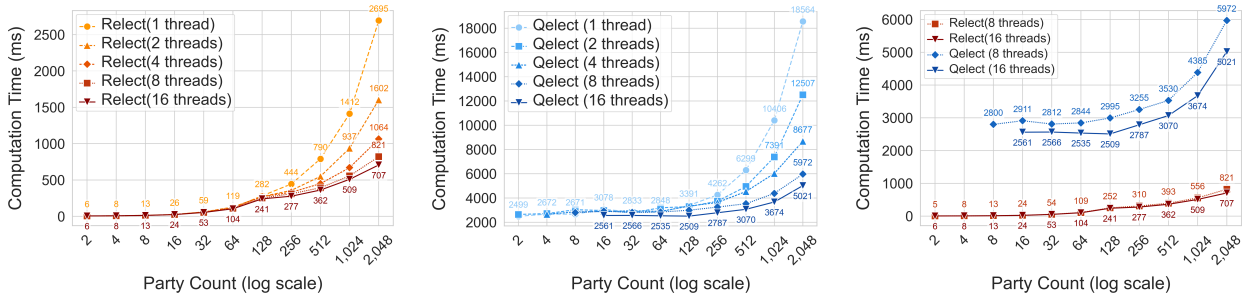
Fig. 3b reports the end-to-end runtime (computation time plus communication time) of Relect and Qelect under both LAN and WAN settings; overall, Relect is consistently faster than Qelect. For 2 to 128 parties, Relect is 2.77 to 345 times faster than Qelect in LAN setting, and 1.94 to 17.2 times faster in WAN setting.

Note that in LAN setting, the local computation of Qelect takes at least 70% of the total runtime, while ours only takes $< 18\%$ of the total runtime for ≥ 32 parties: making the performance bottleneck communication, not computation. This is precisely why we can be 2.77 to 345 times faster, even though our communication time is similar to Qelect. In the WAN setting, while communication can take up a majority of the runtime for 128 parties, for Qelect, the computation still takes at least 42% of the total time. Furthermore, since we reduced the communication size as well, which greatly reduces the communication time in the WAN setting, our total runtime can be 1.94 to 17.2 times faster.

[5]Note that this is an implicit optimization Qelect made: the instance generation is performed offline instead of online, since the randomness generation for each party is finished during trusted setup, which makes their leader election non-dynamic as discussed in § 1.1.

[6]One may also note that the communication time we report for Qelect is significantly shorter than the one reported in [58]. This is because we use a different implementation with the Rust tokio library [55] and use asynchronous sending and receiving to minimize the impact of network latency. However, to guarantee a fair comparison, we use our new implementation for Qelect communication as well.

6.3 Multi-threading



(a) Computation time of Relect with different thread counts.

(b) Computation time of Qelect with different thread counts.

(c) Comparison between Qelect [58] and Relect Algorithm 2.

Figure 4: Comparison of parallel computation times with different thread counts.

Fig. 4a illustrates the computation time of Relect across different thread counts. Three operations within Relect can be parallelized: Oblivious Expansion, Commit Re-randomization, and Encrypted Commitment Generation. However, for a small number of parties, Relect already executes very quickly, meaning multi-threading primarily yields performance gains in large-scale settings. In our implementation, Commit Re-randomization and Encrypted Commitment Generation are only parallelized when the number of parties is at least 256. For Oblivious Expansion, a tree-structured homomorphic operation, requires $O(G \log G)$ key-switching operations, so its parallelization also only shows a noticeable speedup when the party count is sufficiently large. Because the total computation time includes external products between RLWE and RGSW that are difficult to parallelize, the overall speedup does not scale linearly with the number of threads. Fortunately, this overhead is mitigated by the optimization discussed in § 2.5, which restricts the maximum number of external products to 128 (or more generally λ, the security parameter).

Fig. 4b presents the corresponding parallel computation times for Qelect. Due to their implementation constraints of Qelect, the number of allocated threads cannot exceed the total number of parties. Consequently, we only report results for configurations where the party count is greater than or equal to the thread count.

Fig. 4c directly compares the multi-threading computation times of Qelect and Relect. For settings with 32 to 2048 parties, the computation runtime of Relect is 7.27× to 51.8× faster than Qelect when using 8 threads, and 7.10× to 48× faster when using 16 threads. This improvement is comparable to (if not slightly better than) the single-thread computation time advantage we show in Fig. 2a.

Finally, note a minor discrepancy: the single-threaded baseline runtimes reported in Fig. 4a and Fig. 4b were collected during separate execution runs from the single-threaded results shown earlier in Fig. 2a. Although both used identical experimental settings, the reported averages differ slightly. This minor variance is expected from normal run-to-run execution and does not impact the overall performance trends.

6.4 Recursive-doubling Communication

This section briefly discusses an optimization to reduce the communication time applicable to both Qelect [58] and Relect

Instead of the naive fully connected (pairwise) all-to-all exchange, where each party communicates with all other $G-1$ parties, we can organize communication using a recursive-doubling all-to-all pattern. All-to-all means that every party eventually obtains the union of messages from all parties.

Assuming $G = 2^k$, the protocol proceeds in $k = \log G$ synchronous rounds. In round $r \in \{0, \ldots, k-1\}$, each party i exchanges messages with party $i \oplus 2^r$, so that the set of messages held by each party doubles in size each round.

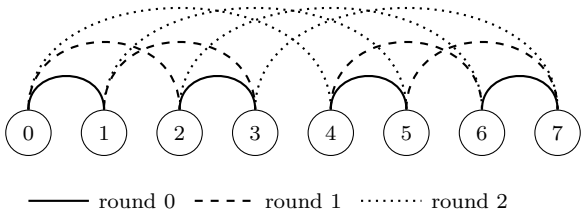


Figure 5: Recursive-doubling party schedule for $G = 8$ ($k = \log G = 3$ rounds). In round r, each party exchanges with party at distance 2^r (equivalently $i \leftrightarrow i \oplus 2^r$).

Recursive doubling completes all-to-all dissemination in $O(\log n)$ rounds with only $O(\log n)$ partners per party, greatly reducing latency and connection-management overhead. Its downside is that the per-round payload grows (often doubling), so total bandwidth and peak buffering can be high, and the protocol remains sensitive to stragglers under synchronous rounds.

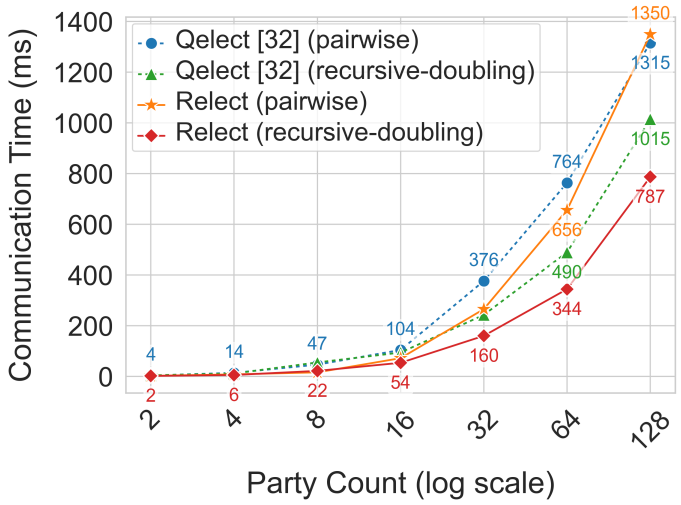


Figure 6: Communication time comparison among Qelect [58] and Relect Algorithm 2 in LAN setting.

Fig. 6 shows that the recursive-doubling communication can improve the communication time in a LAN setting. In a LAN setting, where latency is low and bandwidth is ample, recursive doubling exhibits a more balanced communication load and requires fewer peer connections, making it easier to manage. However, in a WAN setting, the accumulated latency of ($k = \log G$) synchronous rounds, together with bandwidth constraints, can diminish these benefits and make its advantage less pronounced.

Besides, we implement the pairwise communication by asynchronously sending and receiving all messages to and from the other $G - 1$ parties in a concurrent manner. Compared to naive $O(G)$ rounds all-to-all exchange, our implementation has a shorter communication time.

7 Conclusion

This work introduces Relect, an SSLE scheme based on RLWE with low computation time and communication cost. Relect *does not* rely on black-box FHE constructions such as BFV [21, 8] or TFHE [14]. We exploit the algebraic structures of RLWE [46] and RGSW [48] directly, designing novel constructions to avoid costly homomorphic computations. Indeed, Relect achieves a speedup of one to two orders of magnitude compared

to the prior work Qelect in terms of FHE computation, while also reducing the communication costs. As a result, in terms of end-to-end time, in the LAN setting, we are 2.77–345× faster, and in the WAN setting, we are 1.94 to 17.2× faster than Qelect for 2 to 128 parties.

Furthermore, we achieve this great efficiency improvement while *removing* the trusted setup required by Qelect. This eliminates the general security concern with respect to trusted setup while at the same time achieving the flexibility of choosing the leader *dynamically* in each round, instead of fixed in advance in the setup phase as in Qelect. The latter requirement by Qelect could cause additional security concerns as discussed in § 1.1, and Relect can fully avoid it.

Acknoledgement

Yu Yu is supported by the National Natural Science Foundation of China (Grant Nos. 62125204 and 92270201) and Innovation Program for Quantum Science and Technology (No. 2021ZD0302900). We thank ChatGPT for verifying the correctness of the construction, identifying a minor bug, and suggesting simple fixes.

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Journal of Mathematical Cryptology. Volume 9, Issue 3, Pages 169-203, ISSN (Online) 1862-2984, ISSN (Print) 1862-2976 DOI: 10.1515/jmc-2015-0016, 2015.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 21–23, 2018.

[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] Foteini Baldimtsi, Varun Madathil, Alessandra Scafuro, and Linfeng Zhou. Anonymous lottery in the proof-of-stake setting. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 318–333. IEEE, 2020.

[5] 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.

[6] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

[7] 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.

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

[9] 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.

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

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

[12] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM CCS 2018*. ACM Press, October 15–19, 2018.

[13] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS 2017*. ACM Press, October 31 – November 2, 2017.

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

[15] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 670–699, Singapore, December 6–10, 2021. Springer, Cham, Switzerland.

[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] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In *ACM CCS 2021*. ACM Press, November 15–19, 2021.

[18] Ashwini Dalvi, Apoorva Jain, Smit Moradiya, Riddhisha Nirmal, Jay Sanghavi, and Irfan Siddavatam. Securing neural networks using homomorphic encryption. In *2021 International Conference on Intelligent Technologies (CONIT)*, pages 1–7, 2021.

[19] Hanyue Dou, Peifang Ni, Yingzi Gao, and Jing Xu. Mobius: Enabling byzantine-resilient single secret leader election with uniquely verifiable state. NDSS'26, 2025.

[20] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 40–56, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

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

[22] Ben Fisch, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. Thorpir: Single server pir via homomorphic thorp shuffles. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2024.

[23] Ben Fisch, Zeyu Liu, Eran Tromer, and Yunhao Wang. UnifOMR: Oblivious message retrieval with near-optimal concrete efficiency. ACM CCS 2026, 2026.

[24] 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.

[25] Mariana Gama, Emad Heydari Beni, Jiayi Kang, Jannik Spiessens, and Frederik Vercauteren. Blind zkSNARKs: for private proof delegation and verifiable computation over encrypted data. *CiC*, 2(3):10, 2025.

[26] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*, pages 690–719. Springer, 2019.

[27] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Advances in Cryptology - CRYPTO 2010*. Springer, 2010.

[28] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*. ACM, 2009.

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

[30] Tiantian Gong, Ryan Henry, Alexandros Psomas, and Aniket Kate. More is merrier: Relax the non-collusion assumption in multi-server PIR. In *2024 IEEE Symposium on Security and Privacy*, pages 4348–4366, San Francisco, CA, USA, May 19–23, 2024. IEEE Computer Society Press.

[31] Tiantian Gong, Aniket Kate, Hemanta K. Maji, and Hai H. Nguyen. Disincentivize collusion in verifiable secret sharing. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part V*, volume 15605 of *LNCS*, pages 34–64, Madrid, Spain, May 4–8, 2025. Springer, Cham, Switzerland.

[32] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 396–416, New York, NY, USA, 2023. Association for Computing Machinery.

[33] Zhenkai Hu, Haofei Liang, Xiao Wang, Xiang Xie, Kang Yang, Yu Yu, and Wenhao Zhang. Ajax: Fast threshold fully homomorphic encryption without noise flooding. Cryptology ePrint Archive, Paper 2025/1834, 2025.

[34] Wen jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 24–27, 2021.

[35] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security 2018*. USENIX Association, August 15–17, 2018.

[36] 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`.

[37] Joon-Woo Lee, Hyungchul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.

[38] Joon-Woo Lee, Eunsang Lee, Young-Sik Kim, and Jong-Seon No. Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption. In *Advances in Cryptology – ASIACRYPT 2023*, pages 36–68. Springer Nature Singapore, 2023.

[39] Keewoo Lee and Yongdong Yeo. SophOMR: Improved oblivious message retrieval from SIMD-aware homomorphic compression. USENIX Security 2026, 2026.

[40] Haofei Liang, Zeyu Liu, Eran Tromer, Xiang Xie, and Yu Yu. InstantOMR: Oblivious message retrieval with low latency and optimal parallelizability. USENIX Security 2026, 2026.

[41] Haofei Liang, Zeyu Liu, Yunhao Wang, Xiang Xie, Yu Yu, and Fan Zhang. Relect, 2026. https://github.com/haofeiliang/new-ssle.

[42] Zeyu Liu, Katerina Sotiraki, Eran Tromer, and Yunhao Wang. Snake-eye resistant PKE from LWE for oblivious message retrieval and robust encryption. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part III*, volume 15603 of *LNCS*, pages 126–156, Madrid, Spain, May 4–8, 2025. Springer, Cham, Switzerland.

[43] Zeyu Liu and Eran Tromer. Oblivious message retrieval. In *CRYPTO 2022, Part I*, August 15–18, 2022.

[44] Zeyu Liu, Eran Tromer, and Yunhao Wang. Group oblivious message retrieval. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4367–4385, 2024.

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

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

[47] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 22–26, 2022.

[48] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in FHEW-like cryptosystems. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 17–28, 2021.

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

[50] National Institute of Standards and Technology. Post-quantum cryptography standardization, n.d. Accessed: 2025-08-27.

[51] Kabir Peshawaria, Zeyu Liu, Ben Fisch, and Eran Tromer. Laminate: Succinct SIMD-friendly verifiable FHE. Cryptology ePrint Archive, Report 2025/2285, 2025.

[52] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, pages 169–179, 1978.

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

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

[55] Tokio Contributors. Tokio: A runtime for writing reliable network applications with rust, 2026. Accessed: 2026-01-10.

[56] Ruida Wang, Jincheol Ha, Xuan Shen, Xianhui Lu, Chunling Chen, Kunpeng Wang, and Jooyoung Lee. Refined TFHE leveled homomorphic evaluation and its application. Cryptology ePrint Archive, Paper 2024/1318, 2024.

[57] Yunhao Wang and Fan Zhang. Qelect, 2025. https://github.com/wyunhao/Qelect.

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

[59] 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.

[60] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. `https://github.com/zama-ai/tfhe-rs`.

[61] Xinxuan Zhang, Ruida Wang, Zeyu Liu, Binwu Xiang, Yi Deng, Ben Fisch, and Xianhui Lu. Phalanx: An FHE-friendly SNARK for verifiable computation on encrypted data. In *ACM CCS 2025*, pages 4035–4048. ACM Press, November 2025.

[62] Lulu Zhou, Zeyu Liu, Fan Zhang, and Mike Reiter. Cruditee: A stick-and-carrot approach to building trustworthy cryptocurrency wallets with tees. *Advances in Financial Technologies – AFT 2024*, abs/2407.16473, 2024.

[63] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part I*, volume 14004 of *LNCS*, pages 395–425, Lyon, France, April 23–27, 2023. Springer, Cham, Switzerland.

A Parameters

In this section, we provide the parameters of Relect for $G \in \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}$. We test the security of these parameters with lattice-estimator [1]. All parameters provide at least 128 bits of security, and the estimated per-coefficient error probability of threshold decryption is below 2^{-30} under the variance model of § 5.2 and the Gaussian approximation of [33, Theorem 2].

A.1 Commit Parameters

Here shows three sets of the commitment parameters. In Relect, commit is an RLWE ciphertext encrypts $0 \in \mathbb{Z}[X]/(X^n + 1)$. These parameters are suitable for different party count sets. The decryption error probability of the final com''' with the selected leader's secret key sk_r is less than 2^{-44} for all commit parameters and all G according to the improved noise control method introduced in § 5.1.

For the noise with $\sigma_{\mathsf{com}} = 0.849$, it can be sampled with the binary discrete gaussian distribution, which is a discrete gaussian with specific $\sigma = \sqrt{1/(2\ln 2)} \approx 0.849$ and introduced in BLISS [20].

Table 1: Commit Parameters($G \in \{2, 4, 8, 16\}$)

Commit	n	t	q	sk	σ_{com}
	512	2	12289	Ternary	3.7

Table 2: Commit Parameters($G \in \{32, 64, 128\}$)

Commit	n	t	q	sk	σ_{com}
	1024	2	18433	Ternary	0.849

Table 3: Commit Parameters($G \in \{256, 512, 1024, 2048\}$)

Commit	n	t	q	sk	σ_{com}
	1024	2	40961	Ternary	0.849

A.2 ThFHE Parameters

Here shows the main ThFHE parameters without the plaintext modulus T, because the plaintext modulus T equals the ciphertext modulus q of the commit parameters. To provide enough noise budget for threshold decryption, we use the product of multiple prime numbers as ThFHE's ciphertext modulus Q.

Table 4: ThFHE Parameters($G \in \{2, 4, 8, 16, 32, 64, 128\}$)

	N	msk	σ
ThFHE	4096	Ternary	0.849
	Q		
	1125899906826241 * 1125899906629633		

Table 5: ThFHE Parameters($G = 256$)

	N	msk	σ
ThFHE	4096	Ternary	6.4
	Q		
	137438822401 * 68719403009 * 68719230977		

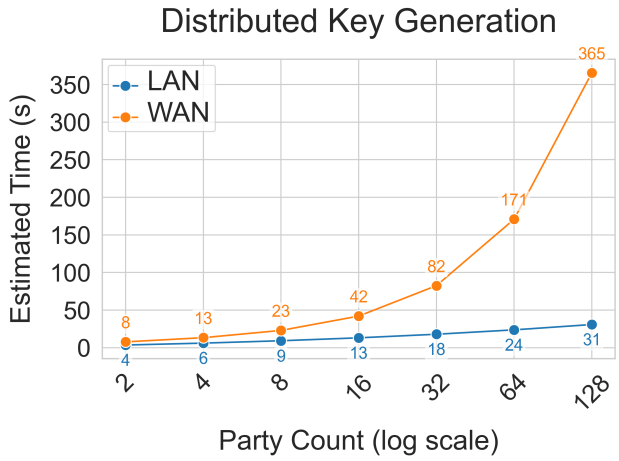


Figure 7: Estimated distributed key generation time of Relect.

A.3 Other Parameters

Besides, the decompose basis and length for RGSW and atk are shown in this subsection. All ℓ_{gsw} and ℓ_{atk} are chosen as small as possible for better performance. But beyond this, $\mathfrak{B}_{\mathsf{gsw}}$, ℓ_{gsw}, $\mathfrak{B}_{\mathsf{atk}}$, and ℓ_{atk} are chosen to keep the estimated error probability of threshold decryption below 2^{-30} under these approximations. Table 8 also shown the noise σ in mpk and atk according the key generation protocol of [33, Figure 9];

B Distributed Key Generation

Ajax reports the distributed key generation runtime for 3 to 31 parties under both LAN and WAN settings. We then fit these measurements to estimate key generation time for other party counts. Since our parameter setting uses $N = 4096$, which is twice that of Ajax, we double the estimated time. In our scheme, similar to

Table 6: ThFHE Parameters($G \in \{512, 1024\}$)

	N	msk	σ
ThFHE	4096	Ternary	12.79
	Q		
	137438822401 * 137438814209 * 68719403009		

Table 7: ThFHE Parameters($G = 2048$)

	N	msk	σ
ThFHE	4096	Ternary	25.58
	Q		
	137438822401 * 137438814209 * 137438773249		

Table 8: Other Parameters

G	noise σ of mpk and atk	$\log_2 \mathfrak{B}_{\mathsf{gsw}}$	ℓ_{gsw}	$\log_2 \mathfrak{B}_{\mathsf{atk}}$	ℓ_{atk}
2	1.200	23	3	33	2
4	1.698	23	3	25	3
8	2.401	23	3	25	3
16	3.396	18	4	25	3
32	4.803	15	5	20	4
64	6.792	15	5	16	5
128	9.605	13	6	16	5
256	13.584	17	5	21	4
512	19.210	17	5	15	6
1024	27.168	14	6	15	6
2048	38.421	13	7	15	6

Ajax, we require a secret key, a public key[7], and key-switching keys (for oblivious expansion, which requires $\log G$ key-switching keys). Therefore, the final estimated time is

$$t = 2 \cdot t_{sk} + (1 + \log G) \cdot 2 \cdot t_{ksk}$$

where t_{sk} denotes the secret key generation time and t_{ksk} denotes the key-switching key generation time.

For the LAN setting, we obtain the following estimation:

$$t_{sk} = -0.016326284 + 0.020461949 \cdot \log(G)$$

$$t_{ksk} = 0.77675919 \cdot G^{0.18536261}$$

For the WAN setting, we obtain the following estimation:

$$t_{sk} = 0.32106321 + 0.05414434 \cdot G$$

$$t_{ksk} = 1.3986241 + 0.16043255 \cdot G$$

The fitted results are shown in Fig. 7. Since these values are obtained through estimation, they are provided for reference only.

Note that one may use an alternative ThFHE scheme instead of Ajax. We choose Ajax since it is the state-of-the-art ThFHE scheme for our purpose, but other schemes may have different advantages suitable for other application scenarios (e.g., Ajax requires an additional round, all parties sending to a single party, during distributed decryption, which might be avoidable in other constructions).

[7]For the public key, we need an RLWE public key instead of an LWE public key as for Ajax [33]. RLWE public key takes less time to generate than the key-switching key, so we estimate the runtime as if it were a key-switching key.