An Efficient Framework for Collusion Resistant Multiparty FHE

Robin Jadoul[1⋆], Barry van Leeuwen[2], and Oliver Zajonc[2,3⋆⋆]

[1] 3MI Labs, Leuven, Belgium.
robin.jadoul@3milabs.tech
[2] COSIC, ESAT, KU Leuven, Leuven, Belgium.
barry.vanleeuwen@kuleuven.be
[3] TU Darmstadt, Darmstadt, Germany
oliver.zajonc@tu-darmstadt.de

Keywords: Multiparty Computation · Fully Homomorphic Encryption · Multiparty FHE · LWE · NTRU · Key Switching

Abstract. Multiparty fully homomorphic encryption (MPFHE) is a generalization of (multi-key) fully homomorphic encryption ((MK)FHE) that lives on the cusp between multiparty computation (MPC) and FHE, enabling a computation over encrypted data using multiple keys. However, contrary to MKFHE, it seeks to reduce the noise inflation based on the number of parties by allowing the parties to first compute shared data in MPC before executing the computation in FHE. However, many works use specific constructions, e.g. Chang-Li, Koirala et. al., and Ma et. al., reliant on the behaviour of a particular scheme and do not consider security against colluding servers and parties, e.g. Passelègue-Stehlé, Okada-Takagi, and Mouchet et. al. In this work we present a new framework that is secure against collusion between servers and parties and allows the use of a broad spectrum of MPC and FHE protocols, while eliminating the noise inflation based on the participating number of parties. We then show a case study of this using the FINAL scheme and show that we reduce the required key material by 60–99.9% compared to the MKFHE FINAL scheme, FINALLY, and 65–70% compared to the Park-Rovira scheme in all cases, and by up to 71% compared to the AKÖ scheme when there are 4 or more parties. Moreover, we reduce the bootstrapping time for the AKÖ, Park-Rovira, and KMS schemes by 75–99.7%.

⋆ Part of this work was performed while the author was at COSIC, KU Leuven.
⋆⋆ This work was done while the author was at COSIC, KU Leuven.

1 Introduction

Fully homomorphic encryption (FHE) is a branch of privacy enhancing technologies (PETs) that allows for a set of parties to jointly compute on private inputs. To do this, each party locally encrypts their message before outsourcing the computation to a server without revealing any inputs or outputs. The mechanism to do this is by exploiting the homomorphic property, allowing for a function evaluation on ciphertext to produce a ciphertext containing the function evaluation on the underlying messages.

However, ever since the introduction by Gentry [Gen09] this has limited the security guarantees that can be offered. First and foremost, due to the nature of FHE, the server needs to be considered semi-honest.[4] Furthermore, a common assumption [LTV12,BCG$^+$17,PS24,OT25] is that in a setting with at least two parties and the server, the server and a user cannot collude. If there is a single key, then the server could receive the encryption of the secret of the honest party and the adversary - knowing the key as there is a dishonest party - could decrypt that message, learning the honest party's private input.

To mitigate this, one could introduce a set of (unique) keys such that each party can encrypt under their own key. This augmentation, called Multi-key FHE [MW16,AJJM20,PVZ24], provides a higher security guarantee, but it also introduces a significant constraint. Most FHE schemes achieve the combination of functionality and security by introducing noise, which prevents an adversary from learning anything about the message in a ciphertext. By introducing noise, any operation done on ciphertexts increases the noise of the resulting ciphertext (e.g. addition adds linear noise, multiplication adds exponential noise). To remove the noise, the decryption step usually has a rounding function which ensures the correct retrieval of the underlying message dependent on the amount of noise present in the ciphertext. It will come as no surprise that after a number of operations the noise can grow so large that the rounding function cannot reliably round back to the underlying message. To accommodate the size of arbitrary computations a new protocol depending on the scheme was introduced called *bootstrapping*. While this allowed for arbitrary computations to be executed, the bootstrapping protocol is often very computationally expensive when compared to the arithmetic operations. One of the largest constraints with Multi-key FHE schemes is that the noise grows significantly faster and is often related to the number of parties that participate in the protocol. As a result, these multi-key protocols often require much larger parameter sets than their single-key counterparts.

Multiparty FHE (MPFHE) mitigates this by using a multiparty computation (MPC) protocol to establish communal keys, reducing the computation to that of single-key FHE while still supporting multiple users. MPC allows a set of users to achieve the same result as FHE: To compute a function on private inputs, however with MPC no server is used to offload the computation and so each party must be involved throughout the evaluation protocol. However, in many cases [PS24,OT25,MTBH21] adding an MPC-based preprocessing phase to the MKFHE protocol to obtain MPFHE once again allow a single party and the server to collude and leak the private information of the honest participants in the protocol. In this paper we present a framework which improves the security offered over the current state-of-the-art by preventing these sorts of semi-honest collusions between the parties and the server from impacting the security of the protocol. Additionally, the framework we present allows us to implement these constructions efficiently agnostic of the underlying MPC and FHE protocols.

[4] There are methods of achieving active security through expensive verification mechanisms.

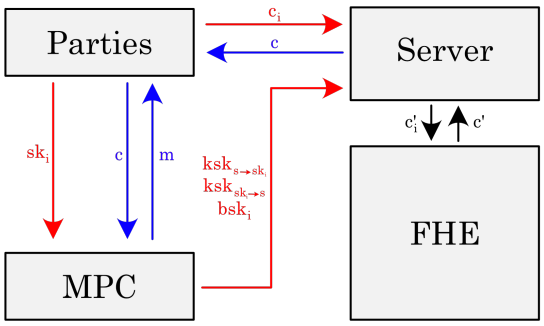


Fig. 1: *A bird's eye overview of the MPFHE framework with encryption in red and decryption in blue .*

The structure of our framework comes in three steps:

Preprocessing: At the start of the protocol, the parties run an efficient semi-honest MPC protocol, e.g. ABY2 [PSSY21], the protocol from [BLO16], or MOTION [BDST22], to generate shared key material. While the exact requirements may change depending on the FHE scheme, usually the material generated consists of keyswitching keys and bootstrapping keys. As this only needs to be executed once at the start and this key material can be reused independent of circuit size and multiple executions as long as the set of parties does not change, this cost can be amortized away.

Evaluation: Once the key material is generated we proceed to the evaluation phase in which the parties and the server execute the FHE scheme to evaluate a function on the encrypted inputs. This execution includes any required bootstrapping procedures, to obtain an encryption containing the function evaluated on the underlying messages. Due to the preprocessing phase our framework avoids the use of the expensive multi-key paradigm and can use the much more efficient single key version for this phase. This directly combats the significantly increased noise growth and limitations that must be set on the number of participating parties.

Decryption: Once each of the parties has obtained the encryption of the output, there are multiple options to proceed with decryption depending on the security guarantees that the framework needs to deliver. To ensure that the framework is resistant to user-server collusion, we will assume that the parties run a round-robin MPC protocol to ensure correct decryption and guarantee output. This means that after the completion of the final bootstrapping procedure, the server only does a single keyswitching to a multi-key LWE ciphertext. Hence, only one keyswitching key needs to be generated for this step. Since this is executed once at the end of the protocol the cost of this step can, similarly to the preprocessing step, simply be amortized away.

If one assumed a-priori that collusion between the server and parties was impossible, then an improvement can be made to our protocol: One could preprocess a shared encryption key, go through the whole process, and at the end the server could keyswitch from the shared encryption key to a single-key ciphertext using the keys of each party. This would require many keyswitching keys and keyswitches, but decryption can now be done without communication. However, as this falls outside our security paradigm we will not treat this as a viable solution in our protocol.

1.1 Our Contributions

A Framework, Multiparty Linear Decryption FHE: We introduce a new framework which allows for the use of any single key linear decryption FHE (LDFHE) scheme, (e.g. BGV/BFV [BGV12,FV12], TFHE [CGGI20], FHEW [MS18,GPvL23], FINAL [BIP$^+$22,JMPP24,PVZ24]) in combination with a semi-honest MPC protocol to build a user-server collusion secure multiparty fully homomorphic encryption (MPFHE) with distinct interactive and non-interactive phases.

An Instantiation, Multiparty FINAL: We instantiate the introduced framework by adapting FINAL [BIP$^+$22] to the new framework to provide the first multiparty FINAL (MP-FINAL) scheme. This scheme provides a significant improvement compared to FINALLY [PVZ24] with a 2.46–7116x reduction in total key material. Moreover, we also provide a thorough analysis on the new scheme and show that the parameters introduced in FINAL (which are shown to not be overstretched) can be implemented directly to obtain an n-party MPFHE scheme for arbitrary n.

An Implementation, Improving the State-of-the-Art: Lastly, we then provide a prototype implementation[5] of MP-FINAL, which we use to benchmark our results against other the current state-of-the-art in Multi-user FINAL, [PVZ24], multiparty FHE, [AKÖ23,PR23], and Multi-Key FHE [KMS24]. In Table 1 we show an abbreviated overview of the improvements.

	Park et al. [PVZ24]	Park & Rovira [PR23]	Kwak et al. [KMS24]	Akin et al. [AKÖ23]	This work
Key size (GB)	5337	2.2	.285	.93	.75
Bootstrap time (ms)	-	900	5650	1800	88.6

Table 1: *A brief overview of our results for $k = 16$ parties.*

1.2 Related Work

To our knowledge, both our formalization of MPFHE and the structure of our framework have not appeared in literature before. Existing concepts in MPFHE, [MTBH21] and the static variant of [AKÖ23], do not consider the security when there is user-server collusion and variants of Threshold FHE, [PS24,OT25] explicitly disallow it. While [MTBH21] is the closest to the framework we propose, the protocol proposed is circuit dependent due to its reliance on BGV/BFV, [BGV12,FV12], rendering comparison between parameter sets impractical. Their bootstrapping also requires collective execution, whereas ours (after preprocessing) can be performed independently by the server. In the Threshold FHE case there is also a scheme that protects against collusions between server and players [CL25], however this scheme has a more widespread use of MPC, requiring it for every input. In our scheme, after the initial preprocessing to obtain keys, no further communication is required throughout the protocol even if additional inputs are required.

Generalizing our approach to a wider net does produce some interesting precursors: MPFHE can be interpreted as a relaxation of the multi-key FHE setting (MKFHE) [LTV12,PVZ24,CDKS19,Par21], where parties are now allowed communication during the setup

[5] The code is available at `https://github.com/KULeuven-COSIC/vinyl`

phase. This has the result that the set of parties has to be, and stay, fixed before the start of the computation and key generation. It is clear that this communication will reduce the overall complexity of the protocols significantly and the proposed framework should translate to these MPFHE schemes without much work.

2 Preliminaries

2.1 Notation

Let R be a ring and let $\mathbb{F}$ be a field. For a polynomial, $g \in R[X]$ or $g \in \mathbb{F}[X]$, we denote by g_i the coefficient corresponding to the ith power of X. We denote vectors by lower-case bold letters, such that $\mathbf{a} = (a_1, \ldots, a_n)$, and matrices by upper-case bold letters, $\mathbf{M} \in M_{r\times c}(\mathbb{F})$, such that $\mathbf{M}^{(i)}$ is the ith row of $\mathbf{M}$ and $\mathbf{M}_{(j)}$ is the jth column of $\mathbf{M}$. As both $\mathbf{M}^{(i)}$ and $\mathbf{M}_{(j)}$ are vectors a single index can be given by $M_j^{(i)} = M_{(j),i} = M_{i,j}$.

Throughout the paper we denote by $\mathcal{R} := \mathbb{Z}[X]/\langle X^N + 1\rangle$ the $2N$-th cyclotomic ring, such that $N = 2^\kappa$ for some $\kappa \in \mathbb{N}$. Similarly, we can define $\mathcal{R}_Q := \mathcal{R}/Q\mathcal{R}$, i.e. the cyclotomic ring with coefficients in $\mathbb{Z}_Q$. Note that any element $g \in R$, $R \in \{\mathcal{R}, \mathcal{R}_Q\}$, can be given as the smallest polynomial of at most degree N in the appropriate coset of R. Hence, we can represent any element, $g \in R$ by a vector representation, $\phi(g) = (g_0, \ldots g_{N-1})$, and this representation is well-defined. As $\mathsf{Im}(\phi) \subset \mathbb{Z}^N$, we can define the infinity norm of an element, g, as $\|g\| = \|\phi(g)\|$.

2.2 Distributions

For a random variable a, we denote by $a \leftarrow D$ the action of sampling a according to the distribution D. We denote by $\mathsf{Var}(a)$ the variance of a. If $D = \mathbb{K}$ for a ring or field $\mathbb{K}$ then the sampling of an element of $\mathbb{K}$ is understood to be uniformly random. Denote by G_σ the discrete Gaussian distribution over a ring $\mathcal{R}$ with variance σ^2 and denote by χ_σ the subgaussian distribution over a ring $\mathcal{R}$. A random variable, V, is called α-subgaussian if the moment generating function fulfills the following equation for some α and all $t \in \mathbb{R}$:

$$\mathbb{E}[\exp(t \cdot V)] \leq \frac{1}{2}\exp(\alpha^2 \cdot t^2).$$

A subgaussian variable $V \leftarrow \chi_\alpha$ fulfills the following properties:

- $\mathsf{Var}(V) \leq \alpha^2$
- If $\mathbf{V} = (V_1, \ldots, V_n)$, $V_i \leftarrow \chi_{\alpha_i}$, then

$$\mathsf{Var}(\mathbf{V}) = \max_{i\in[n]}(\mathsf{Var}(V_i)) = \max_{i\in[n]} \alpha_i.$$

- For $V \leftarrow \chi_\alpha$, $W \leftarrow \chi_\beta$, $t, s \in \mathbb{Z}$

$$\mathsf{Var}(t \cdot V + s \cdot W) = \sqrt{t^2 \cdot \alpha^2 + s^2 \cdot \beta^2}.$$

- Let $v, w \in \mathcal{R}[X]$ such that $v_i \leftarrow \chi_\alpha$, $w_i \leftarrow \chi_\beta$ such that both have mean 0, and $\max(\deg(v), \deg(w)) = N$ then $\mathsf{Var}(v \cdot w) = N \cdot \mathsf{Var}(\phi(v)) \cdot \mathsf{Var}(\phi(w))$.

A specific distribution relevant to our use of NTRU will be the FINAL distribution, which is a subgaussian distribution with variance $\leq \frac{1}{2}$ given as follows:

$$\Pr(f_i = x) = \begin{cases} \frac{1}{4}, & x = -1 \\ \frac{1}{2}, & x = 0 \\ \frac{1}{4}, & x = 1 \end{cases}$$

We denote by $D_{\mathcal{R}}$ the distribution over $\mathcal{R}$ where each of the coefficients is independently drawn from our FINAL distribution.

2.3 Gadget Decomposition

For fixed $Q, B \in \mathbb{Z}$ let $l = \lceil \log_B(Q) \rceil$. The l-dimensional column vector defined by $\mathfrak{g}_{Q,B} := (B^0, \ldots, B^{l-1})$ is called the gadget vector. For any identity matrix $\mathbf{I}_k$ we can then define the gadget matrix $\mathbf{G}_{k,Q,B} = \mathbf{I}_k \otimes \mathfrak{g}_{Q,B}$. To avoid confusion we will write $\mathfrak{g} = \mathfrak{g}_{Q,B} = \mathbf{G}_{1,Q,B}$ dropping parameters if they are clear from context. For any integer $z \in \mathbb{Z}_Q$ represented in the symmetric interval, $[-Q/2, Q/2)$, define its signed decomposition as $\mathfrak{g}^{-1}(z) = (z_0, \ldots, z_{l-1})$ such that $|z_i| < B/2$ for each $i \in [0, \ldots, l-1]$. It is easy to see that $\mathfrak{g}^{-1}(z) \cdot \mathfrak{g} = z$.

This gadget decomposition extends to polynomials in the natural way. For any $f \in \mathcal{R}_Q$ we may define

$$\mathbf{G}^{-1}_{n,Q,B}(f) = \sum_{i=0}^{n-1} \mathbf{G}^{-1}_{n,Q,B}(f_i) \cdot X^i.$$

From this definition it is clear that $\mathbf{G}^{-1}_{n,Q,B}(f) \cdot \mathbf{G}_{n,Q,B} = f$, fulfilling the requirement of a gadget matrix. A well-known consequence of the gadget decomposition is that for any α, $\mathfrak{g}^{-1}(\alpha)$ is γ-subgaussian for $\gamma = O(B)$.

2.4 Multi-key LWE

For the multiparty FINAL scheme, which we introduce in Section 5, we have to introduce two homomorphic encryption schemes each with their own role: The base scheme and the bootstrapping scheme. In this section we introduce the base scheme, which is a multi-key scheme. For this, the most natural choice is multi-key LWE, which we will recall here:

Definition 2.1 (Multi-Key LWE). *Let λ be the security parameter. Then, the Multi-Key* LWE *scheme consists of the following 5 PPT algorithms:* Setup, KeyGen, Enc, Dec, *and* NAND.

- MKLWE.Setup(λ)*: Upon input of a security parameter λ returns the public parameters (q, p, σ). Denote by $\Delta := q/4$.*
- MKLWE.KeyGen(σ)*: Upon input of the public parameters q, p, and σ,* MKLWE.KeyGen *randomly samples $s_i \in \mathbb{F}_2$, $i \in \{0, \ldots, n-1\}$ such that $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{F}_2^n$.*
- MKLWE.Enc$_{\mathbf{s}}(m)$*: Upon input of a key $\mathbf{s}$ and a message $m \in \mathbb{F}_2$,* MKLWE.Enc *generates $\mathbf{a} \leftarrow \mathbb{Z}_q^n$, and $e \leftarrow G_\sigma$. Then, it computes $b = \mathbf{a} \cdot \mathbf{s} + \Delta m + e$. It then outputs a vector of length $k+1$, $\mathsf{ct} = (b, 0, \ldots, 0, \mathbf{a}, 0, \ldots, 0)$ where $\mathbf{a}$ is in the position corresponding to the party generating the encryption, i.e. $\mathcal{P}_i$ is in the ith position.*

– $\mathsf{MKLWE.NAND}(\mathsf{ct}_1, \mathsf{ct}_2)$*: Upon input of two ciphertexts of the form*

$$ct_i = (b_i, \mathbf{a}_{i,1}, \ldots, \mathbf{a}_{i,k}),$$

$\mathsf{MKLWE.NAND}$ *outputs*

$$\mathsf{ct} = \left(\frac{5q}{8} - b_1 - b_2, \mathbf{a}_{1,1} + \mathbf{a}_{2,1}, \ldots, \mathbf{a}_{1,k} + \mathbf{a}_{2,k}\right).$$

We denote a ciphertext encrypting m under multiple keys (such as $\mathsf{sk}_1, \ldots, \mathsf{sk}_k$) as $\mathsf{MKLWE.Enc}_{\mathsf{sk}_1,\ldots,\mathsf{sk}_k}(m)$.

– $\mathsf{MKLWE.Dec}(\mathbf{s}_1, \ldots, \mathbf{s}_k, \mathsf{ct})$*: Upon input of keys $\mathbf{s}_1, \ldots, \mathbf{s}_k$ and ciphertext ct of the form $(b, \mathbf{a}_1, \ldots, \mathbf{a}_k)$* $\mathsf{MKLWE.Dec}$ *computes*

$$m^* = \left\lfloor \frac{b - \sum_{i=1}^{k} (\mathbf{a}_i \cdot \mathbf{s}_i)}{\Delta} \right\rceil.$$

The MKLWE *scheme is correct if*

$$\mathsf{MKLWE.Dec}(\mathsf{sk}_1, \ldots, \mathsf{sk}_k, (\mathsf{MKLWE.Enc}_{\mathsf{sk}_1,\ldots,\mathsf{sk}_k}(m)) = m.$$

To show that the MKLWE.NAND correctly evaluates, the following Lemma was proven in [PVZ24].

We note that this simplifies to single-key LWE by selecting $k = 1$.

Lemma 2.1 (Correctness of LWE-NAND ([PVZ24] Lemma 2)). *For two ciphertexts c_1, c_2 of the form $c_i = (b_i, \mathbf{a}_1, \ldots, \mathbf{a}_k)$, $b_i = \Sigma_{i=1}^{k}(\mathbf{a}_i \cdot \mathbf{s}_i) + \Delta \cdot m_i + \mathbf{e}$ and secret keys sk_i. Consider $\mathbf{c} = \mathsf{MKLWE.NAND}(c_1, c_2)$. If $\|\mathbf{e}_1 + \mathbf{e}_2\| \leq \frac{q}{4}$, then the protocol $\mathsf{MKLWE.Dec}_{\mathsf{sk}_1,\ldots,\mathsf{sk}_k}(\mathbf{c})$ outputs $\mathsf{NAND}(m_1, m_2)$.*

2.5 Single-Key NGS

For the bootstrapping scheme FINAL [BIP+22] introduces an NTRU-based bootstrapping scheme, NGS. Similar to other accumulator based bootstrapping schemes, the goal of the bootstrapping scheme is to allow partial decryption to reduce the noise before returning to the base scheme. We introduce this scheme, as well as properties of the noise generation.

Definition 2.2 (Single-Key NGS). *Let λ be the security parameter. The* NGS *scheme consists of four probabilistic polynomial time algorithms:* Setup, KeyGen, SEnc, VEnc.

– $\mathsf{NGS.Setup}(\lambda)$*: Upon input of a security parameter λ,* Setup *returns the public parameters (N, Q, ζ, B, ℓ).*

– $\mathsf{NGS.KeyGen}(\zeta, N)$*: Upon input of N and ζ,* KeyGen *samples $f' \leftarrow \chi_\zeta^N$ until $f^{-1} = (4 \cdot f' + 1)^{-1}$ exists in $\mathcal{R}_Q$. Then it outputs $\mathsf{sk} = f$.*

– $\mathsf{NGS.SEnc}_{sk}(m)$*: Given a ternary polynomial m, sample $g \leftarrow \chi$, define $\Delta = \lfloor Q/4 \rceil$, and output $c = g/f + \Delta m \in \mathcal{R}_Q$.*

– $\mathsf{NGS.VEnc}_{sk}(m)$*: Given a ternary polynomial m, sample $g_i \leftarrow \chi^N$. Define $\mathbf{g} := (g_0, \ldots, g_{\ell-1})$ and $\mathfrak{g} = (B^0, \ldots, B^{\ell-1})$. Output $c = \mathbf{g}/f + \mathfrak{g}m \in \mathcal{R}_Q^\ell$.*

The NGS scheme defines the external product of a scalar encryption $c = g/f + \Delta m$ and a vector encryption $\mathbf{c} = \mathbf{g}/f + \mathfrak{g}m'$ as

$$c \boxdot \mathbf{c} := \mathfrak{g}^{-1}(c) \cdot \mathbf{c}.$$

Defining the noise of a scalar encryption c as $\mathsf{err}(c) := c \cdot f - \Delta m$ and of a vector encryption as $\mathsf{err}(\mathbf{c}) := \mathbf{c} \cdot f - \mathfrak{g}m$, it can then be verified that the result of this operation is a scalar encryption of $m \cdot m'$ with noise term $\mathfrak{g}^{-1}(c) \cdot \mathbf{g} + g \cdot m'$. Recall that $\mathfrak{g}^{-1}(c)$ is γ-subgaussian, hence we can quantify the noise growth when performing a sequence of external products.

Lemma 2.2 (Noise of a sequence of external products [BIP+22]). *Let $c_0 = g_0/f + \Delta \cdot m_0$ be an* NGS *scalar ciphertext encrypting a binary polynomial $m_0 \in \mathcal{R}_Q$, and let for each $i \in [k]$ $\mathbf{c}_i = \mathbf{g}_i/f + \Delta \cdot m_i$ be an* NGS *vector ciphertext encrypting $m_i \in \mathcal{R}_Q$. If $\mathsf{ct} = c_0 \boxdot_{i=1}^{k} \mathbf{c}_i$, then*

$$\mathsf{Var}(\mathsf{err}(\mathsf{ct})) \leq N \cdot \ell \cdot \gamma^2 \cdot \Sigma_{i=1}^{k} \mathsf{Var}(\mathbf{g}_i) + \mathsf{Var}(g_0) + 4 \cdot \zeta^2.$$

Furthermore, if all ciphertexts are fresh, then

$$\mathsf{Var}(\mathsf{err}(\mathsf{ct})) \leq (4 + ((k+1) \cdot N \cdot \ell \cdot \gamma^2)) \cdot \zeta^2.$$

2.6 Keyswitching

There exist keyswitching procedures that transfer an encrypted value between different keys under the same scheme, or even different keys under different schemes. Here specifically, we recall constructions that keyswitch from NGS to MKLWE and from MKLWE to MKLWE. Note that by setting the number of keys to 1 for the MKLWE setting we obtain single key LWE keyswitching for either construction.

Keyswitching from MKLWE to MKLWE Based on [CDKS21], we generalize the keyswitching method to encompass switching MKLWE to MKLWE ciphertexts. This method consists of two steps:

1. In Algorithm 1 it is described how the keyswitching material is obtained.
2. Based on the material obtained, Algorithm 2 then describes how to do the keyswitching protocol.

Additionally, we provide a proof of correctness and quantify the noise generated by applying the keyswitching procedure.

Algorithm 1: MKLWE to MKLWE keyswitching key generation, $\mathsf{KSKGen}_{\mathsf{MKLWE}}$

Input: A set of source LWE keys $\bar{\mathbf{s}}_1, \ldots, \bar{\mathbf{s}}_t$, and a set of destination LWE keys $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_k$.
Public Parameters : (n, q, B).
Output: Keyswitching keys $\mathsf{ksk}_{i,j,v,w}$.

for *every key* $\bar{\mathbf{s}}_w, w \in [t]$ **do**
  Parse $\bar{\mathbf{s}}_w = (\bar{s}_{w,1}, \ldots \bar{s}_{w,n})$;
  Compute $\mathsf{ksk}_{i,j,v,w} = \mathsf{MKLWE.Enc}_{\mathbf{s}_1,\ldots,\mathbf{s}_k}(v\bar{s}_{w,i}B^j)$ for $i \in [n], j \in [\ell], v \in [B], w \in [t]$ with $\Delta = 1$;
  Output $\mathsf{ksk}_{i,j,v,w}$;

It is clear that by setting $w = 1$, $k = 1$, or $w = k = 1$ in Algorithm 1 and Algorithm 2 we obtain all single key/ multi-key variants of the protocol.

Algorithm 2: MKLWE to MKLWE keyswitching, $\mathsf{KS}_{\mathsf{MKLWE}}$

Input: MKLWE ciphertext $\overline{\mathsf{ct}}$ encrypting m under the set of keys $\overline{\mathbf{s}}_1, \ldots, \overline{\mathbf{s}}_t$. The output of Algorithm 1, $\{\mathsf{ksk}_{i,j,v,w}\}_{i\in[n],j\in[\ell],v\in[B],w\in[t]} = \mathsf{KSKGen}_{\mathsf{MKLWE}}((\overline{\mathbf{s}}_1, \ldots, \overline{\mathbf{s}}_t), (\mathbf{s}_1, \ldots, \mathbf{s}_k))$.

Output: MKLWE ciphertext $\mathsf{ct} = (b, \mathbf{a}_1, \ldots, \mathbf{a}_k)$

1 Parse $\mathsf{ct} = \left(\overline{b}, \overline{\mathbf{a}}_1, \ldots, \overline{\mathbf{a}}_t\right)$;

2 Compute $(b, \mathbf{a}_1, \ldots, \mathbf{a}_k) \leftarrow (\overline{b}, \mathbf{0}, \ldots, \mathbf{0}) - \sum_{i,j,w} \mathsf{ksk}_{i,j,\mathfrak{g}^{-1}(\overline{\mathbf{a}}_w)_{i,j},w}$;

3 Output $(b, \mathbf{a}_1, \ldots, \mathbf{a}_k)$;

Note that $\mathsf{KSKGen}_{\mathsf{MKLWE}}$ theoretically generates a keyswitching key for each possible value of v for every $\overline{s}_{w,i}$. However, in practice (see Algorithm 2) we only use the one for which $\mathfrak{g}^{-1}(\overline{\mathbf{a}}_w)_{i,j} = v$. If this can be determined beforehand we can reduce the number of keys generated by a factor B.

Lemma 2.3 ($\mathsf{KS}_{\mathsf{MKLWE}}$ correctness and noise). *Let* ct' *be an* MKLWE *encryption of* m *under the key* s' *and let* ct *be the output of* $\mathsf{KS}_{\mathsf{MKLWE}}$ *with input* ct'*. Then,* ct *is a valid* MKLWE *encryption of* m *under the keys* $\mathbf{s}_1, \ldots, \mathbf{s}_k$ *and:*

$$\mathsf{Var}(\mathsf{err}(\mathsf{ct})) \leq n \cdot \ell \cdot t \cdot \mathsf{Var}(\mathsf{err}(\mathsf{ksk})) + \mathsf{Var}(\mathsf{err}(\mathsf{ct}')),$$

where $\mathsf{Var}(\mathsf{err}(\mathsf{ksk}))$ *is that of a fresh ciphertext.*

Proof. Let $\mathsf{ct}' = (b', \mathbf{a}'_1, \ldots, \mathbf{a}'_t)$ such that $b' = \sum_{w=1}^{t} \mathbf{a}'_w \cdot \mathbf{s}'_w + \Delta \cdot m + e$ and let $\mathsf{ksk}_{i,j,v,w} = \mathsf{MKLWE.Enc}_{\mathbf{s}_1,\ldots,\mathbf{s}_n}(vs'_{w,i}B^j)$ as given in Algorithm 1. Then, executing the $\mathsf{KS}_{\mathsf{MKLWE}}$ operations in Algorithm 2 allows for the computation of the following:

$$\begin{aligned}
&(b', \mathbf{0}, \ldots, \mathbf{0}) - \sum_{i,j,w} \mathsf{ksk}_{i,j,\mathfrak{g}^{-1}(\mathbf{a}'_w)_{i,j},w} \\
&= \left(b' - \sum_{i,j,w} \left(\mathbf{a}_{i,w} \cdot \mathbf{s}_i + \mathfrak{g}^{-1}(\mathbf{a}'_w)_{i,j} \cdot s'_{w,i} \cdot B^j + e_{i,j,w} \right), \sum_w -\mathbf{a}_{1,w}, \ldots, \sum_w -\mathbf{a}_{n,w} \right)
\end{aligned}$$

Now consider the first element of the resulting vector. It is easy to observe that $\mathfrak{g}^{-1}(\alpha) \cdot \{B^j\}_{j\in[l]} = \alpha$ for any α, hence

$$\begin{aligned}
\sum_w \mathbf{a}'_w \cdot \mathbf{s}'_w + \Delta \cdot m + e - \sum_{i,j,w} \left(\mathbf{a}_{i,w} \cdot \mathbf{s}_i + \mathfrak{g}^{-1}(\mathbf{a}'_w)_{i,j} \cdot s'_{w,i} \cdot B^j + e_{i,j,w} \right) \\
= \sum_{i,w} -\mathbf{a}_{i,w} \cdot \mathbf{s}_i + \Delta m + e + \sum_{i,j,w} e_{i,j,w}
\end{aligned}$$

Setting $\sum_w -\mathbf{a}_{i,w} = \mathbf{a}_i$ we obtain a valid MKLWE ciphertext encrypting m under the keys $\mathbf{s}_1, \ldots, \mathbf{s}_k$. The noise of this ciphertext then consists of the final two terms. Hence

$$\mathsf{Var}(\mathsf{err}(\mathsf{ct})) \leq n \cdot \ell \cdot t \cdot \mathsf{Var}(\mathsf{err}(\mathsf{ksk})) + \mathsf{Var}(\mathsf{err}(\mathsf{ct}')),$$

completing the proof.

In the scheme defined in Section 5 the use of this keyswitching is limited to single key LWE to k-key MKLWE, hence limiting the number of keyswitching keys that need to be generated. Since B is a

fixed constant chosen at initialisation, the number of keyswitching keys required is $O(n \cdot \log(n))$. Moreover, the resultant noise after a keyswitch is limited to

$$\mathsf{Var}(\mathsf{err}(\mathsf{ct}) \leq n \cdot \ell \cdot \sigma^2 + \mathsf{Var}(\mathsf{err}(\mathsf{ct}')),$$

as $\mathsf{ksk}_{i,j,v,w}$ is a fresh MKLWE ciphertext with σ-subgaussian noise.

Remark 2.1. For the preprocessing phase, the only keys that need to be generated are keys from each party's key, sk_i, to the communal key, sk_{com}. Hence, the number of keys generated per user is $O(\log(n))$. As long as the set of parties remains the same, this can be reused.

Keyswitching from NGS to MKLWE For the keyswitching from the bootstrapping scheme, NGS, back to the base scheme we adapt the keyswitching schemes from [BIP+22] to obtain an NGS to MKLWE keyswitching protocol. Once again, this will happen in two parts:

1. In Algorithm 3 we define the procedure to generate the keyswitching keys, $\mathsf{KSKGen}_{\mathsf{NGS}}$.
2. In Algorithm 4 we define the keyswitching procedure using the keyswitching keys that were generated.

To generate the keyswitching key $\mathsf{ksk}_{\mathsf{NGS}}$ the protocol requires the input of an NGS key f and a set of LWE keys, $\mathbf{s}_i$.

Algorithm 3: NGS to MKLWE keyswitching key generation, $\mathsf{KSKGen}_{\mathsf{NGS}}$

Input: An NGS key f, A set of LWE keys, $\mathbf{s}_1, \ldots, \mathbf{s}_n$.
Output: A keyswitching key from $\mathsf{ksk}_{\mathsf{NGS}}$

Sample $\mathbf{A}_i \leftarrow \mathbb{Z}_q^{(N \cdot L) \times n}$ for $i \in [k]$. ;
Sample $e \leftarrow \chi_\sigma$;
Compute $\mathbf{f}_0 = (\phi(f)_0, -\phi(f)_{N-1}, \ldots, -\phi(f)_1)$ and $\mathbf{P} = \mathbf{I}_N \otimes \mathfrak{g}_{q,B}$;
Output $\mathsf{ksk}_{\mathsf{NGS}} = \left(\sum_{i \in [k]} \mathbf{A}_i \cdot \mathbf{s}_i + \mathbf{P} \cdot \mathbf{f}_0 + e, \mathbf{A}_1, \ldots, \mathbf{A}_k\right)$;

Note that only one such key needs to be generated regardless of the number of parties involved. If $k = 1$ then the keyswitching key naturally becomes single key, which, given knowledge of f, can be locally generated by each of the users. To execute the keyswitch we then use Algorithm 4.

Algorithm 4: NGS to MKLWE keyswitching, $\mathsf{KS}_{\mathsf{NGS}}$

Input: An NGS Scalar ciphertext ct', Keyswitching key $\mathsf{ksk}_{\mathsf{NGS}}$
Output: An MKLWE ciphertext ct

Parse $\mathsf{ct}' = \frac{g}{f} + \Delta m$ and $\mathsf{ksk}_{\mathsf{NGS}} = (\mathbf{b}, \mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_k)$;
Set $\mathbf{y} = \mathfrak{g}^{-1}(\phi(\mathsf{ct}'))$;
Compute $b = \mathbf{y} \cdot \mathbf{b}, \mathbf{a}_i = \mathbf{y} \cdot \mathbf{A}_i$ for all $i \in [k]$;
Output $\mathsf{ct} = (b, \mathbf{a}_1, \ldots, \mathbf{a}_k)$;

Lemma 2.4 ($\mathsf{KS}_{\mathsf{NGS}}$ correctness & noise [BIP+22]). *Let* ct' *be a scalar* NGS *encryption of* m *with error* e, $\mathsf{ksk}_{\mathsf{NGS}}$ *be as defined in Algorithm 3, and* ct *be the output of* $\mathsf{KS}_{\mathsf{NGS}}$ *with input* ct'. *Then* ct *is a correct* MKLWE *encryption of* m *under the set of keys* $\mathbf{s}_1, \ldots, \mathbf{s}_n$ *with the following error:*

$$\mathsf{Var}(\mathsf{err}(\mathsf{ct})) \leq N \cdot L \cdot \gamma^2 \cdot \mathsf{Var}(\mathsf{err}(\mathsf{ksk}_{\mathsf{NGS}})) + \mathsf{Var}(\mathsf{err}(\mathsf{ct})),$$

where N *is the* NGS *dimension and* $L = \log_B(Q)$ *and* $\mathsf{Var}(\mathsf{err}(\mathsf{ksk}_{\mathsf{NGS}}))$ *is that of a fresh ciphertext.*

Proof. This Lemma is a straightforward generalization of what has already been discussed in Section 5.2 of [BIP+22] which describes single-key NGS to single-key LWE keyswitching and Lemma 4.3 in [PVZ24] which describes multi-key NGS to multi-key LWE.

In Section 5 details of the exact use of the $\mathsf{KS}_{\mathsf{NGS}}$ protocol will be discussed, however it is easily observed that generating the keyswitching key for $k = 1$ or $k > 1$ does not change the noise after protocol execution. Hence, $\mathsf{KS}_{\mathsf{NGS}}$ requires one MKLWE encryption with $\Delta = 1$ and message $\mathbf{P} \cdot \mathbf{f}_0$ to generate the keyswitching key. To complete the keyswitching protocol, the remaining cost is $k + 1$ plaintext-plaintext vector multiplications and one digit decomposition. As there are no homomorphic operations, we can consider the cost of this amortized.

2.7 Modulus Switching

For the concrete instantiation we provide in Section 5, the final ingredient that is required is a modulus switching mechanism as presented in [BIP+22]. For this, recall the definition of the randomized rounding function from [DM15].

Definition 2.3 (Randomized Rounding Function). *Let* $Q, q \in \mathbb{Z}$ *with* $1 < q < Q$. *The randomized rounding function* $[\cdot]_{Q:q} : \mathbb{Z}_Q \to \mathbb{Z}_q$ *is defined as* $[x]_{Q:q} = \lfloor q \cdot x/Q \rfloor + B$ *where* $B \in \{0, 1\}$ *is a Bernoulli random variable with* $\Pr[B = 1] = (q \cdot x/Q) - \lfloor q \cdot x/Q \rfloor$.

This extends element-wise to vectors, matrices and polynomials. Note that the rounding error is ϵ-subgaussian. Modswitching LWE ciphertexts is done by applying the randomized rounding function element wise to the ciphertext. Denote by $\mathsf{ModSwitch}(c, Q, q) = [c]_{Q:q}$.

Lemma 2.5 (Modulus Switching Noise). *For any* LWE *ciphertext* $\mathbf{c}$ *encrypting a message* m *under the key* s *with noise* η *and modulus* q, *then the output of* $\mathsf{ModSwitch}(\mathbf{c}, Q, q)$ *is an* LWE *ciphertext* c' *encrypting the same message under the same key but with modulus* Q *and noise bounded by*

$$\mathsf{Var}(\mathsf{err}(\mathbf{c}')) \leq \frac{q^2}{Q^2}\eta + (n+1)\sqrt{2\pi}.$$

Proof. Let $\mathbf{c} = (b, \mathbf{a})$ and $\mathsf{ModSwitch}(\mathbf{c}, Q, q) = (b', \mathbf{a}')$. For $i \in 1, \ldots, n$, it holds that $a'_i = \frac{q}{Q}a_i + r_i$ and $b' = \frac{q}{Q}b + r_0$ for independent $\sqrt{2\pi}$-subgaussian rounding errors $r_0, r_1, \ldots, r_n$. The result follows in a straightforward manner.

2.8 Multiparty Computation

One of the key features of our framework is the use of a secure multiparty computation (MPC) protocol to relieve, in large part or in full, the computationally complex multi-key FHE protocols in favor of a single-key FHE protocol. For this we will define an MPC functionality as an arithmetic black box. Leading up to the full functionality, we recall some important definitions of MPC.

The Adversary: Semi-honest and Full Threshold In MPC the adversary, $\mathcal{A}$, can be defined on several axes which define the powers of the adversary, [Lin20].

A first axis answers the question as to *how* the adversary behaves. In the case of *semi-honest* corruption, we assume that the adversary cannot instruct corrupted parties to break away from protocol specification. This means that while the adversary gains access to all internal states of the corrupted parties it can only use those states to learn more about the underlying private information of honest parties. If the adversary exerts *malicious* corruption, however, it is allowed to force corrupted parties to behave against protocol description and deviate arbitrarily. For the purposes of this paper we will consider a semi-honest adversary. Due to the nature of FHE protocols and their malleability, a maliciously secure assumption on the MPC protocol would be too strong for what the FHE can later provide.

Another axis to consider is *how many* honest parties can be corrupted. Here we once again have two options: In the *honest majority* case at most $n/2$ parties can be corrupted, where n is the total number of parties. Alternatively, in the *full threshold* case we allow $n-1$ parties to be corrupted at any time. Again, for the purposes of this paper we will consider the full threshold adversary, allowing corruption of all but one honest party.[6]

Secure Multiparty Computation With the behavior of the adversary in place, we can now turn to the MPC functionality that is secure against the semi-honest, full threshold adversary as defined.

The MPC protocol will be defined by four probabilistic polynomial time (PPT) algorithms: $\mathsf{Init}, \mathsf{Input}, \mathsf{Eval}, \mathsf{Output}$. The exact description of which is given in Figure 2, [DPSZ12].

The MPC Functionality as adapted from [DPSZ12]

The functionality runs with parties $\mathcal{P} = P_1, \ldots, P_n$, an evaluator $\mathcal{E}$ and an ideal adversary, $\mathcal{A}$, that corrupts a set of parties, $U \subset \mathcal{P}$. All values are stored as (varID, x) where $\mathsf{varID} \in I$ for some index set of valid identifiers.

- Init: Upon input of $(\mathsf{init}, \mathbb{K})$ from all parties in $\mathcal{P}$, store the field or ring, $\mathbb{K}$.
- Input: Upon input of $(\mathsf{input}, P_i, \mathsf{varID}, x)$ from party P_i and $(\mathsf{input}, P_i, \mathsf{varID})$ from all other parties in $\mathcal{P}$, where varID is a fresh identifier and $x \in \mathbb{K}$ store (varID, x).
- Eval: Upon input of $(\mathsf{eval}, f, \{\mathsf{varID}_{x_i}\}_{i\in[n_i]}, \{\mathsf{varID}_{z_j}\}_{j\in[n_j]})$ from all parties in $\mathcal{P}$, where varID_{x_i} is already stored in memory for all i, and f is an arithmetic function with n_i inputs and n_j outputs, retrieve the corresponding x_i store $(\{\mathsf{varID}_{z_j}\}, f(x_1, \ldots, x_n)_j)$.
- Output: Upon input of $(\mathsf{output}, \mathsf{varID}, i)$ from all parties in $\mathcal{P}$, where varID is already stored in memory, retrieve (varID, x) and output it to the environment. If the environment returns $\mathsf{Deliver}$, then output x to all parties if $i = 0$, to $\mathcal{E}$ if $i = n+1$, or to P_i else. Otherwise, abort.

Fig. 2: *The MPC Functionality, $\mathcal{F}_{\text{MPC}}$*

Any MPC protocol that instantiates the MPC functionality, $\mathcal{F}_{\text{MPC}}$, correctly and obtains security against a semi-honest, full threshold adversary can be used as an instantiation of the following section. For optimization purposes, our key generation might be instantiated with "imperfect" and single-purpose MPC protocols. For ease of exposition, we do not cover these cases explicitly, but in many cases it will be possible to model these with only a resulting change in the noise of the MPC outputs, which can then be carried forward through the rest of our results.

[6] With some care and appropriate approaches to decryption, our protocols should generalize to arbitrary thresholds and even arbitrary access structures.

3 Linear Decryption-based FHE

To introduce our generalized construction, we need a way to capture a multitude of well-established FHE models. To do this, we consider the definition of linear decryption-based FHE (LD-FHE) as defined in [COS+22]. We abstract a lot of the details of the specific constructions away so that our framework can capture a broad variety of different schemes all with one commonality: the schemes all have the same form of decryption equation, namely one based on a linear product combination of ciphertext and secret key. This allows us to capture BGV/BFV [BGV12,FV12], GSW [GSW13], FHEW [MS18,GPvL23], TFHE [CGGI16], and FINAL [BIP+22,PVZ24].

For this, let λ denote the security parameter. We define LD-FHE as follows.

Definition 3.1 ((Public Key) LD-based FHE).

- $\mathsf{KeyGen}\left(1^\lambda, p\right)$*: Upon input of the security parameter* λ *and a plaintext modulus* p*, this randomized algorithm outputs a tuple* params^* *containing (at least) the value* q *for the ciphertext modulus,*[7] *the value* β *as a "noise bound", the value* d *is one less than the dimension of the ciphertext space, the value* S *which is a bound on the secret key size, and the values* sk *and* pk *as the secret and public keys. It also computes* $\Delta = \lfloor q/p \rfloor$ *and adds this to* params^**. All subsequent algorithms are assumed to take* $\mathsf{params} = \mathsf{params}^* \backslash \{\mathsf{sk}, \mathsf{pk}, S\}$ *as input.*
- $\mathsf{Enc}(\mathsf{pk}, m, \mathsf{type})$*: On input of a message,* $m \in R_p$*, and a (public) key* pk*, this algorithm outputs a ciphertext* $\mathsf{ct} \in R_q^{d+1}$ *such that*

$$\mathsf{ct} \cdot (1, -\mathsf{sk}) = \begin{cases} m + p \cdot \varepsilon \pmod q & \text{if } \mathsf{type} = \mathsf{lsb} \\ \Delta \cdot m + \varepsilon \pmod q & \text{if } \mathsf{type} = \mathsf{msb} \end{cases}$$

  A ciphertext such that $\|\varepsilon\|_\infty \leq \beta$ *is called valid. The precise algorithm used for encryption will depend on the specific scheme.*
- $\mathsf{Eval}\left(\mathsf{pk}, F\left(x_1, \ldots, x_l\right), \mathsf{ct}_1, \ldots, \mathsf{ct}_l\right)$*: On input of* l *ciphertexts* $\{\mathsf{ct}_i\}_{i \in [l]}$ *under a public key* pk*, an arithmetic function* F *requiring* l *inputs, and a public key* pk*, this algorithm homomorphically evaluates the function* F *over the ciphertexts and outputs a valid ciphertext of the form*

$$\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, F(m_1, \ldots, m_l), \mathsf{type})$$

- $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct}, \mathsf{type})$*: On input of a valid ciphertext,* ct*, and the secret key,* sk*, this will compute the message as*

$$m = \begin{cases} \left(\mathsf{ct} \cdot (1, -\mathsf{sk}) (\mathsf{mod}\ q)\right) \pmod p & \text{if } \mathsf{type} = \mathsf{lsb} \\ \lfloor \mathsf{ct} \cdot (1, -\mathsf{sk}) (\mathsf{mod}\ q) \cdot p/q \rceil & \text{if } \mathsf{type} = \mathsf{msb} \end{cases}$$

Remark 3.1. Note that the public key and symmetric key definitions of LD-FHE are practically identical. In case the LD-FHE scheme is symmetric, one can instead consider Enc as making use of sk instead of pk. Similarly, in the Eval functionality, one can consider pk to be the key material needed to enable Eval. Therefore, while we will continue to use public key language throughout, everything that follows also work for symmetric key schemes.

[7] Note that in practice there can be many ciphertext moduli depending on the level a ciphertext might sit at. This can be ignored in theory, but will be important in practice.

LD-FHE based Keyswitching, $\Pi_{\mathrm{LD-KS}}$

The functionality takes as input a valid ciphertext ct^* encrypting a message m under the key pk^*, and the key pairs $(\mathsf{pk}^*, \mathsf{sk}^*)$ and $(\mathsf{pk}, \mathsf{sk})$. Additionally, it uses public parameter type.

$\mathsf{KeyGen}^{\mathsf{KS}}(\mathsf{sk}^*, \mathsf{pk})$: Upon input of secret key sk^* and public key pk, this algorithm calls $\mathsf{Enc}(\mathsf{pk}, \mathsf{sk}^*, \mathsf{type})$ and outputs the resulting ciphertext as ksk.

$\mathsf{KS}(\mathsf{pk}, \mathsf{ksk}, \mathsf{ct}^*)$: Upon input of a valid ciphertext, ct^*, encrypting the message m under the key pk^*, this algorithm calls $\mathsf{Eval}\left(\mathsf{pk}, \mathsf{Dec}(x_1, \mathsf{ct}^*, \mathsf{type}), \mathsf{ksk}\right)$ and outputs the resulting ct.

Fig. 3: *The keyswitching algorithm based on the LD-based FHE definition using only the definitions.*

The correctness requirement simply states that given l valid ciphertexts, Eval outputs a valid encryption of the correct result. The security requirement is the standard IND-CPA notion. Note that the private key, $\mathsf{sk} = (s_1, \ldots, s_d)$, is assumed to be a random element of $\mathcal{R}_q^d$ sampled such that $\|\mathsf{sk}\| \leq S$, hence we do not necessarily assume that this key is uniformly distributed. Throughout the remainder of the paper we will only describe the $\mathsf{type} = \mathsf{msb}$ variant, however the lsb-variant follows analogously.

3.1 Deriving Keyswitching from LD-FHE

One of the key components of a bootstrapping algorithm for LD-based FHE schemes is the concept of keyswitching. Due to the nature of the original LD-based FHE definition, the derivation of the keyswitching protocol was not treated, however it follows elegantly from the definition above.

For this, consider a valid ciphertext, ct^*, encrypting a message m under the key pk^*. The goal of the keyswitching algorithm is then to take ct^*, the key sk^* and a different key pk and return the valid ciphertext ct which encrypts the same message m however under the new key pk. This can be accomplished using only the LD-based FHE instruction as described in Figure 3.

Lemma 3.1 (LD-based FHE Keyswitching). *Given a valid ciphertext* ct^* *as described above,* $\mathsf{KS}\left(\mathsf{KeyGen}^{\mathsf{KS}}(\mathsf{sk}^*, \mathsf{pk}), \mathsf{ct}^*\right)$ *computes a valid ciphertext* ct *encrypting the same message* m *under the key* pk.

Proof. Assume ct^* is a valid ciphertext encrypting m under the key pk^*. Given $\mathsf{ksk} = \mathsf{Enc}(\mathsf{pk}^*, \mathsf{pk})$ then

$$\begin{aligned}\mathsf{KS}(\mathsf{ksk}, \mathsf{ct}^*) &= \mathsf{Eval}(\mathsf{pk}, \mathsf{Dec}\left(x_1, \mathsf{ct}^*, \mathsf{type}\right), \mathsf{ksk}) \\ &= \mathsf{Enc}(\mathsf{pk}, \mathsf{Dec}(\mathsf{sk}^*, \mathsf{ct}^*, \mathsf{type})) \\ &= \mathsf{Enc}(\mathsf{pk}, m, \mathsf{type})\end{aligned}$$

As ct^* is a valid ciphertext, sk^* correctly obtains the message m. Furthermore, re-encrypting under pk is correct as m lies in the plaintext space, which obtains the keyswitching as we want.

3.2 Multi-Key Linear Decryption Based FHE

A more recent development within the FHE space has been the concept of multi-key FHE (MK-FHE), in which each party locally generates a unique key pair, $(\mathsf{pk}_i, \mathsf{sk}_i)$, and can encrypt without communicating with the other parties while still allowing the evaluator to compute a function based

on the encrypted input of each of the parties. The existence of such schemes has been shown for BFV [Par21], BGV [Par21], TFHE [CCS19], and FINAL [PVZ24]. Since these schemes all still enjoy the linear decryption equation, these can be unified into a multi-key version of Definition 3.1 which is provided in Definition 3.2.

Definition 3.2 (Multi-Key LD-based FHE).

- $\mathsf{MK.KeyGen}\left(1^\lambda, p, k\right)$*: Upon input of the security parameter* λ*, a plaintext modulus* p*, and the number of parties* k*, this randomized algorithm outputs a tuple* params^* *containing (at least) the value* q *for the ciphertext modulus, the value* β *as a "noise bound", the value* d *is one less than the dimension of the ciphertext space, the value* S *which is a bound on the secret key size, and the public-secret key pairs* $(\mathsf{pk}_i, \mathsf{sk}_i)$ *for* $i \in [k]$*. The case for a symmetric scheme is analogous to single key LD-FHE. It also computes* $\Delta = \lfloor q/p \rfloor$ *and adds this to* params^**. All subsequent algorithms are assumed to take* $\mathsf{params} = \mathsf{params}^* \backslash \{(\mathsf{pk}_i, \mathsf{sk}_i)_{i\in[k]}, S\}$ *as input.*
- $\mathsf{MK.Enc}\left(\{\mathsf{pk}_i\}_{i\in I\subset[k]}, m, \mathsf{type}\right)$*: On input of a message,* $m \in R_p$*, and an ordered set of (public) keys* $\{\mathsf{pk}_i\}_{i\in I\subset[k]}$*, this algorithm outputs a ciphertext* $\mathsf{ct} \in R_q^{d+1}$ *such that*

$$\mathsf{ct} \cdot \left(1, -\overline{\mathsf{sk}}_1, \ldots, -\overline{\mathsf{sk}}_k\right) = \begin{cases} m + p \cdot \varepsilon \pmod q & \text{if } \mathsf{type} = \mathsf{lsb} \\ \Delta \cdot m + \varepsilon \pmod q & \text{if } \mathsf{type} = \mathsf{msb} \end{cases}$$

  where $\overline{\mathsf{sk}}_i = \mathsf{sk}_i$ *if* $i \in I$ *and* 0 *otherwise. A ciphertext such that* $\|\varepsilon\|_\infty \leq \beta$ *is called valid. The precise algorithm used for encryption will depend on the specific scheme.*
- $\mathsf{MK.Eval}\left(\{\mathsf{pk}_i\}_{i\in I\subset[k]}, F\left(x_1, \ldots, x_l\right), \mathsf{ct}_1, \ldots, \mathsf{ct}_l\right)$*: On input of* l *ciphertexts and an arithmetic function* F *requiring* l *inputs, and a set of public keys* $\{\mathsf{pk}_i\}_{i\in I\subset[k]}$*, this algorithm homomorphically evaluates the function* F *over the ciphertexts and outputs a valid ciphertext of the form*

$$\mathsf{ct} = \mathsf{Enc}(\{\mathsf{pk}_i\}_{i\in I\subset[k]}, F(m_1, \ldots, m_l), \mathsf{type})$$

- $\mathsf{MK.Dec}(\{\mathsf{sk}_i\}_{i\in I\subset[k]}, \mathsf{ct}, \mathsf{type})$*: On input of a valid ciphertext,* ct*, encrypting a message,* m*, under public keys,* $\{\mathsf{pk}_i\}_{i\in I}$*, and the set of secret keys,* $\{\mathsf{sk}_i\}_{i\in I\subset[k]}$*, this will compute the message as*

$$m = \begin{cases} \left(\mathsf{ct} \cdot \left(1, -\overline{\mathsf{sk}}_1, \ldots, -\overline{\mathsf{sk}}_k\right) \pmod q\right) \pmod p & \text{if } \mathsf{type} = \mathsf{lsb} \\ \left\lfloor \mathsf{ct} \cdot \left(1, -\overline{\mathsf{sk}}_1, \ldots, -\overline{\mathsf{sk}}_k\right) \pmod q \cdot p/q \right\rceil & \text{if } \mathsf{type} = \mathsf{msb} \end{cases}$$

The correctness requirement is then simple to extend: Given a fixed set $I \subset [k]$, and l valid ciphertexts, $\mathsf{MK.Eval}$ outputs a valid encryption of the correct result. The security requirement remains the standard IND-CPA notion. Similar to before, we will consider only the $\mathsf{type} = \mathsf{msb}$ variant, leaving the lsb variant to the interested reader.

3.3 Bringing Keyswitching to Multi-Key LD-FHE

Similarly to the single key variant, the multi-key variant of LD-FHE also allows for the definition of a keyswitching algorithm, however at this point the question can be extended: Is it possible, should we so wish, to keyswitch from the one paradigm to the other? A simple observation would be that, given $I, J \subset [k]$, if a multi-key keyswitching algorithm would exist, then by simply restricting either I or J to a single key we would obtain an answer to the question.

Lemma 3.2 (Multi-Key LDFHE Keyswitching). *Let $I, J \subseteq [k]$. Given a valid multi-key ciphertext ct^* encrypting a message m under public key set pk_I^*, $\mathsf{MK.KS}(\mathsf{pk}_J, \mathsf{MK.KeyGen}^{\mathsf{KS}}(\mathsf{sk}_I^*, \mathsf{pk}_J), \mathsf{ct}^*)$ computes a valid ciphertext ct encrypting the same message m under the public key set pk_J.*

Proof. Assume ct^* encrypts a message m under the keys pk_I^*. By the definition of $\mathsf{MK.KeyGen}^{\mathsf{KS}}(\mathsf{pk}_J, \mathsf{sk}_I^*)$,

$$\mathsf{ksk} = \{\mathsf{ksk}_i\}_{i \in I} = \{\mathsf{Enc}(\mathsf{pk}_J, \mathsf{sk}_i, \mathsf{type})\}_{i \in I}$$

In accordance with Figure 4 we then observe that

$$\begin{aligned}
\mathsf{MK.KS}(\mathsf{pk}_J, \mathsf{MK.KeyGen}^{\mathsf{KS}}(\mathsf{sk}_I^*, \mathsf{pk}_J), \mathsf{ct}^*) &= \mathsf{MK.KS}(\mathsf{pk}_J, \mathsf{ksk}, \mathsf{ct}^*) \\
&= \mathsf{MK.Eval}(\mathsf{pk}_J, \mathsf{MK.Dec}(x_1, \mathsf{ct}^*, \mathsf{type}), \mathsf{ksk}) \\
&= \mathsf{MK.Enc}(\mathsf{pk}_J, \mathsf{MK.Dec}(\mathsf{sk}_I^*, \mathsf{ct}^*, \mathsf{type})) \\
&= \mathsf{MK.Enc}(\mathsf{pk}_J, m, \mathsf{type})
\end{aligned}$$

obtaining the correctness result required.

However, as evident from Figure 4, a naive implementation of this would require the full use of $\mathsf{MK.Eval}$. Breaking the abstraction for a moment, this would indicate that such a key switching algorithm perhaps needs to make use of a multi-key bootstrapping phase, something that this framework seeks to avoid.

However, one of the key properties of the (MK.)LDFHE schemes is that the decryption operation is linear, hence the multiplicative depth of this operation is 0. Hence, as long as the noise of the resulting ciphertext $\mathsf{KS}(\mathsf{ksk}, \mathsf{ct}^*)$ produces is small enough, then decryption will not fail. This fully depends on the MK.LDFHE based scheme which is implemented. As we describe in section 2.6, and as visible in the above references, common choices of encryption scheme for FHE schemes, including BGV, BFV, TFHE, and FINAL, come with dedicated keyswitching procedures that already avoid this bootstrapping issue. Therefore, for the remainder of this paper we assume that keyswitching does not require a bootstrapping phase in and of itself.

4 A Framework for LWE-based Multiparty FHE Schemes

In this section, we demonstrate the creation of a multiparty FHE protocol from an MPC protocol and an LDFHE protocol with very minimal restrictions on either. As in Section 2.8, we require that the MPC protocol is secure against a semi-honest, and full threshold adversary where the set of corruptible entities comprises the parties, $\mathcal{P}$, and the evaluator, $\mathcal{E}$.

4.1 Generalized Multiparty FHE

In this section we will formulate the multiparty FHE functionality. After, we will show that this produces a secure protocol under specific, but not wildly restrictive, assumptions.

The multiparty FHE functionality contains 6 probabilistic polynomial time algorithms: $\mathsf{Init}, \mathsf{KeyGen}, \mathsf{Dec}, \mathsf{Input}, \mathsf{Eval}$, and Output. Upon input of a set of parties, $\mathcal{P} = P_1, \ldots, P_k$, an evaluator, $\mathcal{E}$, and an ideal adversary $\mathcal{A}$ that corrupts a set of parties, $\mathcal{C} \subset (\mathcal{P} \cup \{\mathcal{E}\})$ where $\mathcal{C} \cap \mathcal{P} \neq \emptyset$, $\mathcal{F}_{\text{mpFHE}}$ can do the following:

multi-key LD-FHE based Keyswitching, $\Pi_{\text{MK.LD-KS}}$

Let $I, J \subset [k]$. Let $\mathsf{pk}_I, \mathsf{sk}_I, \mathsf{pk}_J, \mathsf{sk}_J$ be the public (resp. secret) keys belonging to the set of parties P_I and P_J respectively. Additionally, it uses public parameter type.

$\mathsf{MK.KeyGen}^{\mathsf{KS}}(\mathsf{sk}_I, \mathsf{pk}_J)$: Upon input of the secret key set sk_I and the public key set pk_J, this algorithm computes the following for every $i \in I$:

$$\mathsf{ksk}_i = \mathsf{Enc}(\mathsf{pk}_J, \mathsf{sk}_i, \mathsf{type})$$

and outputs $\mathsf{ksk} = \{\mathsf{ksk}_i\}_{i \in I}$.

$\mathsf{MK.KS}(\mathsf{pk}_J, \mathsf{ksk}, \mathsf{ct}^*)$: Upon input of the set of keyswitching keys, ksk, and a ciphertext, ct^*, encrypting a message m under the public key set pk_i, this algorithm calls $\mathsf{MK.Eval}$ to compute

$$\mathsf{ct} = \mathsf{MK.Eval}\left(\{\mathsf{pk}_j\}_{j \in J}, \mathsf{Dec}(x_1, \mathsf{ct}^*, \mathsf{type}), \mathsf{ksk}\right)$$

and outputs ct.

Fig. 4: *The keyswitching algorithm based on the Multi-Key LD-based FHE definition using only the definitions.*

- Init allows the functionality to initialize any variables that it may need. This includes the instantiation of the LDFHE and MPC protocols, as well as the key generation for the LDFHE protocol.
- KeyGen allows for the generation of the (unknown) communal key. Here the parties and evaluator receive the key switching keys. The evaluator will use these to keyswitch the inputs from the other parties to the unknown communal key, and later will use another such keyswitching key to generate the output ciphertext encrypted under all parties' keys.
- Input formalizes the encryption and communication of an input to the evaluator, $\mathcal{E}$. The Evaluator also keyswitches to the unknown communal key in this step.
- Eval allows the evaluator, $\mathcal{E}$, to evaluate a given function, f, on given inputs, $\{c_i\}_{i \in I}$.
- Output formalizes the communication of an output ciphertext to the parties, $\mathcal{P}$. The evaluator performs keyswitching to multi-key LDFHE under all the parties' individual secret keys here.
- Dec behaves in line with multi-key LDFHE decryption, taking in a ciphertext, $\mathbf{c}$, and a private key, sk, or a set of private keys, $\{\mathsf{sk}_i\}_{i \in I}$, and outputs the underlying plaintext, m.

The detailed functionality $\mathcal{F}_{\text{mpFHE}}$ can be found in Figure 5.

The communal key is generated by all parties and, as such, will use an MPC protocol in the implementation. The generation of this key is identical to the generation of a single-key LDFHE key, but keeping it secret the parties or the adversary. Concurrently, keyswitching keys from each of the private keys of the parties to this (unknown) communal key are generated. As the private keys of the parties never get shared and the communal key is not known to any of the parties, the communal key and the keyswitching keys can be used repeatedly over multiple executions.

4.2 The Framework in Practice

In Figure 6, we provide a protocol implementing our mp-FHE functionality.

Theorem 4.1. *The protocol* Π_{mpFHE} *UC-realizes* $\mathcal{F}_{\text{mpFHE}}$ *in the* $\{\mathcal{F}_{\text{mpc}}, \mathsf{LDFHE}\}$*-Hybrid model.*

Proof (Outline of Proof).

The mp-FHE functionality, $\mathcal{F}_{\text{mpFHE}}$

The functionality runs with parties $\mathcal{P} = P_1, \ldots, P_k$, an evaluator $\mathcal{E}$ and an ideal adversary, $\mathcal{A}$, that corrupts a set of parties, $U \subset \mathcal{P}$. All values are stored as (varID, x) where $\mathsf{varID} \in I$ for some index set of valid identifiers.

$\mathsf{Init}(\mathbb{K}, \lambda, p)$**:** Upon receiving Init from all parties and the evaluator, the functionality initializes the database of encryptions $\mathbb{S}$ and stores the field or ring $\mathbb{K}$, the security parameter λ and the plaintext modulus p. Furthermore, when receiving Init from $\mathcal{P}_i$, the functionality outputs a tuple params^* as defined in $\mathsf{LDFHE.Init}$. The functionality also outputs an FHE key $(\mathsf{sk}_i, \mathsf{pk}_i)$ to $\mathcal{P}_i$.

$\mathsf{KeyGen}(\mathsf{pk}_i, \mathsf{sk}_i)$**:** Upon input of $(\mathsf{KeyGen}, i, \mathsf{sk}_i, \mathsf{pk}_i)$ from each party other than $\mathcal{E}$, the functionality then generates and stores the required communal keys $(\mathsf{pk}_{com}, \mathsf{sk}_{com})$. Then, the functionality stores $(\mathsf{ksk}_i, \mathsf{pk}_{com}, \mathsf{sk}_i) \in \mathbb{S}$ for each party $\mathcal{P}_i$, and sends ksk_i to $\mathcal{P}_i$ and $\mathcal{E}$.

$\mathsf{Input}(m, \mathsf{pk})$**:** Upon input of $\mathsf{Input}(m, \mathsf{pk})$ from party $\mathcal{P}_i$, generate some c, store $(c, \mathsf{pk}, m) \in \mathbb{S}$, return c to $\mathcal{P}_i$, and send c to $\mathcal{E}$. The evaluator then generates c' and stores $(c', \mathsf{pk}_{com}, m) \in \mathbb{S}$, and outputs c' to $\mathcal{E}$.

$\mathsf{Eval}(\mathsf{pk}_{com}, f, \mathbf{c})$**:** Upon input of $\mathsf{Eval}(\mathsf{pk}_{com}, f, \mathbf{c})$ by $\mathcal{E}$ where f is an arithmetic function, $\mathsf{pk}_{\mathsf{com}}$ are the communal public keys, and $\mathbf{c}$ is the set of ciphertexts to be used as inputs, such that $\forall c_i \in \mathbf{c}, (c_i, \mathsf{pk}_{com}, m_i) \in \mathbb{S}$ with $s \in k_{\mathsf{com}}$, and k_{eval} was generated by $\mathsf{EvalKeyGen}(s)$, the functionality stores $\{(c'_i, \mathsf{pk}_{com}, f(m_1, \ldots, m_k)_i)\}_{i \in [n']} \in \mathbb{S}$ and sends $\{c'_i\}_{i \in [n']}$ to $\mathcal{E}$.

$\mathsf{Output}(c)$**:** Upon input of $\mathsf{Output}(c)$ by $\mathcal{E}$, the functionality sends c to all parties.

$\mathsf{Dec}(c)$**:** Upon input of $\mathsf{Dec}(c)$ by each $\mathcal{P}_i$, if there is a stored tuple $(c, \mathsf{sk}_{com}, m)$ for some m, return m to all parties.

Fig. 5: *The Multiparty-FHE Functionality*

We first describe a simulator $\mathcal{S}$. Recall that we denote by $\mathcal{C}$ the set of corrupted parties. Throughout the protocol, the simulator emulates $\mathcal{F}_{\text{mpc}}$ keeping track of any counters and IDs needed. For expositionary reasons, when addressing $\mathcal{F}_{\text{mpc}}$, we denote by x the variable ID referring to x.

Init: Upon a call of Init by a party in $\mathcal{C}$, the simulator computes $\mathsf{LDFHE.KeyGen}(1^\lambda, p)$, and sends the resulting tuple params^* and key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ to the adversary.

KeyGen: Upon input of $\mathsf{KeyGen}(\mathsf{pk}_i, \mathsf{sk}_i)$ by all parties other than $\mathcal{E}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{MPC}}.\mathsf{Eval}(\mathsf{LDFHE.KeyGen}, (1^\lambda, p), (\mathsf{pk}_{com}, \mathsf{sk}_{com}))$ to produce a key pair $(\mathsf{pk}_{com}, \mathsf{sk}_{com})$, which it stores. The simulator $\mathcal{S}$ also emulates $\mathcal{F}_{\text{mpc}}.\mathsf{Eval}(\mathsf{LDFHE.Enc}, (\mathsf{sk}_i, \mathsf{pk}_{com}, \mathsf{type}), ksk_i)$ for each i. If $\mathcal{E} \in \mathcal{C}$, then the simulator will emulate $\mathcal{F}_{\text{MPC}}.\mathsf{Output}(\mathsf{pk}_{com}, \mathcal{E})$ and $\mathcal{F}_{\text{MPC}}.\mathsf{Output}(\mathsf{ksk}_i, \mathcal{E})$ for all i. The simulator reconstructs this value and sends it to the adversary. Whether or not $\mathcal{E} \in \mathcal{C}$, the simulator receives the shares of sk_{com} held by the parties in $\mathcal{C}$ and sends these to the adversary.

Input: We distinguish a number of cases here:

- **Honest $\mathcal{E}$ and dishonest $\mathcal{P}_i$:** The simulator emulates $\mathsf{LDFHE.Enc}(m, \mathsf{pk}_i)$ and sends the resulting encryption to $\mathcal{E}$.
- **Dishonest $\mathcal{E}$ and dishonest $\mathcal{P}_i$:** The simulator computes $c \leftarrow \mathsf{LDFHE.Enc}(m, \mathsf{pk}_i)$, and $c' \leftarrow \mathsf{LDFHE.KS}(c, \mathsf{ksk}_i)$ and sends the resulting encryption c' to the adversary.
- **Dishonest $\mathcal{E}$ and honest $\mathcal{P}_i$:** Upon receiving a ciphertext c from $\mathcal{P}_i$, the simulator computes $\mathsf{LDFHE.KS}(c, \mathsf{ksk}_i)$ and sends the output of the keyswitching to the adversary.

Eval: If $\mathcal{E}$ is honest, the simulator does not need to do anything. If $\mathcal{E}$ is dishonest, then upon the adversary calling Eval, the simulator emulates a call to $\mathcal{F}_{\mathsf{LDFHE}}.\mathsf{Eval}$ using the same inputs, returning the output ciphertext to the adversary.

Output: If $\mathcal{E}$ is honest, the simulator passes c to the adversary. If $\mathcal{E}$ is dishonest, then the simulator send c to all honest parties and returns it to the adversary.

Dec: The simulator emulates $\mathcal{F}_{\text{MPC}}.\mathsf{Output}$ using the adversary's shares to output m, which it then sends to the adversary.

The mp-FHE protocol with multi-key decryption, Π_{mpFHE} in the $\mathsf{LDFHE}, \mathcal{F}_{\text{MPC}}$-Hybrid Model

The protocol runs with parties $\mathcal{P} = \mathcal{P}_1, \ldots, \mathcal{P}_k$ and an evaluator $\mathcal{E}$. All values are stored as (varID, x) where $\mathsf{varID} \in I$ for some index set of valid identifiers.

$\mathsf{Init}(\mathbb{K}, \lambda, p)$**:** The parties call $\mathcal{F}_{\text{MPC}}.\mathsf{Init}(\mathbb{K})$ and $\mathsf{LDFHE.Init}(1^\lambda, p)$. Each party calls $\mathsf{LDFHE.KeyGen}$ and receives $(\mathsf{sk}_i, \mathsf{pk}_i)$.

$\mathsf{KeyGen}(\mathsf{pk}_i, \mathsf{sk}_i)$**:** Each party $\mathcal{P}_i$ runs $\mathcal{F}_{\text{MPC}}.\mathsf{Input}(\mathsf{sk}_i)$. They then have $\mathcal{F}_{\text{MPC}}$ compute $(\mathsf{pk}_{com}, \mathsf{sk}_{com})$ by calling $\mathcal{F}_{\text{MPC}}.\mathsf{Eval}(\mathsf{LDFHE.KeyGen}(1^\lambda, p))$. Next, they have $\mathcal{F}_{\text{MPC}}$ compute $\{\mathsf{ksk}_i\}_{i\in[k]}$ by calling $\mathcal{F}_{\text{MPC}}.\mathsf{Eval}(\mathsf{LDFHE.Enc}(\mathsf{sk}_i, \mathsf{pk}_{com}, \mathsf{type}))$ for all $i \in [k]$. Finally, the parties run $\mathcal{F}_{\text{MPC}}.\mathsf{Output}$ so that $\mathcal{E}$ gets $\{\mathsf{ksk}_i\}_i, \mathsf{pk}_{com}$.

$\mathsf{Input}(m, \mathsf{pk})$**:** The party calls $\mathsf{LDFHE.Enc}(\mathsf{pk}, m, \mathsf{type})$ which outputs c. The party sends c to $\mathcal{E}$. $\mathcal{E}$ then calls $\mathsf{LDFHE.KS}(c, \mathsf{ksk}_i)$ and stores the output c'.

$\mathsf{Eval}(\mathsf{pk}_{com}, f, \mathbf{c})$**:** The evaluator calls $\mathsf{LDFHE.Eval}(\mathsf{pk}_{com}, f, \mathbf{c})$.

$\mathsf{Output}(c)$**:** The evaluator sends c to all parties.

$\mathsf{Dec}(c)$**:** The parties call $\mathcal{F}_{\text{MPC}}$ to execute $\mathcal{F}_{\text{MPC}}.\mathsf{Input}(c)$, $\mathcal{F}_{\text{MPC}}.\mathsf{Eval}(\mathsf{LDFHE.Dec}, (c, \mathsf{sk}_{com}), m)$, followed by $\mathcal{F}_{\text{MPC}}.\mathsf{Output}(m, i)$ for all i.

Fig. 6: *The Multiparty-FHE Protocol*

Indistinguishability argument: We now argue that the output and transcripts of the adversary are identical for the real and ideal execution with a semihonest adversary.

The output is the same in both executions as the key difference lies in whether the call is to the functionality's Eval or $\mathcal{F}_{\text{LDFHE}}$'s Eval. Both return the same value on the same inputs, so we conclude that the output of the two executions will be the same.

As for the transcripts, we make the argument per sub-protocol.

Init**:** For each party $\mathcal{P}_i$ in $\mathcal{C}$, the adversary receives a key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ and the $\mathsf{LDFHE\ params}^*$ in both cases.

KeyGen**:** Here, if $\mathcal{E}$ is dishonest, then the adversary receives the $\{\mathsf{ksk}_i\}_i, pk_{com}$ in both cases.

Input**:** **Honest $\mathcal{E}$ and dishonest $\mathcal{P}_i$:** The adversary receives nothing in both cases.

Dishonest $\mathcal{E}$ and dishonest $\mathcal{P}_i$: The adversary receives c' in both cases, which are identically distributed due to the nature of LDFHE.

Dishonest $\mathcal{E}$ and honest $\mathcal{P}_i$: The adversary receives the output of the keyswitching in both cases.

Eval**:** If $\mathcal{E}$ is dishonest, the adversary receives the output of the evaluation c in both cases. In case a dishonest evaluator calls Eval while inputting ciphertexts encrypted under different keys, they receive nothing both in the real world and the simulation.

Output**:** If $\mathcal{E}$ is dishonest, then the adversary receives the output ciphertext as any other dishonest parties. Else, the simulator passes the received ciphertext to the adversary. In both cases, the adversary's view is identical.

Dec**:** As long as the ciphertext input is under pk_{com}, this will output the same message in both cases. In the case where the ciphertext input is under a different key, then Dec does not output anything. The adversary will receive this value as any corrupted parties.

Beyond our argument that Π_{mpFHE} UC-realizes $\mathcal{F}_{\text{mpFHE}}$, we argue that the handling of the communal key does not cause any problems. The main issue is that we generate the secret key in MPC. This key is then used to perform the entire computation. However, as this value is intermediate within the MPC, the only way to learn it is to either have all parties collude, which is impossible as at most $k-1$ parties can be corrupt, or to extract the key from the keyswitching keys. Since

the keyswitching keys are opened only to the evaluator $\mathcal{E}$, and there are no keyswitching keys from the communal key to any other key, the adversary can only decrypt values input by other corrupt parties.

5 A Case Study: Multiparty FINAL

To exemplify the above generic construction Π_{mpFHE}, we now present an adaptation of FINAL [BIP+22] to the multiparty setting as a specific instantiation.

In this setting, we also show some of the optimization techniques that may become possible when the underlying single-key FHE scheme is no longer a black box. In section 6, we then implement this instantiation and evaluate how it scales with the number of parties.

5.1 Key Generation

We use MPC during preprocessing to compute the keys used during the protocol. Note that while the communal LWE and NGS keys exist, we never output these values from the MPC. Within our protocol, we require the generation of several keys during the MPC:

- For each party $\mathcal{P}_i$ where $i \in [k]$, the keyswitching key from their LWE key $\mathbf{s}_i$ to the communal LWE key $\mathbf{s}$, using the dedicated construction from section 2.6.
- The keyswitching key from the communal NGS key f to the communal LWE key $\mathbf{s}$, from section 2.6.
- The bootstrapping keys bsk_i for $i \in [n]$ which allows for an indirect keyswitch from sk-LWE to NGS.

To compute the bootstrapping keys, we run the algorithm in Algorithm 5 in MPC to generate the keys listed above. We stress that the communal secret keys $\mathbf{s}$ and f are known to no-one, as these are never output from the MPC. Thus, the server, using the keys generated by this process, has the ability to do the following without compromising security:

- keyswitch a ciphertext of the form $\mathsf{LWE.Enc}_{\mathbf{s}_i}(m)$ to $\mathsf{LWE.Enc}_{\mathbf{s}}(m)$, used during the input phase.
- keyswitch a ciphertext of the form $\mathsf{NTRU.Enc}_f(m)$ to $\mathsf{LWE.Enc}_{\mathbf{s}}(m)$, used during the bootstrapping.

The noise of the keyswitching keys is that of a fresh ciphertext, see section 2.4.

Once the parties have generated their shares of the keyswitching keys ksk_i, they then send this keyswitching key to the server along with their input encrypted under their own LWE key $\mathbf{s}_i$. The server then keyswitches to the communal key $\mathbf{s}$ using the keyswitching from single-key LWE to single-key LWE in section 2.6 with keyswitching key ksk_i.

5.2 Bootstrapping

In this Section we introduce the bootstrapping using LWE as a base scheme as alluded to in [BIP+22]. The bootstrapping has the following structure. After some initial format conversions, a blind rotation is performed using CMux gates. This homomorphically decrypts the pre-processed ciphertext, providing an NGS ciphertext of the same message. Then, this NGS ciphertext is keyswitched back to an LWE ciphertext. Finally, the modulus is switched back to the LWE modulus from the NGS modulus. Each step is covered in more detail in what follows.

Algorithm 5: The Bootstrapping Key Generation Algorithm, Π_{KeyGen}

Input: Each party $\mathcal{P}_i$ inputs their LWE key $\mathbf{s}_i$. The Security parameter λ.
Output: The keyswitching keys ksk_i from $\mathbf{s}_i$ to $\mathbf{s}$ for all $i \in [k]$.
The keyswitching key $\mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}}$ from f to $\mathbf{s}$.

$(N, Q, \zeta, B, l) \leftarrow \mathsf{NGS.Setup}(\lambda)$;
$(q, p, \sigma) \leftarrow \mathsf{LWE.Setup}(\lambda)$;
$\mathbf{s} \leftarrow \mathsf{LWE.KeyGen}(\sigma)$. ;
$f \leftarrow \mathsf{NGS.KeyGen}(\zeta, N)$. ;
$\mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}} \leftarrow \mathsf{KSKGen}_{\mathsf{NGS}}(f, s)$;
for $i \in [k]$ **do**
  $\mathsf{ksk}_i \leftarrow \mathsf{KSKGen}_{\mathsf{LWE}}(\mathbf{s}_i, \mathbf{s})$;
for i *in* $[n]$ **do**
  $\mathbf{g}_i \xleftarrow{\$} R_n$;
  $\mathsf{bsk}_i \leftarrow \frac{\mathbf{g}_i}{f} + \mathfrak{g} s_i$;
Output $\mathsf{ksk}_i, \mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}}$;

Algorithm 6: The Bootstrapping Algorithm, Π_{BS}

Input: $\mathsf{ct} = (b, \mathbf{a})$, an LWE ciphertext encrypting m. The output of $\mathsf{KSKGen}_{\mathsf{NGS}\to\mathsf{LWE}}(f, \mathbf{s}) =: \mathsf{ksk}_{\mathsf{NGS}\to\mathsf{LWE}}$.
Output: ct', an LWE ciphertext encrypting the same m.

$b \leftarrow \left\lfloor \frac{2 \cdot N \cdot b}{q} \right\rceil$;
$\mathbf{a} \leftarrow \left(\left\lfloor \frac{2 \cdot N \cdot (-\mathbf{a})}{q} \right\rceil \right)$;
$\mathsf{ACC} \leftarrow \lfloor \frac{Q}{8} \rceil \cdot X^{N/2} \cdot \sum_{i=0}^{N-1} X^i$;
$\mathsf{ACC} \leftarrow \mathsf{ACC} \boxdot \mathfrak{g} \cdot X^b$;
for $i \leftarrow 0$ *to* $n-1$ **do**
  $c_{mux} \leftarrow \mathsf{CMux}_i(\mathbf{a}_i)$;
  $\mathsf{ACC} \leftarrow \mathsf{ACC} \boxdot c_{mux}$;
$\mathsf{ACC} \leftarrow \mathsf{ACC} + \lfloor \frac{Q}{8} \rceil \cdot \sum_{i=0}^{N-1} X^i$;
$\mathsf{ACC} \leftarrow \mathsf{ModSwitch}(\mathsf{ACC}, Q, q)$;
$\mathsf{ct}' \leftarrow \mathsf{KS}_{\mathsf{NGS}\to\mathsf{LWE}}(\mathsf{ACC}, \mathsf{ksk}_{\mathsf{NGS}\to\mathsf{LWE}})$;

As in [BIP+22] and [PVZ24], the first step of the bootstrapping is to convert the input LWE ciphertext to an NGS ciphertext. This is done via a series of binary CMux gates, described in [CGGI20]. The binary CMux is defined as

$$\mathsf{CMux}_i(c) = \mathfrak{g} + (X^c - 1) \cdot \mathsf{NGS.VEnc}_f(s_i).$$

Note that $\mathfrak{g}$ is a noiseless NGS encryption of 1, and recall that $\mathsf{bsk}_i = \mathsf{NGS.VEnc}_f(s_i)$. We note that

$$\mathsf{CMux}_i(c_i) = \mathfrak{g}(1 + X^{c_i} s_i - s_i) + \frac{\mathbf{g}}{f_i}(X^{c_i} - 1) = \mathsf{NGS.VEnc}_f(X^{c_i \cdot s_i}),$$

with noise $\mathbf{g}(X^{c_i} - 1)$. The last equality holds due to the following:

- If $s_i = 0$, then $X^{c_i s_i} = X^0 = 1$ and $1 + X^{c_i} s_i - s_i = 1$.
- If $s_i = 1$, then $X^{c_i s_i} = X^{c_i}$ and $1 + X^{c_i} s_i - s_i = X^{c_i}$.

As $c_i \in \mathbb{F}_2$, $X^{c_i s_i} \in \mathbb{M}$. We give the following Lemma to bound the noise generated by a CMux gate.

Lemma 5.1. *Let* $E_{bsk_i} = \mathsf{Var}(\mathsf{err}(\mathsf{bsk}_i))$ *and let* E_{CMux_i} *be the noise of the output of the* CMux *gate. Then* $E_{\mathsf{CMux}_i} \leq 2 \cdot E_{\mathsf{bsk}_i}$.

Proof. This Lemma is a more specific version of Lemma 7 of [PVZ24].

Given as input the LWE ciphertext $\mathsf{LWE.Enc}_s(m)$ parsed as $(b, \mathbf{a})$, the first step of the bootstrapping is to pre-process this to obtain:

$$c_0 := \left\lfloor \frac{2 \cdot N \cdot b}{q} \right\rceil,$$
$$\mathbf{c}_1 := \left\lfloor \frac{2 \cdot N \cdot (-\mathbf{a})}{q} \right\rceil.$$

We now would like to perform the following multiplication:

$$\mathbf{g} \cdot X^{c_0} \cdot \prod_{i \in [k]} \mathsf{CMux}_i(c_{1,i}) = \mathsf{NGS.VEnc}_f(X^{\Delta \cdot m + e}),$$

where we note that $\mathbf{g} \cdot X^{c_0} = \mathsf{NGS.VEnc}(X^{c_0})$ with noise 0. The difficulty in performing such a multiplication is that there is no way to directly perform multiplications between NGS vector encryptions. Thus, we add a scalar NGS encryption called the accumulator on the left side of the computation to allow for the use of the external product defined in Section 2.5.

The accumulator is defined as

$$\mathsf{ACC} := \left\lfloor \frac{Q}{8} \right\rceil \cdot X^{N/2} \cdot \sum_{i=0}^{N} X^i \mod (X^N + 1).$$

Using this well-known technique [GPvL23,MS18,CGGI20], the following Theorem can be proven:

Theorem 5.1 ([BIP+22] Section 5.1). *Let* $\mathsf{ct} = (b, \mathbf{a})$ *and let* $c_0, \mathbf{c}_1, \mathsf{bsk}_i$, *and* ACC *be defined as above. Then*

$$c' := (\mathsf{ACC} \boxdot c_0) \boxdot_i \mathsf{CMux}_i(c_{1,i}) + \left\lfloor \frac{Q}{8} \right\rceil \cdot \sum_{i=1}^{N-1} X^i = \mathsf{NGS.SEnc}_f(m),$$

with $\Delta = \frac{Q}{4}$, *such that* $\mathsf{Var}(\mathsf{err}(c')) \leq (n+1) \cdot N \cdot l \cdot \gamma^2$.

Proof. First, we prove correctness, and after we will bound the noise generated. Recall that the accumulator is given by the following:

$$\mathsf{ACC} = \left\lfloor \frac{Q}{8} \right\rceil \cdot X^{N/2} \cdot \sum_{i=0}^{N-1} X^i.$$

Then using the external product in the first step obtains

$$\mathsf{ACC} \boxdot \mathfrak{g} \cdot X^b = \left\lfloor \frac{Q}{8} \right\rceil \cdot X^{N/2+b} \cdot \sum_{i=0}^{N-1} X^i.$$

Then applying the CMux once more for each $i \in [n]$ results in

$$\mathsf{ACC} = \left\lfloor \frac{Q}{8} \right\rceil \cdot \sum_{i=0}^{N-1} X^i \cdot X^{\left\lfloor \frac{2N}{q} b \right\rceil} \cdot \prod_{i=1}^{n} X^{\left\lfloor \frac{2N}{q} a_i s_i \right\rceil}.$$

Note that $b - \sum_{i=0}^{n} a_i s_i = \Delta \cdot m + e$. Since after applying a boolean gate, $\Delta = \left\lfloor \frac{q}{2} \right\rceil$, the following holds:

$$\frac{2 \cdot N}{q}(\Delta \cdot m + e) = N \cdot (m + \frac{e}{\Delta}).$$

Due to our assumptions regarding the error bound, i.e. that $\left\lfloor \frac{e}{\Delta} \right\rceil = 0$, we obtain:

$$\mathsf{ACC} = \left\lfloor \frac{Q}{8} \right\rceil \cdot X^{N/2} \cdot \sum_{i=0}^{N-1} X^i \cdot (-1)^m.$$

We are only interested in the value of the 0th coefficient. Observe that the 0th coefficient is at $i = N/2$, obtaining $X^{N/2} \cdot X^{N/2} = X^N = -1$. This results in

$$\mathsf{ACC} = \left\lfloor \frac{Q}{8} \right\rceil - 1 \cdot (-1)^m.$$

Note that this is a valid encryption $\mathsf{NGS.SEnc}(2m-1)$. By adding a noiseless encryption of 1 using $\Delta = \lfloor Q/8 \rceil$, and observing that in the remainder we use $\Delta = \lfloor Q/4 \rceil$, we obtain:

$$\mathsf{ACC} = \mathsf{NGS.SEnc}(m)$$

as desired.

We now bound the noise. As there are $n+1$ total CMux multiplications, we obtain from Lemma 2.2 that the noise of this sequence is bounded by

$$\mathsf{Var}(\mathsf{err}(c')) \leq (n+1) \cdot N \cdot l \cdot \gamma^2 \cdot \mathsf{Var}(\mathsf{err}(\mathsf{CMux}_i(a_i))) + \mathsf{Var}(\mathsf{err}(T(X))).$$

By Lemma 5.1, note that $\mathsf{Var}(\mathsf{err}(\mathsf{CMux}_i(a_i))) \leq 2 \cdot \mathsf{Var}(g)$, thus:

$$\mathsf{Var}(\mathsf{err}(c')) \leq (n+1) \cdot N \cdot l \cdot \gamma^2.$$

Now that we have the results from Theorem 5.1, we can keyswitch from NGS to LWE using the keyswitching protocol from [BIP+22], recalled in Section 2.6. Doing so produces an LWE ciphertext of the same message as was input, however this encryption is under the NGS modulus Q.

Thus, we must still switch back to the LWE modulus q before we once again have a ciphertext of the same form as the input of the bootstrapping. We use the modulus switching from Section 2.7.

Now the bootstrapping is complete, the following holds:

Theorem 5.2. *Let $\mathcal{P}$ be the set of k participating parties. Let $\mathbf{c} := \mathsf{LWE.Enc}_{\mathbf{s}}(m)$. Then Π_{BS} outputs $\mathbf{c}' = \mathsf{LWE.Enc}_{\mathbf{s}}(m)$ with noise bounded by:*

$$\begin{aligned}\mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{BS}}(\mathbf{c}))) &= \mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{KS}}(\Pi_{\mathrm{MS}}(\mathsf{ACC})))) \\ &\leq N \cdot L \cdot \gamma^2 \cdot \sigma^2 + \frac{q^2}{Q^2} \cdot (n+1) \cdot N \cdot l \cdot \gamma^2 + (n+1) \cdot \sqrt{2\pi}.\end{aligned}$$

Proof. By Theorem 5.1, the error after step 6 is given by

$$\mathsf{Var}(\mathsf{err}(\mathsf{ACC})) \leq (n+1) \cdot N \cdot l \cdot \gamma^2.$$

As this is the final step using the NGS scheme, this is also the bound for the noise while working in this scheme.

By Lemma 2.5, it is known that the noise introduced by the modulus switching algorithm is bounded by:

$$\begin{aligned}\mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{MS}}(\mathsf{ACC}))) &\leq \frac{q^2}{Q^2}(\mathsf{Var}(\mathsf{err}(\mathsf{ACC})) + (n+1)\sqrt{2\pi} \\ &\leq \frac{q^2}{Q^2} \cdot (n+1) \cdot N \cdot l \cdot \gamma^2 + (n+1)\sqrt{2\pi}.\end{aligned}$$

Thus, via Lemma 2.4 the Theorem is proven:

$$\begin{aligned}\mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{BS}}(\mathbf{c}))) &= \mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{KS}}(\Pi_{\mathrm{MS}}(\mathsf{ACC})))) \\ &\leq N \cdot L \cdot \gamma^2 \cdot \mathsf{Var}(\mathsf{err}(\mathsf{ksk})) + \mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{MS}}(\mathsf{ACC}))) \\ &\leq N \cdot L \cdot \gamma^2 \cdot \sigma^2 + \frac{q^2}{Q^2} \cdot (n+1) \cdot N \cdot l \cdot \gamma^2 + (n+1) \cdot \sqrt{2\pi}.\end{aligned}$$

5.3 Heuristics

Denote by $E_{BS} = \mathsf{Var}(\mathsf{err}(\Pi_{\mathrm{BS}}(\mathbf{c})))$. This noise, obtained after bootstrapping, behaves as a Gaussian distribution. Thus, we can apply the Central Limit Theorem and its heuristic to conclude that $\|\mathsf{err}(\Pi_{\mathrm{BS}}(\mathbf{c}))\| \leq 6 \cdot \sqrt{E_{BS}}$.

From Theorem 5.2, we can conclude that

$$\|\mathsf{err}(\mathsf{ct}_{\mathsf{NGS}})\| \leq 6 \cdot \sqrt{N \cdot L \cdot \gamma^2 \cdot \sigma^2 + \frac{q^2}{Q^2} \cdot (n+1) \cdot N \cdot l \cdot \gamma^2 + (n+1) \cdot \sqrt{2\pi}}. \tag{1}$$

Theorem 5.3. *If the* NGS *ciphertexts of the protocol in Algorithm 6 satisfy equation 1 except with negligible probability and $Q = \mathcal{O}(N \log(N))$, then these ciphertexts can be decrypted, except with negligible probability.*

Proof. As $n = \mathcal{O}(N), \{l, L\} = \mathcal{O}(\log Q) = \mathcal{O}(\log N)$ and $\{\gamma^2, \sigma^2\} \in \mathcal{O}(1)$, we have that the noise of NGS ciphertexts in the protocol in Algorithm 6 are $\mathcal{O}(N \log(N))$ except with negligible probability by Eq 1. We require this noise to be less than $Q/4$ for correct decryption, thus it is sufficient to select $Q = \mathcal{O}(N \log(N))$.

5.4 Decryption

After bootstrapping, the server holds an LWE ciphertext of the output encrypted under s. If this is not the end of the protocol, the server then continues the computation, bootstrapping as needed. Otherwise, the server sends the output ciphertext back to the parties for them to decrypt using MPC.

The advantage of describing decryption in this way is a conceptual simplicity that follows from leveraging the full power of an MPC protocol that is already supposed to be present for the key generation phase. The downside, however, is that it introduces an additional communication and coordination burden between the parties beyond the initial setup. Therefore, below, we discuss an alternative method that removes the burden of using a fully general MPC protocol by introducing some additional key material.

Alternative Decryption using Multi-Key LWE In the general functionality, and in the instantiation above, we show decryption by running single key decryption within MPC. Here we note another option. During preprocessing, the parties can instead also compute key switching keys from the communal key s_{com} to multi-key LWE under all parties' individual keys $s_1, \ldots, s_k$.

Then during output, the evaluator keyswitches the output ciphertext to multi-key LWE, which the parties can then decrypt using standard multi-key LWE techniques. We now bound the noise during decryption for this method.

Theorem 5.4. *Let* $\mathbf{c}$ *be the output of a freshly bootstrapped ciphertext and let* $\mathbf{c}'$ *be the output of* $\mathsf{KS}_{\mathsf{LWE}\to\mathsf{MKLWE}}(\mathbf{c}, \mathsf{ksk})$*. Then:*

$$\mathsf{Var}(\mathsf{err}(\mathbf{c}')) \leq 2 \cdot N \cdot L \cdot \gamma^2 \cdot \sigma^2 + \frac{q^2}{Q^2} \cdot (n+1) \cdot N \cdot l \cdot \gamma^2 + (n+1) \cdot \sqrt{2\pi}.$$

Furthermore, $\mathbf{c}'$ *decrypts correctly with overwhelming probability if* $q = \mathcal{O}(n^2)$

Proof. The noise bound follows directly from Theorem 5.2 and Lemma 2.3. We use the Central Limit Theorem heuristic to conclude that

$$\|\mathsf{err}(\mathbf{c}')\| \leq 6 \cdot \sqrt{\mathsf{Var}(\mathsf{err}(c'))}.$$

As $N = \mathcal{O}(n)$, $\{l, L\} = \mathcal{O}(\log n)$, $\left\{\frac{q^2}{Q^2}, \gamma, \sigma\right\} = \mathcal{O}(1)$,

$$\|\mathsf{err}(c')\| = \mathcal{O}(n \log n).$$

The correct decryption follows as $\mathcal{O}(n \log n) < \mathcal{O}(n^2)$.

A further improvement can be made to this by breaking the black box of the decryption. In this improvement, instead of generating a keyswitching key from sk-LWE to mk-LWE, one instead has the parties generate a keyswitching key from sk-NGS to mk-LWE, allowing to replace the keyswitch in the final bootstrapping with a keyswitch to mk-LWE instead. This saves the effort of an extra key switch at the end of the computation.

5.5 Key Compression

We observe that in practice, storing the keyswitching keys comes at a noticeable cost in memory usage. Storing a single $\mathsf{ksk}_{\mathsf{LWE}\to\mathsf{LWE}}$ incurs a total of $n \cdot \ell \cdot B \cdot (n+1) \cdot \log q$ bits of storage, with $\mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}}$, $\mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{MKLWE}}$ and $\mathsf{ksk}_{\mathsf{LWE}\to\mathsf{MKLWE}}$ following similar patterns. Following ideas from [CDKS21,CNT12], we can compress the constituent LWE ciphertexts by sampling a single random seed of length λ that can be expanded into the (public, uniformly random) $\mathbf{a}$ component of the ciphertext. This in turn leads to only $n \cdot \ell \cdot B \cdot \log q + \lambda$ bits of storage needed for a $\mathsf{ksk}_{\mathsf{LWE}\to\mathsf{LWE}}$. In MPC, the key generation is still easily achieved by establishing and opening the random seed, after which the parties can locally expand the seed and consider the resulting values as public information. The same random seed can be used for all different required keyswitching keys, as long as distinct randomness is extracted from it for each key.

Similarly, when parties need to send many inputs to the server, it may be beneficial to apply the same technique to the LWE ciphertexts encrypted under s_i, reducing the cost of sending t inputs from $t \cdot (n+1) \cdot \log q$ to only $t \cdot \log q + \lambda$ bits. In either of these cases, the impact on the

computational cost would be minor in comparison to the rest of the scheme. Where needed for performance, the expansion can still be performed a single time up-front and stored in memory, keeping the improvement of this compression confined to the amount of data communicated over the network.

6 Implementation

In order to show the practicality and correctness of our construction, we implement the multiparty version of FINAL as described in Section 5. Note that this is a prototype implementation and does not realize the full capabilities of FINAL (e.g. our bootstrapping takes 88.6ms while the original implementation manages to do this in 48ms). Using similar adaptations any alternative implementation can be modified to provide multiparty functionality.

As our transformation works with any appropriate MPC preprocessing for key generation, our code implements the ideal functionality directly as a single program, and does not try to emulate network communication or any specific MPC protocols. Further on, we do analyze the number of multiplications required and the multiplicative depth of a potential instantiation of the key generation circuit. This allows for a good first order approximation of the preprocessing cost, based on a chosen MPC protocol.

Our benchmarks use the original parameter from [BIP+22] with minor adjustments to the noise parameters in the LWE scheme, i.e. $\sigma = 3.5$ instead of $\sigma = 4.39$. This change does not affect the security level and, according to [APS15], still provides a security level of $> 2^{110}$ as guaranteed by FINAL. Moreover, this guarantees that the decryption error of MPFHE FINAL is $< 2^{-75}$ in the NGS scheme and 2^{-40} in the LWE scheme. Note that these changes do not obstruct the improvements to FINAL as suggested in [JMPP24].

Due to the application of the framework, the only limitation that the number of parties can impose on the scalability of the scheme is the number and size of the keyswitching keys. As can be seen in Table 2, we note that the storage of the key material is not restrictive requiring at most 22.7 GB when the protocol is executed by 512 parties. While this is not part of our implementation, we recall that the memory footprint of the keyswitching keys could be significantly reduced by generating random seeds in the key generation algorithm that can be expanded into full LWE samples. Additionally, the work and time required for the multiparty key generation naturally scales linearly with the number of parties.

Protocol	Key Size in GB								
Number of parties	2	4	8	16	32	64	128	256	512
PvLZ [PVZ24]	.32	4.50	119.00	5337.00	-	-	-	-	-
PR [PR23, Table 3][8]	.42	.63	1.10	2.20	-	-	-	-	-
KMS [KMS24, Table 3]	.22	.29	.25	.29	.32	-	-	-	-
AKÖ [AKÖ23, Table 3]	.08	.24	.66	.93	2.00	4.10	9.10	37.00	80.00
This work	.13	.22	.39	.75	1.45	2.87	5.70	11.40	22.70

Table 2: *A comparison of total key material in GB*

[8] Counts only the size of bsk and not ksk.

Concretely, our implementation generates the communal LWE and NGS keys $\mathbf{s}_{com}$ and f_{com} (but does not output these), and the associated evaluation keys for the regular FINAL protocol. Additionally, it generates k input keys $\mathbf{s}_i$, one per party, along with the keyswitching keys $\mathsf{ksk}_{\mathbf{s}_i \to \mathbf{s}_{com}}$ and a (NGS to LWE) keyswitching key $\mathsf{ksk}_{f_{com} \to \mathbf{s}_{1\ldots k}}$ for the output, which will both be part of the evaluation keys for the server. Evidently, each party receives their personal key $\mathbf{s}_i$. Inputs are provided by the parties as LWE encryptions under their personal keys,[9] which the server can transform into encryptions under $\mathbf{s}_{com}$ through $\mathsf{ksk}_{\mathbf{s}_i \to \mathbf{s}_{com}}$ followed by a single bootstrap operation for noise management. From there, any choice of logic gates can be executed by following the single-key scheme. Output towards the parties is performed by executing a final bootstrap operation on the relevant bits, where the final keyswitching step makes use of $\mathsf{ksk}_{f_{com} \to \mathbf{s}_{1\ldots k}}$ instead of the usual $\mathsf{ksk}_{f_{com} \to \mathbf{s}_{com}}$. We take this approach, rather than performing a keyswitch from $\mathbf{s}_{com}$ so $\mathbf{s}_{1\ldots k}$, to limit the final output noise and prevent decryption failures. Depending on the specific application,[10] this need not be a separate bootstrap operation, but could this be integrated into the bootstrap associated with the operation the computed the output bit. To decrypt these final MKLWE ciphertexts, we simply assume all parties are present and can decrypt in a round-robin fashion.

6.1 Performance

The benchmarks were performed single-threaded, on a laptop with 64 GB of RAM and an Intel Core Ultra 7 155H CPU. As we do not aim for our implementation to be specifically optimized for running time, we focus in our presentation of these benchmarks on the effect of scaling to higher numbers of parties, and as such do not aim for minute control on timing fluctuations and the potential impact of other running processes on the machine. In particular, we aim to show that key generation, as well as output and decryption operations scale linearly with the number of parties, while everything else performs at a constant cost similar to what we observe in our single-key FINAL figures.

Protocol	Bootstrap time in ms								
Number of parties	2	4	8	16	32	64	128	256	512
PvLZ [PVZ24]	No Implementation Data Available								
PR [PR23, Table 2]	200	330	460	900	-	-	-	-	-
KMS [KMS24, Table 3]	240	880	2230	5650	13940	-	-	-	-
AKÖ [AKÖ23, Table 3]	190	560	1200	1800	4300	8600	18000	-	-
This work	88.6 (Ours) / 48 (Original, [BIP$^+$22])[11]								

Table 3: *A comparison of time required to perform a bootstrapping operation*

In Table 4, we first summarize a few running times of our implementation of the single-key FINAL scheme. Then, we present our measurements of multiparty key generation, the final keyswitching output, and multi-key decryption for 2, 4, 8, 16 and 32 parties in section 6.1. The implementation is not limited to the number of parties being a power of two, and can scale to arbitrarily many parties

[9] Observe that here too, network communication could be reduced (in the case of many inputs) by compressing the ciphertexts into a single random seed with one extra field element per input.

[10] For instance, when output bits cannot be used as inputs to later Boolean gate.

[11] The original implementation of FINAL in [BIP$^+$22] was faster for the bootstrapping, but incompatible with our expansion to multiple parties. As such we have a less optimized implementation which can be found at `https://github.com/KULeuven-COSIC/vinyl`. We list both bootstrapping runtimes here.

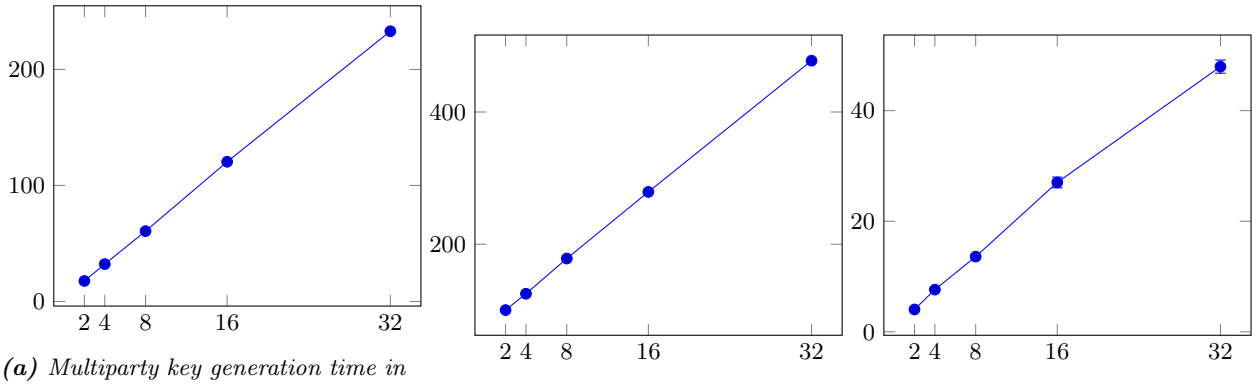
without any increase in the parameters, but these choices should give a clear overview of the scaling behavior for our scheme.

LWE.Enc	LWE.Dec	Gate evaluation (Bootstrap)	Key gen	$\mathsf{KS}_{\mathsf{LWE}\to\mathsf{LWE}}$
178 μs	1.39 μs	88.65 ms	3.43 s	2.71 ms

Table 4: *An overview of our running times for basic operations in the single-key FINAL and LWE schemes.*

As we expect, we observe that input and gate evaluation timings are in line with a single-key bootstrap, with some minor overhead to perform the keyswitch or addition of the two ciphertexts respectively. For key generation, output and decryption, we observe a linear scaling in the number of parties, matching the expected extra costs of needing MKLWE with one key per party and the generation of keyswitching keys per party.

In Table 2 and Table 3, we compare the size of our required key material and the time necessary to perform a single bootstrap to prior work. One of the compared works, [PVZ24], is a multi-key scheme, so the comparison is naturally favorable towards our scheme, but the other schemes we compare to are for the multiparty setting.[12] We also again remark that our implementation is less performant than the original implementation of the FINAL scheme, but that this is purely an artifact of the implementation itself and replacing our inner single-key FHE implementation with the more performant version would be possible without any further changes to the key size, parameter choices, or security analysis.



(a) *Multiparty key generation time in s*

(b) *Multiparty output time in ms*

(c) *Multi-key decryption time in μs*

Fig. 7: *Experimental verification that the running time (y-axis) for multiparty key generation, keyswitching from* NGS *to* MKLWE*, and* MKLWE *decryption grows linearly in the number of parties (x-axis).*

[12] Notably, for [AKÖ23], we compare against the *static* scheme.

6.2 Key generation in MPC

A wide variety of different MPC protocols exist that may be useful in a variety of situations, with support for different types of access structures and many tradeoffs that depend on where the protocol is applied or deployed. As our framework is agnostic to the specific flavor of MPC being employed, and we want to enable a wide variety of potential choices, including e.g. choosing between an honest majority assumption or a full-threshold collusion assumption, we do not provide an implementation of a full MPC protocol for our key generation. Instead, we provide an estimation of the required communication and round cost of such a protocol, in the linear secret sharing-based MPC paradigm. We identify four different components of the key generation that can be analyzed independently. By inspection, each of the components can be seen to require only a constant amount of rounds, and hence so does the overall key generation. If so desired, the components can be interleaved to even further reduce the total number of required rounds. Communication overhead that grows quadratically in the number of parties can be avoided, where needed, by applying the king paradigm from [DN07].

For the base MPC protocol, we assume the following functionalities and building blocks to be present, and we do not make any assumptions on the access structure underlying the secret sharing. These functionalities are commonly found in existing protocol, and can frequently be built from more fundamental building blocks. Where relevant, we suggest such constructions. Observe that additionally, when dealing with an honest majority or $\mathcal{Q}_2$ access structure, some of the operations may be optimized further, for instance by letting the result of a multiplication be opened directly during the execution of the multiplication.

Open A share can be opened $x \leftarrow \mathsf{Open}([x])$ in a constant number of rounds and with a constant number of field elements communicated per party.

Multiplication As a basic operation, the multiplication of two shares $[a \cdot b] \leftarrow \mathsf{Mult}([a], [b])$ can be performed in a constant number of rounds and with a constant number of field elements per party.

Pseudo-random secret sharing Parties can establish, without communication, a sharing of a random, unknown value, uniformly over the field: $[x] \leftarrow \mathsf{PRSS}()$. This may, for instance, be achieved with techniques from [CDI05]. Observe that for an additive secret sharing scheme, it suffices for every party to independently sample their own share.

Pseudo-random bit sharing Parties can establish a sharing of a random bit (0 or 1 in the field) at the cost of a squaring and an opening: $[b] \leftarrow \mathsf{PRBS}()$. One way to do this stems from [DKL$^+$13]: obtain $[r] \leftarrow \mathsf{PRSS}()$, then reveal $r^2 \leftarrow \mathsf{Open}(\mathsf{Mult}([r], [r]))$. Now the random bit sharing can be computed as $[b] = \frac{1+(\sqrt{r^2})^{-1} \cdot [r]}{2}$.

Multiplicative inverse It is possible to find the multiplicative inverse of a shared value at the cost of a multiplication and an opening. One way to achieve this is to obtain a mask $a \leftarrow \mathsf{PRSS}()$, then reveal $\alpha \leftarrow \mathsf{Open}(\mathsf{Mult}([a], [x]))$ and obtain $[x^{-1}] = \alpha^{-1}[a]$. For our purposes, any inverse of zero that shows up in this procedure can be ignored, as they will lead to retrying the computation with fresh randomness.

From these primitives, we can build up the required four components of the key generation. For reasons that will become clear below, we change the LWE modulus $q = 133121 = 2^{11} \cdot 5 \cdot 13 + 1$, which does not notably change any correctness, security or performance characteristics of FINAL. Additionally, we also will rely on LWE noise following the centered binomial distribution

$e = \sum_{i=1}^{38} b_i - b'_i$ with b_i and b'_i uniformly random bits. This can be confirmed to have almost the same standard deviation as the discrete Gaussian distribution used by FINAL for the noise growth analysis, and to not lose any security as computed by the lattice estimator [APS15].

Communal LWE key s_{com} The communal LWE key s_{com} consists of n random bits, so the parties can invoke n parallel instances of PRBS.

Bootstrapping key bsk The bootstrapping key consists of encryptions of s_{com} under an NTRU key. Generating the NTRU key polynomial f can be done in $2 \cdot N$ invocations of PRBS, as the distribution of the coefficients can be achieved by computing $\mathsf{PRBS}() - \mathsf{PRBS}()$. To compute the polynomial f^{-1}, we can make use of our change in modulus, which now allows for the easy computation of an NTT of degree N. The NTT computation is linear, so this step can happen entirely locally. In the NTT representation, we can then simply invert f component-wise. If any NTT component is zero, or the mask used for inversion is zero, a new candidate f is sampled. All bsk_i can now be computed as $\frac{\mathbf{g}}{f} + \mathfrak{g}s_i$, requiring $2 \cdot \ell \cdot N \cdot n$ total random bits to sample $\mathbf{g}$ similar to f and $N \cdot n$ multiplications (in the NTT domain).

Output keyswitching key $\mathsf{ksk}_{\mathsf{NGS}\to\mathsf{MKLWE}}$ The final keyswitch of the bootstrapping, and from our optimization, the final keyswitch to the output MKLWE key encrypts the coefficients of f respectively under the communal LWE key s, or jointly under all individual keys. We write $\hat{\mathbf{f}} = (f_0, -f_{N-1}, -f_{N-2}, \ldots, -f_1)$, and $(\mathsf{ksk}_{\mathsf{NGS}\to\mathsf{MKLWE}})_j = \sum_i A_{j,i} \cdot s_i + \mathbf{e}_j + \mathfrak{g}\hat{\mathbf{f}}_j$.
All matrices A_i can be publicly computed from shared PRNG seeds, and hence have no communication cost. The matrix-vector multiplication and multiplication by the gadget vector are linear, and can be computed locally. This only leaves $\mathbf{e}_j$ to be sampled from the centered binomial distribution, at a total cost of $2 \cdot 38 \cdot \ell \cdot N$ calls to $\mathsf{PRBS}()$.

Input keyswitching keys $\mathsf{ksk}_{\mathsf{LWE}\to\mathsf{LWE}}$ For the input keyswitching keys, transforming ciphertexts from the individual key s_i of a party to the communal key s, we need to compute $(\mathsf{ksk}_{\mathsf{LWE}\to\mathsf{LWE}})_j = \langle \mathbf{a_j}, s \rangle + \Delta(s_i)_j + e_j$. Since the privacy of only s_i is at stake, the e_j may be directly secret shared by the owner of s_i, for a total cost of n sharings per party.

In total, this results in an MPC protocol for the key generation that runs in a constant number of rounds, and requires a total of $((2 \cdot 2 \cdot 38 + 2 \cdot n) \cdot \ell + 2) \cdot N + n = \mathcal{O}(n \cdot \ell \cdot N)$ random bit generations, $N \cdot n$ multiplications, N inversions, and $2 \cdot n$ input sharings per party. For the parameters of our protocol, this amounts to fewer than ten million multiplications and openings.

7 Conclusion

In this work, we have introduced a UC framework for building a multiparty fully homomorphic encryption schemes from MPC and LDFHE. We provided a security argument in the semihonest setting, and we showed for which kinds of MPC and FHE schemes our framework functions with no assembly required.

Of this new framework, we provide an instantiation using the FINAL [BIP+22] scheme. This scheme was implemented and shown to have asymptotically less key material than the state of the art [AKÖ23], allowing for our scheme to support a theoretically unlimited number of players.

Finally, we provided an open source implementation of this instantiation, improving the state of the art in multi-user FHE.

7.1 Future Work

This work has many possibilities for extension. Further study of the various constructions of MP-FHE could lead to the discovery of more efficient schemes. In particular, studying which MPC protocols lead to the most optimized key generation could reduce the computation and communication required from the participants. Likewise, replacing the LDFHE scheme with a Linear Decryption Threshhold FHE scheme could allow for a subset of parties to decrypt instead of requiring the complete set.

The particular instantiation we introduce might be improved by applying approximate gadget decomposition [JMPP24].

Acknowledgements

This work was supported by the Flemish Government through the Cyber Security Research Program with grant number VOEWICS02 and VR20192203. This work has been supported in part by the FWO under an Odysseus project GOH9718N. This work was supported by Research Council KU Leuven C3 on Accelerating towards Deployment: NTRU-Based Scalable Fully Homomorphic Encryption with contract number C3/25/058. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Cyber Security Research Flanders or the FWO.

References

AJJM20. Prabhanjan Ananth, Abhishek Jain, Zhengzhong Jin, and Giulio Malavolta. Multi-key fully-homomorphic encryption in the plain model. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 28–57, Durham, NC, USA, November 16–19, 2020. Springer, Cham, Switzerland.

AKÖ23. Yavuz Akin, Jakub Klemsa, and Melek Önen. A practical TFHE-based multi-key homomorphic encryption with linear complexity and low noise growth. In Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis, editors, *ESORICS 2023: 28th European Symposium on Research in Computer Security, Part I*, volume 14344 of *Lecture Notes in Computer Science*, pages 3–23, The Hague, The Netherlands, September 25–29, 2023. Springer, Cham, Switzerland.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

BCG+17. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, and Michele Orrù. Homomorphic secret sharing: Optimizations and applications. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 2105–2122, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

BDST22. Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion – a framework for mixed-protocol multi-party computation. *ACM Trans. Priv. Secur.*, 25(2), March 2022.

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

BIP+22. Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. FINAL: Faster FHE instantiated with NTRU and LWE. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 188–215, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland.

BLO16. Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 578–590, Vienna, Austria, October 24–28, 2016. ACM Press.

CCS19. Hao Chen, Ilaria Chillotti, and Yongsoo Song. Multi-key homomorphic encryption from TFHE. In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology – ASIACRYPT 2019, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 446–472, Kobe, Japan, December 8–12, 2019. Springer, Cham, Switzerland.

CDI05. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362, Cambridge, MA, USA, February 10–12, 2005. Springer Berlin Heidelberg, Germany.

CDKS19. Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 395–412, London, UK, November 11–15, 2019. ACM Press.

CDKS21. Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 2021: 19th International Conference on Applied Cryptography and Network Security, Part I*, volume 12726 of *Lecture Notes in Computer Science*, pages 460–479, Kamakura, Japan, June 21–24, 2021. Springer, Cham, Switzerland.

CGGI16. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany.

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

CL25. Yijia Chang and Songze Li. Arbitrary-threshold fully homomorphic encryption with lower complexity. In Lujo Bauer and Giancarlo Pellegrino, editors, *USENIX Security 2025: 34th USENIX Security Symposium*, pages 8403–8422, Seattle, WA, USA, August 13–15, 2025. USENIX Association.

CNT12. Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464, Cambridge, UK, April 15–19, 2012. Springer Berlin Heidelberg, Germany.

COS+22. Ilaria Chillotti, Emmanuela Orsini, Peter Scholl, Nigel P. Smart, and Barry van Leeuwen. Scooby: Improved multi-party homomorphic secret sharing based on FHE. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22: 13th International Conference on Security in Communication Networks*, volume 13409 of *Lecture Notes in Computer Science*, pages 540–563, Amalfi, Italy, September 12–14, 2022. Springer, Cham, Switzerland.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer Berlin Heidelberg, Germany.

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

DN07. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 572–590, Santa Barbara, CA, USA, August 19–23, 2007. Springer Berlin Heidelberg, Germany.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

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

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

GPvL23. Antonio Guimarães, Hilder V. L. Pereira, and Barry van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part VI*, volume 14443 of *Lecture Notes in Computer Science*, pages 3–35, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore.

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

JMPP24. Robin Jadoul, Axel Mertens, Jeongeun Park, and Hilder V. L. Pereira. NTRU-based FHE for larger key and message space. In Tianqing Zhu and Yannan Li, editors, *ACISP 24: 29th Australasian Conference on Information Security and Privacy, Part I*, volume 14895 of *Lecture Notes in Computer Science*, pages 141–160, Sydney, NSW, Australia, July 15–17, 2024. Springer, Singapore, Singapore.

KMS24. Hyesun Kwak, Seonhong Min, and Yongsoo Song. Towards practical multi-key TFHE: Parallelizable, key-compatible, quasi-linear complexity. In Qiang Tang and Vanessa Teague, editors, *PKC 2024: 27th International Conference on Theory and Practice of Public Key Cryptography, Part IV*, volume 14604 of *Lecture Notes in Computer Science*, pages 354–385, Sydney, NSW, Australia, April 15–17, 2024. Springer, Cham, Switzerland.

Lin20. Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020.

LTV12. Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *44th Annual ACM Symposium on Theory of Computing*, pages 1219–1234, New York, NY, USA, May 19–22, 2012. ACM Press.

MS18. Daniele Micciancio and Jessica Sorrell. Ring packing and amortized FHEW bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *ICALP 2018: 45th International Colloquium on Automata, Languages and Programming*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 100:1–100:14, Prague, Czech Republic, July 9–13, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

MTBH21. Christian Mouchet, Juan Ramón Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(4):291–311, October 2021.

MW16. Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 735–763, Vienna, Austria, May 8–12, 2016. Springer Berlin Heidelberg, Germany.

OT25. Hiroki Okada and Tsuyoshi Takagi. Low communication threshold FHE from standard (module-)LWE. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025, Part VII*, volume 16251 of *Lecture Notes in Computer Science*, pages 165–198, Melbourne, VIC, Australia, December 8–12, 2025. Springer, Singapore, Singapore.

Par21. Jeongeun Park. Homomorphic encryption for multiple users with less communications. *IEEE Access*, 9:135915–135926, 2021.

PR23. Jeongeun Park and Sergi Rovira. Efficient TFHE bootstrapping in the multiparty setting. *IEEE Access*, 11:118625–118638, 2023.

PS24. Alain Passelègue and Damien Stehlé. Low communication threshold fully homomorphic encryption. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part I*, volume 15484 of *Lecture Notes in Computer Science*, pages 297–329, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

PSSY21. Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 2165–2182. USENIX Association, August 11–13, 2021.

PVZ24. Jeongeun Park, Barry Van Leeuwen, and Oliver Zajonc. FINALLY: A multi-key FHE scheme based on NTRU and LWE. *IACR Communications in Cryptology (CiC)*, 1(3):15, 2024.