Multiparty FHE Redefined: A Framework for Unlimited Participants

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

[1] 3MI Labs, Leuven, Belgium.
robin.jadoul@3milabs.tech
[2] COSIC, ESAT, KU Leuven, Leuven, Belgium.
{barry.vanleeuwen, oliver.zajonc}@kuleuven.be

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. Generally, MPFHE protocols have required ad-hoc constructions and adaptations to already existing protocols. In this work we present a new framework that standardizes the approach of MPFHE to allow the use of a broad spectrum of MPC and FHE protocols, while eliminating the noise inflation based on the participating number of parties. This presents the first ever multiparty FHE protocol which allows an arbitrary number of participants. We then show a case study of this using the FINAL scheme and show that we reduce the required key material by 40-99.9% compared to the MKFHE FINAL scheme, FINALLY, 8-71% compared to the AKÖ scheme, and 65-70% compared to the Park-Rovira scheme. Moreover, we reduce the bootstrapping time for the AKÖ, Park-Rovira, and KMS schemes by 75-99.7%.

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

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

Table of Contents

1 Introduction

Computing on Encrypted Data (COED) is an application for a number of different forms of privacy enhancing technologies (PET). Within this field there are two main families of work: multiparty computation (MPC) and fully homomorphic encryption (FHE). While both MPC and FHE allow a set of parties to communally compute a function on their various secret inputs, the way that they achieve this is starkly different.

In most cases, MPC distributes the secrets in a safe manner (such as secret sharing), allowing each of the parties to perform computation on their own machine before communicating the results with the other parties in the protocol. These systems allow obtaining high security guarantees at the cost of high communication and computation complexity. A drawback as a result of this is that each of the parties require a computationally powerful machine.

Contrasting, FHE allows for the delegation of these computationally expensive operations to a third party server without revealing each party's input or output, reducing the communication and computation complexity of each of the participants at the cost of much lower security guarantees. Multi-key FHE takes this a step further by allowing for these computations to happen in a situation where each participating party has their own (unique) key.

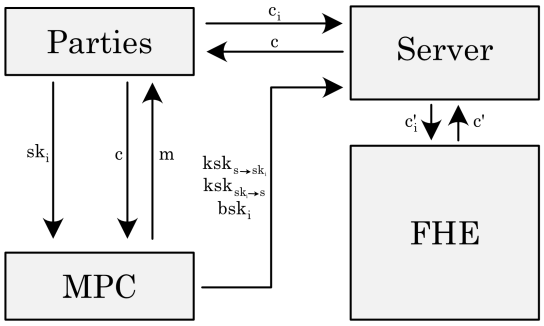
Most FHE schemes achieve the combination of functionality and security by introducing noise, which prevents an adversary from learning anything about the message in a ciphertext. To remove the noise, the decryption step usually has a rounding function which ensures the correct retrieval of the underlying message dependent on the amount of noise present in the ciphertext. With every operation the noise grows: linearly for addition, but exponentially for multiplication. 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.

A possible solution to this is multiparty FHE (MPFHE), which allows for a short burst of communication using an MPC protocol so that the parties can agree on communal keys with which to then outsource the computation to an evaluator while preserving the secrecy of their inputs both to one another and to the evaluator. Thus, the parties, at the cost of some communication, reduce the computational overhead of performing the FHE to that of the single key variant while still allowing multiple parties to participate. While these constructions have been studied in the past, they were often based on exotic assumptions or challenging to verify claims. In this paper we present a framework which allows us to implement these constructions efficiently agnostic of the underlying MPC an FHE protocols.

First, the MPC protocol is used to generate all keys required for the FHE protocol to follow. Depending on the adopted FHE protocol the required key material can change, but in general we need two sets of key material: key switching keys and bootstrapping keys. Secondly, the parties input using ciphertexts under their own individual keys, which the server can then key switch to a common key for the computation itself. Within the computation, the server can bootstrap using the bootstrapping keys, and parties can also supply additional inputs without further communication among the parties themselves, should that be needed. When outputting, the server can simply use

the key switching keys for output, so the output ciphertext will be under the required key. Finally, the parties can now decrypt the output.

Fig. 1: *A bird's eye overview of the MPFHE framework*



Decryption poses some challenges of its own, so we provide some possible approaches that each come with their own set of protocols and tradeoffs.

The first option, single-key decryption, is advantageous in that the parties need not communicate to decrypt. However, there are some drawbacks. The first is that the server must perform as many key switches as there are parties, which can be costly. The second is similar to the first, and that is that as many key switching keys as there are parties must be computed during the MPC preprocessing. The third is that the server must not collude with any party in this scenario, as else the server may key switch any intermediate value, or even another party's input to another key. The (colluding) owner of that key could then learn any value of the computation they wish.

The second option, multi-key decryption, has the advantages of security. The server may be considered semi-honest without breaking security. Furthermore, only a single key switching needs to be computed during the server and only a single key switching key needs to be computed during the MPC preprocessing. The downside is that the parties must communicate for both the input and output of the protocol.

Because this framework allows for the use of a single key FHE protocol over the multi-key variants, this directly combats the incredible noise growth and limitations that must be set on the number of participating parties.

1.1 Our Contributions

A Framework, Multiparty FHE: We introduce a new framework for multiparty fully homomorphic encryption (MPFHE) which allows for the use of any single key FHE protocol featuring linear decryption (e.g. BGV/BFV [BGV12,FV12], TFHE [CGGI20], FHEW [MS18,GPvL23],FINAL [BIP$^+$22,JMPP24,PvLZ24]) in combination with a (semi-honest) MPC protocol to build a multiparty FHE scheme: namely one that requires some communication during key generation, but after which the server can perform the computation with the same complexity as the single key FHE scheme.

An Instantiation, Multiparty FINAL: We instantiate our new framework by adapting FINAL [BIP$^+$22] to the new framework and provide an improvement compared to FINALLY [PvLZ24]

with a 2.46-71.16x reduction in total key material. We also provide an analysis of the noise growth and prove that this is still within the bounds of the lattice attacks known to affect NTRU.

An Implementation, Improving the State-of-the-Art: We provide a prototype implementation of MP-FINAL, which we use to benchmark our results against other MPFHE protocols, which are considered the current state of the art, [AKÖ23,PR23,KMS24].[3]

	PvLZ [PvLZ24]	PR [PR23]	KMS [KMS24]	AKÖ [AKÖ23]	This work
Key size (GB)	5337	2.2	.285	.93	.75
Bootstrapping 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 the literature before. Some existing concepts can be seen as precursors, however. MPFHE can be interpreted as a relaxation of the multi-key FHE setting (MKFHE)[LTV12,PvLZ24,CDKS19,Par21], where parties are now allowed communication during the setup 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. Under a similar constraint, we find threshold FHE (ThFHE) protocols[AKÖ23,AJL$^+$12,KLSW21,MBH22], where a threshold MPC protocol attempts to emulate a single FHE client and enable encryption and decryption when a threshold majority of the parties is honest. Lastly, a line of work building constant-round MPC protocol from FHE attempts to perform a constant-round, multiparty key generation phase, encryption and decryption inside an MPC protocol, with the main computation performed locally through FHE by all parties[BGG$^+$18,MTBH21].

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 $mM^{(i)}$ 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 $M^{(i)}_j = M_{(j),i} = M_{i,j}$.

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 $\|g\| = \|\phi(g)\|$.

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

2.2 Distributions

For a random variable a we define $a \leftarrow G_\sigma$ as the action of sampling a from the discrete Gaussian distribution over a ring R with variance σ^2. Moreover, by $\mathsf{Var}(a)$ we denote the maximum variance of a.

A random variable, V, is called α-subgaussian if the moment generating function fulfills the following equation for some α and all $t \in \mathbb{R}$:

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

We will use the fact $\mathsf{Var}(V) \leq \alpha^2$ and the following property for a vector $\mathbf{a} := (a_1, a_2, \ldots, a_n)$ for our noise analysis:

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

Furthermore, we employ the benefit of subgaussian variables called the Pythagorean Additivity property, which means that given an α-subgaussian distribution χ_α, a β-subgaussian distribution χ_β, $a \leftarrow \chi_\alpha$, $b \leftarrow \chi_\beta$, and any $t, s \in \mathbb{Z}$, it holds that $\mathsf{Var}(t \cdot a + s \cdot b) = \sqrt{t^2 \cdot \alpha^2 + s^2 \cdot \beta^2}$. Lastly, for subgaussian random variables centered around zero, $a, b \in R[X]$, it holds that $\mathsf{Var}(a \cdot b) \leq N \cdot \mathsf{Var}(\phi(a)) \cdot \mathsf{Var}(\phi(b))$.

A specific distribution relevant to our use of NTRU will be the FINAL distribution, which is $\frac{1}{2}$-subgaussian:

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

2.3 Gadget Decomposition

For fixed integers Q, B, let $\ell = \lceil \log_B(Q) \rceil$. 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.

The gadget decomposition associated with a particular gadget vector $\mathfrak{g}$ is a function, denoted by $\mathfrak{g}^{-1}$, such that for $x \in \mathbb{Z}_Q$, $\mathfrak{g}^{-1}(x) = (x_0, \ldots, x_{\ell-1})$ with $|x_i| < B/2$. For the notation as inverse to make sense, we require that $\mathfrak{g}^{-1}(x) \cdot \mathfrak{g} = x$ holds. The gadget decomposition naturally extends elementwise to vectors and polynomials.

2.4 Single-Key NGS

Within the bootstrapping, we will need single-key NGS, introduced in [BIP+22]. We introduce this scheme, as well as properties of the noise generation.

Definition 2.1 (Single-Key NGS). *Let λ be the security parameter. The NGS scheme consists of four probabilistic polynomial time algorithms: Setup, KeyGen, SEnc, VEnc.*

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

2. $\mathsf{NGS.KeyGen}(\zeta, N)$: *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* $\mathsf{sk} = f$.
3. $\mathsf{NGS.SEnc}_{sk}(m)$: *Given a ternary polynomial* m, *sample* $g \leftarrow \chi$, *define* $\Delta = \lfloor Q/4 \rceil$, *and output* $c = g/f + \Delta m \in \mathcal{R}_Q$.
4. $\mathsf{NGS.VEnc}_{sk}(m)$: *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 $\mathfrak{g}^{-1}(c) \cdot \mathbf{g} + g \cdot m'$. If we introduce the notation γ^2 for the variance of the noise of a fresh encryption, we can furthermore quantify the noise growth when performing a sequence of external products.

Lemma 2.1 (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

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

2.5 Multi-key LWE

We recall here the multi-key FHE scheme based on the LWE-problem. We note that this simplifies to single-key LWE by selecting $k = 1$.

Definition 2.2 (Multi-Key LWE). *Let* λ *be the security parameter. Then, the Multi-Key* LWE *scheme consists of the following 5 PPT algorithms:* $\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}$, *and* NAND.

- $\mathsf{MKLWE.Setup}(\lambda)$: *Upon input of a security parameter* λ *returns the public parameters* (q, p, σ). *Denote by* $\Delta := q/4$.
- $\mathsf{MKLWE.KeyGen}(\sigma)$: *Upon input of the public parameters* q, p, *and* σ, $\mathsf{MKLWE.KeyGen}$ *randomly samples* $s_i \in \mathbb{F}_2$, $i \in \{0, \ldots, n-1\}$ *such that* $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{F}_2^n$.
- $\mathsf{MKLWE.Enc}_{\mathbf{s}}(m)$: *Upon input of a key* $\mathbf{s}$ *and a message* $m \in \mathbb{F}_2$, $\mathsf{MKLWE.Enc}$ *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* i*th 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 $(b, \mathbf{a}_1, \ldots, \mathbf{a}_k)$* 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*

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

2.6 Key Switching

We briefly recall two forms of key switching: from (single-key) LWE and NGS to MKLWE, where we note that for both the instantiation of MKLWE with a single key constitutes a key switch to single-key LWE. We use the former in the input phase and the latter during the bootstrapping phase of section 4.

Key Switching from LWE to MKLWE Based on [CDKS21], we describe a key switching method from LWE to MKLWE. The key switching key generation is described in Figure 2, and the actual key switching then takes place in Figure 3. The noise generated by applying the key switching procedure can be quantified. Note that we generate a key switching key for each possible value of each $a_{i,j}$.

Lemma 2.2 ($\mathsf{KS}_{\mathsf{LWE}\to\mathsf{MKLWE}}$ noise). *Let c be an* LWE *encryption of m and let c' be the output of $\mathsf{KS}_{\mathsf{LWE}\to\mathsf{MKLWE}}$ with input c. Then, the following holds:*

$$\mathsf{Var}(\mathsf{err}(c')) \leq n \cdot \ell \cdot \mathsf{Var}(\mathsf{err}(\mathsf{ksk})) + \mathsf{Var}(\mathsf{err}(c)),$$

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

Fig. 2: LWE *to* MKLWE *key switching key generation*

$\mathsf{KSKGen}_{\mathsf{LWE}}$
Input: LWE keys $\mathbf{s}, \mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_k$. Public parameters (n, q, B). Output: Key switching keys $\mathsf{ksk}_{i,j,v} := \mathsf{MKLWE.Enc}_{s_1,s_2,\ldots,s_k}(vs_iB^j)$ for $i \in [n], j \in [\ell], v \in [B]$ with $\Delta = 1$

Key Switching from NGS to MKLWE For the second kind of key switching, we slightly modify the procedure from [BIP+22]. We define the key switching key generation procedure $\mathsf{KSKGen}_{\mathsf{NGS}}$ with the key switching procedure in Figure 4, and again quantify the noise growth of the operation. The same idea as before applies where we keep MKLWE encryptions of the NGS key with a gadget

Fig. 3: LWE *to* MKLWE *key switching*

$\mathsf{KS}_{\mathsf{LWE}\to\mathsf{MKLWE}}$

Input: LWE ciphertext under key $\mathbf{s}$: $\mathbf{c} = (b, \mathbf{a})$. Output of $\{\mathsf{ksk}_{i,j,v}\}_{i,j,v} := \mathsf{KSKGen}_{\mathsf{LWE}}(s, s')$.
Output: $(b', \mathbf{a}') \leftarrow (b, \mathbf{0}) - \sum_{i,j} \mathsf{ksk}_{i,j,\mathfrak{g}^{-1}(\mathbf{a})_{i,j}}$

decomposition, and then rely on the linearity of decryption (for the 0th coefficient of the ciphertext). To generate the key switching key $\mathsf{ksk}_{\mathsf{NGS}\to\mathsf{MKLWE}}$, from an NGS key f to an LWE key $\mathbf{s}$, sample the matrices $\mathbf{A}_i \leftarrow \mathbb{Z}_q^{(N\cdot L)\times n}$, $e \leftarrow \chi_\sigma$, define $\mathbf{f}_0 = (\phi(f)_0, -\phi(f)_{N-1}, -\phi(f)_{N-2}, \ldots, -\phi(f)_1)$ and $\mathbf{P} = \mathbf{I}_N \otimes \mathfrak{g}_{q,B_{\mathsf{ksk}}}$. Then the output becomes $\mathsf{ksk}_{\mathsf{NGS}\to\mathsf{MKLWE}} = (\mathbf{b} := \sum_{i\in[k]} \mathbf{A}_i \cdot \mathbf{s} + e + \mathbf{P}\cdot\mathbf{f}_0, \mathbf{A}_1, \ldots, \mathbf{A}_k)$.

Lemma 2.3 ($\mathsf{KS}_{\mathsf{NGS}\to\mathsf{MKLWE}}$ noise [BIP+22]). *Let* $\mathbf{c}$ *be a scalar* NGS *encryption of* m *with error* e *and* $\mathbf{c}'$ *be the output of* $\Pi_{KS_{\mathsf{NGS}\to\mathsf{MKLWE}}}$ *with input* $\mathbf{c}$*. Then*

$$\mathsf{Var}(\mathsf{err}(\mathbf{c}')) \leq N \cdot L \cdot \gamma^2 \cdot \mathsf{Var}(\mathsf{err}(\mathsf{ksk})) + \mathsf{Var}(\mathsf{err}(\mathbf{c})),$$

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

Fig. 4: NGS *to* MKLWE *key switching*

$\mathsf{KS}_{\mathsf{NGS}\to\mathsf{MKLWE}}$

Input: $\mathbf{c}, ksk_{\mathsf{NGS}\to\mathsf{MKLWE}}$

1. Parse $(\mathbf{b}, \mathbf{A}) \leftarrow \mathsf{ksk}_{\mathsf{NGS}\to\mathsf{MKLWE}}$
2. $\mathbf{y} \leftarrow \mathfrak{g}^{-1}(\phi(\mathbf{c}))$
3. for i in [k]:
   - $\mathbf{a}_i \leftarrow \mathbf{y} \cdot \mathbf{A}_i$
4. $b \leftarrow \mathbf{y} \cdot \mathbf{b}$
5. Output $\mathbf{c}' := (b, \mathbf{a}_1, \ldots, \mathbf{a}_k)$

2.7 Modulus Switching

We implement the method of modulus switching LWE ciphertexts as in [BIP+22]. 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 elementwise to the ciphertext. Denote by $\mathsf{ModSwitch}(c, Q, q) = [c]_{Q:q}$.

Lemma 2.4 (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 can not 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.[4]

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 five probabilistic polynomial time (PPT) algorithms: $\mathsf{Init}, \mathsf{Input}, \mathsf{Add}, \mathsf{Multiply}, \mathsf{Output}$. The exact description of which is given in Figure 5, [DPSZ12].

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

Fig. 5: *The MPC Functionality, $\mathcal{F}_{\text{MPC}}$*

The MPC Functionality as adapted from [DPSZ12]

The functionality runs with parties $\mathcal{P} = P_1, \ldots, P_n$ 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, 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, where varID is a fresh identifier and $x \in \mathbb{K}$ store (varID, x).
- Add: Upon input of $(\mathsf{add}, \mathsf{varID}_x, \mathsf{varID}_y, \mathsf{varID}_z)$ from all parties, where $\mathsf{varID}_x, \mathsf{varID}_y$ are already stored in memory, retrieve the corresponding x and y and store $(\mathsf{varID}_z, x + y)$.
- Multiply: Upon input of $(\mathsf{mult}, \mathsf{varID}_x, \mathsf{varID}_y, \mathsf{varID}_z)$ from all parties, where $\mathsf{varID}_x, \mathsf{varID}_y$ are already stored in memory, retrieve the corresponding x and y and store $(\mathsf{varID}_z, x \cdot y)$.
- Output: Upon input of $(\mathsf{output}, \mathsf{varID}, i)$ from all parties, where varID is already stored in memory, retrieve (varID, x) and output it to the environment. If the environment returns Deliver, then output x to all parties if $i = 0$ or to P_i if $i \neq 0$. Otherwise, abort.

Any MPC protocol that instantiates the MPC functionality, $\mathcal{F}_{\text{MPC}}$, correctly and obtains security against an 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.

3 A Framework for LWE-based Multiparty FHE Schemes

In this section, we demonstrate the creation a multiparty FHE protocol from an MPC protocol and an sk-FHE 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 party set comprises of the party set, $\mathcal{P}$, and evaluator, $\mathcal{E}$. We will now introduce the functionalities of single key FHE, the multiparty functionality, and finally the multiparty protocol.

3.1 Single-Key FHE: Security versus Efficiency

One of the main benefits of the multiparty framework that we present is the ability to preprocess a large amount of the key generation using an MPC protocol. This allows us to generate a common key without any of the parties involved ever having access to this common key. However, this causes an issue to arise when we arrive at the decryption phase of the protocol; We can not key switch to the common key after the bootstrap as this is unknown to the parties involved. Hence, we must amend the standard single-key FHE functionality to include a key switching functionality to key switch from the unknown common key to the private keys of each of the parties. The functionality for this can be found in Figure 6.

Any single-key FHE protocol extended with multi-key key switching and decryption which implements the functionality securely will provide security against semi-honest corruption of each of the parties and/or the server.

Fig. 6: *The single-key FHE Functionality with Multi-key Decryption*

The sk-FHE functionality with multi-key decryption, $\mathcal{F}_{\text{skFHE+}}$

The functionality runs with k parties $\mathcal{P}_1, \ldots, \mathcal{P}_k$ and a semi-honest evaluator $\mathcal{E}$.

Init(): Upon receiving Init from the party and the evaluator, the functionality initializes the database of single-key encryptions $\mathbb{S}$ and the database of key switching keys KSK.

KeyGen: Upon input of KeyGen without further inputs from $\mathcal{P}_i$, the functionality outputs an FHE key s to $\mathcal{P}$.

EvalKeyGen(s_i): Upon input of EvalKeyGen(s) by $\mathcal{P}_i$, the functionality generates k_{eval} and sends it to $\mathcal{E}$.

KSKGen(s, s'): Upon input of KSKGen(s, s'), the functionality stores the tuple (s, s', ksk) in KSK.

KS(c, ksk): Upon input of KS(c, ksk) where $(c, s, m) \in \mathbb{S}$ and $(s, s', \mathsf{ksk}) \in \mathsf{KSK}$, the functionality stores $(c', s', m) \in \mathbb{S}$ and outputs c'.

Enc(s, m): Upon input of Enc(s, m) from $\mathcal{P}$, send $\mathsf{Enc}_s(m)$ to $\mathcal{P}$.

Eval(f, k_{eval}, inputs): Upon input of Eval(f, k_{eval}, inputs) by $\mathcal{P}$, where inputs is a set of encryptions of inputs each under the same key s to the function f and k_{eval} was generated by EvalKeyGen(s), the functionality returns $f(m_1, \ldots, m_n)$ to the parties.

Dec(c, s): Upon input of Dec(c, s), if there is a stored tuple (s, m, c) for some m, return m. Else, return $\perp$.

MKKSKGen($s, \{s'_i\}_i$): Upon input of MKKSKGen($s, \{s'_i\}_i$), the functionality stores the tuple $(s, \{s'_i\}_i, \mathsf{ksk})$ in KSK.

MKKS(c, ksk): Upon input of MKKS(c, ksk) where $(c, s, m) \in \mathbb{S}$ and $(s, \{s'_i\}_i, \mathsf{ksk}) \in \mathsf{KSK}$, the functionality stores $(c', \{s'_i\}_i, m) \in \mathbb{S}$ and outputs c'.

MKDec($c, \{s_i\}_i$): Upon input of MKDec($c, \{s_i\}_i$) by a subset of parties $\mathcal{P} \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$ where each call has the same input ciphertext c but potentially different sets of keys $(\{s_i\}_i)_j$. If $(c, \{s_l\}_l, m) \in \mathbb{S}$ and $\{s_l\}_l = \Cup_j(\{s_i\}_i)_j$, return m to all parties that called MKDec with c as an input.

One of the improvements that one could provide is to simplify the protocol by replacing the multi-key key switching by a parallel execution of single-key key switching from the unknown common key to the private key of each of the parties. This would significantly reduce the noise level and computational load on the server. However, this introduces a problem; At any point an adversary which corrupts a single party and the server can choose to key switch any private input or intermediate values to the private key of the corrupted party and decrypt to obtain the underlying plaintext.

In Figure 7, you can find the standard single-key FHE functionality supporting this version of the protocol, however due to the attack described above we *must* assume that the server can not collude with the parties.

3.2 Generalized Multiparty FHE

In this section we will formulate the multiparty FHE functionality to accommodate both the multi-key and single-key decryption situations discussed above. We will present each of them, focusing first on those parts that overlap and then following each of the diverging paths. Finally, we will show that this produces a secure protocol under specific, but not wildly restrictive, assumptions.

The multiparty FHE functionality contains 10 PPT algorithms: $\mathsf{Init}, \mathsf{KeyGen}_{\mathsf{MPC}}, \mathsf{KeyGen}_{\mathsf{FHE}}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Input}, \mathsf{KS}_{\mathsf{toComm}}, \mathsf{KS}_{\mathsf{fromComm}}, \mathsf{Eval}$, and Output. Upon input of a set of parties, $\mathcal{P} = P_1, \ldots, P_n$, an evaluator, $\mathcal{E}$, and an ideal adversary $\mathcal{A}$ that corrupts a set of parties, $U \subset \mathcal{P}$, $\mathcal{F}_{\text{mpFHE+}}$ can do the following:

Fig. 7: *The single-key FHE Functionality*

The sk-FHE functionality, $\mathcal{F}_{\text{skFHE}}$

The functionality runs with a single honest party $\mathcal{P}$ and a semi-honest evaluator $\mathcal{E}$.

$\mathsf{Init}()$**:** Upon receiving Init from the party and the evaluator, the functionality initializes the database of encryptions $\mathbb{S}$ and the database of key switching keys KSK.
KeyGen**:** Upon input of KeyGen without further inputs from $\mathcal{P}$, the functionality outputs an FHE key s to $\mathcal{P}$.
$\mathsf{EvalKeyGen}(s)$**:** Upon input of $\mathsf{EvalKeyGen}(s)$ by $\mathcal{P}$, the functionality generates k_{eval} and sends it to $\mathcal{E}$.
$\mathsf{KSKGen}(s, s')$**:** Upon input of $\mathsf{KSKGen}(s, s')$, the functionality stores the tuple (s, s', ksk) in KSK.
$\mathsf{KS}(c, \mathsf{ksk})$**:** Upon input of $\mathsf{KS}(c, \mathsf{ksk})$ where $(c, s, m) \in \mathbb{S}$ and $(s, s', \mathsf{ksk}) \in \mathsf{KSK}$, the functionality stores $(c', s', m) \in \mathbb{S}$ and outputs c'.
$\mathsf{Enc}(s, m)$**:** Upon input of $\mathsf{Enc}(s, m)$ from $\mathcal{P}$, send $\mathsf{Enc}_s(m)$ to $\mathcal{P}$.
$\mathsf{Eval}(f, k_{\mathsf{eval}}, \mathsf{inputs})$**:** Upon input of $\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$ by $\mathcal{E}$, where $C := \{c_i\}_i$ is a set of encryptions such that $(s, c_i, m_i) \in \mathbb{S})$, m_i are inputs to the function f and k_{eval} was generated by $\mathsf{EvalKeyGen}(s)$, the functionality stores $(s, c', f(m_1, \ldots, m_n)$ and returns c' to $\mathcal{E}$.
$\mathsf{Dec}(c, s)$**:** Upon input of $\mathsf{Dec}(c, s)$, if there is a stored tuple (s, m, c) for some m, return m. Else, return $\perp$.

- Init allows the functionality to initialize any variables that it may need. This includes, if necessary, the instantiation of the FHE and MPC protocols.
- $\mathsf{KeyGen}_{\mathsf{FHE}}$ allows a party, P_i, to generate the private FHE key, $\mathbf{s}_i$, for their own use.
- $\mathsf{KeyGen}_{\mathsf{MPC}}$ allows for the generation of the (unknown) communal key. As this is the main branch of the diverging paths we will discuss these in detail later.
- Enc behaves in line with FHE Encryption, taking in a message, m, and a private key, sk, and outputting a ciphertext, $\mathbf{c}$.
- Dec behaves in line with FHE 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.
- Input formalizes the communication of a ciphertext to the evaluator, $\mathcal{E}$.
- $\mathsf{KS}_{\mathsf{toComm}}$ is a key switching algorithm allowing the server to switch from a base key to the (unknown) communal key.
- $\mathsf{KS}_{\mathsf{fromComm}}$ is a key switching algorithm allowing the server to switch from the (unknown) communal key to a key or set of keys.
- 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 to the parties, $\mathcal{P}$.

The detailed functionality with multi-key decryption, $\mathcal{F}_{\text{mpFHE+}}$, can be found in Figure 8 while the detailed functionality without multi-key decryption, $\mathcal{F}_{\text{mpFHE}}$, can be found in Figure 9.

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 FHE key, however remains secret from each of the parties, along with key switching keys from each of the private keys of the parties to this (unknown) communal key. Additionally, it also generates key switching keys from the communal key to some combination of the secret keys of the parties. However, this depends on the version of the functionality that we seek to realize:

In the case of $\mathcal{F}_{\text{skFHE+}}$, decryption is done communally using MPC. This requires the existence of a multi-key encryption scheme which can be key switched to, and as such the key switching is

Fig. 8: *The Multiparty-FHE Functionality with Multi-key Decryption*

The mp-FHE functionality with multi-key decryption, $\mathcal{F}_{\text{mpFHE+}}$

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.

$\mathsf{Init}(\mathbb{K})$**:** Upon receiving Init from all parties and the evaluator, the functionality initializes the database of encryptions $\mathbb{S}$ and stores the field or ring, $\mathbb{K}$.

$\mathsf{KeyGen}_{\mathsf{FHE}}$: Upon input of $\mathsf{KeyGen}_{\mathsf{FHE}}$ without further inputs from $\mathcal{P}_i$, the functionality outputs an FHE key $\mathbf{s}_i$ to $\mathcal{P}_i$.

$\mathsf{KeyGen}_{\mathsf{MPC}}$**:** Upon input of $(\mathsf{KeyGen}_{\mathsf{MPC}}, i, \mathbf{s}_i)$ from each party, the functionality then generates and stores the required communal keys k_{com}. Furthermore, the functionality generates and outputs the input, output, and evaluation keys $k_{\mathsf{input}}, k_{\mathsf{eval}}, k_{\mathsf{output}}$ to $\mathcal{E}$.

$\mathsf{Enc}(s, m)$**:** Upon input of $\mathsf{Enc}(s, m)$ by $\mathcal{P}_i$, generate some c, store $(s, m, c) \in \mathbb{S}$, return c to $\mathcal{P}_i$.

$\mathsf{Input}(c)$**:** Upon input of $\mathsf{Input}(c)$ from party $\mathcal{P}_i$, send c to $\mathcal{E}$.

$\mathsf{KS}_{\mathsf{toComm}}(c)$**:** Upon input of $\mathsf{KS}_{\mathsf{toComm}}(c)$ by $\mathcal{E}$, the functionality checks whether $(s, c, m) \in \mathbb{S}$ with $s \notin k_{\mathsf{com}}$. If so, the functionality stores $(c', s', m) \in \mathbb{S}$, where $s' \in k_{\mathsf{com}}$ and outputs c' to $\mathcal{E}$.

$\mathsf{KS}_{\mathsf{fromComm}}(c)$**:** Upon input of $\mathsf{KS}_{\mathsf{fromComm}}(c)$ by $\mathcal{E}$, the functionality checks whether $(s, c, m) \in \mathbb{S}$ with $s \in k_{\mathsf{com}}$. If so, the functionality stores $(\{s_i\}_{i\in[k]}, c', m) \in \mathbb{S}$, outputs c' to $\mathcal{E}$.

$\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$**:** Upon input of $\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$ by $\mathcal{E}$ where f is some function, k_{eval} are the evaluation keys, and C is the set of ciphertexts to be used as inputs, such that $\forall c_i \in C, (s, c_i, m_i) \in \mathbb{S}$ with $s \in k_{\mathsf{com}}$, and k_{eval} was generated by $\mathsf{EvalKeyGen}(s)$, the functionality stores $(c', s, f(m_1, \ldots, m_n)) \in \mathbb{S}$ and sends c' to $\mathcal{E}$.

$\mathsf{Output}(c, i)$**:** Upon input of $\mathsf{Output}(c, i)$ by $\mathcal{E}$, the functionality outputs c to $\mathcal{P}_i$.

$\mathsf{Dec}(c, \mathbf{s}_i)$**:** Upon input of $\mathsf{Dec}(c, \mathbf{s}_i)$ by each $\mathcal{P}_i$, if there is a stored tuple $(\{\mathbf{s}_i\}_{i\in[k]}, c, m)$ for some m, return m to all parties.

done to this multi-key scheme. $\mathsf{KeyGen}_{\mathsf{FHE}}$ then also only includes the generation of this single key switching key instead of one per party.

In the case of $\mathcal{F}_{\text{skFHE}}$, decryption is done by key switching the output k times, once to each party's secret key, which parties can then decrypt individually. Communal key generation includes the generation of key switching keys from the communal secret key to each of the individual secret keys of the parties.

3.3 The Framework in Practice

In Figure 11, we provide a protocol implementing our mp-FHE functionality. We argue now that this construction faithfully implements the functionality and that the security of this construction only relies on the security of the underlying MPC and FHE protocols. We will make this argument by first arguing that $\Pi_{\text{mpFHE+}}$ implements $\mathcal{F}_{\text{mpFHE+}}$.

Consider $\Pi_{\text{mpFHE+}}$ and note that $\mathsf{Init}, \mathsf{KeyGen}_{\mathsf{FHE}}, \mathsf{Enc}, \mathsf{KS}_{\mathsf{toComm}}$, and $\mathsf{KS}_{\mathsf{fromComm}}$ are trivially implemented as they are drawn directly from $\mathcal{F}_{\text{MPC}}$ and $\mathcal{F}_{\text{skFHE+}}$. Moreover, Input and Output are just sending a single message, hence also trivially fulfilled. This means that the only two algorithms that remain are Dec and $\mathsf{KeyGen}_{\mathsf{MPC}}$.

The Dec Algorithm Upon input of (Dec, c, s_i) by a subset of parties, $A \in \mathcal{P}$, the parties instantiate $\mathcal{F}_{\text{MPC}}$ and run MKDec as described in $\mathcal{F}_{\text{skFHE+}}$ in MPC. As $\mathcal{F}_{\text{MPC}}$ receives a ciphertext, c_i, and a set of keys, $\{\mathbf{s}_i\}_{P_i\in A}$ from each party it can first check if $c_i = c_j$ for all i, j. If this is not the

Fig. 9: *The Multiparty-FHE Functionality*

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

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.

$\mathsf{Init}(\mathbb{K})$**:** Upon receiving Init from all parties and the evaluator, the functionality initializes the database of encryptions $\mathbb{S}$ and stores the field or ring, $\mathbb{K}$.

$\mathsf{KeyGen}_{\mathsf{FHE}}$: Upon input of $\mathsf{KeyGen}_{\mathsf{FHE}}$ without further inputs from $\mathcal{P}_i$, the functionality outputs an FHE key s_i to $\mathcal{P}_i$.

$\mathsf{KeyGen}_{\mathsf{MPC}}$**:** Upon input of $(\mathsf{KeyGen}_{\mathsf{MPC}}, i, s_i)$ from each party, the functionality then generates and stores the required communal keys k_{com}. Furthermore, the functionality generates and outputs the input, output, and evaluation keys $k_{\mathsf{input}}, k_{\mathsf{eval}}, k_{\mathsf{output}}$ to $\mathcal{E}$.

$\mathsf{Enc}(s, m)$**:** Upon input of $\mathsf{Enc}(s, m)$ by $\mathcal{P}_i$, generate some c, store $(s, m, c) \in \mathbb{S}$, return c to $\mathcal{P}_i$.

$\mathsf{Input}(c)$**:** Upon input of $\mathsf{Input}(c)$ from party $\mathcal{P}_i$, send c to $\mathcal{E}$.

$\mathsf{KS}_{\mathsf{toComm}}(c, k_{\mathsf{input}})$**:** Upon input of $\mathsf{KS}_{\mathsf{toComm}}(c, k_{\mathsf{input}})$ by $\mathcal{E}$, the functionality checks whether $(c, s, m) \in \mathbb{S}$ with $s \notin k_{\mathsf{com}}$. If so, the functionality stores $(c', s', m) \in \mathbb{S}$, where $s' \in k_{\mathsf{com}}$ and outputs c' to $\mathcal{E}$.

$\mathsf{KS}_{\mathsf{fromComm}}(c, k_{\mathsf{output}})$**:** Upon input of $\mathsf{KS}_{\mathsf{fromComm}}(c, k_{\mathsf{output}})$ by $\mathcal{E}$, the functionality checks whether $(c, s, m) \in \mathbb{S}$ with $s \in k_{\mathsf{com}}$. If so, the functionality stores $(c', s_i, m) \in \mathbb{S}$, outputs c' to $\mathcal{E}$.

$\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$**:** Upon input of $\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$ by $\mathcal{E}$ where f is some function, k_{eval} are the evaluation keys, and C is the set of ciphertexts to be used as inputs, the functionality stores $(c', s, f(m_1, \ldots, m_n)) \in \mathbb{S}$ and sends c' to $\mathcal{E}$.

$\mathsf{Output}(c, i)$**:** Upon input of $\mathsf{Output}(c, i)$ by $\mathcal{E}$, the functionality outputs c to $\mathcal{P}_i$.

$\mathsf{Dec}(c, s)$**:** Upon input of $\mathsf{Dec}(c, s)$, if there is a stored tuple (s, m, c) for some m, return m. Else, return $\perp$.

case it aborts. Otherwise, it calls MKDec to see if the tuple $(c, \{\mathbf{s}_i\}_{P_i \in A}, m)$ exists. As the parties in A only hold their own keys, the only ciphertexts for which MKDec will contain such a tuple are those which (a subset of) the parties in A called to encrypt. Hence, they are also allowed to obtain the decryption. Otherwise, the parties in A will not hold the required keys to decrypt, and so $s_l \not\subset \{\mathbf{s}_i\}_{P_i \in A}$ and so the tuple does not exist. In the latter case, the parties in A learn nothing as required. Hence, we can conclude that $\Pi_{\text{mpFHE+}}.\mathsf{Dec}$ correctly implements $\mathcal{F}_{\text{mpFHE+}}.\mathsf{Dec}$.

The $\mathsf{KeyGen}_{\mathsf{FHE}}$ and Enc Algorithm An equivalent argument can be made to show the equivalent conditions for $\mathcal{F}_{\text{mpFHE}}.\mathsf{KeyGen}_{\mathsf{FHE}}$ and Enc, and as such we make the argument for KeyGen_{FHE} and leave the other as an exercise. Upon calling $\mathsf{KeyGen}_{\mathsf{FHE}}$ in $\mathcal{F}_{\text{mpFHE+}}$, a party would expect a key $\mathbf{s}_i$. When running the protocol, the party instead calls $\mathcal{F}_{\text{skFHE+}}.\mathsf{KeyGen}$, which returns the same result.

The KeyGen_{MPC} Algorithm Upon input of $(\mathsf{KeyGen}_{MPC}, i, \mathbf{s}_i)$ by each party $\mathcal{P}_i$ in the functionality, the expected output is that the functionality stores the communal key s, and outputs $k_{\mathsf{input}}, k_{\mathsf{eval}}$, and k_{output} to the server $\mathcal{E}$.

In the protocol however, the parties all run the MPC protocol. We argue that collectively using the MPC protocol to generate the secret key and not outputting it is equivalent to the functionality storing the secret key. There are three places where the functionality uses the stored key s: $\mathsf{KS}_{\mathsf{toComm}}$, $\mathsf{KS}_{\mathsf{fromComm}}$, and Eval. Each of these is equivalent to the server possessing the correct keys used during the appropriate calls to $\mathcal{F}_{\text{skFHE+}}$. In $\mathsf{KS}_{\mathsf{toComm}}$ the check is equivalent to the server possessing the right keyswitching key in the protocol as the only way to perfrom $\mathsf{KS}_{\mathsf{toComm}}$ is to pass

Fig. 10: *The Multiparty-FHE Protocol*

The mp-FHE protocol, Π_{mp} in the $\mathcal{F}_{\text{skFHE}}, \mathcal{F}_{\text{MPC}}$-Hybrid Model

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

$\mathsf{Init}(\mathbb{K})$**:** The parties call $\mathcal{F}_{\text{MPC}}.\mathsf{Init}(\mathbb{K})$ and $\mathcal{F}_{\text{skFHE}}.\mathsf{Init}$.
KeyGen **:** The party calls $\mathcal{F}_{\text{skFHE}}.\mathsf{KeyGen}$.
$\mathsf{KeyGen}(s_i)$**:** The parties run $\mathcal{F}_{\text{MPC}}.\mathsf{Input}$, followed by a mix of $\mathcal{F}_{\text{MPC}}.\mathsf{Add}$ and $\mathcal{F}_{\text{MPC}}.\mathsf{Multiply}$ such that the following is computed:

- $s \leftarrow \mathcal{F}_{\text{skFHE}}.\mathsf{KeyGen}$
- for $i \in [k]$:
  - $\mathsf{ksk}_i \leftarrow \mathcal{F}_{\text{skFHE}}.\mathsf{KSKGen}(s_i, s)$
  - $\mathsf{ksk}'_i \leftarrow \mathcal{F}_{\text{skFHE}}.\mathsf{KSKGen}(s, s_i)$
- $k_{\mathsf{eval}} \leftarrow \mathcal{F}_{\text{skFHE}}.\mathsf{EvalKeyGen}$.

Finally, the parties run $\mathcal{F}_{\text{MPC}}.\mathsf{Output}$ so that $\mathcal{E}$ gets $k_{\mathsf{input}} := \{\mathsf{ksk}_i\}_i, k_{\mathsf{eval}}, k_{\mathsf{output}} := \{\mathsf{ksk}'_i\}_i$.
$\mathsf{Enc}(s, m)$**:** The party calls $\mathcal{F}_{\text{skFHE}}.\mathsf{Enc}(s, m)$.
$\mathsf{Input}(c)$**:** The party sends c to $\mathcal{E}$.
$\mathsf{KS}_{\mathsf{toComm}}(c)$**:** The evaluator calls $\mathcal{F}_{\text{skFHE}}.\mathsf{KS}(c, \mathsf{ksk})$ where ksk is the appropriate element of k_{input}.
$\mathsf{KS}_{\mathsf{fromComm}}(c)$**:** The evaluator calls $\mathcal{F}_{\text{skFHE}}.\mathsf{KS}(c, \mathsf{ksk})$ where ksk is the appropriate element of k_{output}.
$\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$**:** The evaluator calls $\mathcal{F}_{\text{skFHE}}.\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$.
$\mathsf{Dec}(c, s)$**:** The party calls $\mathcal{F}_{\text{skFHE}}.\mathsf{Dec}(c, s)$.

$\mathcal{F}_{\text{skFHE+}}.\mathsf{KS}$ a keyswitching key to the communal key s. Such keys are also generated and stored by $\mathcal{E}$ in k_{input}. In $\mathsf{KS}_{\mathsf{fromComm}}$, the check is likewise equivalent to $\mathcal{E}$ possessing the correct ksk for the call to $\mathcal{F}_{\text{skFHE+}}.\mathsf{MKKS}$ with a ksk from k. The only such key is stored in k_{output}, generated during the MPC protocol. Finally, in Eval, the check is equivalent to the server possessing k_{eval} as this is generated using $\mathcal{F}_{\text{skFHE+}}.\mathsf{EvalKeyGen}$ called in MPC, with s as input.

The generation of the remaining keys is indistinguishable, as will be demonstrated in the algorithm in which they are used.

The Input Algorithm The server receives the input ciphertext in both cases.

The Output Algorithm The appropriate party receives the output ciphertext in both cases.

The $\mathsf{KS}_{\mathsf{toComm}}$ Algorithm In the functionality, this protocol returns a keyswitched ciphertext $\iff$ (s,c,m) is a stored ciphertext and that $s \notin k_{\mathsf{com}}$. If so, (s',c',m) is stored and c' is output to $\mathcal{E}$.

In the protocol, the check that (s, c, m) is a stored ciphertext is performed by $\mathcal{F}_{\text{skFHE+}}.\mathsf{KS}$ and the check that $s \notin k_{\mathsf{com}}$ is equivalent to $\mathcal{E}$ passing the correct ksk as $\mathcal{E}$ will possess this key (and thus the check $(s, s', \mathsf{ksk}) \in \mathsf{KSK}$ will pass) $\iff$ this ksk is generated in $\mathsf{KeyGen}_{\mathsf{MPC}}$ as generating others outside the protocol would require the knowledge of s, which $\mathcal{E}$ does not have. The only such keys generated are those to s, the communal key.

The $\mathsf{KS}_{\mathsf{fromComm}}$ Algorithm In the functionality, this algorithm returns a keyswitched ciphertext $\iff$ (s,c,m) is a stored ciphertext and that $s \in k_{\mathsf{com}}$. If so, $(\{s_i\}_i, c', m)$ is stored and c' is output to $\mathcal{E}$.

Fig. 11: *The Multiparty-FHE Protocol With Multikey Decryption*

The mp-FHE protocol with multikey decryption, $\Pi_{\text{mpFHE+}}$ in the $\mathcal{F}_{\text{skFHE+}}, \mathcal{F}_{\text{MPC}}$-Hybrid Model

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

$\mathsf{Init}(\mathbb{K})$**:** The parties call $\mathcal{F}_{\text{MPC}}.\mathsf{Init}(\mathbb{K})$ and $\mathcal{F}_{\text{skFHE+}}.\mathsf{Init}$.
KeyGen **:** The party calls $\mathcal{F}_{\text{skFHE+}}.\mathsf{KeyGen}$.
$\mathsf{KeyGen}(s_i)$**:** The parties run $\mathcal{F}_{\text{MPC}}.\mathsf{Input}$, followed by a mix of $\mathcal{F}_{\text{MPC}}.\mathsf{Add}$ and $\mathcal{F}_{\text{MPC}}.\mathsf{Multiply}$ such that the following is computed:

- $s \leftarrow \mathcal{F}_{\text{skFHE+}}.\mathsf{KeyGen}$
- for $i \in [k]$:
  - $\mathsf{ksk}_i \leftarrow \mathcal{F}_{\text{skFHE+}}.\mathsf{KSKGen}s_i, s$
- $k_{\mathsf{output}} \leftarrow \mathcal{F}_{\text{skFHE+}}.\mathsf{MKKSKGen}(s, \{s_i\}_{i\in[k]}$
- $k_{\mathsf{eval}} \leftarrow \mathcal{F}_{\text{skFHE}}.\mathsf{EvalKeyGen}$.

Finally the parties run $\mathcal{F}_{\text{MPC}}.\mathsf{Output}$ so that $\mathcal{E}$ gets $k_{\mathsf{input}} := \{\mathsf{ksk}_i\}_i, k_{\mathsf{eval}}, k_{\mathsf{output}} := \{\mathsf{ksk}'_i\}_i$.
$\mathsf{Enc}_s^{(}, m)$**:** The party calls $\mathcal{F}_{\text{skFHE+}}.\mathsf{Enc}_s^{(}, m)$.
$\mathsf{Input}(c)$**:** The party sends c to $\mathcal{E}$.
$\mathsf{KS}_{\mathsf{toComm}}(c, k_{\mathsf{input}})$**:** The evaluator calls $\mathcal{F}_{\text{skFHE+}}.\mathsf{KS}(c, \mathsf{ksk})$ where ksk is the appropriate element of k_{input}.
$\mathsf{KS}_{\mathsf{fromComm}}(c, k_{\mathsf{output}})$**:** The evaluator calls $\mathcal{F}_{\text{skFHE+}}.\mathsf{MKKS}(c, \mathsf{ksk})$ where ksk is the appropriate element of k_{output}.
$\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$**:** The evaluator calls $\mathcal{F}_{\text{skFHE}}.\mathsf{Eval}(f, k_{\mathsf{eval}}, C)$.
$\mathsf{Output}(c, i)$**:** The evaluator sends c to $\mathcal{P}_i$.
$\mathsf{Dec}(c, s)$**:** The parties instantiate $\mathcal{F}_{\text{MPC}}$ to execute $\mathcal{F}_{\text{skFHE+}}.\mathsf{MKDec}(c, s_{ii})$ and output to all parties.

In the protocol, the check that (s, c, m) is a stored ciphertext is performed by $\mathcal{F}_{\text{skFHE+}}.\mathsf{MKKS}$ and the check that $s \in k_{\mathsf{com}}$ is equivalent to $\mathcal{E}$ passing the correct ksk as $\mathcal{E}$ will possess this key (and thus the check $(s, \{s_i\}_i, ksk) \in KSK$ will pass) $\iff$ this ksk is generated in $\mathsf{KeyGen}_{\mathsf{MPC}}$ as generating others outside the protocol would require the knowledge of s, which $\mathcal{E}$ does not have. The only such keys generated are those from s, the communal key.

The Eval Algorithm In the functionality, this algorithm returns c' encrypting $f(m_1, \ldots, m_n)$ with the appropriate data stored $\iff$ all input ciphertexts c_i are in $\mathbb{S}$ with the same associated key s, and k_{eval} was generated by $\mathcal{F}_{\text{mpFHE+}}.\mathsf{EvalKeyGen}$ using s as input.

In the protocol, the same checks are performed by $\mathcal{F}_{\text{skFHE+}}.\mathsf{Eval}$, and the same things are stored and returned by the same call to $\mathcal{F}_{\text{skFHE+}}.\mathsf{Eval}$

To show that security holds note that $\Pi_{\text{mpFHE+}}$ inherits most of its security guarantees from the hybrid construction and the underlying $\mathcal{F}_{\text{MPC}}$ and $\mathcal{F}_{\text{skFHE+}}$. The only consideration that is left to make are the generation and handling of the common key. The main issue is that we generate the secret key that is used to perform the entire computation in MPC. 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 $n-1$ parties can be corrupt, or to extract the key from the key switching keys. Since the key switching keys are opened only to the evaluator, $\mathcal{E}$, who doesn't simultaneously know all of the parties' private keys this would break the security of $\mathcal{F}_{\text{skFHE+}}$.

A similar argument for security and that Π_{mpFHE} implements $\mathcal{F}_{\text{mpFHE}}$ can be made.

3.4 Practical Considerations for the Choice of MPC and FHE Schemes

There are a number of considerations to consider when choosing the MPC and FHE protocols which realize the functionalities given above in a practical setting.

The FHE protocol needs to have an efficient key switching protocol of which the keys do not depend on the ciphertext, as the generation of keys midway through the FHE protocol would require another execution of the MPC protocol. In order to have have security against a server colluding with a player, i.e. $\mathcal{F}_{\text{mpFHE+}}$, the FHE scheme needs a sister multi-key protocol or the ability to support multi-key key switching and decryption with specialized dedicated algorithms. A general class of FHE protocols that support this include the linear decryption FHE protocols, [COS+22], such as BGV [BGV12], BFV [FV12], and TFHE [CGGI16,CGGI20], as well as FINAL [BIP+22,JMPP24,PvLZ24] in particular.

The MPC protocol has far less stringent requirements on it. All we require is a full threshold protocol which is secure against semi-honest adversaries. Security against a malicious adversary would only provide marginal benefits, as a single malicious entity, server or party, could perform a trivial denial of service attack against the FHE protocol. Examples of these (and overperforming) are plentiful, e.g. GMW [GMW91], SPDZ [DPSZ12], and BGW [BGW88].

4 A Case Study: Multiparty FINAL

To exemplify the above generic construction $\Pi_{\text{mpFHE+}}$, 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 5, we then implement this instantiation and evaluate how it scales with the number of parties.

4.1 Key Generation

We use MPC during preprocessing to compute the keys used during the protocol. Note that while some communal LWE and NGS keys theoretically exist, we never reveal these values. Instead, we compute the key switching keys from the individual party's LWE keys to the communal LWE key, as the key switching keys from the communal NGS ciphertext to the communal LWE key, and the key switching key from the single communal LWE key to multi-key LWE under all of the participating parties' keys.

In our protocol, we require the generation of several keys during the MPC:

- For each party $\mathcal{P}_i$ where $i \in [k]$, the key switching key from their LWE key $\mathbf{s}_i$ to the communal LWE key $\mathbf{s}$.
- The key switching key from the communal NGS key f to the communal LWE key $\mathbf{s}$.
- The bootstrapping keys bsk_i for $i \in [n]$ which allows for an indirect key switch from sk-LWE to NGS.
- One of the following sets of key switching keys for decryption:
  - The key switching keys from the communal LWE key $\mathbf{s}$ to single-key LWE under each party's key $\mathbf{s}_i$.
  - The key switching key from the communal LWE key $\mathbf{s}$ to multi-key LWE under all parties' keys $\mathbf{s}_1, \ldots, \mathbf{s}_k$.

Fig. 12: *The algorithm for computing the bootstrapping keys*

The Bootstrapping Key Generation Algorithm, Π_{KeyGen}

Input: Each party $\mathcal{P}_i$ inputs their LWE key $\mathbf{s}_i$. The Security parameter λ. A boolean $\mathsf{IsMKDec}$.
Output: The key switching keys ksk_i from $\mathbf{s}_i$ to $\mathbf{s}$ for all $i \in [k]$.
The key switching key $\mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}}$ from f to $\mathbf{s}$.
The key switching key ksk_{out} from $\mathbf{s}$ to $\mathbf{s}_{1,\ldots,k}$.
Algorithm:

1. $(N, Q, \zeta, B, l) \leftarrow \mathsf{NGS.Setup}(\lambda)$
2. $(q, p, \sigma) \leftarrow \mathsf{MKLWE.Setup}(\lambda)$
3. $\mathbf{s} \leftarrow \mathsf{LWE.KeyGen}(\sigma)$.
4. $f \leftarrow \mathsf{NGS.KeyGen}(\zeta, N)$.
5. $\mathbf{A} \xleftarrow{\$} \mathbb{F}_q$.
6. $e \xleftarrow{\$} \chi_\sigma$.
7. $\mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}} \leftarrow (\mathbf{A}, b := \mathbf{A}\cdot\mathbf{s} + e + G\cdot\phi(f))$.
8. for $i \in [k]$ do:
   (a) $\mathsf{ksk}_i \leftarrow \mathsf{KSKGen}_{\mathsf{LWE}}(\mathbf{s}_i, \mathbf{s})$.
9. for i in $[n]$ do:
   (a) $\mathbf{g}_i \xleftarrow{\$} R_n$
   (b) $\mathsf{bsk}_i \leftarrow \frac{\mathbf{g}_i}{f} + \mathfrak{g}s_i$.
10. if $\mathsf{IsMKDec}$:
    (a) $\mathsf{ksk}_{out} \leftarrow \mathsf{KSKGen}_{\mathsf{MKLWE}}(\mathbf{s}, (\mathbf{s}_1, \ldots, \mathbf{s}_n))$.
11. else:
    (a) for i in $[k]$:
        i. $\mathsf{ksk}_{out,i} \leftarrow [\mathsf{KSKGen}_{\mathsf{LWE}}(\mathbf{s}, \mathbf{s}_i)]$.
12. Output $\mathsf{ksk}_i, \mathsf{ksk}_{out}, \mathsf{ksk}_{\mathsf{NTRU}\to\mathsf{LWE}}$.

To compute the bootstrapping keys, we run the algorithm in Figure 12 in MPC to generate the keys listed above. We note 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:

- key switch a ciphertext of the form $\mathsf{LWE.Enc}_{\mathbf{s}_i}(m)$ to $\mathsf{LWE.Enc}_{\mathbf{s}}(m)$, used during the input phase.
- key switch a ciphertext of the form $\mathsf{NTRU.Enc}_f(m)$ to $\mathsf{LWE.Enc}_{\mathbf{s}}(m)$, used during the bootstrapping.
- one of the following for use during the output phase:
  - key switch a ciphertext of the form $\mathsf{LWE.Enc}_{\mathbf{s}}(m)$ to $\mathsf{MKLWE.Enc}_{\mathbf{s}_1,\ldots,\mathbf{s}_k}(m)$.
  - key switch a ciphertext of the form $\mathsf{LWE.Enc}_{\mathbf{s}}(m)$ to $\mathsf{LWE.Enc}_{\mathbf{s}_i}(m)$ for any i.

The noise of the key switching keys is that of a fresh ciphertext, see section 2.5.

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

4.2 Bootstrapping

In this section we introduce the bootstrapping. This is almost exactly the bootstrapping in [BIP+22], except for the fact that we explicitly describe the LWE based version alluded to in FINAL.

The bootstrapping has the following structure. First, there's a minor step doing some preprocessing. Second, a blind rotation is done using CMux gates. This homomorphically decrypts the preprocessed ciphertext, providing an NGS ciphertext of the same message. Third, this NGS ciphertext is key switched 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.

Fig. 13: *Bootstrapping a Single-Key LWE Ciphertext*

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.

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

As in [BIP+22] and [PvLZ24], 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 4.1. *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. Follows from lemma 7 of [PvLZ24].

Given as input the LWE ciphertext $\mathsf{LWE.Enc}_s(m)$ parsed as $(b, \mathbf{a})$, the first step of the bootstrapping is to preprocess 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's 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 2.4.

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 lemma can be proven:

Theorem 4.1 ([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 2.1 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 4.1, 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 4.1, we can key switch from NGS to LWE using the key switching 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 4.2. *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 4.1, 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 2.4, 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 2.3 the theorem is proven:

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

4.3 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 4.2, 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 4.3. *If the* NGS *ciphertexts of the protocol in Figure 13 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 Figure 13 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 $Q = \mathcal{O}(N \log(N))$.

4.4 Decryption

After bootstrapping, the server holds an LWE ciphertext of the output encrypted under s. If this isn't the end of the protocol, the server then continues the computation, bootstrapping as needed. If this is the end of the protocol, then the server must perform some kind of keyswitching before sending the output ciphertext back to the parties.

For this keyswitching and its associated decryption, we have a number of options. The first is to have the server key switch from the common key to a multi-key ciphertext under the individual keys for all the parties, then decrypting as in $\Pi_{\mathrm{mpFHE+}}$. The second is to key switch to single-key LWE once for each party as in Π_{mpFHE} (Figure 10). We briefly discuss the ramifications of each of these options in this section.

We first discuss the option of key switching to multi-key. The upsides of this are that there's only a single key switch required and that the server cannot collaborate with a single user to prevent the other users from obtaining the output. The downside is that all the parties must once again collaborate at the end of the protocol to obtain the output.

Second, we discuss the option of key switching once each for each party's key. The upside is that none of the parties need interact with each other once the keys are agreed upon in the preprocessing. However, this leaves the door open to a trivial attack in which the server can allow any party to read any intermediate value or even the output of the computation by simply key switching to that party's individual key and send the desired ciphertexts their way. Furthermore, this requires k executions of the key switching procedure, though this is amortized over the cost of the computation as a whole.

A Note on Breaking the Black Box We also note that an improvement can be made to this instantiation by breaking the black box of the decryption. In this improvement, instead of generating a keyswitching key from sk-LWE to mk-LWE, you instead have the parties generate a keyswitching key from sk-NGS to mk-LWE, and then replace the keyswitch in the final bootstrapping with a keyswitch to mk-LWE instead. This saves the effort of an extra keyswitch. However, we choose to not do this to faithfully implement our functionality.

Theorem 4.4. *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 4.2 and Lemma 2.2. 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 $\mathcal{O}(n \log n) < \mathcal{O}(n^2)$.

4.5 Key Compression

We observe that in practice, storing the key switching 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 key switching 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 $t \cdot q + \lambda$ 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.

5 Implementation

In order to show the practicality and correctness of our construction, we implement the multiparty version of FINAL as described in Section 4. Note that this is a prototype implementation and does not realize the full capabilities of FINAL (i.e. 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 we treat the MPC preprocessing and key generation as a black box, our implementation implements the ideal functionality directly as a single program, and does not try to emulate network communication or any specific MPC protocols. 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 key switching keys. As can be seen in Table 3, we note that the storage of the key material is not restrictive requiring at most 22.7GB when the protocol is executed by 512 parties. While this is not part of our implementation, we recall that the memory footprint of the key switching 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.

Concretely, our implementation generates the communal LWE and NGS keys $\mathbf{s}$ and f (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 key switching keys $\mathsf{ksk}_{\mathbf{s}_i \to \mathbf{s}}$ and a (NGS to LWE) key switching key $\mathsf{ksk}_{f \to \mathbf{s}_{1\ldots k}}$ 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,[5] which the server can transform into encryptions under $\mathbf{s}$ through $\mathsf{ksk}_{\mathbf{s}_i \to \mathbf{s}}$ 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 key switching step makes use of $\mathsf{ksk}_{f \to \mathbf{s}_{1\ldots k}}$ instead of the usual $\mathsf{ksk}_{f \to \mathbf{s}}$. We take this approach, rather

[5] 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.

than performing a key switch from $\mathbf{s}$ so $\mathbf{s}_{1\ldots k}$, to limit the final output noise and prevent decryption failures. Depending on the specific application,[6] 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. The decrypt these final MKLWE ciphertexts, we simply assume all parties are present and can decrypt in a round-robin fashion.

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.

In table 2, 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, input, gate evaluation, output, and multi-key decryption for 2, 4, 8, 16 and 32 parties in Figures 14, 15, 16, 17, and 18 respectively. The implementation is not limited to the number of parties being a power of two, and can scale to arbitrarily many parties without any increase in the parameters, but these choices should give a clear overview of the scaling behavior for our scheme.

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 key switch 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 key switching keys per party.

In table 3 and table 4, we compare the size of our required key material and the time necessary to perform a single bootstrap to prior work. 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.

Protocol	Key Size in GB								
k	2	4	8	16	32	64	128	256	512
PvLZ [PvLZ24]	.32	4.5	119	5337	-	-	-	-	-
PR [PR23]	.42	.63	1.1	2.2	-	-	-	-	-
KMS [KMS24]	.215	0.285	.25	.285	.321	-	-	-	-
AKÖ [AKÖ23]	.08	.24	.66	.93	2.0	4.1	9.1	37	80
This work	.13	.22	.39	.75	1.45	2.87	5.7	11.4	22.7

Table 3: *A comparison of total key material in GB where k is the number of participating parties, [PR23] counts only the size of* bsk *and not* ksk.

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

LWE.Enc	LWE.Dec	Bootstrap	Key gen
178 μs	1.39 μs	88.65 ms	3.43 s

Table 2: *An overview of our running times for basic operations in the single-key FINAL scheme.*

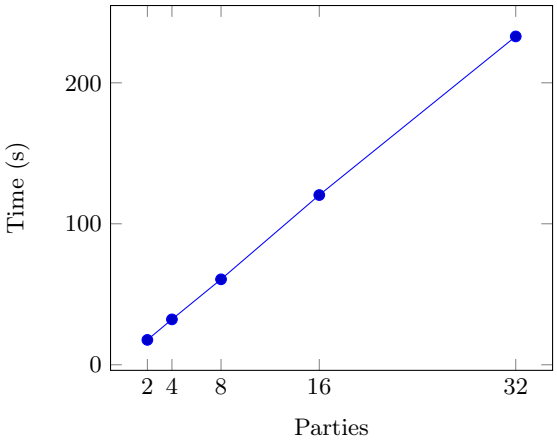


Fig. 14: *Multiparty key generation*

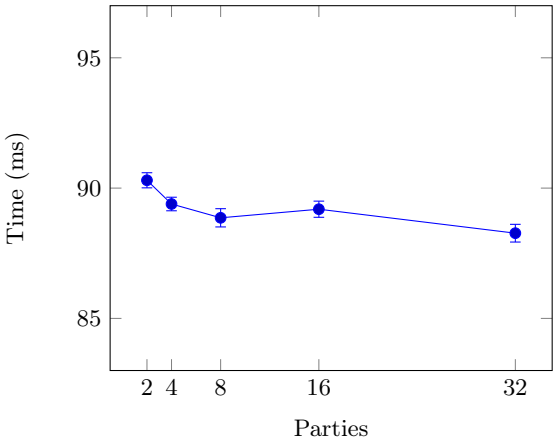


Fig. 15: *Multiparty input*

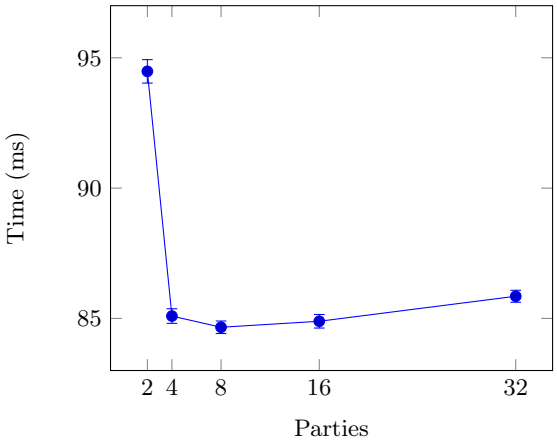


Fig. 16: *Multiparty gate evaluation*

Acknowledgements

This work was supported by Cyber Security Research Flanders with reference number VR20192203 and by the FWO under an Odysseus project GOH9718N.

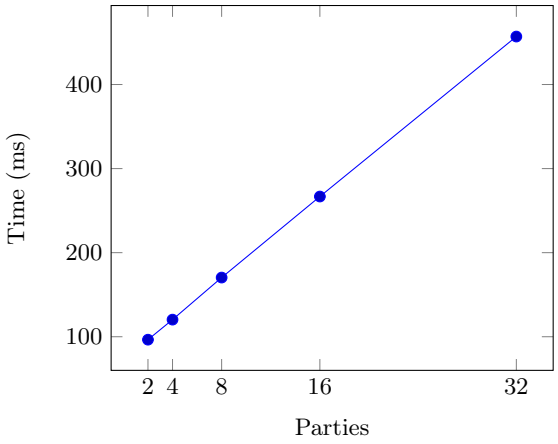


Fig. 17: *Multiparty output*

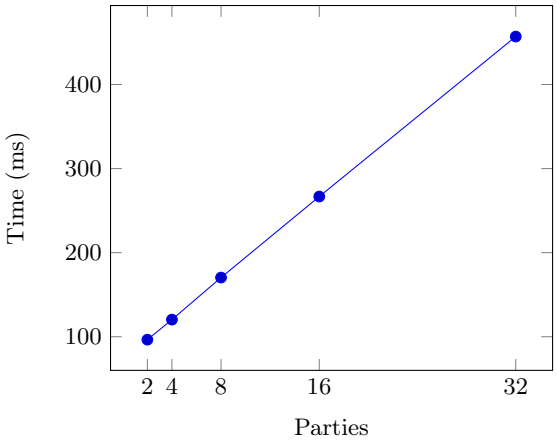


Fig. 18: *Multi-key decryption*

Protocol	Bootstrap time in ms								
k	2	4	8	16	32	64	128	256	512
PvLZ [PvLZ24]	No Implementation Data available								
PR [PR23]	200	330	460	900	-	-	-	-	-
KMS [KMS24]	240	880	2230	5650	13940	-	-	-	-
AKÖ [AKÖ23]	190	560	1200	1800	4300	8600	18000	-	-
This work	88.6 (Ours) / 48 (Original, [BIP$^+$22])								

Table 4: *A comparison of time required to perform a bootstrapping operation, where k is the number of participating parties.*

References

AJL$^+$12. Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Berlin, Heidelberg, April 2012.

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, Part I*, volume 14344 of *LNCS*, pages 3–23. Springer, Cham, September 2023.

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.

BGG$^+$18. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596. Springer, Cham, August 2018.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

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, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 188–215. Springer, Cham, December 2022.

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*, pages 395–412. ACM Press, November 2019.

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 21International Conference on Applied Cryptography and Network Security, Part I*, volume 12726 of *LNCS*, pages 460–479. Springer, Cham, June 2021.

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, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33. Springer, Berlin, Heidelberg, December 2016.

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.

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, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 446–464. Springer, Berlin, Heidelberg, April 2012.

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*, volume 13409 of *LNCS*, pages 540–563. Springer, Cham, September 2022.

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 617–640. Springer, Berlin, Heidelberg, April 2015.

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, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Berlin, Heidelberg, August 2012.

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

GMW91. Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, July 1991.

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, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 3–35. Springer, Singapore, December 2023.

JMPP24. Robin Jadoul, Axel Mertens, Jeongeun Park, and Hilder V. L. Pereira. NTRU-based FHE for larger key and message space. In Yannan Li Tianqing Zhu, editor, *ACISP 24, Part I*, volume 14895 of *LNCS*, pages 141–160. Springer, Singapore, July 2024.

KLSW21. Hyesun Kwak, Dongwon Lee, Yongsoo Song, and Sameer Wagh. A unified framework of homomorphic encryption for multiple parties with non-interactive setup. Cryptology ePrint Archive, Report 2021/1412, 2021.

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, Part II*, volume 14604 of *LNCS*, pages 354–385. Springer, Cham, April 2024.

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 ACM STOC*, pages 1219–1234. ACM Press, May 2012.

MBH22. Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. An efficient threshold access-structure for RLWE-based multiparty homomorphic encryption. Cryptology ePrint Archive, Report 2022/780, 2022.

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*, volume 107 of *LIPIcs*, pages 100:1–100:14. Schloss Dagstuhl, July 2018.

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

Par21. Jeongeun Park. Homomorphic encryption for multiple users with less communications. Cryptology ePrint Archive, Report 2021/1085, 2021.

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

PvLZ24. Jeongeun Park, Barry van Leeuwen, and Oliver Zajonc. FINALLY: A multi-key FHE scheme based on NTRU and LWE. *IACR Communications in Cryptology*, 1(3), 2024.