Noah's Ark: Efficient Threshold-FHE Using Noise Flooding

Morten Dahl[1], Daniel Demmler[1], Sarah El Kazdadi[1], Arthur Meyre[1], Jean-Baptiste Orfila[1], Dragos Rotaru[1], Nigel P. Smart[1,2], Samuel Tap[1], and Michael Walter[1]

[1] Zama, Paris, France.
[2] COSIC, KU Leuven, Leuven, Belgium.
morten.dahl@zama.ai,daniel.demmler@zama.ai,sarah.elkazdadi@zama.ai,
arthur.meyre@zama.ai,jb.orfila@zama.ai,dragos.rotaru@zama.ai,
nigel.smart@kuleuven.be,samuel.tap@zama.ai,michael.walter@zama.ai,

Abstract. We outline a secure and efficient methodology to do threshold distributed decryption for LWE based Fully Homomorphic Encryption schemes. Due to the smaller parameters used in some FHE schemes, such as Torus-FHE (TFHE), the standard technique of "noise flooding" seems not to apply. We show that noise flooding can also be used with schemes with such small parameters, by utilizing a switch to a scheme with slightly higher parameters and then utilizing the efficient bootstrapping operations which TFHE offers. Our protocol is proved secure via a simulation argument, making its integration in bigger protocols easier to manage.

Table of Contents

1 Introduction

The problem of threshold decryption for Fully Homomorphic Encryption (FHE) schemes, called threshold-FHE from henceforth, is as old as FHE itself. The problem is for a set of n parties to have a secret sharing of the underlying FHE secret key so that they can between them decrypt a given FHE ciphertext correctly, in the case where at most t of the parties are corrupt. Indeed, Gentry's original thesis [Gen09] mentioned threshold-FHE as a way of utilizing FHE to perform a very low round complexity semi-honest MPC protocol.

To understand the technical problem with threshold-FHE it is worth considering the "format" of a simple FHE - either public or private key - scheme To explain we utilize the format of

BFV/TFHE [FV12, CGGI16, CGGI20] ciphertexts, but a similar discussion can be provided for other FHE schemes such as BGV [BGV12]. Consider encrypting an element $m \in \mathbb{Z}_p$, using a standard Learning-With-Errors (LWE) ciphertext of the form $(\mathbf{a}, b)$ with ciphertext modulus q, where $\mathbf{a} \in \mathbb{Z}_q^\ell$ and $b \in \mathbb{Z}_q$, using the equation

$$b = \mathbf{a} \cdot \mathbf{s} + e + \Delta \cdot m \pmod q$$

where $\Delta = \lfloor q/p \rfloor$, e is some "noise" term and $\mathbf{s} \in \mathbb{Z}_q^\ell$ is the secret key. Usually, in the FHE setting, $\mathbf{s}$ is chosen to be a vector of small norm, for example $\mathbf{s} \in \{0,1\}^\ell$.

To enable threshold-FHE we first secret share the secret key $\mathbf{s}$ among n parties, a process which we shall denote by $[\mathbf{s}]^{\langle t,q \rangle}$ to signal a sharing modulo q with respect to a threshold $t < n$ linear secret sharing scheme. On input of the ciphertext $(\mathbf{a}, b)$ we can then produce trivially a secret sharing of the value $e + \Delta \cdot m$ by computing

$$[t]^{\langle t,q \rangle} = b - \mathbf{a} \cdot [\mathbf{s}]^{\langle t,q \rangle} = [e + \Delta \cdot m]^{\langle t,q \rangle}.$$

By opening the value of $[t]^{\langle t,q \rangle}$ all parties can then perform rounding to obtain m. However, this reveals the value of e, which combined with the ciphertext and the message, will reveal information about the secret key $\mathbf{s}$.

The way around this is to add some additional noise into the secret sharing before the opening. Thus the decrypting parties somehow generate an additional secret shared noise term $[E]^{\langle t,q \rangle}$, and the value which is opened is now

$$[t]^{\langle t,q \rangle} = b - \mathbf{a} \cdot [\mathbf{s}]^{\langle t,q \rangle} + [E]^{\langle t,q \rangle} = [e + E + \Delta \cdot m]^{\langle t,q \rangle}.$$

The key concern is then that E should introduce enough randomness to mask the e value after the shared value $[t]^{\langle t,q \rangle}$ is opened. If E is too small then too much information about e is revealed, if E is too big then the final rounding will not reveal the correct value of m. Diagrammatically we can consider this process as approximated by the diagram in Figure 1.

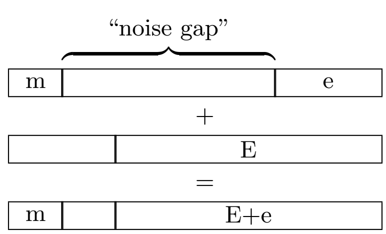

Fig. 1. Representation of the noise addition for threshold decryption

To mask, statistically, all information in e we would (naively) require E to be chosen uniformly from a range which is 2^{stat} larger than e. Thus if we can bound the ciphertext noise by $|e| < B$, then we would require E to be chosen uniformly in the range $[-2^{\mathsf{stat}} \cdot B, \ldots, 2^{\mathsf{stat}} \cdot B]$. This process is often dubbed "noise flooding" in the literature. However, this would mean we require $\Delta > 2^{\mathsf{stat}} \cdot B$, which in turn means that the ciphertext modulus q needs to be "large".

We show that by adding an E term on, which is itself the addition of at least two uniform distributions in the range $[-2^{\mathsf{stat}} \cdot B, \ldots, 2^{\mathsf{stat}} \cdot B]$, we are able to obtain a statistical distance of actually $2^{-2\cdot\mathsf{stat}}$. We can, hence, obtain enough security by selecting $\mathsf{stat} \approx 40$, and so reduce the need for very much larger q values.

In theory it may be possible to select E from a smaller range, and rely on game based security assumptions. This approach is taken in two recent papers, [BS23] and [CSS+22], via the Renyi divergence. This methodology enables parameters to be chosen in which q is much smaller than the above analysis would require. As we discuss below this approach leads to additional problems in the larger protocols in which we embed our threshold decryption. Thus the use of Renyi divergence is not without problems in this situation.

Before proceeding we note that in many situations there is no problem with the increased size of q that the noise flooding approach requires. The key observation is that FHE enables the use of a bootstrapping operation. The purpose of this operation is to reduce the size of the noise e in the ciphertext $(\mathbf{a}, b)$ to be as small as possible. Thus if bootstrapping is performed, and the FHE scheme is such that the noise gap between e and m in Figure 1 is large enough, then the noise flooding methodology will work "out-of-the-box". Thus for BFV/BGV implementations which enable bootstrapping there is no problem to solve, as noise flooding is enough.

When using BFV/BGV in an SHE leveled mode then the problem also does not occur. In such schemes each level essentially adds an extra 14-24 bits (depending on the implementation) into the noise gap [KPR18, BCS19, OSV20]. Thus by simply increasing the number of levels by a small constant (say, two or three) one can obtain a noise gap which is enough to apply the flooding technique. Thus in such schemes our methodology in Section 4 can be applied, without any need for prior pre-processing.

Thus the only place where noise flooding is in practice a problem is when the FHE parameters are such that the noise gap is tiny, even after a bootstrapping operation is performed. This is exactly the situation in TFHE where one (usually) selects a relatively small q value (for example $q = 2^{64}$). This small q value, and associated small LWE dimension ℓ, requires the size of the noise even after bootstrapping to be around 2^{30} in order to ensure security. This means the noise gap is too small, but only by tens of bits. In this work we solve this problem for TFHE, by utilizing the fast bootstrapping enabled by TFHE. In some sense we protect the initial FHE ciphertext from the flooding operation, by placing the underlying message in a larger ciphertext (a kind of protective Noah's Ark).

1.1 Historical Discussion

At about the time of Gentry's thesis on FHE in 2009 [Gen09], the first threshold key generation and decryption for LWE based ciphertexts was given by Bendlin and Damgård [BD10]. Their methodology used replicated secret sharing to split the secret key, a method whose complexity scales with $\binom{n}{t}$. The simpler case of full-threshold, i.e. $t = n - 1$, decryption for LWE ciphertexts was combined with SHE and formed the basis of the SPDZ MPC protocol [DPSZ12]. This utilized the BGV encryption scheme, supporting circuits of multiplicative depth one, and used the noise flooding technique mentioned above.

The same techniques were then used in the context of FHE by Asharov et al [AJL+12] in the full threshold setting. To obtain active security in settings with dishonest majority one needs to add zero-knowledge proofs into the mix, see [ABGS22] which gives a practical instantiation using noise flooding for BGV (in the context of a voting application). A similar application of noise flooding for BGV was given in [CLO+13], which considered the threshold setting of $t < n/3$ via Shamir sharing. This enabled active security, without needing to resort to zero-knowledge proofs. In our work we shall adopt the methodology of [CLO+13] for our main threshold decryption protocol.

A generic thresholdizer for arbitrary protocols was given by Boneh et al. in [BGG+18] using threshold-FHE. The construction of Boneh et al. utilizes a special form of secret sharing called $\{0,1\}$-LSSS, which is closely related to replicated sharing.

All of these prior works utilized noise flooding as a methodology. As remarked above this requires a super-polynomial gap between the bound on the noise term e and the ciphertext modulus q. Such super-polynomial blow-ups in other areas of cryptography based on LWE have recently been avoided by utilizing the Renyi divergence [BLR+18]. This, as an approach to threshold-FHE, was recently examined by [BS23] and [CSS+22].

The problem with using the Renyi divergence in the context of distributed decryption is that the general technique of Renyi divergence is hard to apply to security problems which are inherently about distinguishing one distribution from another. In [CSS+22] and [BS23] a way around this was found by designing special security games for threshold-FHE usage, which enabled the use of the Renyi divergence. The problem is that these games need to cope with the homomorphic nature of the underlying encryption scheme, and thus cannot be adaptive. In the applications (such as to MPC) mentioned above we really require a threshold-FHE protocol which is indistinguishable, to an adversary, with a simulation interacting with an ideal functionality. The security games presented in [CSS+22] and [BS23] do not allow such a usage.

Another approach is to apply generic MPC to the problem of threshold decryption of FHE ciphertexts. Here one avoids the noise flooding operation, and one executes the rounding operation inherent in decryption via a generic MPC protocol. This is relatively straight forward to implement using modern LSSS-based MPC systems, however the round complexity is very high. This can be a problem when entities are separated by large distances.

Thus we are led back to considering noise flooding. However, as detailed above, for FHE schemes such as BGV and BFV this is not a problem. The only issue comes with schemes such as TFHE, which utilize small parameters in order to achieve very fast bootstrapping operations. However, perhaps the very fast bootstrapping operation itself can be used to solve the problem?

1.2 Our Contribution

We present a simple method for threshold decryption for TFHE ciphertexts in the presence of $t < n/3$ actively (but statically) corrupted adversarial parties. Our methodology produces a threshold decryption functionality which is in the simulation paradigm, this makes it more amenable to being used as a black box in larger protocols than the game-based approaches based on Renyi divergence.

Our approach works for arbitrary prime power values of q, including the important case of $q = 2^{64}$. Adapting it to the case of non-prime power values of q is immediate via the Chinese-Remainder-Theorem. In doing so we utilize the (relatively standard) trick of applying Shamir secret sharing over Galois rings [ACD+19], thus we do not need to go via a replicated style secret sharing. In Shamir secret sharing the share sizes do not grow exponentially with the value of $\binom{n}{t}$.

When $\binom{n}{t}$ is "small" we apply a trick, which first appeared in [CLO+13], to enable threshold-FHE using a modified Pseudo-Random Secret Sharing (PRSS). In such a situation our protocol is a simple one round protocol, which is robust[3] and works over asynchronous networks when $t < n/3$. When $t < n/2$ we note that we obtain a non-robust protocol, but one which has active-with-abort

[3] i.e. it outputs the correct decryption even in the presence of malicious parties.

security. The proof of security in [CLO+13] has a number of minor bugs/missing details in it, and is overly complex, thus we also re-prove the main threshold-FHE result from this paper. It turns out that using a PRSS for small values of $\binom{n}{t}$, automatically means we are adding a sum of at least two uniform distributions in the flooding term; and thus we can apply our improved statistical distance analysis in this case.

When $\binom{n}{t}$ is large we require slightly more work. In particular we divide our threshold-FHE protocol into two phases, an online and an offline phase. In the offline phase a "generic" MPC protocol is used to generate random shares of bits, which are used to produce two uniformly random noise flooding terms of the correct size. Thus again, we are able to apply our improved statistical distance analysis in this case. In the online phase we consume these random shares of bits to perform the threshold-FHE operation. The online phase is again robust and works over asynchronous networks, when $t < n/3$; and is only active-with-abort secure when $t < n/2$. The security properties of the offline phase are inherited from the underlying MPC protocol used to generate the shares of random bits; if the underlying MPC protocol is robust over asynchronous networks then so is the offline phase of our threshold-FHE protocol; if it only provides active-with-abort security over synchronous networks then they are the properties of our offline phase.

Our methodology for threshold-FHE follows in two conceptually simple steps:

1. We take the input ciphertext with LWE parameters (ℓ, q) and then transform this into a ciphertext with LWE parameters with slightly larger parameters (L, Q) which encrypts the same message, where Q is a prime power with $q|Q$, and with relatively small noise. This switching to larger parameters is performed during a bootstrapping operation, which enables us to simultaneously reduce the noise, so that the noise gap is sufficiently large. We call this operation $\mathsf{Switch}\text{-}n\text{-}\mathsf{Squash}$, as it both switches the (ℓ, q) values, and also squashes the noise.
2. We apply the traditional noise flooding operation, followed by a robust opening procedure on the secret shared value.

In practice the value q will be 2^{64}, and we will only need to boost the modulus to a value of $Q = 2^{128}$ in order to have a sufficient noise gap to perform threshold decryption. With such a value of Q it turns out that TFHE bootstrapping is still efficient, and thus the entire threshold decryption process is efficient. In particular it is very low round (requiring only one round in the online phase), thus it is also preferable to techniques based on generic MPC.

Note, the noise-to-modulus ratio after our $\mathsf{Switch}\text{-}n\text{-}\mathsf{Squash}$ operation is much smaller. This is the key fact which enables our threshold decryption operation to proceed. That such a smaller ratio still maintains security is because the dimension has increased from ℓ to L.

In the special case when $\binom{n}{t}$ is small (say less than 100) we in addition obtain a one-round, threshold decryption protocol which is robustly secure when $t < n/3$, with no offline phase, and which assumes only asynchronous, as opposed to synchronous, networks.

2 Preliminaries

2.1 Notation

Our basic input ciphertexts will come with ciphertext modulus q, and plaintext modulus p. For the underlying bootstrapping keys for TFHE we will utilize a cyclotomic ring of two-power degree N. The ring we define as

$$\mathcal{R} = \mathbb{Z}[X]/(X^N + 1),$$

with the reduction modulo the ciphertext (resp. plaintext) modulus q (resp. p) being given by

$$\mathcal{R}_q = (\mathbb{Z}/q\mathbb{Z})[X]/(X^N + 1) \quad (\text{ resp. } \mathcal{R}_p = (\mathbb{Z}/p\mathbb{Z})[X]/(X^N + 1) \ \ ).$$

We fix the global Δ as $\Delta = \lfloor q/p \rfloor$. This is the ratio between the ciphertext modulus q, and the *application* plaintext modulus p.

Elements in $\mathcal{R}$ (resp. $\mathcal{R}_q$, $\mathcal{R}_p$, etc) will be considered as vectors $\mathbf{A}$, $\mathbf{B}$, etc where we apply the component-wise addition operation. Multiplication, however, is performed with respect to the ring multiplication operation. Normal vectors, i.e. non-ring elements, will be written with lower case boldface, $\mathbf{a}$, $\mathbf{b}$, etc.

We let $\mathbf{a}[i]$ denote the i-th component of the vector $\mathbf{a}$, and $\mathbf{A}[i]$ denote the i-th coefficient of the ring element $\mathbf{A}$ when considered in the polynomial embedding. We assume the underlying ring is obvious from the context.

Multiplication of vectors $\mathbf{a} \cdot \mathbf{b}$ is assumed to be the normal dot-product, which results in a scalar value. We abuse notation by allowing $\mathbf{A} \leftarrow \mathbf{a}$ to denote a ring element is defined from a vector $\mathbf{a}$ of the same size. Thus, if $\mathbf{a} = (a_0, \ldots, a_{N-1})$ then we have

$$\mathbf{A} = a_0 + a_1 \cdot X + \cdots + a_{N-1} \cdot X^{N-1}.$$

2.2 Statistical Distance

Let $U(-B, B)$ denote the uniform distribution on the integer interval $(-B, \ldots, B]$ and $U(-B, B)^m$ be m samples from the respective distribution. Define $\Delta_{SD}(D_1, D_2)$ as the standard statistical distance between two distributions D_1 and D_2 which are defined over a common domain X, i.e.

$$\Delta_{SD}(D_1, D_2) = \frac{1}{2} \sum_{x \in X} |D_1(x) - D_2(x)|.$$

Security for our threshold-FHE protocol when $\binom{n}{t}$ is small will rely on the following Lemmas, all of which are variants of the standard Smudging Lemma (see for example Lemma 2.1 of [AJW11])

Lemma 2.1 (Standard Smudging Lemma). *Let $e \in \mathbb{Z}$ and $B, m \in \mathbb{N}$ denote fixed integers, then we have*

$$\Delta_{SD}\left((e + U(-B, B))^m \ , \ U(-B, B)^m\right) \le \frac{m \cdot |e|}{B},$$

From the data processing inequality, which says that the statistical distance between two distributions cannot increase by applying any (possibly randomized) function to them, one can immediately deduce

Lemma 2.2. *Let $e \in \mathbb{Z}$ and $B, m, v \in \mathbb{N}$ denote fixed integers, then we have*

$$\Delta_{SD}\left((e + \sum_{i=1}^{v} U(-B, B))^m \ , \ \sum_{i=1}^{v} U(-B, B)^m \right) \le \frac{m \cdot |e|}{B},$$

However, a more accurate estimation, when $v \ge 2$, can be given by Lemma 2.4, which follows, via the data processing inequality, from the following Lemma, whose proof is given in Appendix A,

Lemma 2.3. *Let $e \in \mathbb{Z}$ and $B, m \in \mathbb{N}$ denote fixed integers, and let $\mathcal{P} = U(-B, B) + U(-B, B)$. Then*

$$\Delta_{SD}(\mathcal{P}^m, (e + \mathcal{P})^m) \le \frac{m \cdot |e|}{B^2} + \sqrt{m \cdot \frac{|e|^2 \cdot \log B + 2}{2 \cdot (B^2 + B)}}.$$

Lemma 2.4. *Let $e \in \mathbb{Z}$ and $B, m, v \in \mathbb{N}$ denote fixed integers with $v \ge 2$, then we have,*

$$\begin{aligned}\Delta_{SD}\Big((e + \sum_{i=1}^{v} U(-B, B))^m \ , \ \sum_{i=1}^{v} U(-B, B)^m\Big) \\ \le \frac{m \cdot |e|}{B^2} + \sqrt{m \cdot \frac{|e|^2 \cdot \log B + 2}{2 \cdot (B^2 + B)}},\end{aligned}$$

In our application we always utilize $v \ge 2$, in which case we apply Lemma 2.4. When we apply this for m distributed decryption queries we are actually sampling a different value of e per query. On each application, the specific e value used is the output noise term from a bootstrapping operation for a given input ciphertext. Thus the above distances are simplified, upper bounds in our application scenario of the actual statistical distances between the various distributions we analyze.

In our application we will set $B = 2^{\mathsf{stat}} \cdot |e|$, where $\mathsf{stat} = 40$, since Lemma 2.4 tells us that distinguishing the two distributions (for fixed e) requires around

$$\begin{aligned}\frac{B^2}{|e|^2 \cdot \log B} &= \frac{2^{2 \cdot \mathsf{stat}} \cdot |e|^2}{|e|^2 \cdot (\mathsf{stat} + \log |e|)} \\ &= \frac{2^{2 \cdot \mathsf{stat}}}{\mathsf{stat} + \log |e|} \approx 2^{2 \cdot \mathsf{stat}}\end{aligned}$$

samples.

2.3 Learning-With-Errors (LWE)

The (decision) LWE problem is to distinguish between samples drawn from the two distributions

$$\begin{aligned}D_1 &= \{ \ (\mathbf{a}, b) \ : \ \mathbf{a} \leftarrow \mathbb{Z}_q^\ell, \ b \leftarrow \mathbb{Z}_q \ \}, \\ D_2 &= \{ \ (\mathbf{a}, b) \ : \ \mathbf{a} \leftarrow \mathbb{Z}_q^\ell, \ e \leftarrow \mathcal{D}, \ b = \mathbf{a} \cdot \mathbf{s} + e \ \},\end{aligned}$$

where $\mathbf{s} \in \mathbb{Z}_q^\ell$ is a fixed (secret) value, and $\mathcal{D}$ is the LWE-error distribution. In practice $\mathcal{D}$ is usually a discrete form of the Gaussian distribution with "small" standard deviation. For appropriate values of the parameters (q, ℓ) the problem is believed to be hard.

The Ring-LWE problem we define as trying to distinguish the two distributions

$$\begin{aligned}D_1 &= \{ \ (A, B) \ : \ A, B \leftarrow \mathcal{R}_q \ \}, \\ D_2 &= \{ \ (A, B) \ : \ A \leftarrow \mathcal{R}_q, \ E \leftarrow \mathcal{D}_\mathcal{R}, \ B = A \cdot S + E \ \},\end{aligned}$$

where $S \in \mathcal{R}_q$ is a fixed (secret) value, and $\mathcal{D}_\mathcal{R}$ is the Ring-LWE-error distribution on elements of $\mathcal{R}$. We can think of the Ring-LWE problem as being a special version of the LWE problem in which N LWE samples of dimension N are obtained on every iteration.

To enable easier selection of such parameters we approximate the required standard deviation for the distribution $\mathcal{D}$ for given values of q and ℓ, and a given security level. In this work we select an LWE security level of 128, namely distinguishing D_1 from D_2 should require a work effort of 2^{128}. We represent this function of the standard deviation for 128-bit security, as a function of q and ℓ, as the function $\Sigma_{\mathsf{LWE}}(q, \ell)$. This function can be approximated by fitting curves to the output of the LWE-estimator [APS15], when $\mathcal{D}$ is a discrete Gaussian distribution. One should strictly speaking give a separate approximation for each value of q, but it turns out for the two values of q which are important to us, namely $q = 2^{64}$ and $q = 2^{128}$, the same approximation can be used. For $\ell \geq 450$ we have the approximation[4]

$$\begin{aligned} \alpha &= -0.02659946234310527, \\ \beta &= 2.98154318414599, \\ \Sigma_{\mathsf{LWE}}(q, \ell) &= \max(q \cdot 2^{\alpha \cdot \ell + \beta}, 4). \end{aligned}$$

We insert a minimum standard deviation of four into the approximation function to avoid problems when ℓ is very, very large.

2.4 TFHE

Our basic input TFHE ciphertext will be of the form $(\mathbf{a}, b)$ where $\mathbf{a} \in \mathbb{Z}_q^\ell$ and $b \in \mathbb{Z}_q$ such that

$$b = \mathbf{a} \cdot \mathbf{s} + e + \Delta \cdot m$$

for the message $m \in \mathbb{Z}_p$ and a noise value e. We assume the plaintext space $p = 2^{\varrho+1}$, where ϱ is the number of bits of plaintext and we add one bit to enable efficient non-negacylic operations. For our purposes we will not require specific details of the operations on TFHE ciphertexts, however we will require a detailed understanding of the associated noise growths in each operation. For the reader interested in the specific algorithm details we refer to [CLOT21] (also [CGGI20] and [CJP21]) as well as the details of how the following noise formulae are derived.

The operations we will perform (*modulus switch*, *keyswitch* and *bootstrap*) may require additional encryptions of the secret keys with respect to different LWE-style encryption schemes. Thus we have to also keep track of the different types of ciphertexts which each operation is performed on. For our purposes we can focus on just the basic LWE ciphertexts as above, plus a so-called "flattened-GLWE" ciphertext, or F-GLWE, which one can think of as a normal LWE ciphertext but with dimension $w \cdot N$, for the ring-LWE dimension N used in the GLWE ciphertexts and w an associated parameter. GLWE ciphertexts are a generalization of the RLWE ciphertexts introduced above.

For simplicity in this paper we present the noise formulae only for the case of q a power of two. This is the main application area of our work; small changes are needed for other prime power values of q. Note that with q and p both powers of two we have that p exactly divides q, which is what makes the noise formulae slightly easier to describe. We note that all the following operations are deterministic in nature; thus every party executing these operations will produce the same output; this assumes that the parties execute the same Fast Fourier Transform (FFT) algorithms internally to multiply polynomials and are working on identical hardware. This requirement of operating the same FFT algorithm on identical hardware can be relaxed, see Section 5.7 of [BBB+23].

[4] The coefficients α and β were estimated with the commit made on January 5, 2023: https://github.com/malb/lattice-estimator/tree/f9f4b3c69d5be6df2c16243e8b1faa80703f020c

Modulus Switch This operation takes an LWE ciphertext, with ciphertext modulus q, and switches it to an LWE ciphertext with modulus $2 \cdot N$. We present the high-level view of this operation in Figure 2. This algorithm is never *explicitly* called by our algorithms, however it is the first stage of bootstrapping and thus we do need to take into account the noise added by this operation in our analysis. This algorithm will be correct (with probability pr_{MS}) if we have that

$$c_{MS} \cdot \sqrt{\sigma^2 + \sigma_{MS}^2} < \frac{\Delta}{2} \tag{1}$$

where

$$\mathsf{pr}_{MS} = 1 - \mathsf{erfc}\left(\frac{c_{MS}}{\sqrt{2}}\right) = \mathsf{erf}\left(\frac{c_{MS}}{\sqrt{2}}\right)$$

and

$$\sigma_{MS}^2 = \frac{q^2}{48 \cdot N^2} - \frac{1}{12} + \ell \cdot \left(\frac{q^2}{96 \cdot N^2} + \frac{1}{48}\right),$$

where erf (resp. erfc) is the error (resp. complementary error) function. The function $\mathsf{erfc}(x)$ measures the chance of a Gaussian variable with zero mean and variance $\sigma = 0.5$ (or standard normal distribution) to fall outside the bounds $[-x, x]$. We will take $c_{MS} \approx 7.2$ in our analysis, leading to an error probability of $\mathsf{erfc}(7.2/\sqrt{2}) = 2^{-40}$ on homomorphic operations.

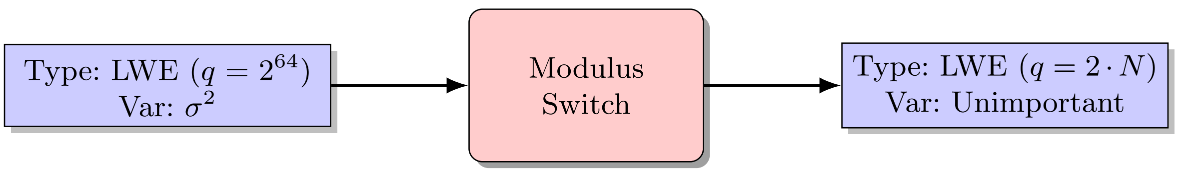

Fig. 2. Modulus Switch

Key Switch The KeySwitch operation takes a F-GLWE ciphertext and returns a normal LWE ciphertext. We require this operation as the bootstrapping operation below produces an F-GLWE ciphertext, and we need to translate it back to a standard LWE ciphertext for further processing by our algorithm. The high-level view of this algorithm is presented in Figure 3 where we have

$$\begin{aligned}
\sigma_{KS}^2 &= w \cdot N \cdot \left(\frac{q^2}{12 \cdot \beta_{ksk}^{2 \cdot \nu_{ksk}}} - \frac{1}{12}\right) \cdot \left(\mathrm{Var}(s_i) + \mathbb{E}^2(s_i)\right) \\
&\quad + \frac{w \cdot N}{4} \cdot \mathrm{Var}(s_i) + w \cdot N \cdot \nu_{ksk} \cdot \sigma_{ksk}^2 \cdot \left(\frac{\beta_{ksk}^2 + 2}{12}\right) \\
&= \frac{w \cdot N}{2} \cdot \left(\frac{q^2}{12 \cdot \beta_{ksk}^{2 \cdot \nu_{ksk}}} - \frac{1}{12}\right) \\
&\quad + w \cdot N \cdot \left(\frac{1}{16} + \nu_{ksk} \cdot \sigma_{ksk}^2 \cdot \left(\frac{\beta_{ksk}^2 + 2}{12}\right)\right) \\
&= w \cdot N \cdot \left(\frac{q^2}{24 \cdot \beta_{ksk}^{2 \cdot \nu_{ksk}}} + \frac{1}{48} + \nu_{ksk} \cdot \sigma_{ksk}^2 \cdot \left(\frac{\beta_{ksk}^2 + 2}{12}\right)\right)
\end{aligned}$$

since for a binary secret key we have $Var[s_i] = 1/4$ and $\mathbb{E}[s_i] = 1/2$. The values ν_{ksk} and β_{ksk} are parameters associated with the key-switching keys, in particular how the decomposition gadget is formed. The value σ_{ksk} is the standard deviation used to generate the noise term in the key-switching keys. The latter is selected such that an LWE problem with dimension ℓ, modulus q and standard deviation for the noise term σ_{ksk} is hard to solve, i.e. $\sigma_{ksk} = \Sigma_{\mathsf{LWE}}(q, \ell)$.

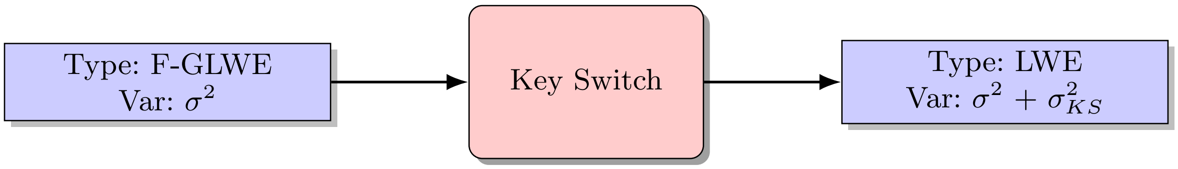

Fig. 3. Key Switch

Bootstrap Bootstrapping takes an LWE ciphertext and outputs a F-GLWE ciphertext but with (potentially) smaller noise, see Figure 4 for a high-level overview. The first thing a bootstrap operation performs is a modulus switch, therefore the input to the bootstrap operation (for it to be correct with a given probability) must satisfy equation (1). The specific details of how a bootstrap is performed is outside the scope of this paper, here we just describe its behavior. The noise output from bootstrap has variance σ_{BR}^2 where

$$\sigma_{BR}^2 = \ell \cdot \Bigg(\nu_{bk} \cdot (w+1) \cdot N \cdot \left(\frac{\beta_{bk}^2 + 2}{12}\right) \cdot \sigma_{bk}^2 + \left(\frac{q^2 - \beta_{bk}^{2\cdot\nu_{bk}}}{24 \cdot \beta_{bk}^{2\cdot\nu_{bk}}}\right) \cdot \left(1 + \frac{w \cdot N}{2}\right) + \frac{w \cdot N}{32} + \frac{1}{16} \cdot \left(1 - \frac{w \cdot N}{2}\right)^2 \Bigg)$$

Again, the values ν_{bk} and β_{bk} are parameters associated with the decomposition gadget associated to the bootstrapping keys, and the value σ_{bk} is the standard deviation used to generate the noise term in the bootstrapping keys. The latter is selected such that an LWE problem with dimension $w \cdot N$, modulus q and standard deviation for the noise term σ_{bk} is hard to solve, $\sigma_{bk} = \Sigma_{\mathsf{LWE}}(q, w \cdot N)$.

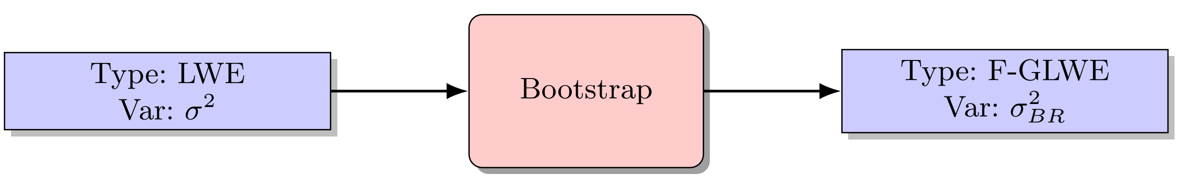

Fig. 4. Bootstrap

Refresh Refresh is the key operation behind our method for threshold-FHE. It is the combination of bootstrap and keyswitch, see Figure 5. We shall refer to this operation by the notation $(\mathbf{a}, b) \leftarrow \mathsf{Refresh}((\mathbf{a}', b'), \mathfrak{pk})$, where $\mathfrak{pk}$ is the public key. As such the operation will be correct (with a given probability) only if the input noise satisfies equation (1), with the output noise being $\sigma_{BR}^2 + \sigma_{KS}^2$.

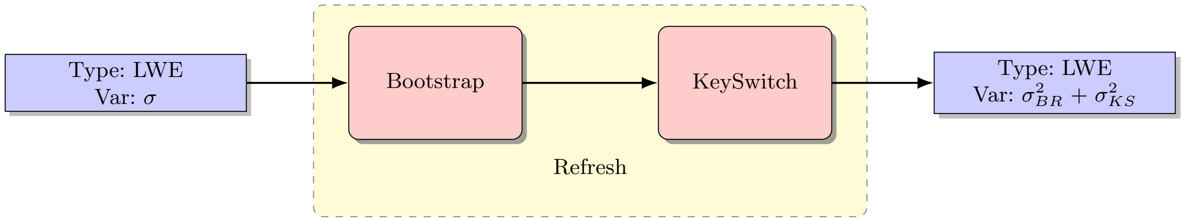

Fig. 5. The Refresh Algorithm

2.5 Secret Sharing

We want to utilize basic Shamir secret sharing over the ring $\mathbb{Z}_Q$ for the larger ciphertext modulus Q. For ease of exposition we will assume that Q is a prime power, with the most challenging case being $Q = 2^K$. To cope with Q being a power of two we need to use Shamir sharing over Galois rings.

Galois Ring Structures The use of Galois rings for Shamir sharing has a long history, going back to (at least) Serge Fehr's masters thesis [Feh93]. For more modern usage see [ACD+19, JSL22]. We first need to fix a Galois ring extension, and write $Q = \mathfrak{p}^K$, where in our case of interest $\mathfrak{p} = 2$. We then define

$$d = \lceil \log_{\mathfrak{p}}(n+1) \rceil$$

this means that the finite field $\mathbb{F}_{\mathfrak{p}^d}$ contains at least $n+1$ values where n is the number of parties. Fix an irreducible polynomial $F(Y)$, of degree d, for this finite field

$$\mathbb{K} = \mathbb{F}_{\mathfrak{p}^d} = \mathbb{F}_{\mathfrak{p}}[Y]/F(Y).$$

Elements in $\mathbb{F}_{\mathfrak{p}^d}$ will be represented by polynomials of degree less than d in a formal root θ of $F(Y)$, i.e. we write $\gamma = c_0 + c_1 \cdot \theta + \cdots + c_{d-1} \cdot \theta^{d-1} \in \mathbb{F}_{\mathfrak{p}^d}$ with $c_i \in \mathbb{F}_{\mathfrak{p}}$. We shall use *the same* polynomial to define the Galois ring extension

$$\mathcal{G} = \mathbb{Z}_Q[\theta] = \mathbb{Z}_Q[Y]/F(Y).$$

We assume that $\mathbb{F}_{\mathfrak{p}^d}$ is embedded into $\mathcal{G}$ in the obvious way, and so can freely talk about elements in $\mathbb{F}_{\mathfrak{p}^d}$ as if they are also in $\mathcal{G}$. Note, in the case where $Q = \mathfrak{p}$ (i.e. Q is a "large" prime) we have that $\mathcal{G} = \mathbb{F}_Q$.

We enumerate the non-zero elements in $\mathbb{F}_{\mathfrak{p}^d}$ as $\{\gamma_1, \ldots, \gamma_{\mathfrak{p}^d-1}\}$, and so for every player $\mathcal{P}_i$ we can refer to "their" element γ_i. Note, in the case where $Q = \mathfrak{p}$ we have $\gamma_i = i$ for $i \in [1, \ldots, n]$. Note that when thinking of the γ_i as elements of $\mathcal{G}$ we have that $\gamma_i - \gamma_j$ is invertible for every distinct pair (i, j). This allows us to define the following polynomials in $\mathcal{G}[X]$, for $i \in \{1, \ldots, n\}$.

$$\delta_i(X) = \prod_{j \neq i} \frac{X - \gamma_j}{\gamma_i - \gamma_j}.$$

Note that

1. $\delta_i(\gamma_i) = 1$.
2. $\delta_i(\gamma_j) = 0$, if $i \neq j$.
3. $\deg \delta_i(X) = n - 1$.

More generally, $\delta_i(X)$ can be defined for any subset of at least $t + 1$ players.

Shamir Sharing over $\mathbb{Z}_Q$ We now define a secret sharing scheme for elements $a \in \mathbb{Z}_Q$, given in Figure 6, which has threshold t out of n players. This means that the scheme perfectly hides a value if at most t parties combine their share, however if $t+1$ parties come together then the share value can be perfectly reconstructed (if no party deliberately introduces an error into their share value). We write $[a]^{\langle t,Q \rangle}$ to denote that a value $a \in \mathbb{Z}_Q$ is secret shared according to the sharing, and we write $[a]_i^{\langle t,Q \rangle} \in \mathcal{G}$ to denote player $\mathcal{P}_i$'s share. Note, that our sharing can also share elements in $\mathcal{G}$ and not just elements in $\mathbb{Z}_Q$, in which case upon opening such an element the opening procedure will abort.

The Secret Sharing Scheme $[x]^{\langle t,Q \rangle}$

$\mathsf{Share}(a)$: Given $a \in \mathbb{Z}_Q$ this produces a sharing, i.e. values $[a]_i^{\langle t,Q \rangle} \in \mathcal{G}$
1. Generate a polynomial $g_a(X) \in \mathcal{G}[X]$ of degree at most t such that $g_a(0) = a$.
2. Define $[a]_i^{\langle t,Q \rangle} = g_a(\gamma_i)$.

$\mathsf{Open}([a]_1^{\langle t,Q \rangle}, \ldots, [a]_n^{\langle t,Q \rangle})$:
1. Compute the polynomial
$$g_a(X) \leftarrow \sum_i [a]_i^{\langle t,Q \rangle} \cdot \delta_i(X).$$
2. If $\deg g_a(X) > t$ then abort.
3. If $g_a(0) \notin \mathbb{Z}_Q$ then abort.
4. Return $g_a(0)$.

Fig. 6. The Secret Sharing Scheme $[x]^{\langle t,Q \rangle}$.

Notice, that the opening algorithm, given in Figure 6, will abort if any of the n-parties send in a share value which is inconsistent. In addition it will abort if the shared value is not in $\mathbb{Z}_Q$, but in $\mathcal{G} \setminus \mathbb{Z}_Q$.

The secret sharing scheme is linear, namely given secret sharings $[a]^{\langle t,Q \rangle}$ and $[b]^{\langle t,Q \rangle}$ we can produce a secret sharing of the value $\alpha \cdot a + \beta \cdot b + \gamma$ for any values $\alpha, \beta, \gamma \in \mathbb{Z}_Q$ with no interaction. This is done by each party $\mathcal{P}_i$ computing

$$[\alpha \cdot a + \beta \cdot b + \gamma]_i^{\langle t,Q \rangle} \leftarrow \alpha \cdot [a]_i^{\langle t,Q \rangle} + \beta \cdot [b]_i^{\langle t,Q \rangle} + \gamma.$$

We shall write this as global operation in the notation

$$[\alpha \cdot a + \beta \cdot b + \gamma]^{\langle t,Q \rangle} \leftarrow \alpha \cdot [a]^{\langle t,Q \rangle} + \beta \cdot [b]^{\langle t,Q \rangle} + \gamma.$$

Error Correction Over Galois Rings In this section we explain how to do Reed-Solomon error correction over the Galois ring $\mathcal{G}$ when $t < n/3$. The methodology is taken from [ACD$^+$19, Figure 1].

The standard Berlekamp–Welch or Gao algorithms for error correcting Reed-Solomon codes over $\mathbb{K}$ take as input $(x_1, \ldots, x_n)$ where $x_i \in \mathbb{K}$. We denote this by $\mathsf{RS\text{-}Decode}_{\mathbb{K}}(x_1, \ldots, x_n)$. It is assumed on input that $x_i = f(\gamma_i)$, for all except at most t values, and for a polynomial $f \in \mathbb{F}_{\mathfrak{p}^d}[X]$ of degree at most t. The "error" values x_i can either be incorrect values x_i or the $\perp$ symbol. One could think of $\perp$ as zero, but sometimes in decoding algorithms it is faster to keep data around which we know to be a definite error. The output of $\mathsf{RS\text{-}Decode}_{\mathbb{K}}(x_1, \ldots, x_n)$ is the polynomial $f(X)$.

The Berlekamp–Welch and Gao algorithms can take an additional parameter r which specifies the maximum expected number of errors, with the algorithm returning $\perp$ if more than r errors are detected. In this context we write $\mathsf{RS\text{-}Decode}^r_{\mathbb{K}}(x_1, \ldots, x_n)$, with $r = \perp$ denoting the usual operation of no assumption on the errors.

In our Galois ring we have a similar decoding problem but now we have $x_i = f(\gamma_i)$, where f is a polynomial in $\mathcal{G}[X]$ of degree at most t, and the γ_i have been (trivially) lifted from $\mathbb{F}_{\mathfrak{p}^d}$ to $\mathcal{G}$. Note that every element $\alpha \in \mathcal{G}$ can be written as

$$\alpha = a_0 + a_1 \cdot \mathfrak{p} + \cdots + a_{K-1} \cdot \mathfrak{p}^{K-1}$$

where $a_i \in \mathbb{F}_{\mathfrak{p}^d}$. We will write the polynomial f in a similar manner as

$$f(X) = f_0(X) + f_1(X) \cdot \mathfrak{p} + \cdots + f_{K-1}(X) \cdot \mathfrak{p}^{K-1}$$

and we will recover the f_i values recursively using the standard algorithm $\mathsf{RS\text{-}Decode}_{\mathbb{K}}(x_1, \ldots, x_n)$ as a subroutine. This is explained in Figure 7, where $\pi : \mathcal{G} \longrightarrow \mathbb{K}$ denotes the reduction modulo p map. We adopt the convention that passing $\perp$ into π then the output is also $\perp$, and that any arithmetic operation on $\perp$ results in $\perp$.

Robust Opening We can now define an opening procedure called $\mathsf{RobustOpen}$, in Figure 8, which will robustly open the shared value, depending on the relationship between t and n, and the underlying network properties. We present the robust opening protocol for Shamir sharing of arbitrary degree d. When $d = t$ robust opening is only available when $t < n/3$. Note, that for asynchronous networks and $t = d < n/4$ we can execute less computational steps than for the case $t = d < n/3$ by simply waiting for more data to arrive. The method for asynchronous networks and $t = d < n/3$ is called "online error correction", and was first presented in [BCG93].

Assuming the input sharing is of an element in $\mathbb{Z}_Q$ then robust open protocol in Figure 8 will output the value in $\mathbb{Z}_Q$, even if adversarial parties introduce errors. This is despite the shares themselves, and the Lagrange interpolation coefficients, being defined by elements in $\mathcal{G}$.

Decoding in Galois Rings

$\mathsf{RS\text{-}Decode}^r_{\mathcal{G}}(x_1, \ldots, x_n)$.

1. $\mathbf{x} \leftarrow (x_1, \ldots, x_n)$.
2. $\mathbf{y} \leftarrow \mathbf{x}$.
3. For $i = 0, \ldots, K-1$ do
   (a) $\mathbf{z} \leftarrow \pi(\mathbf{y}/\mathfrak{p}^i)$.
   (b) $f_i(X) \leftarrow \mathsf{RS\text{-}Decode}^r_{\mathbb{K}}(\mathbf{z})$.
   (c) If $f_i(X) = \perp$ then return $\perp$. In this case either there are more than t errors, if $r = \perp$, or there are more than r errors, if $r \neq \perp$.
   (d) For $j = 1, \ldots, n$ set $t_j \leftarrow \sum_{l=0}^{i} f_l(\gamma_j) \cdot \mathfrak{p}^l \in \mathcal{G}$.
   (e) $\mathbf{y} \leftarrow \mathbf{x} - \mathbf{t}$.
   (f) If y_j is not divisible by $\mathfrak{p}^{i+1}$ then $y_j \leftarrow \perp$.
4. Output $\sum_{l=0}^{K-1} f_l(X) \cdot \mathfrak{p}^l \in \mathcal{G}[X]$.

Fig. 7. Reed-Solomon error decoding in Galois rings

RobustOpen

This protocol depends on the ratio between d, t, and n, and whether the underlying network is synchronous or asynchronous. It is run either by a player $\mathcal{P} = \mathcal{P}_i$ who already holds their share $[a]_i^{\langle d,Q \rangle}$ (we call this Case A), or by an external player $\mathcal{P}$ (which we call Case B).

$\mathsf{RobustOpen}(\mathcal{P}, [a]^{\langle d,Q \rangle})$:

1. Player $\mathcal{P}_i$ sends $[a]_i^{\langle d,Q \rangle}$ securely to player $\mathcal{P}$.
2. If $d + 3 \cdot t < n$ and the network is asynchronous, or $d + 2 \cdot t < n$ and the network is synchronous
   (a) $\mathcal{P}$ waits until they have received $d + 2 \cdot t$ (case A) or $d + 2 \cdot t + 1$ (case B) share values $\{[a]_j^{\langle d,Q \rangle}\}_j$.
   (b) Apply the Reed-Solomon decoding algorithm, $\mathsf{RS\text{-}Decode}^t_{\mathcal{G}}(\cdots)$, to the $d + 2 \cdot t + 1$ shares they hold to robustly compute $F(X)$,
   (c) Return $a = F(0)$.
3. If $d + 2 \cdot t < n$ and the network is asynchronous
   (a) For $r = 0, \ldots, t$ do
      i. $\mathcal{P}$ waits until $d + t + r$ (case A) / $d + t + r + 1$ (case B) shares have been received.
      ii. Apply the Reed–Solomon decoding algorithm, $\mathsf{RS\text{-}Decode}^r_{\mathcal{G}}(\cdots)$, on the $d + t + r + 1$ shares, assuming there are r errors in these shares.
      iii. If error correction outputs a degree d degree poly then, if there are at least $d+t+1$ shares (out of the $d + t + r + 1$ shares) which lie on the polynomial, then this is the correct polynomial so output the constant term and exit the loop. Note, this step requires just scanning the $d + t + r + 1$ shares and counting how may lie on the polynomial.

$\mathsf{RobustOpen}(\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}, [a]^{\langle d,Q \rangle})$: This is a short hand for all $i \in \{1, \ldots, n\}$ players executing $\mathsf{RobustOpen}(\mathcal{P}_i, [a]^{\langle d,Q \rangle})$ in parallel. Hence, all n players will obtain the value a.

Fig. 8. Robust Opening Protocol when $d + 2 \cdot t < n$

3 The Switch-n-Squash Operation

The first step in our threshold decryption operation is to take an LWE ciphertext $(\mathbf{a}, b)$ defined with parameters (q, ℓ), with respect to a secret key $\mathbf{s} \in \{0,1\}^\ell$, and with noise variance σ^2. Then we switch it to a ciphertext $(\mathbf{a}', b')$ defined for parameters (Q, L) with $q|Q$, $L > \ell$, and for a secret key $\mathbf{s}' \in \{0,1\}^L$, and with a new noise variance σ'^2_{BR}, for a suitably small noise variance. Thus we increase both the ciphertext modulus and the LWE dimension, but we also increase the noise gap.

We perform this switch by performing a bootstrapping operation which outputs a ciphertext with an LWE dimension $L = \omega \cdot N$ and ciphertext modulus Q. Indeed, one can see the entire method a just bootstrapping, with specially designed bootstrapping keys in order to result in a ciphertext with output parameters (L, Q). A summary of the procedure is given in figure 9.

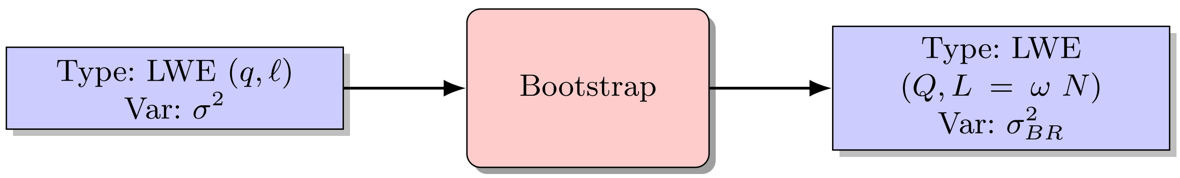

Fig. 9. The Switch-n-Squash Method

The reason for moving a ciphertext from parameter set (q, ℓ) to (Q, L) is to enable us to have a lot more room between the noise bound and the value of $\Delta' = \frac{Q}{p}$. In particular the noise-gap should be big enough to enable noise flooding for threshold decryption. Thus, we need to select large enough cryptographic parameters to enable this refresh operation to output a suitably small noise value.

If our input ciphertext with parameter set (q, ℓ) has noise variance σ^2, then, after the modulus switch inside the bootstrap, we obtain a ciphertext with parameter set $(2 \cdot N, \ell)$ with noise variance

$$\sigma'^2 = \sigma^2 + \sigma^2_{MS}$$

To guarantee correctness up to a probability of failure pr_{MS}, we need the condition in equation (1) to be met. After the bootstrapping, we end up with a ciphertext with parameter set (Q, L) with noise variance σ'^2_{BR}, with σ'_{BR} a function of L, Q, N', w', σ'_{bk} etc as described earlier in the case of (q, l). We use N', w' etc to differentiate these values from the "normal" values used in standard FHE operations.

In the next section we will require the following equations to be satisfied, for some integer parameter pow. The parameter pow denotes the extra factor of noise we will add during flooding, i.e. it is approximately $\log_2 |E/e|$. We make it slightly larger than stat (by an extra additive term of $\log_2 100$) in order to cope with a non-uniform value of E which will be used in our procedure when $\binom{n}{t}$ is small (see later for a further discussion of this case).

$$\begin{aligned} \mathsf{Bd} &= c_{Dec} \cdot \sigma'_{BR}, \\ 2^{\mathsf{pow}} \cdot \mathsf{Bd} &\leq \frac{\Delta'}{2}, \\ \mathsf{pow} &\geq \mathsf{stat} + \log_2 100, \end{aligned}$$

where $c_{Dec} \approx 7.2$. Hence, combining these all together we have that

$$\mathsf{stat} + \log_2 100 \leq \mathsf{pow} \leq \log_2 \left(\frac{\Delta'}{2} \right) - \log_2 \left(c_{Dec} \cdot \sigma'_{BR} \right).$$

In particular this means that we must have

$$\log_2 \left(c_{Dec} \cdot \sigma'_{BR} \right) \leq \log_2 \left(\frac{\Delta'}{2} \right) - \mathsf{stat} - \log_2 100. \tag{2}$$

Given $\mathsf{stat} \approx 40$, we thus need to select parameters so that the noise after bootstrapping for these large parameters is at least stat bits smaller than the decryption correctness bound of $\Delta'/2$.

To find cryptographic parameters that guarantee the correctness, the efficiency and the security, we used the optimization method introduced in [BBB+23]. In a nutshell, it consists into solving the following optimization problem. We aim to minimize the function $\left(\mathsf{Cost}\,(BS)\right)$ which is a surrogate of the execution time of the bootstrapping as defined in [BBB+23], subject to the two constraints

$$c_{MS} \cdot \sqrt{\sigma^2 + \sigma_{MS}^2} < \frac{\Delta}{2},$$
$$\log_2\left(c_{Dec} \cdot \sigma'_{BR}\right) \le \log_2\left(\frac{\Delta'}{2}\right) - \mathsf{stat} - \log_2 100,$$

where σ^2 is the variance of the input ciphertext, $\Delta = \frac{q}{p}$, $\Delta' = \frac{Q}{p}$, and $c_{MS} = c_{Dec} \approx 7.2$.

A summary of four potential parameter sets are given in Table 1. We give four sets of parameters; two for each plaintext size of $\varrho = 1$ and $\varrho = 4$, and for each plaintext size we give a variant with ℓ a non-power of two and ℓ a power of two. The former for use with the "traditional" methodology of giving out many encryptions of zero, and the latter for use with the more compact public key encryption methodology given in [Joy23].

Table 1. Parameters for switching up operations with the four sets of basic parameters.

	$\varrho = 1$	$\varrho = 4$	$\varrho = 1$	$\varrho = 4$
(q, ℓ)	$(2^{64}, 777)$	$(2^{64}, 870)$	$(2^{64}, 1024)$	$(2^{64}, 1024)$
(Q, L)	$(2^{128}, 4096)$	$(2^{128}, 4096)$	$(2^{128}, 4096)$	$(2^{128}, 4096)$
pow	47	47	47	47
N'	1024	2048	1024	2048
w'	4	2	4	2
β'_{bk}	2^{32}	2^{32}	2^{32}	2^{32}
ν'_{bk}	2	2	2	2
$\log_2 \sigma'_{bk}$	22.0	22.0	22.0	22.0
$\log_2 \sigma'_{BR}$	72.0	72.1	72.2	72.2

We see that the standard deviation of the output noise after bootstrapping is around 2^{72}, which is gives us around 50 bits of noise gap for a ciphertext modulus of 2^{128}. Which is enough to fit in our flooding by a value of approximately $72 + 40 = 112$ bits. Note that for the input ciphertext, with parameters (q, ℓ), the noise gap is with overwhelming probability much smaller than 2^{50}, indeed it is less than 2^{10}.

4 Threshold Decryption Operation

After applying the methods from the previous sections we now have a ciphertext

$$(\mathbf{a}, b) = (\mathbf{a}, \mathbf{a} \cdot \mathbf{s}' + e + \Delta' \cdot m)$$

where $\mathbf{a} \in \mathbb{Z}_Q^L$, $\mathbf{s}' \in \{0,1\}^L$, the message m lies in $\mathbb{Z}_p$, and $\Delta' = Q/p$, and e is a noise term. The noise term is assumed to have variance ${\sigma'_{BR}}^2$, i.e. the LWE ciphertext instance is an output of the

Refresh operation from the previous section. In what follows we shall assume $|e| \leq \mathsf{Bd}$, where we assume (with overwhelming probability) that

$$\mathsf{Bd} = c_{Dec} \cdot \sigma'_{BR},$$

where $c_{Dec} \approx 7.2$. To fix ideas think of $Q = 2^{128}$ and the variance being of size roughly 2^{140}, and so $\mathsf{Bd} \approx 2^{70}$. Thus we have a noise gap of around 50 bits (assuming a plaintext space of at most 10 bits).

We assume the secret key $\mathbf{s}'$ has been secret shared with respect to our secret sharing scheme, i.e. we have a sharing $[\mathbf{s}']^{\langle t,Q\rangle}$. Formally we define the threshold decryption for the parameters (Q, L) via two ideal functionalities. The first $\mathcal{F}_{\mathsf{KeyGen}}$, in Figure 10, acts as a set-up assumption for our protocol, needed for the UC proof we provide. It generates a key pair, and secret shares the secret key among the players using the secret sharing scheme. One can realize this functionality using a generic MPC protocol, see Appendix D for an outline. Note, despite wanting active security we do not "complete" adversarially input shares into a complete sharing (as is often done in such situations), as the implementing actively protocol for $\mathcal{F}_{\mathsf{KeyGen}}$ does not actually need to do this.

$\mathcal{F}_{\mathsf{KeyGen}}$

Init():
1. Execute $(\mathfrak{pk}, \mathbf{s}') \leftarrow \mathsf{KeyGen}(1^\kappa)$ for the underlying TFHE encryption scheme with parameters (Q, L).
2. Generate a secret sharing $[\mathbf{s}']^{\langle t,Q\rangle}$ of the secret key.
3. Send $\mathfrak{pk}$ to all players (including the adversary), and send $[\mathbf{s}']_i^{\langle t,Q\rangle}$ to player $\mathcal{P}_i$ (including adversarially controlled players).

Fig. 10. The ideal functionality for distributed key generation

The key functionality we want to implement is $\mathcal{F}_{\mathsf{KeyGenDec}}$ given in Figure 11. Note, that this functionality always returns the correct result, irrespective of what the adversary does.

$\mathcal{F}_{\mathsf{KeyGenDec}}$

Init():
1. Execute $(\mathfrak{pk}, \mathbf{s}') \leftarrow \mathsf{KeyGen}(1^\kappa)$ for the underlying TFHE encryption scheme with parameters (Q, L).
2. Send $\mathfrak{pk}$ to all players, including the adversary and store the value $\mathbf{s}'$.

DistDecrypt($\mathfrak{ct}, \mathcal{U}$): For a ciphertext $\mathfrak{ct}$ with error e such that $|e| < \mathsf{Bd}$.
1. Compute $m \leftarrow \mathsf{Dec}(\mathfrak{ct}, \mathbf{s}')$.
2. If $\mathcal{U}$ is adversarially controlled then send $(\mathfrak{ct}, m)$ to the adversary.
3. Otherwise send m to player $\mathcal{U}$ and $\mathfrak{ct}$ to the adversary.

Fig. 11. The ideal functionality for distributed key generation and decryption

Our threshold decryption protocol comes in two flavours, one where $\binom{n}{t}$ is "small" and one where $\binom{n}{t}$ is "large"[5]. When $\binom{n}{t}$ is small our threshold decryption protocol requires only one round of interaction, whilst when $\binom{n}{t}$ is large the online phase of our threshold decryption still requires only one round, however there is a (slightly) complex, ciphertext independent, offline phase which needs to be completed first.

In both cases we assume $t < n/3$, as we wish to have a robust asynchronous threshold decryption protocol; at least in the online phase of our protocol. We also recall we require that the protocol's security should come via a simulation, as opposed to a game based argument. This is to enable composition of the threshold decryption protocol easily within other larger protocols.

4.1 Threshold Decryption for "Small" $\binom{n}{t}$

We start with the case of $\binom{n}{t}$ being small, where we can utilize a variant of the standard Pseudo-Random Secret Sharing (PRSS), originally introduced in [CDI05]. The problem is that the complexity of a PRSS depends on $\binom{n}{t}$, which can become exponentially big as n increases. Thus this method can only be used when $\binom{n}{t}$ is small. We use a slightly modified form of PRSS in that we do not output sharings of uniformly random values from $\mathbb{Z}_Q$, but from a different range. This form of PRSS was originally used in [CLO+13], for exactly the purpose of threshold-FHE.

The algorithms for a non-interactive PRSS are defined in Figure 12. The algorithm $\mathsf{PRSS.Init}()$ iterates over all sets A of size $n-t$. Thus the complexity of $\mathsf{PRSS}, \mathsf{Init}()$, i.e. the number of sets A we need to deal with, depends on $\binom{n}{t}$, which can become very large for large n and t.

The PRSS makes use of a PRF ψ of the form

$$\psi : \begin{cases} \{0,1\}^{\mathsf{sec}} \times S \longrightarrow \quad \mathbb{Z} \\ \quad (\kappa, \mathsf{cnt}) \quad \longmapsto \psi(\kappa, \mathsf{cnt}) \end{cases}$$

where $\{0,1\}^{\mathsf{sec}}$ is the keyspace and S is a set of counters. The output of the function ψ is assumed to be bounded in absolute value by

$$\mathsf{Bd}_1 = \frac{(2^{\mathsf{pow}} - 1) \cdot \mathsf{Bd}}{\binom{n}{t}},$$

recall that pow is roughly speaking $\log_2 |E/e|$.

One can implement ψ using AES in an obvious counter mode, e.g. as $\log_2 \mathsf{Bd}_1 < 256$ one can set

$$\psi(\kappa, \mathsf{cnt}) = \Big(\mathsf{AES}_\kappa(0\|\mathsf{cnt}) + 2^{128} \cdot \mathsf{AES}_\kappa(1\|\mathsf{cnt})\Big) \pmod{\mathsf{Bd}_1},$$

where we treat the output block of the AES cipher as an integer in $[0, \ldots, 2^{128} - 1]$. Note, the output of ψ is only statistically uniform in the required range here if $\log_2 \mathsf{Bd}_1 < 256 - \mathsf{stat}$, which will be true in our usage. Since the output of ψ is bounded as above, we have that the value E is bound by $(2^{\mathsf{pow}} - 1) \cdot \mathsf{Bd}$, as the sum used in the PRSS has at most $\binom{n}{t}$ terms. Note, the shared value which is output by the PRSS invocation is the sharing of the value

$$E \leftarrow \sum_A \psi(r_A, \mathsf{cnt}).$$

[5] Think of the small/large regime being divided at a value such as 100

PRSS

PRSS.Init(): For every set $A \subseteq \{1, \ldots, n\}$ of size $n - t$:
1. $S \leftarrow \{\mathcal{P}_i\}_{i \in A}$.
2. Players $\mathcal{P}_i$ with $i \in A$ execute $r_A \leftarrow \mathsf{AgreeRandom}(S, \mathsf{sec})$, from Appendix B.
3. Define $f_A(X) \in \mathbb{Z}_q[X] = \mathbb{Z}_{2^k}[X]$ to be the polynomial of degree t such that $f_A(0) = 1$ and $f_A(\gamma_i) = 0$ for all $i \notin A$. Each party $\mathcal{P}_i$ only needs store $f_A(\gamma_i)$ though.
4. $\mathsf{cnt}_{\mathsf{PRSS}} \leftarrow 0$.

PRSS.Next():
1. Party $\mathcal{P}_i$ computes, where the sum is over every set A containing i,
$$[E]_i^{\langle t,Q \rangle} \leftarrow \sum_{A : i \in A} \psi(r_A, \mathsf{cnt}_{\mathsf{PRSS}}) \cdot f_A(\gamma_i).$$
2. $\mathsf{cnt}_{\mathsf{PRSS}} \leftarrow \mathsf{cnt}_{\mathsf{PRSS}} + 1$.
3. Return $[E]^{\langle t,Q \rangle}$.

Fig. 12. Pseudo-Random Secret Sharing PRSS

Given this PRSS we can define our threshold decryption protocol, which we give in Figure 13, where we assume a dedicated player $\mathcal{U}$ (possibly not one of the threshold decryption parties) will receive the final output. If all threshold decryption parties are to receive the output of the threshold decryption, or the output is to be public and not just to player $\mathcal{U}$, then the communication in step 3 does not need to be done securely.

Threshold Decryption - Protocol 1

Init():
1. The parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ execute PRSS.Init().
2. The parties obtain $[\mathbf{s}']^{\langle t,Q \rangle}$ via a threshold key generation protocol, see Appendix D.

DistDecrypt($\mathfrak{ct}$, $[\mathbf{s}']^{\langle t,Q \rangle}$, $\mathcal{U}$): On input of $\mathfrak{ct} = (\mathbf{a}, b) \in \mathbb{Z}_Q^{L+1}$ this executes the following steps:
1. The parties $\mathcal{P}_i$ execute $[E]^{\langle t,Q \rangle} \leftarrow \mathsf{PRSS.Next}()$.
2. The parties $\mathcal{P}_i$ compute $[v]^{\langle t,Q \rangle} \leftarrow b - \mathbf{a} \cdot [\mathbf{s}']^{\langle t,Q \rangle} + [E]^{\langle t,Q \rangle}$.
3. Party $\mathcal{P}_i$ sends the value $[v]_i^{\langle t,Q \rangle}$ *securely* to the player $\mathcal{U}$.
4. Player $\mathcal{U}$ applies algorithm RobustOpen to robustly reconstruct the value $b - \mathbf{a} \cdot \mathbf{s}' + E$, and hence m.

Fig. 13. Threshold Decryption - Protocol 1

For correctness we require that
$$2^{\mathsf{pow}} \cdot \mathsf{Bd} \leq \frac{\Delta'}{2},$$
since then the PRSS addition will not effect the correctness of the final result as $E + \mathsf{Bd} \leq (2^{\mathsf{pow}} - 1) \cdot \mathsf{Bd} + \mathsf{Bd} = 2^{\mathsf{pow}} \cdot \mathsf{Bd} < \Delta'/2$.

On the other hand (see below) for security we require that
$$\mathsf{pow} \geq \mathsf{stat} + \log_2 \binom{n}{t},$$

where stat is the security parameter related to statistical distance. Thus this method is only applicable when we have a large Δ' in comparison to the noise bound Bd. This is why we needed to boost the ciphertext from one with parameters (q, ℓ) to one with parameters (Q, L) in the previous sections. Since we are assuming the small regime for $\binom{n}{t}$ is when $\binom{n}{t} \leq 100$, and we used the inequality

$$\mathsf{pow} \geq \mathsf{stat} + \log_2 100$$

in the previous section to derive the bounds on the noise after refreshing to ensure that

$$2^{\mathsf{pow}} \cdot \mathsf{Bd} \leq \frac{\Delta'}{2}$$

we are assured that the conditions of the following theorem are satisfied for our refresh parameters.

Simulator Threshold Decryption

On input of

1. A ciphertext $\mathfrak{ct} = (\mathbf{a}, b)$ and a public key $\mathfrak{pk}$.
2. The underlying message m encrypted by $\mathfrak{ct}$.
3. A set of adversarial parties I with $|I| \leq t$.
4. The share values $[\mathbf{s}']_i^{\langle t,Q\rangle}$ for $i \in I$.
5. The PRSS secret keys r_A for all sets A such that $A \cap I \neq \emptyset$.

this algorithm outputs the simulated shares $\{[v]_j^{\langle t,Q\rangle}\}_{j \notin I}$.

$\mathsf{Sim} - \mathsf{DistDecrypt}$:

1. The simulator computes, for $i \in I$,
$$[\hat{v}]_i^{\langle t,Q\rangle} = b - \mathbf{a} \cdot [\mathbf{s}']_i^{\langle t,Q\rangle} + \sum_{A: i \in A} \psi(r_A, \mathsf{cnt}_{\mathsf{PRSS}}) \cdot f_A(\gamma_i).$$
2. The simulator computes
$$E' = \sum_{A: A \cap I \neq \emptyset} \psi(r_A, \mathsf{cnt}_{\mathsf{PRSS}}) + \sum_{B: B \cap I = \emptyset} r_B$$
where r_B is chosen uniformly at random so that $|r_B| \leq \mathsf{Bd}_1$.
3. The simulator computes $v = \Delta' \cdot m + E'$.
4. The simulator generates the decryption shares $\{[\hat{v}]_j^{\langle t,Q\rangle}\}_{j \notin I}$ via Lagrange interpolation (and possibly generating random shares if $|I| < t$) from v and the values $\{[\hat{v}]_i^{\langle t,Q\rangle}\}_{i \in I}$.
5. The simulator outputs $\{[v]_j^{\langle t,Q\rangle}\}_{j \notin I}$.

Fig. 14. Simulator for $\mathsf{DistDecrypt}(\mathfrak{ct}, [\mathbf{s}']^{\langle t,Q\rangle}, \mathcal{U})$

Theorem 4.1. *Assuming*

$$\mathsf{pow} \geq \mathsf{stat} + \log_2 \binom{n}{t},$$

in the $\mathcal{F}_{\mathsf{KeyGen}}$-hybrid model the protocol in Figure 13 implements $\mathcal{F}_{\mathsf{KeyGenDec}}$ with statistical security against any static active adversary corrupting I parties, with $|I| \leq t$, making at most $2^{2 \cdot \mathsf{stat}}$ threshold decryption queries.

Assuming

$$2^{\mathsf{pow}} \cdot \mathsf{Bd} \leq \frac{\Delta'}{2},$$

the protocol is correct.

Proof. Correctness follows, even in the presence of $t < n/3$ fully malicious parties, on noticing that the bounds on the noise, described above, imply that the value v does encode the original message correctly when it is robustly opened.

Security of the protocol follows by showing that the output of simulator in Figure 14 is statistically indistinguishable, from the output of an adversary controlling I parties, with $|I| \leq t$, in a real execution of the protocol.

The proof of this security claim follows essentially the argument in Section 7.5 of the full version of [CLO+13]; where we have to switch from a BGV style of looking at ciphertexts to one of BFV. However, the proof in [CLO+13] is overly complex and has a few minor bugs, which we correct here.

First note, the values $\{[v]_j^{\langle t,Q \rangle}\}_{j \in I}$ produced by the simulator are the true decryption share values which the adversary should broadcast (even if he does not) if they acted honestly. The bounds on the noise described above then imply that the value v does encode the original message correctly. This means that the Lagrange interpolation in the simulation will recover the shares for the honest players $\{[v]_j^{\langle t,Q \rangle}\}_{j \notin I}$ as required.

Now, let e denote the value of $b - \mathbf{a} \cdot \mathbf{s'} - \Delta' \cdot m \pmod{Q}$.

In a real execution of the protocol the shares output by the honest players are consistent and are enough to allow the honest parties to decrypt correctly, since $t < n/3$. The simulation has exactly the same properties.

The value E is the value output by the PRSS in the real execution of the protocol, and the value E' is the value simulated for the PRSS in the simulated protocol.

In the real protocol the adversary sees the value

$$\Delta' \cdot m + e + E$$

whereas in the simulated protocol he sees the value

$$\Delta' \cdot m + E'.$$

By the security of the PRSS the value of $e + E$ and $e + E'$ are indistinguishable. Thus we only need to show that $e + E'$ and E' are indistinguishable.

However, by Lemmas 2.4 and 2.3 (applied with $v = \binom{n}{t} \geq n$, $B = \mathsf{Bd}_1$, and $|e| = \mathsf{Bd}$) we have that, when executing at most d distributed decryption operations,

$$\begin{aligned}
\Delta_{SD}\Big((e + \sum_{i=1}^{v} U(-B,B))^d\ ,\ \sum_{i=1}^{v} U(-B,B)^d\Big) &\leq \frac{d \cdot \mathsf{Bd}}{\mathsf{Bd}_1^2} + \sqrt{d \cdot \frac{\mathsf{Bd}^2 \cdot \log \mathsf{Bd}_1 + 2}{2 \cdot (\mathsf{Bd}_1^2 + \mathsf{Bd}_1)}} \\
&\approx \frac{d \cdot v^2 \cdot \mathsf{Bd}}{2^{2 \cdot \mathsf{pow}} \cdot \mathsf{Bd}^2} + \sqrt{d \cdot \frac{v^2 \cdot \mathsf{Bd}^2 \cdot \log (2^{\mathsf{pow}} \cdot \mathsf{Bd}) + 2}{2^{2 \cdot \mathsf{pow}+1} \cdot \mathsf{Bd}^2}} \\
&\approx \frac{d \cdot v^2}{2^{2 \cdot \mathsf{pow}} \cdot \mathsf{Bd}} + \sqrt{d \cdot \frac{v^2 \cdot \log (2^{\mathsf{pow}} \cdot \mathsf{Bd}) + 2}{2^{2 \cdot \mathsf{pow}+1}}}
\end{aligned}$$

$$\approx \frac{v}{2^{\mathsf{pow}}} \cdot \sqrt{d \cdot (\mathsf{pow} + \log \mathsf{Bd})}$$
$$\leq c \cdot \sqrt{d} \cdot 2^{-\mathsf{stat}},$$

for some relatively 'small' constant c. Lemma 2.4/2.3 applies, since the number of uniform random variables $U(-B, B)$ added by the honest players is lower bounded by $n - t \geq 2$. Thus to distinguish the two distributions the adversary would need to sample $d > 2^{2 \cdot stat}$ operations. □

4.2 Threshold Decryption for "Large" $\binom{n}{t}$

When $\binom{n}{t}$ is large we can no longer relay on a non-interactive PRSS. We can also not rely on "standard" interactive PRSS's, as our PRSS was used to create a small-ish element above and not a uniformly random one. Thus when $\binom{n}{t}$ is large we generate the masking value $[E]^{\langle t,Q\rangle}$ above, as a sum of two uniformly random values, using random bits provided by an "offline" phase. This offline phase is abstracted in the ideal functionality $\mathcal{F}_{\mathsf{Offline}}$ in Figure 15. We discuss how this can be implemented in Appendix C.

The Functionality $\mathcal{F}_{\mathsf{Offline}}$

We describe this functionality as a robust ideal functionality, the modifications to make a functionality which is only secure in an active-with-abort setting are easily made.

$\mathcal{F}_{\mathsf{Offline}}.\mathsf{Bits}(b)$:
1. The functionality samples uniformly random bits $b_i \in \{0, 1\}$ for $i = 1, \ldots, b$.
2. The functionality creates random sharings $[b_i]^{\langle t,Q\rangle}$ of these bits.
3. The functionality distributes the shares $[b_i]_j^{\langle t,Q\rangle}$ to each player $\mathcal{P}_j$.

Fig. 15. The Offline Functionality $\mathcal{F}_{\mathsf{Offline}}$

In Protocol 1 the value E was a sum of $\binom{n}{t}$ uniform random variables in $[-2^{\mathsf{pow}-1} \cdot \mathsf{Bd}, \ldots, 2^{\mathsf{pow}-1} \cdot \mathsf{Bd})$, only two of which had to be truly random to ensure security. In Protocol 2 the value E is selected by adding two values obtained uniformly from the range $[-2^B, \ldots, 2^B))$ where $B = \lceil \log_2 \mathsf{Bd} \rceil + \mathsf{pow}$. The full procedure is given in Figure 16; the call to the Offline procedure in lines 1 and 2, indicate that this is where data is obtained from the offline procedure. This "offline" operation can either be executed in place (in which case it is not offline but online) or it is the place where the data is fetched from the prior offline execution. The correctness and security of the protocol follows from similar (but simpler) arguments to those presented above.

5 Experiments

We can now present our threshold decryption procedure for TFHE ciphertexts, which we give in Figure 17. Recall, from the introduction, for BGV or BFV ciphertexts by selecting parameters suitably or by bootstrapping, one can proceed directly to step 2 in Figure 17.

We note that line 1 of Figure 17 does not require interaction, whereas line 2 does. We thus first present experimental times for the evaluation of line 1 (Switch-n-Squash) for our four parameter sets. These we present in Table 2 of our Rust implementation. These results were obtained on an

Threshold Decryption - Protocol 2

Init():
1. The parties obtain $[\mathbf{s}']^{\langle t,Q\rangle}$ via a threshold key generation protocol.

DistDecrypt($\mathfrak{ct}, [\mathbf{s}']^{\langle t,Q\rangle}, \mathcal{U}$): On input of $\mathfrak{ct} = (\mathbf{a}, b)$ this executes the following steps:
1. $([b_i]^{\langle t,Q\rangle})_{i=0}^{B} \leftarrow \mathcal{F}_{\mathsf{Offline}}.\mathsf{Bits}(B+1)$.
2. $([b_i']^{\langle t,Q\rangle})_{i=0}^{B} \leftarrow \mathcal{F}_{\mathsf{Offline}}.\mathsf{Bits}(B+1)$.
3. The parties $\mathcal{Q}_i$ compute $[E]^{\langle t,Q\rangle} \leftarrow (-2^B + \sum_{i=0}^{B}[b_i]^{\langle t,Q\rangle} \cdot 2^i) + (-2^B + \sum_{i=0}^{B}[b_i']^{\langle t,Q\rangle} \cdot 2^i)$.
4. The parties $\mathcal{Q}_i$ compute $[v]^{\langle t,Q\rangle} \leftarrow b - \mathbf{a} \cdot [\mathbf{s}']^{\langle t,Q\rangle} + [E]^{\langle t,Q\rangle}$.
5. Party $\mathcal{Q}_i$ sends the value $[v]_i^{\langle t,Q\rangle}$ *securely* to the player $\mathcal{U}$.
6. Player $\mathcal{U}$ applies algorithm RobustOpen to robustly reconstruct the value $b - \mathbf{a} \cdot \mathbf{s} + E$, and hence m.

Fig. 16. Threshold Decryption - Protocol 2

Complete Threshold Decryption

FullDistDecrypt($\mathfrak{ct}, [\mathbf{s}']^{\langle t,Q\rangle}, \mathcal{U}$): On input of $\mathfrak{ct} = (\mathbf{a}, b) \in \mathbb{Z}_q^{\ell+1}$ this executes the following steps:
1. Execute $\mathfrak{ct}' \leftarrow$ Switch-n-Squash($\mathfrak{ct}$) to obtain $\mathfrak{ct}' \in \mathbb{Z}_Q^{L+1}$ encrypting the same value under the key $\mathbf{s}' \in \{0,1\}^L$, with noise with variance ${\sigma'_{BR}}^2$.
2. Execute $m \leftarrow$ DistDecrypt($\hat{\mathfrak{ct}}, [\mathbf{s}']^{\langle t,Q\rangle}, \mathcal{U}$) to obtain m.

Fig. 17. The complete threshold decryption protocol for TFHE ciphertexts

AWS `m6i.metal` instance with 128 Intel Xeon Gen 3 vCPUs and 512 GiB RAM, taking an average execution time over 100 runs of the relevant algorithms.

Table 2. Execution times (in milliseconds) for line 1 (Switch-n-Squash) of Figure 17

Parameters	Switch-n-Squash
$(2^{64}, 777) \to (2^{128}, 4096)$	241.01
$(2^{64}, 870) \to (2^{128}, 4096)$	265.80
$(2^{64}, 1024) \to (2^{128}, 4096)$	316.77
$(2^{64}, 1024) \to (2^{128}, 4096)$	314.19

Recall these timings are for a part of the computation which does not requite interaction, and which are amenable to acceleration by the FHE accelerators currently being developed. For example, the paper [BDV22] shows a three orders of magnitude acceleration using only FPGA acceleration (as opposed to ASIC acceleration).

To time line 2 of Figure 17 we need to consider various other factors; the number of parties n performing the distributed decryption, the threshold t, the type of network, the number of active corruptions. For each of our four parameter sets we utilized three different sets of (n, t) values; namely $(n, t) = (4, 1), (10, 3)$ and $(40, 13)$. For the first of these one can utilize the PRSS-based distributed decryption method, for the other two one needs to utilize the methodology requiring an offline phase. In our experiments we only timed the online phase for the latter two cases. In all cases we present the average run-time over 1000 iterations for a single honest party. Recall this party will terminate as soon as it has received enough shares to robustly reconstruct the underlying encrypted value.

We also investigated the effect of a LAN-like setting (1 Gbit/s with small ping times of ≈1 ms) versus a WAN-like setting (100 Mbit/s with high ping times of ≈100 ms), and whether we are optimistic or pessimistic in terms of the number of errors introduced by the adversary during the distributed decryption. If there are no errors then the online-error correction method underlying RobustOpen will execute faster than if there are maximal, i.e. t, adversarial errors. The asynchronous channels are implemented using gRPC with `tokio` and `tonic` Rust crates.

We measured our experiments on a single AWS `m6i.metal` instance as above. We ran the n protocol parties as individual docker containers and simulated the LAN/WAN connection between them. Our full results are given in Table 3 of Appendix E.

In the most favorable situation, namely four parties where we can tolerate one dishonest party over a LAN, we obtain execution times for line 2 of Figure 17 of under 2 milliseconds. In the least favorable situation we investigated, namely 40 parties of which thirteen are malicious (and send invalid share values), and over a WAN, we are able to execute line 2 of Figure 17 in under 100 milliseconds on average.

Acknowledgements

The work of Nigel P. Smart on this work was supported by CyberSecurity Research Flanders with reference number VR20192203, and by the FWO under an Odysseus project GOH9718N. We would like to thank Fre Vercauteren for conversations during the course of this work.

References

ABGS22. Diego F. Aranha, Carsten Baum, Kristian Gjøsteen, and Tjerand Silde. Verifiable mix-nets and distributed decryption for voting from lattice-based assumptions. Cryptology ePrint Archive, Report 2022/422, 2022. https://eprint.iacr.org/2022/422.

ACD⁺19. Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part I*, volume 11891 of *Lecture Notes in Computer Science*, pages 471–501, Nuremberg, Germany, December 1–5, 2019. Springer, Heidelberg, Germany.

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, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 483–501, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

AJW11. Gilad Asharov, Abhishek Jain, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. Cryptology ePrint Archive, Report 2011/613, 2011. https://eprint.iacr.org/2011/613.

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

BBB⁺23. Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization and larger precision for (t)fhe. *Journal of Cryptology*, 36(3):28, 2023.

BCG93. Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th Annual ACM Symposium on Theory of Computing*, pages 52–61, San Diego, CA, USA, May 16–18, 1993. ACM Press.

BCS19. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using TopGear in overdrive: A more efficient ZKPoK for SPDZ. In Kenneth G. Paterson and Douglas Stebila, editors, *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*, volume 11959 of *Lecture Notes in Computer Science*, pages 274–302, Waterloo, ON, Canada, August 12–16, 2019. Springer, Heidelberg, Germany.

BD10. Rikke Bendlin and Ivan Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In Daniele Micciancio, editor, *TCC 2010: 7th Theory of Cryptography Conference*, volume 5978 of *Lecture Notes in Computer Science*, pages 201–218, Zurich, Switzerland, February 9–11, 2010. Springer, Heidelberg, Germany.

BDV22. Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. FPT: a fixed-point accelerator for torus fully homomorphic encryption. Cryptology ePrint Archive, Report 2022/1635, 2022. https://eprint.iacr.org/2022/1635.

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, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 565–596, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

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

BLR+18. Shi Bai, Tancrède Lepoint, Adeline Roux-Langlois, Amin Sakzad, Damien Stehlé, and Ron Steinfeld. Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance. *Journal of Cryptology*, 31(2):610–640, April 2018.

BS23. Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from LWE with polynomial modulus. Cryptology ePrint Archive, Report 2023/016, 2023. https://eprint.iacr.org/2023/016.

BSS99. I F Blake, G Seroussi, and N P Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, United Kingdom, 1999.

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

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

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

CJP21. Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*, volume 12716 of *Lecture Notes in Computer Science*, pages 1–19, Be'er Sheva, Isreal, 2021. Springer.

CLO+13. Ashish Choudhury, Jake Loftus, Emmanuela Orsini, Arpita Patra, and Nigel P. Smart. Between a rock and a hard place: Interpolating between MPC and FHE. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 221–240, Bengalore, India, December 1–5, 2013. Springer, Heidelberg, Germany.

CLOT21. Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699, Singapore, December 6–10, 2021. Springer, Heidelberg, Germany.

CSS+22. Siddhartha Chowdhury, Sayani Sinha, Animesh Singh, Shubham Mishra, Chandan Chaudhary, Sikhar Patranabis, Pratyay Mukherjee, Ayantika Chatterjee, and Debdeep Mukhopadhyay. Efficient threshold FHE with application to real-time systems. Cryptology ePrint Archive, Report 2022/1625, 2022. https://eprint.iacr.org/2022/1625.

DKL+13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil

Jajodia, and Keith Mayes, editors, *ESORICS 2013: 18th European Symposium on Research in Computer Security*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18, Egham, UK, September 9–13, 2013. Springer, Heidelberg, Germany.

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

Feh93. Serge Fehr. Span programs over rings and how to share a secret from a module. Masters Thesis, ETH Zurich, 1993. https://crypto.ethz.ch/publications/Fehr98.html.

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

Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

Joy23. Marc Joye. Tfhe public-key encryption revisited. Cryptology ePrint Archive, Paper 2023/603, 2023. https://eprint.iacr.org/2023/603.

JSL22. Robin Jadoul, Nigel P. Smart, and Barry Van Leeuwen. MPC for Q_2 access structures over rings and fields. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021: 28th Annual International Workshop on Selected Areas in Cryptography*, volume 13203 of *Lecture Notes in Computer Science*, pages 131–151, Virtual Event, September 29 – October 1, 2022. Springer, Heidelberg, Germany.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

OSV20. Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 254–283, San Francisco, CA, USA, February 24–28, 2020. Springer, Heidelberg, Germany.

RST+22. Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *Journal of Cryptology*, 35(1):5, January 2022.

A Proof of Lemma 2.3

Proof. In the following, we show the case of $e = 1$. The general case follows from the triangle inequality and $|e|$ applications of the lemma for $e = 1$.

Let $b = 2 \cdot B + 1$ and note that $\mathcal{P}(x) = (b - |x|)/b^2$ for all $x \in [-2 \cdot B, 2 \cdot B]$. We begin by introducing two truncated distributions, $\mathcal{P}_\bot$ and $\mathcal{P}_\top$. Define $\mathcal{P}_\bot$ such that it is proportional to $\mathcal{P}$, except that $\mathcal{P}_\bot(-2 \cdot B) = 0$, i.e. we have $\mathcal{P}_\bot(x) = S \cdot (b - |x|)/b^2$ for all $x \in [-2 \cdot B + 1, 2 \cdot B]$, where $S = b^2/(b^2 - 1)$ is the normalization factor. Similarly, we define $\mathcal{P}_\top$ to be proportional to $1 + \mathcal{P}$ but truncated at the top such that we have $\mathcal{P}_\top(2 \cdot B + 1) = 0$. Note that the normalization factor of $\mathcal{P}_\top$ matches the one of $\mathcal{P}_\bot$ and that the two distributions have the same support. We will show the result by bounding $\Delta_{SD}(\mathcal{P}^m, \mathcal{P}_\bot^m)$, $\Delta_{SD}(\mathcal{P}_\bot^m, \mathcal{P}_\top^m)$, and $\Delta_{SD}(\mathcal{P}_\top^m, (1 + \mathcal{P})^m)$. The rest follows by the triangle inequality.

First note that $\Delta_{SD}(\mathcal{P}, \mathcal{P}_\bot) = \Delta_{SD}(\mathcal{P}_\top, 1 + \mathcal{P}) = 1/b^2$. Accordingly, we have $\Delta_{SD}(\mathcal{P}^m, \mathcal{P}_\bot^m) \le m/b^2$ and $\Delta_{SD}(\mathcal{P}_\top^m, (1 + \mathcal{P})^m) \le m/b^2$.

It remains to bound $\Delta_{SD}(\mathcal{P}_\bot^m, \mathcal{P}_\top^m)$. We first consider the KL-divergence between $\mathcal{P}_\bot$ and $\mathcal{P}_\top$:

$$\begin{aligned} \Delta_{KL}(\mathcal{P}_\bot, \mathcal{P}_\top) &= -\sum_{x=-2\cdot B+1}^{2\cdot B} \mathcal{P}_\bot(x) \cdot \log \frac{\mathcal{P}_\top(x)}{\mathcal{P}_\bot(x)} \\ &= -S \cdot \sum_{x=-2\cdot B+1}^{2\cdot B} \mathcal{P}(x) \cdot \log \frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} \end{aligned}$$

$$= -S \cdot \Big[\left(\mathcal{P}(0) \cdot \log \frac{\mathcal{P}(-1)}{\mathcal{P}(0)} \right) + \left(\mathcal{P}(2 \cdot B) \cdot \log \frac{\mathcal{P}(2 \cdot B - 1)}{\mathcal{P}(2 \cdot B)} \right)$$
$$+ \sum_{x=-2 \cdot B+1}^{-1} \mathcal{P}(x) \cdot \left(\log \frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} + \log \frac{\mathcal{P}(x+1)}{\mathcal{P}(x)} \right) \Big]$$

Note that we have

$$\mathcal{P}(2 \cdot B) \cdot \log \frac{\mathcal{P}(2 \cdot B - 1)}{\mathcal{P}(2 \cdot B)} = \frac{\log 2}{b^2} \geq 0$$

so this term may be ignored (due to the negative sign of the expression). In the following, we make use of the fact that $\log(1 - 1/x) \geq -2/x$ for all $x \geq 2$. Then we have

$$\mathcal{P}(0) \cdot \log \frac{\mathcal{P}(-1)}{\mathcal{P}(0)} = \frac{1}{b} \cdot \log \frac{b-1}{b} = \frac{1}{b} \cdot \log(1 - 1/b) \geq -2/b^2 \ .$$

Similarly, we have for all $x \in [-2 \cdot B + 1, -1]$

$$\begin{aligned}
\mathcal{P}(x) \cdot & \left(\log \frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} + \log \frac{\mathcal{P}(x+1)}{\mathcal{P}(x)} \right) \\
&= \mathcal{P}(x) \cdot \left(\log \left(\frac{\mathcal{P}(x-1)}{\mathcal{P}(x)} \frac{\mathcal{P}(x+1)}{\mathcal{P}(x)} \right) \right) \\
&= \frac{b+x}{b^2} \cdot \log \left(\frac{(b+x-1)(b+x+1)}{(b+x)^2} \right) \\
&= \frac{b+x}{b^2} \cdot \log \left(\frac{(b+x)^2 - 1}{(b+x)^2} \right) \\
&= \frac{b+x}{b^2} \cdot \log \left(1 - \frac{1}{(b+x)^2} \right) \\
&\geq -\frac{2}{b^2 \cdot (b+x)} \ .
\end{aligned}$$

Combined, we get

$$\begin{aligned}
\Delta_{KL}(\mathcal{P}_\perp, \mathcal{P}_\top) &\leq S \cdot \left[\frac{2}{b^2} + \sum_{x=-2 \cdot B+1}^{-1} \frac{2}{b^2 \cdot (b+x)} \right] \\
&= \frac{2 \cdot S}{b^2} \cdot \left(1 + \sum_{x=2}^{2 \cdot B} 1/x \right) \\
&= \frac{2 \cdot S}{b^2} \cdot \sum_{x=1}^{2 \cdot B} 1/x \\
&\leq \frac{2 \cdot (\log(2 \cdot B) + 1)}{(b^2 - 1)} \\
&= \frac{\log B + 2}{2 \cdot (B^2 + B)}.
\end{aligned}$$

By the sub-additive property of Δ_{KL} we now have

$$\Delta_{KL}(\mathcal{P}_{\perp}^{m}, \mathcal{P}_{\top}^{m}) \leq m \cdot \frac{\log B + 2}{2 \cdot (B^2 + B)}$$

and by Pinsker's inequality

$$\Delta_{SD}(\mathcal{P}_{\perp}^{m}, \mathcal{P}_{\top}^{m}) \leq \sqrt{\Delta_{KL}(\mathcal{P}_{\perp}^{m}, \mathcal{P}_{\top}^{m})} = \sqrt{m \cdot \frac{\log B + 2}{2 \cdot (B^2 + B)}}.$$

□

B Auxillary Protocols

B.1 Commitment Schemes

We will need a commitment scheme. The one we choose is secure in the random oracle model, and thus uses a hash function. The hash function $\mathcal{H} : \{0,1\}^* \longrightarrow \{0,1\}^{|\mathcal{H}|}$ is assumed to be one such as SHA-256, or SHA-3. The output length $|\mathcal{H}|$ should be at least $(2 \cdot \mathsf{sec})$-bits in length. The scheme is defined in Figure 18. If we want to specify the randomness externally to the commitment then we write $\mathsf{Commit}(m; r)$.

Commitment Scheme

$\mathsf{Commit}(m)$: On input of a message $m \in \{0,1\}^*$ this proceeds as follows, to produce the commitment c and the opening information o.
1. Generate a random bit string $r \leftarrow \{0,1\}^{\mathsf{sec}}$.
2. Set $o \leftarrow m \| r$.
3. Compute the commitment $c \leftarrow \mathcal{H}(o)$.
4. Output (c, o).

$\mathsf{Open}(c, o)$: This opens the commitment.
1. Compute $c' \leftarrow \mathcal{H}(o)$.
2. If $c' \neq c$ then abort.
3. Write $m \| r \leftarrow o$, where r is sec-bits long.
4. Output m.

Fig. 18. Commitment Scheme

B.2 Agree on a Random Number

We require a protocol which enables a subset $S \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ of our set of parties need to agree on a random value, often the key for a PRF, which is outside the control of all parties. This is done via the protocol $\mathsf{AgreeRandom}$ in Figure 19. Note, the protocol does not verify that the players all obtain the same random value. The idea being that if the same random value is not obtained then this should become apparent once the value is used in a PRF.

Protocol $\mathsf{AgreeRandom}(S, k)$

The input is a subset $S \subseteq \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ and a value $k \in \mathbb{N}$ which is the length of the output. Let I denote the index set of the players in S, i.e. $S = \{\mathcal{P}_i\}_{i \in I}$.

1. Each $\mathcal{P}_i \in S$ generates $s_i \leftarrow \{0,1\}^k$.
2. Each $\mathcal{P}_i \in S$ executes $(c_i, o_i) \leftarrow \mathsf{Commit}(s_i)$.
3. Each $\mathcal{P}_i \in S$ sends c_i to all other players in S.
4. When $\mathcal{P}_i$ has received c_j from all $j \in I \setminus \{i\}$ it sends o_i to all other players in S.
5. Each $\mathcal{P}_i \in S$ executes $s_j \leftarrow \mathsf{Open}(c_j, o_j)$ for all $j \in I \setminus \{i\}$.
6. Each $\mathcal{P}_i \in S$ outputs $s \leftarrow \bigoplus_{j \in I} s_i$.

Fig. 19. Protocol AgreeRandom

C Offline Phase

Our methodology for large values of $\binom{n}{t}$ requires a method to generate shared random bits within an "offline" phase. We now elaborate on how this could be done. We first assume a generic MPC protocol which works with the secret sharing scheme $[\cdot]^{\langle t,Q \rangle}$, which recall shares elements in the Galois ring $\mathcal{G}$.

We present the ideal functionality $\mathcal{F}_{\mathsf{MPC}}$ in Figure 20, note we only require the ability to multiply elements and generate shares of random elements in this functionality. We describe this in terms of sharings in order to facilitate ease of understanding, however it can also be expressed in terms of the usual "handles" to variables maintained inside the functionality.

Ideal Functionality $\mathcal{F}_{\mathsf{MPC}}$

$\mathcal{F}_{\mathsf{MPC}}.\mathsf{RandomSharing}()$:

1. The ideal functionality generates a uniformly random $a \leftarrow \mathcal{G}$.
2. The ideal functionality generates a random sharing $[a]^{\langle t,Q \rangle}$.
3. The value $[a]_i^{\langle t,Q \rangle}$ is given to player $\mathcal{P}_i$.

$\mathcal{F}_{\mathsf{MPC}}.\mathsf{Mult}([a]^{\langle t,Q \rangle}, [b]^{\langle t,Q \rangle})$:

1. The functionality forms the a random sharing $[c]^{\langle t,Q \rangle}$ of the product $a \cdot b$.
2. The functionality passes $[c]_i^{\langle t,Q \rangle}$ to player $\mathcal{P}_i$.

Fig. 20. The MPC Ideal Functionality $\mathcal{F}_{\mathsf{MPC}}$

To implement such a functionality there are a number of possibilities. To obtain a fully robust offline protocol over synchronous networks one could utilize the protocol from [ACD+19]. If one only requires an offline phase which is actively secure up to a possible abort then the potentially simpler protocols from [JSL22] may be preferred.

The standard method to generate shared random bits in an MPC protocol (for odd values of Q) is to use the $2:1$ nature of the squaring operation to produce random bits, see [DKL+13] for example. However, when Q is even we cannot use the above technique, since the squaring map is now a $4:1$ mapping. Instead we utilize a technique from [OSV20], which we simplify a little below.

We require the the trace and half-trace functions by

$$\mathsf{Tr}(x) = \sum_{j=0}^{(d-1)} x^{2^j},$$

$$\overline{\mathsf{Tr}}(x) = \sum_{j=0}^{(d-1)/2} x^{2^{2j}}.$$

Our method to generate bits, when $Q = 2^K$ requires access to a function function $\mathsf{Solve}(v, Q)$, which solves the equation

$$X^2 + X = v \pmod{Q}, \tag{3}$$

assuming such a solution exists. Such a solution exists if we have $\mathsf{Tr}(v) = 0 \pmod 2$. In our application we are looking for solutions in $\mathcal{G}$.

To solve this equations requires two steps: First we solve the equation modulo 2, i.e. in the finite field $\mathbb{F}_{2^d}$, and then we lift this solution using Hensel's Lemma to the whole ring $\mathcal{G}$. Then to solve

$$X^2 + X = v \pmod 2 \tag{4}$$

we apply the classical method, to be found in [BSS99, page 26], which depends on the parity of d.

d Odd: In this case we compute

$$x_0 = \mathsf{Solve}(v, 2) = \overline{\mathsf{Tr}}(v) \pmod 2.$$

d Even: This case is slightly more complex. We first find an element $\delta \in \mathbb{F}_{2^d}$ such that $\mathsf{Tr}(\delta) = 1$. This is in fact easy, as half the elements in $\mathbb{F}_{2^d}$ have trace one. We then can write

$$x_0 = \mathsf{Solve}(v, 2) = \sum_{i=0}^{d-2} \Big(\sum_{j=i+1}^{d-1} \delta^{2^j} \Big) \cdot v^{2^i} \pmod 2.$$

We can lift the solution to equation (4), to a solution of equation (3) by executing the following recursion $\lceil \log_2 K \rceil$ times

$$x_0 = \mathsf{Solve}(v, 2),$$

$$x_{n+1} = \frac{x_n^2 + v}{1 + 2 \cdot x_n} \pmod Q, \quad \text{for } n \geq 0.$$

This appears to require a full outer Newton iteration in order to find the sucessive x_i, and a full inner Newton iteration to find the inverse of $(1 + 2 \cdot x_n)$. However, this initial $O(\lceil \log_2 K \rceil^2)$ estimate of operations can be replaced with $O(\lceil \log_2 K \rceil)$ iterations, using the algorithm in Figure 21.

This then gives us the algorithm given in Figure 22 to generate a random bit in $\mathcal{G}$. What is nice, from an MPC perspective, about the even Q case is that we do not need to loop to produce a non-zero value. This leads to the MPC-version of the bit generation protocol in Figure 23.

Solve($v, Q = 2^K$)

1. $x \leftarrow \mathsf{Solve}(v, 2)$.
2. $y \leftarrow 1$.
3. For $i = 1, \ldots, \lceil \log_2 K \rceil$ do
   (a) $m \leftarrow 2^{2^i}$.
   (b) $z \leftarrow 1 + 2 \cdot x \pmod m$.
   (c) $y \leftarrow y \cdot (2 - z \cdot y) \pmod m$.
   (d) $y \leftarrow y \cdot (2 - z \cdot y) \pmod m$.
   (e) $x \leftarrow (x \cdot x + v) \cdot y \pmod m$.
4. Return $x \pmod Q$.

Fig. 21. Solving $X^2 + X = v \pmod Q$ Using Hensel Lifting

Bit Generation: Q Even

1. $a \leftarrow \mathcal{G}$.
2. $v \leftarrow a + a^2 \pmod Q$.
3. $r \leftarrow \mathsf{Solve}(v, Q)$.
4. $d \leftarrow (-1 - 2 \cdot r) \pmod Q$.
5. $b \leftarrow (a - r)/d \pmod Q$.
6. Return b.

Fig. 22. Random bit generation when Q is Even

Example: We go through a small worked example to demonstrate this actually works using the simple ring $\mathcal{G} = \mathbb{Z}_q$, for $q = 2^3 = 8$.

1. $a \leftarrow \mathbb{Z}_{2^3}$. So take, for example $a = 3$.
2. $v \leftarrow a + a^2 \pmod 8$. So in our example $v = 4$.
3. Applying $r \leftarrow \mathsf{Solve}(v, 8)$ gives us $r = 3$.
4. $d \leftarrow -1 - 2 \cdot r$, gives us $d = 1$.
5. $b \leftarrow (a - r) \ / \ d \pmod 8 = (3 - 3) \ / \ 1 \pmod 8 = 0$.

D Threshold Key Generation

In this appendix we briefly discuss how a threshold key $[\mathbf{s}']^{\langle t,Q \rangle}$ can be generated.

The first technique would be to utilize the homomorphic properties of the underlying LWE encryption to “combine” different users individual keys into a single threshold key $[\mathbf{s}']^{\langle t,Q \rangle}$ with the correct properties. This approach typifies what is called *multi-key homomorphic encryption*. However, the approach tends to produce a threshold public key with different underlying noise distributions to that which would arise from a trusted third party generating the shared public key. This results in great inefficiencies in practice as (especially for TFHE) performance of homomorphic evaluation and bootstrapping is highly dependent on choosing exactly the correct parameters and noise distributions.

The second technique is to apply an MPC protocol to generate the underlying secret key data in an secret shared form. For LWE based ciphertexts this is relatively straight forward. One can

Bit Generation and Usage

Offline.GenBit():

1. Execute the following a "sufficient" number of times in parallel:
   (a) $[a]^{\langle t,Q\rangle} \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{RandomSharing}()$.
   (b) $[u]^{\langle t,Q\rangle} \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{Mult}([v]^{\langle t,Q\rangle}, [v]^{\langle t,Q\rangle})$.
   (c) $[v]^{\langle t,Q\rangle} \leftarrow [a]^{\langle t,Q\rangle} + [u]^{\langle t,Q\rangle}$.
   (d) $v \leftarrow \mathsf{RobustOpen}([v]^{\langle t,Q\rangle})$.
   (e) $r \leftarrow \mathsf{Solve}(v, Q)$.
   (f) $d \leftarrow (-1 - 2 \cdot r) \pmod{Q}$.
   (g) $[b]^{\langle t,Q\rangle} \leftarrow (a - r)/d$.
   (h) $\mathcal{B} \leftarrow \mathcal{B} \cup \{[b]^{\langle t,Q\rangle}\}$.

Offline.Bits(v):

1. While $|\mathcal{B}| < v$ then execute Offline.GenBit().
2. Write $\mathcal{B} = \{[b_i]^{\langle t,Q\rangle}\}_{i=1}^{N}$.
3. $\mathcal{B} \leftarrow \mathcal{B} \setminus \{[b_i]^{\langle t,Q\rangle}\}_{i=1}^{v}$.
4. Return $\{[b_1]^{\langle t,Q\rangle}, \ldots, [b_v]^{\langle t,Q\rangle}\}$.

Fig. 23. Bit Generation and Bit Usage

utilize the MPC functionality from earlier, to obtain many sharings of random bits $[b]^{\langle t,Q\rangle}$. Given these almost all the operations needed to perform a threshold key generation are linear operations, followed by openings of the shared values (when wishing to output the shared public values themselves). This approach has been used to generate threshold BGV style keys for the SPDZ MPC system [RST+22]. In our situation (generating TFHE keys) the application is even easier due to the ciphertext modulus being a power of a prime and not a product of multiple distinct primes.

E Timings for Distributed Decryption

The results for our various experiments for distributed decryption are given in Table 3.

Table 3. Execution times for line 2 (DistDecrypt) of Figure 17 when $(Q, L) = (2^{128}, 4096)$.

(n,t)	Ping Time	Number Errors	DistDecrypt (ms)
$(4,1)$	≈ 1 ms	0	1.43
$(4,1)$	≈ 1 ms	1	1.64
$(4,1)$	≈ 100 ms	0	50.78
$(4,1)$	≈ 100 ms	1	53.68
$(10,3)$	≈ 1 ms	0	2.40
$(10,3)$	≈ 1 ms	3	3.31
$(10,3)$	≈ 100 ms	0	53.85
$(10,3)$	≈ 100 ms	3	57.90
$(40,13)$	≈ 1 ms	0	18.57
$(40,13)$	≈ 1 ms	13	41.58
$(40,13)$	≈ 100 ms	0	68.38
$(40,13)$	≈ 100 ms	13	91.59