Distributed Key Generation for Efficient Threshold-CKKS

Seonhong Min[1], Guillaume Hanrot[2], Jai Hyun Park[2], Alain Passelègue[2], and Damien Stehlé[2]

[1] Seoul National University, Seoul, Republic of Korea
[2] CryptoLab Inc., Lyon, France

Abstract. Threshold fully homomorphic encryption provides efficient multi-party computation with low round-complexity. Among fully homomorphic encryption schemes, CKKS (Cheon-Kim-Kim-Song) enables high-throughput computations on both approximate and exact data. As most interesting applications involve deep computations, they require bootstrapping, the most efficient variants of which rely on sparse ternary secret keys. Unfortunately, so far, key generation protocols for threshold-CKKS either assume a trusted dealer, or lead to dense and non-ternary secret keys that severely damage computational throughput. In the latter case, the impact is so large that one often considers off-loading bootstrapping to an interactive protocol [Mouchet et al., PETS'21].

We introduce a novel Distributed Key Generation (DKG) protocol for threshold-CKKS. At a high level, it consists in running the existing distributed key generation algorithm from Mouchet et al. resulting in large secret keys, and using it to homomorphically evaluate the sparse-secret key generation algorithm. At the end, the parties obtain additive shares of a sparse secret key. The main technical challenge is to obtain an algorithm for sampling sparse ternary vectors of prescribed Hamming weight that can be CKKS-evaluated in an efficient manner. In the process, we design a new sampler of one-hot vectors that outperforms the one from [Boneh et al., AFT'20]. We also design a rejection-sampling algorithm to map several one-hot vectors into a vector of prescribed Hamming weight. The whole process can be performed with only two CKKS bootstraps, even for a significant number of users.

We present several variants of the DKG protocol, with 2 to 4 communication rounds, as well as an extension to key generation delegation. We implemented the 4-round protocol; its computational components run in under 2s on GPU (RTX4090) for up to 1024 parties.

1 Introduction

Fully Homomorphic Encryption (FHE) enables arbitrary computations on data provided in encrypted form. Since the first FHE scheme [Gen09], numerous others have been proposed. Among them, CKKS [CKKS17] stands out for enabling high-throughput computations on real-valued data, making it a desirable approach for privacy-preserving machine learning. Its applicability was recently extended to computations on discrete data [DMPS24, BCKS24, BKSS24].

When performing multiplications, CKKS consumes modulus: ciphertexts are defined modulo some integer, and the modulus of the output is lower than the modulus of the multiplicands. To make the Ring-LWE [SSTX09, LPR10] instances intractable, the ciphertext modulus must remain bounded from above, which implies that only a few multiplications can be performed sequentially without other ingredients. CKKS bootstrapping [CHK+18] is a ciphertext maintenance operation that takes a low-modulus ciphertext and transforms it into a high-modulus ciphertext for the same underlying plaintext, up to some limited numerical error. Bootstrapping drives the efficiency of a CKKS instantiation in two ways: it is an expensive operation, and it may consume a significant amount of modulus so that there may not be much of it left for the computations that we want to perform. If Q_{comp} denotes the modulus left after bootstrapping and T_{BTS} denotes the runtime of bootstrapping, the key objective in bootstrapping design is to maximize Q_{comp}/T_{BTS}. As it is the core of CKKS efficiency, bootstrapping has been intensively studied and optimized.

In order to obtain an efficient CKKS instantiation, the best known approach is to rely on the Sparse Secret Encapsulation (SSE) technique from [BTH22]. It is used in all competitive implementations of CKKS (see [MBTPH20, BBB+22, Cry22]). As the bootstrapping multiplicative depth depends on the Hamming weight of the secret key, SSE generates two (ternary) secret keys: a sparse key and a less-sparse key. The less-sparse key may have Hamming weight ranging from 192 to $2N/3$ where N is the dimension of the key (a power-of-two integer typically between 2^{15} and 2^{17}) and is used most of the time. The sparse key may have Hamming weight as low as 32 and is used inside bootstrapping: the secret for a ciphertext is temporarily switched to the sparse key, and then switched back to the less-sparse key. To enable this, two switching keys are required: an encryption of sk under the sparse key sk' at a low modulus, and an encryption of sk' under sk at a high modulus. We note that sparse Ring-LWE secrets have been used in other FHE schemes as well to accelerate their bootstrapping algorithms (see, e.g., [GHS12, LMSS23, BTH22, MHWW24]).

Threshold-FHE (ThFHE) [AJL+12] is an extension of FHE where the decryption key is distributed among several users, specific subsets of which must collaborate to decrypt. As more broadly in threshold cryptography, this reduces the risks incurred by hosting a secret key in a single machine: if that machine gets compromised by an adversary, then it can decrypt all available ciphertexts; if that machine is out of service, then no ciphertext can be decrypted. Further, ThFHE enables low-round secure multiparty computations [AJL+12], removing the latency bottleneck often encountered when multiplying communication rounds. Finally, ThFHE can be used to "thresholdize" cryptographic algorithms, leading, among others, to round-optimal threshold signatures [BGG+18]. It is important to note that ThFHE puts an additional constraint on bootstrapping: its correctness not only impacts the usability of the scheme, but also its security. Indeed, it was shown in [CCP+24] that the bootstrapping algorithms of several FHE schemes lead to key-recovery attacks when bootstrapping failures

are observed. To obtain sufficiently low failure probability without damaging performance, the only known approach is SSE [BTH22].

To enable a high-throughput threshold-CKKS, one hence needs a key generation protocol that produces properly distributed key material. The current solutions are far from satisfactory. Key generation algorithms that rely on generic secure multi-party computation or on a trusted dealer fail in terms of efficiency and security, respectively. The state-of-the-art Distributed Key Generation (DKG) protocol for threshold-CKKS [MTBH21] has each party generate its own key pair and, as there is an additive homomorphism between the secret key and the public key in CKKS (up to small noise and assuming that the uniform part of the key is generated using a common reference string), one can then set the joint public key (resp. secret key) as the sum of all public keys (resp. secret keys). Even though the evaluation key involves a quadratic function of the secret key, it is possible to obtain a joint evaluation key for the joint secret key in one more round, or without any extra round but with a produced evaluation key that differs from usual and results in lower throughput [KLSW24, BMM25]. The algorithm from [MTBH21] was extended from the n-out-of-n setting (also referred to as multi-party FHE) to the t-out-of-n setting in [MBH23]. As the joint secret key is the sum of independently sampled keys, with overwhelming probability, it is not sparse, even if the individual keys were sparse. The joint key is unusable for SSE, and bootstrapping then becomes less efficient and consumes more modulus. For 32 users, we estimate that the throughput would be at least 1.71x lower (see App. H). We stress that this slowdown stems from the DKG protocol but impacts the throughput of all computations. Further, as the magnitude of the key depends on the number of users and the bootstrapping circuit depends on the magnitude of the key, bootstrapping circuits for diverse numbers of users should be considered (or a single one for a large maximum number of users, but that will be unnecessarily inefficient for a smaller number of users). As bootstrapping is a complex algorithm with numerous optimizations, this may lead to significant software development complications. For these reasons, threshold-CKKS implementations typically off-load bootstrapping to an interactive protocol, damaging throughput with communication delays. Collective bootstrapping was initially proposed in [MBH23] and is available in [BBB$^+$22, MBTPH20]. We are thus led to the following question.

Does there exist a practical DKG protocol for threshold-CKKS that does not rely on a trusted dealer and that enables homomorphic evaluations that are as efficient as (non-threshold) CKKS, independently of the number of parties?

Contributions. We describe a DKG protocol between an arbitrary number of parties n that only assumes a common reference string (which can be implemented with a hash function modeled as a random oracle). It outputs a public key and an evaluation key that enable homomorphic evaluations that are near-identical to state-of-the-art non-threshold CKKS based on sparse secret encapsulation. Each party obtains a decryption key share, for n-out-of-n threshold decryption. It extends to t-out-of-n for any $t \leq n$ by relying on [MBH23] and, for this reason, this work focuses on the n-out-of-n case.

We give several variants of the protocol, with 2 to 4 rounds of communication and various trade-offs between the run-time of the DKG and the throughput performance and precision of the homomorphic computations. We implemented a 4-round variant with an evaluation precision that is slightly degraded. For this purpose, we used a variant of the HEaaN library [Cry22] based on [CCK+25]. In our experiments, the dominating step of DKG runs in slightly less than 2 seconds on a commodity GPU (RTX4090) for up to 1024 parties. This computation can be performed publicly, by any of the parties or by external servers, making it likely that GPUs would be available. Using the output key, the computation throughput is the same as non-threshold CKKS, and the precision degrades by approximately 1.28 bits.

The techniques developed can also be used to delegate key generation (in a single-party or multi-party CKKS). Assume a limited client, for example an IoT device, wishes to externalize computations. It may not have the computational resources to generate and transmit an evaluation key, which can be of the order of 5-10GB. Instead, it can ask a number of external parties to call key generators to collectively run a variant of the DKG algorithm so that only the client gets to know the secret key whereas the public key and evaluation key are made public. Security relies on the assumption that the key generators do not collude.

1.1 Technical Overview.

At a conceptual level, the proposed DKG protocol runs the ThFHE-based universal thresholdizer framework from [BGG+18] on the key generation algorithm for a single-key FHE, with a suboptimal but temporary ThFHE scheme. First, the parties collaboratively run a setup protocol of some (n-out-of-n) ThFHE scheme with low communication. Note that the resulting scheme generally has a different key-pair structure from the single-key scheme, but it is only a temporary situation. Then, the parties homomorphically sample from the secret key distribution (optionally, from the noise distribution as well), using encrypted random bits from the parties, using this suboptimal ThFHE scheme. Finally, the parties can jointly decrypt this ciphertext to obtain a share of this secret and run another setup protocol of the ThFHE scheme with these new shares. This new scheme has the same key distribution as the conventional single-key FHE scheme, and ThFHE homomorphic computations can be performed exactly and as efficiently as in the single-key case. We emphasize that this framework can be applied to obtain a DKG protocol for any ThFHE scheme, temporarily relying on an existing ThFHE scheme that is used to homomorphically evaluate the key generation algorithm of the target ThFHE.

In this work, we focus on CKKS, both as target and temporary ThFHE schemes, as it provides high-throughput computations on approximate and exact data. In particular, this allows us to enjoy an efficient evaluation of the key generation circuit. Our main DKG protocol variant first runs the 2-round protocol from [MTBH21] to obtain temporary key material: this step outputs a public key and an evaluation key whose underlying secret key is the sum of n individual secret keys independently sampled by the n users, as well as an n-out-of-n

decryption key share for each user. As already mentioned, this key material does not allow computations that are as efficient as single-key CKKS, but they suffice for temporary computations internal to DKG. Once we have those keys, we homomorphically run the secret key generation algorithm of the SSE variant of CKKS before the third communication round, to obtain shares of a sparse secret key that can be used for bootstrapping and a less-sparse secret key that can be used during the rest of homomorphic computations, as well as partial evaluation key material. The protocol uses a fourth communication round to create the non-linear component of the evaluation keys (namely, the relinearization key), as in [MTBH21].

We observe that the last round can be removed by homomorphically running the public and evaluation key generation algorithms from [BTH22] at the third round, on top of the secret key generation algorithm. Another round can be saved by handling relinearization keys of the temporary ThFHE scheme as in [KLSW24, BMM25]. In all these variants, the generated global keys have noise terms that grow with the number of users, leading to lower computation precision. Noise terms as in single-key CKKS can be obtained via homomorphic generation as well, at the expense of a slower DKG protocol.

The only computationally intensive component of our main DKG protocol variant is the homomorphic evaluation of the SSE secret key generation, at the third round. For the sparse key, we have to homomorphically sample a uniform ternary vector of dimension N and Hamming weight exactly h, where N is large (e.g., $N = 2^{16}$) and h is small (e.g., $h = 32$). Note that the value of h is very precise, as it directly impacts the performance of bootstrapping, the security of the evaluation key, and the failure probability of bootstrapping. Comparatively, sampling the less-sparse key is easier, as its precise Hamming weight is not so impactful: only its rough value matters.

Let us now focus on the sparse key. Our goal is to design an algorithm for sampling a uniform ternary vector of dimension N and Hamming weight $h \ll N$, which can be CKKS-evaluated. For the input randomness, each one of the n parties sends an encryption of a uniform bit-string: these are added modulo 2, by placing the plaintext bits in the most significant bits of the ciphertexts. The sum is bootstrapped using [BCKS24, BKSS24], so that the uniform bits are now at a high level. Up to multiplying (element-wise) by a vector of uniform signs, we can concentrate on using those bootstrapped uniform bits to generate a uniform binary vector of Hamming weight h. For this purpose, we start with two simpler sub-tasks. We first propose an algorithm for evaluating an integer indicator function whose distinguished point is at the boundary: for $B \geq 0$, the function δ_B maps $x = 0$ to 1 and $x \in \{1, \ldots, B\}$ to 0. By relying on Chebyshev polynomials, we obtain an algorithm with multiplicative depth $\frac{1}{2} \log B$ for any fixed output precision and consuming only $O(\log B)$ multiplications. In particular, the number of multiplications is exponentially smaller than the state-of-the-art approach from [LLL+21]. Second, we consider the task of sampling a uniform one-hot vector, i.e., a vector that is 1 in one of its entries and 0 everywhere else. The state-of-the-art approach for this task was introduced in [BEHG20] in the

context of secret single-leader elections, and later refined in [ADK+23]. It was described at a high-level that is independent of the underlying FHE scheme, but if we instantiate it in the context of CKKS, then it may be checked that it has multiplicative depth $\log\log K + O(1)$ where K is the one-hot vector dimension and its main cost component is $O((\log K)(\log\log K))$ vector rotations. By relying on a matrix-vector product, and an indicator function to make the entries binary, we obtain an algorithm with depth $\approx \frac{1}{2}\log\log K$ and which consumes only $O(\log K)$ rotations. Interestingly, if we batch-sample $H \geq 1$ one-hot vectors, the amortized cost per one-hot vector decreases to $O(\log H + (\log K)/H + \sqrt{\log K})$ rotations. Finally, our sparse secret key sampler takes as input $H \geq h$ one-hot vectors, adds the first i for all $i \in [h, H]$ and keeps the first sum that has Hamming weight h. This is achieved by relying on integer indicator functions. The value of H is set so that the sum of H one-hot vectors has Hamming weight $\geq h$ with overwhelming probability: by a balls-and-bins argument, it is shown that H does not need to be much larger than h, allowing the design of an efficient algorithm.

On active security. Our protocol only provides passive security against static corruptions, similarly as prior protocols [MTBH21, MBH23]. It is an interesting question to strengthen it to active security, though an independent one in terms of techniques. Zero-knowledge proofs could ensure that all values communicated by the users have correct forms, and the collaborative generation of randomness may guarantee that the final key material is securely distributed when at least one user is honest. For zero-knowledge proofs, those developed in [CMS+23, HLSS25, HMSS25] could serve as starting points.

1.2 Related works.

A DKG protocol for BGV [BGV12] was given in [RST+22], using a secret-sharing-based protocol. It samples from a ternary distribution by multiplying several secret-shared bits, resulting in a ternary vector with a randomized Hamming weight. However, the method cannot sample a vector with a small fixed weight, which is necessary for efficient CKKS (and BGV) bootstrapping. This suffices in [RST+22] as the application does not require bootstrapping. DKGs relying on secret-sharing are considered in [STH+23, DDEK+23] for TFHE [CGGI16], for which sparse keys are not required. This approach was later formalized and concretely instantiated in [BCD+25] for a t-out-of-n threshold TFHE scheme, against malicious adversaries. Their solution relies on the MPC protocol from [BBG+21, EHL+23] when $\binom{n}{t}$ is small, and the MPC protocol from [DN07] when $\binom{n}{t}$ is large. For 13 parties, they require 54 hours of computation and 50GB of memory usage, with a network bandwidth of 1Gbps. Total data transfer per party is 10.5TB. Considering that the communication complexity of their DKG protocol grows quadratically with n, we expect that their method will be unusable when n is large, for example, $n = 1024$, which is one of the parameters our protocol can facilitate. On the other hand, their DKG protocol achieves active security per se, while for our method, one would need to verify the computation either by reproducing the result or via verifiable

computation, with additional zero-knowledge proofs for the rest of the protocol (threshold decryption).

Another DKG protocol for BFV [Bra12, FV12] is considered in [GKS24] but without any evaluation key. A generic DKG protocol for BGV, BFV, CKKS and TFHE is provided in [BFM+25], but it does not produce sparse secret keys.

Several FHE libraries [MBTPH20, BBB+22] provide threshold versions of CKKS, following the approaches given in [MTBH21, MBH23]. These works and libraries do not provide (non-interactive) bootstrapping and require communication every time a bootstrap is required. The limitation stems from the lack of shared secret keys that are sufficiently sparse.

2 Preliminaries

Vectors are in bold and in column format. For an integer i and a vector $\mathbf{v} = (v_0, \ldots, v_{k-1})^T$, we define the rotation of $\mathbf{v}$ by i positions as $\mathtt{Rot}_i(\mathbf{v}) = (v_i, \ldots, v_{k-1}, v_0, \ldots, v_{i-1})^T$. For a power-of-two integer N and an integer $q > 0$, we let $R = \mathbb{Z}[X]/(X^N + 1)$ denote the $(2N)$-th cyclotomic ring, and $R_q = R/qR$. The notations log and ln respectively refer to the base-2 and base-e logarithms.

2.1 Cheon-Kim-Kim-Song (CKKS)

The CKKS fully homomorphic encryption scheme [CKKS17] enables approximate real/complex arithmetic over encrypted data.

Key generation. The secret key sk is a ring element $s \in R$ that has small coefficients (e.g., ternary). The public key pk is a pair $(p_0, p_1) \in R_q^2$ such that $p_1 \cdot s + p_0 = e$, for some error $e \in R$ that has small coefficients sampled from a distribution χ_e. Key generation also outputs an evaluation key evk, which can be made public and enables homomorphic evaluation of circuits on encrypted data. We will describe its components while discussing homomorphic evaluation.

Encoding/decoding. Encoding and decoding are formatting functions, to present the data (cleartext) in a way that is algebraically convenient for encryption (plaintext), decryption and homomorphic evaluation. CKKS has two main encoding modes: coefficients encoding and slots encoding. The coefficients encoding method takes a vector $\mathbf{m} \in \mathbb{R}^N$ as input, views it as a polynomial $\mu(X) = \sum_{0 \le i < N} m_i X^i \in \mathbb{R}[X]/(X^N + 1)$, multiplies it by some sufficiently large real scaling factor $\Delta > 0$ that drives precision, and rounds. In other words, we compute $\lfloor \Delta \cdot \mu(X) \rceil$. Decoding divides by Δ.

The slots encoding mode 'packs' a vector $\mathbf{m} = (m_i)_{0 \le i < N/2} \in \mathbb{C}^{N/2}$ into a polynomial in $\mathbb{R}[X]/(X^N+1)$ so that the arithmetic operations over polynomials translate to the element-wise counterpart operations over vectors. Specifically, given a vector $\mathbf{m}$, we first compute $\widetilde{\mu}(X) \in \mathbb{R}[X]/(X^N+1)$ such that $\widetilde{\mu}(\zeta_{2N}^{5^i}) = m_i$ for all $0 \le i < N/2$, where ζ_{2N} is a primitive $(2N)$-th root of unity and the $\zeta_{2N}^{\pm 5^i}$'s form the multiplicative group of all primitive $(2N)$-th roots of unity. As $\widetilde{\mu}$ is assumed to have real coefficients, the interpolation above uniquely determines

it. Then, this polynomial $\widetilde{\mu}$ is discretized into $\mu = \lfloor \Delta \cdot \widetilde{\mu} \rceil$. This may introduce a small error to the message, and the scaling factor Δ should be chosen sufficiently large to keep sufficient precision. Conversely, given a slots encoding $\mu \in R$, the decoding process outputs a vector $\mathbf{m} = (m_i)_{0 \leq i < N/2} \in \mathbb{C}^{N/2}$ such that $m_i = \mu(\zeta_{2N}^{5^i})/\Delta$ for all $0 \leq i < N/2$. Note that the rotation by $i \geq 0$ indices (resp. conjugation) of $\mathbf{m}$ can be instantiated on $\widetilde{\mu}$ with the automorphism $X \mapsto X^{5^i}$ (resp. $X \mapsto X^{-1} = -X^{N-1}$).

Encryption. Let $\mathsf{pk} = (p_0, p_1) \in R_q^2$ be the public key. CKKS encryption of a plaintext $\mu \in R_q$ is obtained by sampling $u \in R$ with coefficients in $\{-1, 0, 1\}$ and small error polynomials $e_0, e_1 \in R$ from a specified distribution, and defining

$$\mathsf{Enc}_{\mathsf{pk}}(\mu) = (u \cdot p_0 + \mu + e_0, u \cdot p_1 + e_1) \in R_q^2 \ .$$

Ciphertexts and decryption. The ciphertext modulus q depends on the computations performed so far. CKKS involves a chain $(Q_\ell)_{0 \leq \ell < L}$ of increasing moduli that define so-called levels. A level-ℓ ciphertext is of the form $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$. If $\mathsf{sk} = s \in R$ is the secret key, then ct decrypts to

$$\mathsf{Dec}_s(\mathsf{ct}) = \langle \mathsf{ct}, (1, s) \rangle = c_0 + c_1 s \pmod{Q_\ell} \ . \tag{1}$$

If this is equal to $\mu + e$ for some small-magnitude noise $e \in R$, we say that ct decrypts to $\mu \in R$. If the scaling factor Δ is specified, we also say that ct is an encryption of the cleartext $\mathbf{m}$, the decoding of μ, with scaling factor Δ.

Key-switching. Let $\mathsf{ct} = (c_0, c_1) \in R_{Q_\ell}^2$ be a level-ℓ encryption of a plaintext $\mu \in R$ under a secret key s. We define a *key-switching key* from s to s' to be an encryption $(\mathsf{swk}_0, \mathsf{swk}_1) \in R_{PQ_\ell}^2$ of Ps under s' with modulus PQ_ℓ, where $P \geq Q_\ell$ is an *auxiliary modulus*.[3] The key switching procedure consists in computing

$$\mathsf{ct}' = (c_0 + \lfloor \mathsf{swk}_0 \cdot c_1 / P \rceil \, , \lfloor \mathsf{swk}_1 \cdot c_1 / P \rceil) \in R_{Q_\ell}^2 \ .$$

It then holds that ct' is an encryption of μ under the secret key s'.

Native homomorphic operations. CKKS supports element-wise addition, subtraction, and multiplication (denoted by $\odot$) of cleartext vectors. Using the automorphisms of R, it also supports rotations and element-wise conjugation. We note that multiplications, rotations, and conjugations require switching keys; the switching key required for the multiplication, from s^2 to s, is called *relinearization key*. The set of all required switching keys is the *evaluation key*. If given as inputs ciphertexts at level $0 \leq \ell < L$, the output of addition, subtraction, rotation, and conjugation has level ℓ: these operations do not consume level. If given as inputs ciphertexts at level $0 < \ell < L$, the output of multiplication has level $\ell - 1$. We refer to [CKKS17] for detailed descriptions of these algorithms.

Bootstrapping. As a multiplication consumes a level, one can perform at most $L - 1$ sequential multiplications before reaching level 0. To allow further

[3] For simplicity, we assume that the gadget rank dnum [BEHZ16] is set to 1. Our results extend to arbitrary dnum, in which case it suffices to have $P^{\mathsf{dnum}} \geq Q_\ell$.

multiplications, a ciphertext-maintenance algorithm known as *bootstrapping* can be applied to a low-level ciphertext [CHK+18]. This creates a new ciphertext for the same cleartext (up to a small numerical error), at a higher level. Bootstrapping raises the decryption identity (1) from modulus Q_0 to modulus Q_{L-1} as

$$c_0 + c_1 s = \mu + e + I \cdot Q_0 \pmod{Q_{L-1}} \ , \tag{2}$$

where $I \in R$ is an integer polynomial with small-magnitude coefficients. Assuming that the coefficients of $\mu + e$ are small compared to Q_0 and that s is ternary of Hamming weight h, expanding (2) shows that the coefficients of $I \cdot Q_0$ are rounded sums (up to signs) of $h + 1$ coefficients of c_0 and c_1. Heuristically assuming that the coefficients of c_0 and c_1 are uniformly distributed, this implies that, with some probability $1 - p$, the coefficients of I have absolute values bounded from above by a certain constant $\mathcal{K}_p$. In the sequel, we set $p \leq 2^{-128}$ and write $\mathcal{K}$ instead of $\mathcal{K}_p$. Else, one can mount attacks against the security of threshold-CKKS (see, e.g., [CCP+24, Sec. 3]).

Typically, a bootstrapping process for CKKS ciphertexts consists of four components: Slots2Coeffs, ModRaise, Coeffs2Slots and EvalMod. In this work, we focus on Slots2Coeffs-first bootstrapping, which uses these components in that specific order. We provide a concise explanation of each step in App. A.

The most depth-consuming step in bootstrapping is EvalMod. It evaluates a polynomial approximation for the modular function $x/Q_0 \bmod 1$ over a union of intervals $\bigcup_{-\mathcal{K} \leq k \leq \mathcal{K}} [k-\alpha, k+\alpha]$ for a small $\alpha \in \mathbb{R}$. It consumes a number of levels that is essentially linear in $\log \mathcal{K} = O(\log h)$. For this reason, in the competitive bootstrapping methods, an ephemeral sparse secret is often considered for bootstrapping [BTH22]: one switches the main secret key to the sparse secret key before ModRaise, and switches the sparse key to the main key after ModRaise. These key switchings are made possible by disclosing public key material as part of the evaluation key. Overall, the bound $\mathcal{K}$ depends on the temporary sparse key and depth can be saved in EvalMod.

It was noted in [CHK+21] that Slots2Coeffs can be skipped if the encoding of the input ciphertext is already in coefficients mode, and the encoding of the ciphertext output by bootstrapping is in slots mode. We refer to the original bootstrapping by Bootstrap, and to this bootstrapping variant by HalfBootstrap.

Security. The security of CKKS relies on Ring-LWE [SSTX09, LPR10] for secret distributions corresponding to the secret keys (including the sparse one) and encryption randomness u, and error distributions chosen in key generation and encryption. It is also assumed that Ring-LWE remains secure when the adversary is given encryptions of specific functions of the secret key, as well as encryptions of the ephemeral sparse key under the main key and vice-versa. This circularity assumption allows one to make the evaluation key public.

2.2 Threshold-FHE

A threshold-FHE scheme (ThFHE) is an extension of FHE in which the secret key sk is distributed between n parties $P_1, \ldots, P_n$, each holding a share

$\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ respectively. To decrypt, parties partake in an interactive decryption protocol that requires some number t of them to participate. We focus on the n-out-of-n setting, as a t-out-of-n protocol may be derived via [MBH23].

Definition 1. *A ThFHE scheme is a tuple of PPT protocols* $\mathsf{ThFHE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *with the following functionalities:*

- $\mathsf{Setup}(1^\lambda, 1^n)$ *is an algorithm that takes the security parameter* λ*, a number of parties* n*, and outputs public parameters* pp*. It defines a plaintext space* $\mathfrak{M}$ *and a ciphertext space* $\mathfrak{C}$*.*
- $\mathsf{KeyGen}(\mathsf{pp})$ *is a protocol that takes public parameters* pp*, and returns a public key* pk*, an evaluation key* evk *and secret shares* $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ *(one per participant). It can be run by a trusted dealer, or be an interactive protocol between the* n *parties. All protocols below implicitly take* pp *as inputs.*
- $\mathsf{Enc}(\mathsf{pk}, \mu)$ *is an algorithm that takes a public key* pk *and a plaintext* $\mu \in \mathfrak{M}$*, and returns a ciphertext* $\mathsf{ct} \in \mathfrak{C}$*.*
- $\mathsf{Eval}(\mathsf{evk}, \mathcal{C}, \mathsf{ct}_1, \ldots, \mathsf{ct}_\ell)$ *is an algorithm that takes an evaluation key* evk*, a circuit* $\mathcal{C} : \mathfrak{M}^\ell \to \mathfrak{M}$ *for an arbitrary* $\ell \geq 0$ *and* ℓ *ciphertexts* $\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell \in \mathfrak{C}$ *encrypting* $m_1, \ldots, m_\ell \in \mathfrak{M}$*, and returns an encryption of* $\mathcal{C}(m_1, \ldots, m_\ell)$*.*
- $\mathsf{Dec}(\mathsf{ct}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n)$ *is an interactive protocol that takes a ciphertext* $\mathsf{ct} \in \mathfrak{C}$ *and secret key shares* $\mathsf{sk}_1, \ldots, \mathsf{sk}_n$ *(one per participant), and returns a plaintext* $\mu' \in \mathfrak{M} \cup \{\perp\}$*.*

Our goal being to design a DKG protocol, we do not define the security of encryption and refer to [BGG+18]. However, we give the security model for the DKG protocol, formalizing that the result of the interactive protocol is as good as that of the following ideal protocol run by a trusted dealer:

- Run (non-threshold) $(\mathsf{pk}, \mathsf{sk}, \mathsf{evk}) \leftarrow \mathsf{KeyGen}_{\mathsf{CKKS}}$;
- Sample an additive secret sharing $(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$ of sk;
- Return $(\mathsf{pk}, \mathsf{evk}, \mathsf{sk}_i)$ to P_i, for all $i \leq n$.

We let $\mathsf{View}_{\mathcal{A}}^{\mathsf{ideal}}$ denote the view of an adversary that corrupts $n-1$ parties in this ideal protocol, i.e., $(\mathsf{pk}, \mathsf{evk}, \mathsf{sk}_2, \ldots, \mathsf{sk}_n)$ assuming (without loss of generality) that the adversary corrupts all parties except P_1. Like prior works [MTBH21, MBH23], we only consider static corruptions and leave open the question of achieving security against adaptive corruptions. We then define security of a DKG protocol as the fact that the whole view $\mathsf{View}_{\mathcal{A}}^{\mathsf{real}}$ of an adversary in the interactive protocol can be simulated from its ideal view $\mathsf{View}_{\mathcal{A}}^{\mathsf{ideal}}$.

Definition 2 (DKG Security). *We consider* n *parties* $P_1, \ldots, P_n$ *running* $\mathsf{KeyGen}(\mathsf{pp})$*, where* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^n)$*. Let* $\mathcal{A}$ *be a semi-honest adversary with* $n-1$ *static corruptions, and* $\mathsf{View}_{\mathcal{A}}^{\mathsf{real}}$ *be its view of an execution of the protocol. We say that* $(\mathsf{Setup}, \mathsf{KeyGen})$ *is a simulation-secure DKG protocol if there is an efficient simulator* $\mathsf{Sim}_{\mathsf{DKG}}$ *that takes as input* $\mathsf{View}_{\mathcal{A}}^{\mathsf{ideal}}$ *and returns a simulated transcript* $\mathsf{tr}_{\mathsf{sim}}$ *that is computationally indistinguishable from* $\mathsf{View}_{\mathcal{A}}^{\mathsf{real}}$*.*

3 Homomorphic Sparse Key Sampling

In this section, we describe two homomorphic secret key sampling algorithms. The first one (Fig. 3) produces a ternary key with prescribed Hamming weight (sparse key), while the second one (Sec. 3.4) produces a ternary key with a typically much higher weight that is bounded from below (less-sparse key), or a generic ternary key with weight $\approx 2N/3$. This is the core computational component of our DKG protocol.

We describe the algorithms in a model of computation that corresponds to CKKS operations with slots encoding: we have vectors over $\mathbb{R}^{N/2}$ (where N is the Ring-LWE degree), on which we can perform element-wise additions and multiplications, as well as rotations of coordinates. We shall assume, for simplicity, that our algorithms aim at generating vectors of dimension $K \leq N/2$; they extend to larger K by using several vectors of slots (i.e., ciphertexts). In our DKG protocol, we will use $K = N$. We further assume that K is a power of 2. We identify the key that we aim at creating and the vector of its coefficients.

We outline our strategy for the sparse case. If we target Hamming weight h, we build $H > h$ one-hot vectors $(\mathbf{v}_i)_{0 \leq i < H}$ (Sec. 3.2) and deduce $H - h + 1$ candidate vectors $\mathbf{w}_j = \sum_{0 \leq i \leq j} \mathbf{v}_i$, for $h \leq j \leq H$. For a well-chosen H (see App. B.6), one of these candidates has weight h with overwhelming probability. As they may contain coefficients ≥ 2, we convert the candidates into binary vectors: we use the indicator function described in Sec. 3.1 for this task. We select a right candidate (see Sec. 3.3), and make it ternary by multiplying it by a uniform vector with coordinates in $\{-1, 1\}$. Implementing this strategy requires H one-hot vectors, hence $H \log K$ random bits; we assume that $H \log K \leq N/2$ so that these random bits can be encrypted into a single slot-encoded CKKS ciphertext. More bits are required to obtain the signs of the coordinates of the output sparse ternary. Here, we assume that all these bits are given as input, and refer to Sec. 4.1 for their sampling in the DKG protocol.

3.1 Evaluating an Integer Indicator Function

We consider integer indicator functions, where the input x is an integer that belongs to a prescribed interval $[0, B]$ and the distinguished point u is an integer:

$$\delta_{B,u}(x) = \begin{cases} 0 & \text{if } x \in \{0, \ldots, B\} \setminus \{u\} \ , \\ 1 & \text{if } x = u \ . \end{cases}$$

We are interested in the element-wise evaluation of $\delta_{B,u}$ on vectors, but, as slots are acted upon in parallel, it suffices to consider a single slot. The direct approach based on Lagrange interpolation is numerically unstable and only moderately efficient. We distinguish the interior case, where the distinguished point u is in $\{1, \ldots, B-1\}$, from the boundary case, where it is either 0 or B. For the interior case, we rely on the classical approach of minimax approximation using the (multi-interval) Remez algorithm. For the boundary case, we introduce a more efficient solution that exploits properties of the Chebyshev polynomials.

Interior integer indicator function. Let $u \in \{1, \ldots, B-1\}$. We consider the step function $f_{B,u}(x) = \mathbf{1}_{[u-1/2,u+1/2]}$ defined over the domain $\cup_{0\le i\le B}[i-\eta, 1+\eta]$, which is 0 over $[0, u-1/2) \cup (u+1/2, B]$ and 1 in $[u-1/2, u+1/2]$ for some 'threshold' η. We fix an approximation degree d and 'threshold' η and use the multi-interval Remez algorithm [LLL+21] to obtain a degree-d polynomial approximation to $f_{B,u}$. The evaluation of this polynomial requires $\lceil \log(d+1) \rceil$ levels and its cost is dominated by $O(\sqrt{d})$ ciphertext-ciphertext multiplications (using the Paterson-Stockmeyer algorithm).

Note that the interior setting reduces to the boundary setting: the change of variable $x \mapsto (u-x)^2$ sends u to 0 and $[0, B]$ to $[0, \max(u^2, (B-u)^2)]$. This however increases level consumption by one unit for the change of variable, and significantly enlarges the interval under study (which typically also impacts level consumption). Using the method below for boundary indicator functions can decrease the number of ciphertext-ciphertext multiplications but, depending on the context, this may not compensate for the possibly higher level consumption.

Boundary integer indicator function. Without loss of generality, we assume that $u = 0$. We define the Chebyshev polynomials $(T_d)_{d\ge 0}$ by $T_0 = 1$, $T_1 = X$ and $T_{d+1} = 2XT_d - T_{d-1}$ for $d \ge 1$, and introduce $W_{B,d} = T_d\left(\frac{B+1-2X}{B-1}\right) / T_d\left(\frac{B+1}{B-1}\right) \in \mathbb{R}[X]$. With these notations, we have the following result, whose proof is in App. B.2.

Lemma 1. *For all integers $B \ge 2$, $d \ge 0$, we have $W_{B,d}(0) = 1$ and*

$$\forall x \in \{1, \ldots, B\} : \ |W_{B,d}(x)| \le T_d\left(\frac{B+1}{B-1}\right)^{-1} \le 2\left(\frac{\sqrt{B}-1}{\sqrt{B}+1}\right)^d .$$

In particular, for $\varepsilon > 0$ and $d \ge \frac{\sqrt{B}+1}{2}\ln\left(\frac{2}{\varepsilon}\right)$, we have $W_{B,d}(0) = 1$ and $|W_{B,d}(x)| \le \varepsilon$ for all $x \in \{1, \ldots, B\}$.

Algorithm ScaledChebyshev from Fig. 1 efficiently computes $W_{B,d}$ when d is a power of 2. It can be extended to the general case by using the identity $T_{2d+1} = 2T_dT_{d+1} - X$.

Lemma 2. *Given m, x and B, the* ScaledChebyshev *algorithm computes $W_{B,2^m}(x)$ using $m+1$ multiplicative levels and $O(m)$ ciphertext-ciphertext multiplications.*

The algorithm is based on the recurrence relation $T_{2\ell}(X) = 2T_\ell(X)^2 - 1$ for $\ell \ge 0$. One level is used for the transform $X \leftarrow (B+1-2X)/(B-1)$,and m levels are used for the m iterations of this recurrence relation.

For improved stability, we integrate the denominator $T_d((B+1)/(B-1))$ at the start rather than at the end, under the form:

$$c = \left(T_d\left(\frac{B+1}{B-1}\right)\right)^{-\frac{1}{d}}, \ w_1 = c \cdot \frac{B+1-2X}{B-1}, \ w_{k+1} = 2w_k^2 - c^{2^{k+1}} .$$

In terms of numerical stability, a strong advantage of using $W_{B,d}$ is that the bound on $|W_{B,d}(x)|$ from Lemma 1 actually holds for all real numbers $x \in [1, B]$:

```
ScaledChebyshev(x, m, B)
1 : res ← c · (B + 1 − 2x)/(B − 1)
2 : for 1 ≤ i ≤ m
3 :     c ← c^2
4 :     res ← 2 · res^2 − c
5 : return res
```

Fig. 1: Algorithm for computing $W_{B,2^m}$.

this handles the inaccuracy of the input, which is inherent to the approximate nature of CKKS computations. The situation is more complex around 0 where the derivative of $W_{B,d}$ is $\approx d/\sqrt{B}$. We refer to App. B.3 for a detailed discussion, and report experimental behaviour in Table 1 for illustration.

We finally observe that ScaledChebyshev has interesting properties compared to classical polynomial evaluation algorithms, which make it possible to bootstrap while evaluating the polynomial: first, it does not expand the number of ciphertexts (oppositely to Paterson-Stockmeyer); second, if the input is in $[0, B]$, all intermediate values are guaranteed to lie in $[-1, 1]$.

Table 1 compares the behaviour of the Chebyshev method with multi-interval Remez. It demonstrates comparable precision, with a smaller evaluation cost. In addition, the Chebyshev method only requires an assumption on input precision at u, while multi-interval Remez requires a guaranteed precision around every point. The table also shows the expected degradation at the boundary point of the Chebyshev method applied to boundary indicator function. In that setting, one subsequent application of the map $x \mapsto 1 - (x-1)^2$ doubles the precision while degrading the precision at other points by ≈ 1 bit.

u	d	$W_{B,d}$ prec. (at u)	$W_{B,d}$ prec. (at $\neq u$)	Remez prec.	$W_{B,d}$ #KS	Remez #KS
0	16	8.00	10.8	12.8	4	7
	32	7.00	22.6	17.9	5	11
4	32	12.9	6.71	10.2	6	11
	64	17.6	14.4	21.7	7	15
8	32	18.0	10.6	9.19	6	11
	64	17.0	22.2	20.5	7	15

Table 1: Comparison between $W_{B,d}$ and Remez of degree d for indicator functions, for $B = 16$ and various u, d. We assume input precision of 10 bits; at $u \neq 0$, we consider $W_{B,d}((u-x)^2)$ of degree $2d$.

3.2 Sampling One-Hot Vectors

Sampling a single one-hot vector. We consider the sampling of a uniform one-hot vector of dimension $K = 2^k$. As a one-hot vector is uniquely determined by the index of its non-zero entry, information-theoretically, a uniform one-hot vector can be sampled using k uniform bits. We describe an algorithm that converts k bits $b_0, \ldots, b_{k-1}$ stored in a vector into a K-dimensional one-hot vector with non-zero index $b = \sum_{0 \le i < k} b_i 2^i$.

We define $\beta_j(i)$ as the j-th bit in the binary expansion of i, and consider the following vectors $\mathbf{v}_j$ for $0 \le j < k$:

$$\mathbf{v}_j = (1 - \beta_j(i) + b_j \cdot (2\beta_j(i) - 1))_{0 \le i < K}^T \quad .$$

The i-th entry of $\mathbf{v}_j$ is 1 if the j-th bit of i is b_j and 0 otherwise.

The vector we want to build is the coordinate-wise product of the $\mathbf{v}_j$'s. This solution consumes $O(\log^2 K)$ rotations (to create the $\mathbf{v}_j$'s), $O(\log K)$ ciphertext-ciphertext multiplications, and $1 + \log\log K$ multiplicative levels (for the binary tree). A more efficient algorithm to compute the product was proposed in [BEHG20] (see also [ADE+23, NHY+25]): it first generates the vectors $\mathbf{e}_i = (1 - b_i, b_i, 0, \ldots, 0)^T$ for $0 \le i < k$ and then applies a binary tree of tensor products. The tree consumes the same number of levels and multiplications as the previous approach but requires only $O(\log K)$ rotations per level, for a total of $O((\log K)(\log\log K))$ rotations (see App. B.1).

We describe an alternative approach, using sums rather than products. For our parameters and asymptotically, this approach is more efficient than the previously described tensoring methods –see Table 2. It relies on the following observation.

Lemma 3. *Let $\mathbf{v}, \mathbf{v}' \in \{-1, 1\}^k$. Then the inner product $\langle \mathbf{v}, \mathbf{v}' \rangle \in [-k, k]$ is congruent to $k \bmod 2$, and $\mathbf{v} = \mathbf{v}'$ if and only if $\langle \mathbf{v}, \mathbf{v}' \rangle = k$.*

We build the $K \times k$ matrix $\mathbf{M} = (1 - 2\beta_j(i))_{0 \le i < K, 0 \le j < k}$. Given $\mathbf{b} = (b_j)_{0 \le j < k} \in \{0, 1\}^k$, we compute

$$\mathbf{w} = \frac{k}{2} \cdot \mathbf{1}_K - \frac{1}{2} \cdot \mathbf{M} \cdot (\mathbf{1}_k - 2\mathbf{b}) \quad , \tag{3}$$

where the notation $\mathbf{1}_d$ refers to the all-1 vector of dimension d for any $d \ge 0$. The ℓ-th coordinate of $\mathbf{w}$ is

$$w_\ell = \frac{1}{2} \left(k - \langle (1 - 2\beta_j(\ell)_{0 \le j < k}, (1 - 2b_j)_{0 \le j < k} \rangle\right) \quad .$$

Using Lemma 3, we deduce that w_ℓ is an integer in $[0, k]$, and that $w_\ell = 0$ if and only if $b_j = \beta_j(\ell)$ for $0 \le j < k$. As $0 \le \ell < 2^k$, this is equivalent to $\ell = \sum_{0 \le j < k} b_j 2^j$.

As a result, the desired one-hot vector can be obtained from $b_0, \ldots, b_{k-1}$ by evaluating Eq. (3) and then the indicator function that sends every $x \in \{1, \ldots, k\}$ to 0, and $x = 0$ to 1. We now analyze the cost of this approach. The proof of the following theorem is deferred to App. B.4.

Theorem 1. *Given a vector* $(b_0, \ldots, b_{k-1}, 0, \ldots, 0)^T \in \{0,1\}^K$*, the* OneHotVecGen *algorithm from Fig. 2 computes the* K*-dimensional one-hot vector with a nonzero entry in index* $\sum_{0 \le j < k} b_j 2^j$ *using* $O(\log K)$ *rotations,* $O(\log \log K)$ *ciphertext-ciphertext multiplications, and* $\frac{1}{2} \log \log K + O(1)$ *multiplicative levels.*

Method	size	# of levels	# of KS
[BEHG20]	2^{16}	5	95
This work	2^{16}	5	23
[BEHG20]	2^k	$1 + \log k$	$(2 + \log k)k - 1$
This work	2^k	$2 + \log((\log \varepsilon)^2 k)/2$	$k(1 + o(1)) + \log \log(\varepsilon^{-1})$

Table 2: Comparison between our approach and [BEHG20] for generating one-hot vectors of dimension 2^{16}, on input $(b_0, \ldots, b_{15})$. SIMD packing is not considered in [BEHG20]; these estimates are based on the description of Appendix B.1. For $2^k = 2^{16}$, the output of our method is expected to be within 2^{-8} of the correct result; in the general case, we assume target precision ε.

OneHotVecGen$(\mathbf{b} = (b_0, \ldots, b_{k-1}, 0, \ldots, 0)^T)$

1 : $\mathbf{b}_{\text{ext}} \leftarrow \mathbf{b}$

2 : **for** $0 \le i \le \log(K/k) - 1$

3 : $\quad \mathbf{b}_{\text{ext}} \leftarrow \mathbf{b}_{\text{ext}} + \text{Rot}_{2^i \cdot k}(\mathbf{b}_{\text{ext}})$

4 : $(\mathbf{v})_\ell \leftarrow \frac{k}{2} \cdot \mathbf{1}_{\frac{K}{k}} - \mathbf{M}_\ell \cdot \left(\frac{\mathbf{1}_k}{2} - (\mathbf{b}_{\text{ext}})_\ell \right)$ // for $0 \le \ell < K/k$ in SIMD

5 : **return** $\delta_{k,0}(\mathbf{v})$ // in SIMD

Fig. 2: One-hot vector sampler.

Sampling several one-hot vectors. The dominant cost of the previous algorithm comes from the initial copying task (Steps 1-3). When generating several one-hot vectors, this part can be partly batched using replication-type techniques from [HS14] to reduce the initial cost from $O(\log K)$ rotations down to $o(\sqrt{\log K})$ rotations per one-hot vector – the dominant component then becomes Step 4. We describe this variant in App. B.5.

3.3 Sampling a Sparse Vector

Let h denote the target Hamming weight and K the dimension. A naive strategy would be to sample and sum h one-hot vectors. Unfortunately, as some one-hot

vectors may collide, this approach makes it difficult to ensure exact Hamming weight h. Further, when a collision occurs, the resulting vector is not binary. We address these difficulties in the FixedHWSampler algorithm from Fig. 3, which takes as input H one-hot vectors for some $H \geq h$. To prove its (probabilistic) correctness, we start with a combinatorial lemma [Ram88].

Lemma 4. *Let $S(n,k) = \frac{1}{k!}\sum_{0\leq i\leq k}(-1)^{k-i}\binom{k}{i}i^n$ denote the Stirling number of the second kind for $n \geq k$. Then, the probability that H uniformly thrown balls go into exactly i bins out of a total of K bins is given by $\frac{i!\cdot S(H,i)}{K^H}\binom{K}{i}$.*

Theorem 2. *Let $\varepsilon > 0$, h and H be such that*

$$\sum_{i=h}^{H} \frac{i!\cdot S(H,i)}{K^H}\cdot\binom{K}{i} \geq 1-\varepsilon. \tag{4}$$

Then, with probability $\geq 1-\varepsilon$, FixedHWSampler *returns a $\{0,1\}$-vector with Hamming weight equal to h.*

We prove in App. B.6 that (4) can be replaced by the stronger but explicit bound $H - h \geq (h + \ln(1/\varepsilon))/\ln(K/h)$. We stress that in our application, the dimension K is much larger than h, so H only needs to be slightly larger than h.

Proof. In view of the definition of $\delta_{H-h,0}$, the vector $\mathbf{v}_i$ contains 1 in coordinates where $\mathbf{tmp}_i$ is nonzero, and 0 otherwise. Steps 4-6 compute h_i, the weight of $\mathbf{v}_i$, i.e., the number of non-zero coordinates of $\mathbf{tmp}_i$. As $\mathbf{tmp}_i - \mathbf{tmp}_{i-1} = \mathbf{e}_i$ is a one-hot vector, we note in particular that $h_i - h_{i-1} \in \{0,1\}$ and $h_i \leq i$.

Finally, we have $w_i = 1$ if and only if $i - h_i = i - h$, or equivalently $h_i = h$. As a consequence, we have $(h_i - h_{i-1})w_i = 1$ if and only if $h_i = h$ and $h_{i-1} = h_i - 1$; this can occur for at most one index i, thus the sum $\mathbf{res} = \sum_{h-1\leq i\leq H}(h_i - h_{i-1})w_i\mathbf{v}_i$ is equal to 0 if none of the $\mathbf{v}_i$'s has weight h, and else to $\mathbf{v}_i$, where i is the first index such that the weight of $\mathbf{v}_i$ is h.

To prove correctness, it suffices to check that the inputs of the δ functions lie in correct intervals. This follows from the definition of $\mathbf{tmp}_i$ for $\delta_{H-h,0}(\mathbf{tmp}_i)$. For the second call to δ, the fact that $h_i - h_{i-1} \in \{0,1\}$ implies that $i - h_i$ is a non-decreasing sequence. We thus have $i - h_i \in [0, H - h_H]$ throughout the algorithm. A sufficient condition for correctness is thus $h_H \geq h$: Lemma 4 and our assumption on H guarantees that this holds with probability $\geq 1-\varepsilon$.

In case of failure (i.e., if $h_H < h$), we return garbage rather than the zero vector; this happens with negligible probability but can be fixed by replacing Step 7 by $w_i \leftarrow \delta_{H,h}(h_i)$, at a higher cost for evaluating δ.

To extend to the ternary case, it suffices to multiply a uniform sign vector with a binary vector of the desired weight. This requires K extra bits of randomness. We wrap up all the steps into the SparseKeySampler algorithm of Fig. 3. In this case, we take as input two binary random vectors $\mathbf{b}_0, \mathbf{b}_1$ in $\{0,1\}^K$. In order to have enough random bits, SparseKeySampler requires that $Hk \leq K$.

This step is the one requiring the most multiplicative depth in the entire sampling procedure. We note that the level consumption can be reduced if

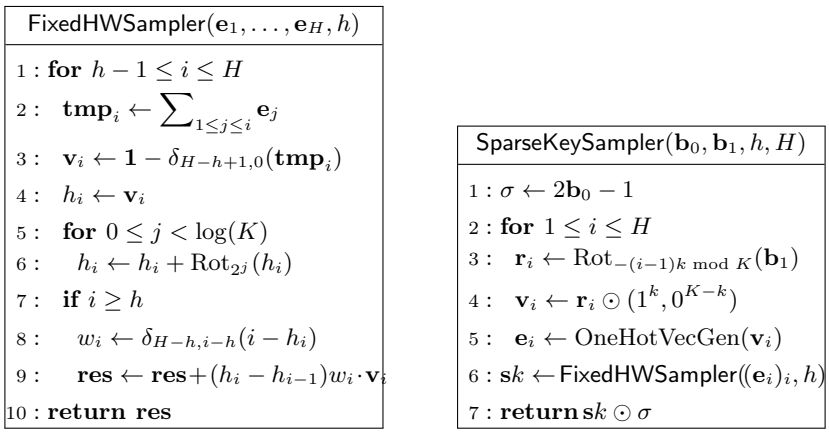


Fig. 3: Left: Binary sampler with prescribed Hamming weight; Right: Sparse ternary sampler. SparseKeySampler requires that $Hk \le K$.

we force the secret to follow a *block structure*, where the non-zero coefficients are forced to be in some prescribed intervals (such keys have been considered in [LMSS23, CHKS25]). We describe a dedicated sampling algorithm in App. B.8.

3.4 Sampling Less-Sparse and Dense Vectors

So far, we gave a sampler for sparse vectors with efficient homomorphic evaluation. The corresponding sparse keys are only used for the sparse secret encapsulation during bootstrapping. For the remaining homomorphic computations, one still needs another set of keys. Depending on the precise requirements of the computations that will be performed, this key can be *less sparse*[4] or *dense* (each coefficient is uniform in $\{-1, 0, 1\}$). In the dense case, it suffices to sample a uniform vector mod 3; we let DenseSampler denote the corresponding algorithm.

We now outline a strategy to sample a less-sparse key. Assume we want to sample a vector in $\{-1, 0, 1\}^K$ with Hamming weight bounded from below by h'.[5] Let $\rho \ge 2$, and $\mathbf{u} \in \mathbb{Z}_\rho^N$ be uniform. By mapping two values of $\mathbb{Z}_\rho$ (we shall use 0 and $\rho - 1$ to enjoy the performance of ScaledChebyshev) to 1 and -1 and the others to 0, we expect to obtain a ternary vector with weight $\approx 2N/\rho$. A probabilistic analysis can be found in App. B.7. For $h' = 192$ and $N = 65536$, we find that for $\rho = 200$, the vector $(\delta_{\rho-1,0}(u_i) - \delta_{\rho-1,\rho-1}(u_i))_{0 \le i < N}$ has weight in $[192, 1024]$ with probability $\ge 1 - 2^{-128}$. Overall, we obtain an algorithm LessSparseSampler taking as input ρ and a uniform vector in $\mathbb{Z}_\rho^K$ (obtained via RandGen), and sampling a less-sparse key. It is formally described in App. C.4.

This optimization provides control on the rescaling error, which tends to be the main source of error in CKKS computations. It grows as $\approx \sqrt{h}$ where h is

[4] In libraries such as [Cry22, MBTPH20], the default weight is often 192.
[5] A lower bound for the Hamming weight is required for Ring-LWE security.

the weight of the secret key. We use the Irwin Hall model to explicitly bound the precision loss [BTH22]. Recall that the rescaling error is of the form $\mathbf{e} \cdot \mathbf{s}$ where $\mathbf{s}$ is the (less-sparse) ternary secret key and $\mathbf{e}$ is modeled as a ring element with uniform entries in $[-1/2, 1/2]$. If the Hamming weight of $\mathbf{s}$ is 192 (resp. 1024), then the probability that an entry of $\mathbf{e} \cdot \mathbf{s}$ has magnitude above 53 (resp. 128) is $\leq 2^{-128}$. The precision loss is hence $\approx \log_2(128/53) \leq 1.28$ bits.

4 A DKG Protocol for Efficient Threshold-CKKS

The purpose of this section is to describe our DKG protocol for threshold-CKKS, formally given in Fig. 6. Specifically, we show that our sparse (and less-sparse) vector generation algorithm can be plugged into existing threshold-CKKS protocols [MTBH21, KLSW24, MBH23]. We focus on the n-out-of-n setting, and recall that a t-out-of-n ThFHE scheme can be derived from an n-out-of-n scheme through share re-sharing [MBH23]: each party shares their partial secret key in a t-out-of-n way with the other parties at the end of the key generation protocol. We emphasize that this does not add any round of communication to the DKG, as the key shares are recovered by parties at the end of Round 3 (the goal of Round 4 is to complete the evaluation key); and therefore the parties can t-out-of-n share their partial secret key during the fourth and final round.

We consider n parties, denoted by $P_1, \ldots, P_n$, and our goal is to distribute between them the key material for CKKS. Concretely, we aim to give them: (1) A public key and an evaluation key associated to a less-sparse secret key, (2) Shares of the less-sparse secret keys, as partial decryption keys, and (3) Key-switching keys for switching between the sparse and less-sparse keys.

4.1 Sub-protocols

We first provide sub-protocols $\mathsf{RandBitsGen}$, ShGen, $\mathsf{PubKeyGen}$, $\mathsf{RelinKeyGen}$, and $\mathsf{KSKeyGen}$ that will simplify the DKG description. The last three require parties to generate common uniform ring elements during the sub-protocol. We rely on a hash function $\mathcal{H}$ modeled as a random oracle to let each party non-interactively generate them (e.g., using a unique public identifier for the sub-protocol execution). We could alternatively assume that these ring elements are provided in a common reference string contained in the public parameters. Due to space constraints, $\mathsf{PubKeyGen}$, $\mathsf{RelinKeyGen}$, and $\mathsf{KSKeyGen}$ are provided in App. C.

Random bits generation. We describe in Fig. 4 an algorithm, $\mathsf{RandBitsGen}$, to create 2ℓ slots-encoded CKKS encryptions of N-dimensional vectors $\mathbf{b}_0, \ldots, \mathbf{b}_{\ell-1}$, where each entry is uniform in $\mathbb{Z}_2$ (recall that N is the ring degree). We describe in App. C.3 a variant taking as extra argument an integer ρ, $\mathsf{RandGen}$, which generates a high-level slots-encoded encryption of a uniform vector in $\mathbb{Z}_\rho^{N/2}$

In plain, a uniform vector in $\mathbb{Z}_\rho^N$ can be obtained by having each party draw a uniform vector over $\mathbb{Z}_\rho^N$ and add these vectors modulo ρ: as soon as one vector

contributing to the sum is uniform, so is the result. To implement modulo-ρ reduction, we encode the plaintexts in the most significant bits of the ciphertexts, *à la* Regev [Reg05]. If Q denotes the ciphertext modulus, a cleartext $m \in R_\rho$ is encoded as a plaintext $\mu = \lfloor \frac{Q}{\rho} \cdot m \rceil \in R_Q$. Note that this can also be interpreted as CKKS encoding/decoding for coefficients mode and scaling factor $\Delta = Q/\rho$. Encryption of plaintexts is as usual. It can be seen that addition over the ciphertext space corresponds to addition over the plaintext space, which itself corresponds to addition over the cleartext space modulo ρ. Overall, to generate the joint randomness, each party outputs a properly scaled encryption of an element of R_ρ, under a shared public key, and we publicly sum them. The ciphertext modulus Q can be chosen to minimize the amount of communication.[6] Using this, we can generate a ciphertext of uniform elements modulo ρ. This ciphertext is then (half-)bootstrapped to make it high-level and slots-encoded. We obtain the $\mathsf{RandGen}$ protocol (App. C.3).

If $\rho = 2^\ell$, we have a ciphertext that encrypts a uniform vector modulo ρ and may like to obtain several ciphertexts corresponding to the binary decomposition. For this purpose, we replace the bootstrapping call by the $\mathsf{SI\text{-}BTS}$ (small-integer bootstrapping) and bit extraction algorithms from [BKSS24]. $\mathsf{SI\text{-}BTS}$ can be viewed as a component of the TFHE-to-CKKS conversion from [BGGJ20]. It takes as input a coefficients-encoded ciphertext at the lowest-level modulus Q_0 and with plaintext placed in the most significant bits of the ciphertext, and which outputs a slots-encoded high-level ciphertext. The bit extraction algorithm from [BKSS24] takes a ciphertext for an N-dimensional vector of integers in $\mathbb{Z} \cap [0, 2^\ell - 1]$ for $\ell \geq 1$ and outputs 2ℓ ciphertexts encrypting in slots format the vectors of bits of the element-wise binary decomposition.

Additive share generation. In Fig. 4, we describe a 1-round protocol ShGen whose purpose is to transform a ciphertext ct encrypting a key sk' under a public key associated to a secret key sk that is shared as $(\mathsf{sk}_i)_i$ between n parties $(P_i)_i$, into additive shares of sk'. In the DKG protocol, we homomorphically sample the two (sparse and less-sparse) keys and rely on ShGen to share the secret keys.

Each party P_i partially decrypts the ciphertext encrypting sk', and adds a flooding noise fl_i on top of the message sk' during partial decryption by adding $\Delta \cdot \mathsf{fl}_i$ to its partial decryption share. Then, combining all partial decryption shares, each P_i recovers $\mathsf{m} = \mathsf{sk}' + \sum_j \mathsf{fl}_j$. We obtain an n-out-of-n secret sharing sk'_i of sk' where the share of P_i is $-\mathsf{fl}_i + \frac{1}{n} \cdot \mathsf{m} \pmod{Q_0}$, where Q_0 is the lowest-level modulus. One can set Q_0 prime to ensure that n and Q_0 are coprime for all $n < Q_0$ (this holds in any realistic scenario).

[6] To further reduce communication, we could have the parties jointly generate a λ-bit seed using the above strategy, and use this seed in a pseudo-random function (PRF) to homomorphically generate the appropriate amount of random bits (e.g., as efficiently done in [PPS26] using a variant of the PRF from [BIP+18]). This has the advantage of requiring fewer collaborative random bits, at the cost of more homomorphic computation.

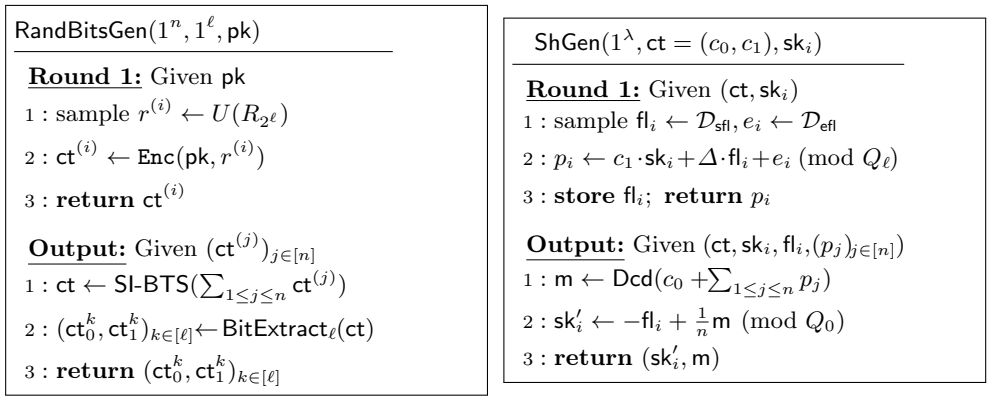


Fig. 4: Left: generation of random bits; Right: additive share generation.

The protocol also returns m. Note that it can be seen as a common $(n+1)$-th share if the i-th share was simply $-\mathsf{fl}_i$. We rely on this observation to save one round for generating the relinearization keys in the DKG protocol.

For security, we flood the partial decryption shares with a fresh noise term e_i. The amount of flooding is $\lambda/2$ bits more than the maximum error contained in ct, for λ bits of security. Hence, for correctness, we need at least $\lambda/2$ bits of gap between the noise and the message (in this case, the secret key sk'). We explain in Sec. 5 how to create such a gap before running ShGen in the DKG protocol. We let $\mathcal{D}_{\mathsf{efl}}$ denote the noise flooding distribution for partial decryption. To hide sk' in m, we also flood it with fl_i's. These are also $\lambda/2$ bits larger than sk, which in turn leads us to require a gap between Δ and the ciphertext modulus, for correctness. We let $\mathcal{D}_{\mathsf{sfl}}$ denote this flooding distribution for the plaintext.

4.2 The DKG Protocol

The DKG consists of two protocols: a Setup algorithm, which generates the public parameters, and an interactive KeyGen protocol, which is run by the parties $P_1, \ldots, P_n$ to generate and distribute the key material. For each sub-protocol from Sec. 4.1 and App. C, we let Proc_i denote the i-th round of the protocol, and by $\mathsf{Proc}_{\mathsf{out}}$ its output step, where Proc is the name of the sub-protocol. For simplicity, we consider a relinearization key instead of a full evaluation key, as it requires two rounds to generate, while the rest (conjugation and rotation keys) is simpler and can be generated in a single round.

During Round 1, each P_i samples a temporary secret key sk_{tmp}^i and publishes $a \cdot \mathsf{sk}_{tmp}^i + e_i$ for some common randomness a. These are combined in Round 2 to generate a temporary public key pk_{tmp}, so that each P_i can encrypt random coins. Parties also generate the associated temporary evaluation key (the relinearization key takes two rounds to create) during Rounds 1 and 2. At the end

of Round 2, parties share a temporary public key with a temporary evaluation key, and ciphertexts of random coins under this temporary key.

Round 3 consists in (locally) homomorphically generating the sparse and less-sparse secret keys using the temporary evaluation key and the encrypted random coins as the common source of randomness. This is achieved using the algorithms of Sec. 3. Each party can then run ShGen_1 on the resulting encrypted (sparse and less-sparse) secret keys using their share of the temporary secret key. At the beginning of Round 4, each party can then compute its share of the less-sparse secret key, generate the associated public key as well as the key-switching keys to switch between the sparse and less-sparse keys. Parties can also start the generation of the relinearization key for the less-sparse key, which will take an additional round to complete. This leads to a 5-round protocol.

The protocol follows this roadmap, except that it avoids the fifth round thanks to the following observation: to generate the final relinearization key, parties need their shares of the less-sparse secret key, which are of the form $\mathsf{sk}_i = -\mathsf{fl}_i + \frac{1}{n}\mathsf{m}$ where m is part of $\mathsf{ShGen}_{\mathsf{out}}$ and can be computed at the beginning of Round 4. The secret key is then $\mathsf{sk} = \mathsf{m} - \sum_j \mathsf{fl}_j$. Yet, we can alternatively think about sk as being shared between $n+1$ parties, where the imaginary $(n+1)$-th party holds m and other parties hold fl_i. As fl_i can be sampled in Round 1 by all parties, we can actually start generating the final relinearization key during Round 3 or earlier, using fl_i as P_i's share of the less-sparse key. To complete $\mathsf{RelinKeyGen}_1$, parties also need the contribution of the imaginary $(n+1)$-th party. As its share m is known to all in Round 4, each party can locally generate its contribution to both rounds of $\mathsf{RelinKeyGen}$ during Round 4 after computing m, and can therefore complete $\mathsf{RelinKeyGen}_2$ in Round 4 as if the imaginary party P_{n+1} had participated in $\mathsf{RelinKeyGen}_1$ during Round 1. Hence, the parties recover the relinearization key at the end of Round 4. The overall protocol is visualized in Fig. 5.

Setup returns the following public parameters pp including the number of parties n, a Ring-LWE degree N, a temporary secret key distribution $\mathcal{D}_{\mathsf{tmp}}$, a less-sparse key distribution $\mathcal{D}_{\mathsf{lsp}}$, a sparse key distribution $\mathcal{D}_{\mathsf{sp}}$, and an error distribution $\mathcal{D}_{\mathsf{err}}$ (over R), an integer ρ used to sample from $\mathcal{D}_{\mathsf{lsp}}$, flooding distributions $\mathcal{D}_{\mathsf{sfl}}$ and $\mathcal{D}_{\mathsf{efl}}$, a hash function $\mathcal{H}$ modeled as a random oracle, which serves to generate all common randomness, and the moduli and scaling factors for ciphertexts and evaluation keys.

The $\mathsf{KeyGen}(\mathsf{pp})$ protocol involves n parties $(P_i)_{1\leq i\leq n}$, takes as input the pp above, and proceeds as described in Fig. 6.

Noise distribution in the keys. In the DKG protocol described above, the noise underlying the public key and the evaluation key components grows with the number of participants n. This is because these keys are obtained by summing up partial public keys pk_i's of the form $a \cdot \mathsf{sk}_i + e_i$, where sk_i is P_i's share of the secret key and e_i is an error term, hence the resulting noise term grows as $\approx \sqrt{n}$. We propose two ways to address this issue. The first solution is to rescale the modulus by a factor $\sqrt{n}$ (for keys and ciphertexts). This costs a few bits of ciphertext modulus, out of hundreds. The second solution is to homomorphically

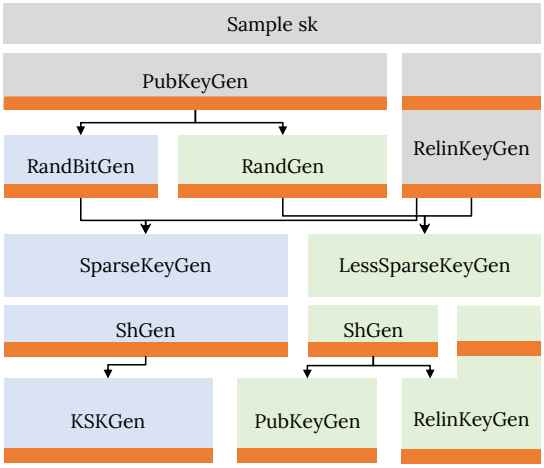


Fig. 5: Visualization of the 4-round protocol. The gray rectangles represent temporary key generation, while the blue and green rectangles generate keys for sparse and less sparse secrets, respectively. The horizontal orange bars represent the communication steps between parties.

generate correctly distributed noise terms from a common source of (encrypted) randomness, and have each party compute shares of the noise term using ShGen. Then, the parties can use their shares of the noise to obtain a public/evaluation key that has the appropriate error term. This is more accurate than the first solution, but is costlier as a lot more random bits need to be generated to homomorphically create all the noise terms (as outlined in Foot. 6, a PRF may be used to reduce communication). We use the first solution.

4.3 Security Analysis

Due to space constraints, we provide the detailed analysis in App. D and only sketch it here. As customary with FHE, we rely on a circular security assumption to argue security when the evaluation key is provided to the adversary. We provide an analysis which ignores the evaluation keys, and extend our claim to the full protocol assuming circular security, as stated in the claim below.

Claim. *Assuming Ring-LWE, perfect correctness of the ThFHE scheme, IND-CPA security of the FHE scheme, as well as circular-security, our distributed key generation is simulation-secure against passive adversaries with $n-1$ static corruptions.*

Proof sketch. Consider an adversary that corrupts all parties except P_1. Ignoring the evaluation keys thanks to the circular security assumption, and ignoring the sparse secret key for simplicity (it can be handled the same way as the less-sparse key), we can simplify the adversary's view to $\{\mathsf{pk}^1_{tmp}, \mathsf{ct}^1_{lsp,r}, p^1_{lsp}, \mathsf{pk}^1_{lsp}\}$.

Round 1:
1: sample $\mathsf{sk}^i_{tmp} \leftarrow \mathcal{D}_{\mathsf{tmp}}$
2: $\mathsf{pk}^i_{tmp} \leftarrow \mathsf{PubKeyGen}_1(\mathsf{sk}^i_{tmp})$
3: $h^i_{tmp} \leftarrow \mathsf{RelinKeyGen}_1(\mathsf{sk}^i_{tmp})$
4: **broadcast** $(\mathsf{pk}^i_{tmp}, h^i_{tmp})$

Round 2: Given $(\mathsf{sk}^i_{tmp}, (\mathsf{pk}^j_{tmp}, h^j_{tmp})_{j\in[n]})$:
1: $\mathsf{pk}_{tmp} \leftarrow \mathsf{PubKeyGen}_{\mathsf{out}}((\mathsf{pk}^j_{tmp})_{j\in[n]})$
2: $k^i_{tmp} \leftarrow \mathsf{RelinKeyGen}_2(\mathsf{sk}^i_{tmp}, (h^j_{tmp})_{j\in[n]})$
3: $\mathsf{ct}^i_{sp,r} \leftarrow \mathsf{RandBitsGen}_1(\mathsf{pk}_{tmp})$
4: $\mathsf{ct}^i_{lsp,r} \leftarrow \mathsf{RandGen}_1(\rho, \mathsf{pk}_{tmp})$
5: **broadcast** $(\mathsf{ct}^i_{sp,r}, \mathsf{ct}^i_{lsp,r}, k^i_{tmp})$

Round 3: Given $(\mathsf{sk}^i_{tmp}, (\mathsf{ct}^j_{sp,r}, \mathsf{ct}^j_{lsp,r}, k^j_{tmp})_{j\in[n]})$:
1: $\mathsf{rlk}_{tmp} \leftarrow \mathsf{RelinKeyGen}_{\mathsf{out}}((k^j_{tmp})_{j\in[n]})$
2: $\mathsf{ct}_{sp,r} \leftarrow \mathsf{RandBitsGen}_{\mathsf{out}}((\mathsf{ct}^j_{sp,r})_{j\in[n]})$
3: $\mathsf{ct}_{lsp,r} \leftarrow \mathsf{RandGen}_{\mathsf{out}}(\rho, (\mathsf{ct}^j_{lsp,r})_{j\in[n]})$
4: $\mathsf{ct}_{sp} \leftarrow \mathsf{SparseKeySampler}(\mathsf{ct}_{sp,r})$
5: $\mathsf{ct}_{lsp} \leftarrow \mathsf{LessSparseKeySampler}(\rho, \mathsf{ct}_{lsp,r})$
6: $p^i_{sp} \leftarrow \mathsf{ShGen}_1(\mathsf{ct}_{sp}, \mathsf{sk}^i_{tmp})$, internally sampling a share fl^i_{sp}
7: $p^i_{lsp} \leftarrow \mathsf{ShGen}_1(\mathsf{ct}_{lsp}, \mathsf{sk}^i_{tmp})$, internally sampling a share fl^i_{lsp}
8: $h^i_{lsp} \leftarrow \mathsf{RelinKeyGen}_1(\mathsf{fl}^i_{lsp})$
9: **broadcast** $(p^i_{sp}, p^i_{lsp}, h^i_{lsp})$

Round 4: Given $(\mathsf{sk}^i_{tmp}, \mathsf{fl}^i_{sp}, \mathsf{fl}^i_{lsp}, (p^j_{sp}, p^j_{lsp}, h^j_{lsp})_{j\in[n]})$:
1: $(\mathsf{sk}^i_{sp}, \mathsf{m}_{sp}) \leftarrow \mathsf{ShGen}_{\mathsf{out}}(\mathsf{ct}_{sp}, \mathsf{sk}^i_{tmp}, \mathsf{fl}^i_{sp}, (p^j_{sp})_{j\in[n]})$
2: $(\mathsf{sk}^i_{lsp}, \mathsf{m}_{lsp}) \leftarrow \mathsf{ShGen}_{\mathsf{out}}(\mathsf{ct}_{lsp}, \mathsf{sk}^i_{tmp}, \mathsf{fl}^i_{lsp}, (p^j_{lsp})_{j\in[n]})$
3: $\mathsf{pk}^i_{lsp} \leftarrow \mathsf{PubKeyGen}_1(\mathsf{sk}^i_{lsp})$
4: $h^{n+1}_{lsp} \leftarrow \mathsf{RelinKeyGen}_1(\mathsf{m}_{lsp})$
5: $k^i_{lsp} \leftarrow \mathsf{RelinKeyGen}_2(\mathsf{fl}^i_{lsp}, (h^j_{lsp})_{j\in[n+1]})$
6: $k^{n+1}_{lsp} \leftarrow \mathsf{RelinKeyGen}_2(\mathsf{m}_{lsp}, (h^j_{lsp})_{j\in[n+1]})$
7: $\mathsf{ksk}^i_{sp\to lsp} \leftarrow \mathsf{KSKeyGen}_1(\mathsf{sk}^i_{sp}, \mathsf{sk}^i_{lsp}, PQ_L)$
8: $\mathsf{ksk}^i_{lsp\to sp} \leftarrow \mathsf{KSKeyGen}_1(\mathsf{sk}^i_{lsp}, \mathsf{sk}^i_{sp}, PQ_0)$
9: **broadcast** $(\mathsf{pk}^i_{lsp}, k^i_{lsp}, \mathsf{ksk}^i_{sp\to lsp}, \mathsf{ksk}^i_{lsp\to sp})$

Output: Given $(\mathsf{sk}^i_{lsp}, (\mathsf{pk}^j_{lsp}, k^j_{lsp}, \mathsf{ksk}^j_{sp\to lsp}, \mathsf{ksk}^j_{lsp\to sp})_{j\in[n]})$:
1: $\mathsf{pk}_{lsp} \leftarrow \mathsf{PubKeyGen}_{\mathsf{out}}((\mathsf{pk}^j_{lsp})_{j\in[n]})$
2: $\mathsf{rlk}_{lsp} \leftarrow \mathsf{RelinKeyGen}_{\mathsf{out}}((k^j_{lsp})_{j\in[n+1]})$
3: $\mathsf{ksk}_{sp\to lsp} \leftarrow \mathsf{KSKeyGen}_{\mathsf{out}}((\mathsf{ksk}^j_{sp\to lsp})_{j\in[n]})$
4: $\mathsf{ksk}_{lsp\to sp} \leftarrow \mathsf{KSKeyGen}_{\mathsf{out}}((\mathsf{ksk}^j_{lsp\to sp})_{j\in[n]})$
5: $\mathsf{pk} \leftarrow \mathsf{pk}_{lsp}$
6: $\mathsf{evk} \leftarrow (\mathsf{rlk}_{lsp}, \mathsf{ksk}_{lsp\to sp}, \mathsf{ksk}_{sp\to lsp})$
7: $\mathsf{sk}_i \leftarrow \mathsf{sk}^i_{lsp}$
8: **return** $(\mathsf{pk}, \mathsf{evk}, \mathsf{sk}_i)$

Fig. 6: Key generation protocol. SparseKeySampler and LessSparseKeySampler are homomorphic implementations, using slot-encoding and $K = N$, of SparseKeyGen and LessSparseKeyGen. Note that LessSparseKeyGen can be replaced by DenseKeyGen.

First, via correctness of decryption, the quantity p^1_{lsp} satisfies:

$$p^1_{lsp} = \Delta \cdot (\mathsf{sk}_{lsp} + \mathsf{fl}^1_{lsp}) + e' - \Big(b_2 + \sum_{1<j\le n} a_2 \cdot \mathsf{sk}^j_{tmp}\Big) + e^1_2 \ ,$$

where e' is a small error underlying $\mathsf{ct}_{lsp} = (b_2, a_2)$, $\mathsf{fl}^1_{lsp} \leftarrow \mathcal{D}_{\mathsf{sfl}}$, and $e^1_2 \leftarrow \mathcal{D}_{\mathsf{efl}}$. By construction, we have $\mathsf{sk}_{lsp} \ll \mathsf{fl}^1_{lsp}$ and $e' \ll e^1_2$. Hence, using two flooding arguments, the quantity p^1_{lsp} is statistically close to:

$$p^1_{lsp} = \Delta \cdot \mathsf{fl}_{\mathsf{sim}} - \Big(b_2 + \sum_{1<j\le n} a_2 \cdot \mathsf{sk}^j_{tmp}\Big) + e_{\mathsf{sim}} \ ,$$

where $\mathsf{fl}_{\mathsf{sim}} \leftarrow \mathcal{D}_{\mathsf{sfl}}$, $e_{\mathsf{sim}} \leftarrow \mathcal{D}_{\mathsf{efl}}$.

Second, the above simulation only requires the knowledge of $\mathsf{ct}_{lsp} = (b_2, a_2)$, which is computed from $\mathsf{ct}^1_{lsp,r}$ and information known to the adversary. We can rely on IND-CPA security of the FHE scheme to argue that the adversary cannot

distinguish $\mathsf{ct}^1_{lsp,r}$ from an encryption of 0, as the adversary has no information about the temporary secret key (it misses P_1's share of it).

Therefore, the adversary's view is computationally indistinguishable from a distribution sampleable from pk_{lsp} and pk_{tmp}.

4.4 Other Approaches

In Fig. 6, the public and evaluation keys are generated from the encrypted secret key by relying on [MTBH21]. The protocol consumes four communication rounds. We introduce alternative approaches, providing trade-offs between computation and communication. Specifically, the evaluation key structure in [MTBH21] is the same as the single-key scheme at the expense of 2 communication rounds, while [KLSW24] supports a one-round setup with an unconventional relinearization key structure, which results in a 1.5× larger key size and 2.4-3.0 times slower key-switching latency. It would be suboptimal to use [KLSW24] for the final key generation since our primary goal is to make the homomorphic computation with the final key set as efficient as possible. However, one can utilize [MTBH21] for the temporary keys (Rounds 1 and 2) to achieve a 3-round protocol. We note that a 2-round protocol is also feasible with more FHE computations. Essentially, we can sample the noise for the key-switching keys by homomorphically evaluating a pseudo-random function (PRF). This allows us to generate the key-switching key homomorphically. For a more detailed explanation, refer to section E. They are compared in Tab. 3. While we believe that our main 4-round protocol in Fig. 6 provides the best performance for general settings, the following solutions with fewer rounds could be preferable when communication is extremely costly; e.g., in environments with a vast number of parties or significantly high network latency. Note that our implementation focuses on the 4-round solution, as we consider it the most optimal choice for typical use cases. The other approaches are given in App. E.

Temporary KeyGen	Final KeyGen	# Rounds	Remarks
Sec. 4.1	Sec. 3 and 4.1	4	Fig. 6 (using [MTBH21])
App. E.2	Sec. 3 and 4.1	3	Using [KLSW24] and [MTBH21]
App. E.2	Sec. 4.1 and PRF evaluation	2	Using [KLSW24] and more FHE computations

Table 3: Comparison between different approaches for DKG.

4.5 Delegated Key Generation

The large evaluation key size of FHE is problematic for clients with small devices: generating and sending the evaluation key may be too demanding for such clients.

To this end, we consider a delegated key generation scenario involving three parties: a client, key generators, and an evaluator. The client aims at delegating computations because it is resource-constrained and owns the decryption key. Classically, the client would generate the key material and send the evaluation key to the evaluator. Instead, our limited client delegates (most of) the key generation to the key generators.

Suppose a ThFHE system has been set up between the key generators. A client generates FHE sparse and less-sparse secret keys, encrypts them under the public key of key generators, and sends them to the key generators. The key generators use PubKeyGen (Fig. 7), RelinKeyGen (Fig. 8), KSKeyGen (Fig. 11), or techniques from Sec. 4.4 to compute the encrypted FHE keys from the received encrypted secret key. To minimize communications, the client can use a small modulus ciphertext, which the key generators will bootstrap to subsequently perform their computations. As the secret keys are ternary, one may rely on the SI-BTS algorithm. Then, the key generators jointly and securely compute the evaluation key of the client and send it to the FHE evaluator. The latter can now perform homomorphic computations for the client. Note that the client's secret key is not revealed unless many key generators collude and break the ThFHE system. An illustration is provided in App. F.

The description above assumes that the client encrypts the data, making it unnecessary to compute a client public key. If the data that the client decrypts can derive from ciphertexts generated by other users, then we may also ask the key generators to output a public key in addition to the evaluation key.

5 Implementation

We now focus on our proof-of-concept implementation with additional optimizations (Section G), based on the HEaaN library [Cry22]. The experiments were carried out with an Intel Xeon Gold 2.8GHz processor (for CPU experiments) or on an NVIDIA RTX-4090 (for GPU experiments). Our code implements the costliest component of the DKG protocol, i.e., the homomorphic sampling of a sparse vector from Section 3, of dimension $K = N$ where N is the Ring-LWE target degree of the threshold scheme.

5.1 Parameter Selection

In the experiments, we generate the secret keys for the most commonly used CKKS parameters. We consider the cases of $n = 32$, $n = 256$ and $n = 1024$ parties. The target ring degree for the generated key is $N = 2^{16}$ and the target Hamming weight of the sparse key for sparse secret encapsulation is $h = 32$. For these parameters, the probabilistic bound H in the sparse vector algorithm can be set as $H = 45$. For the homomorphic DKG, we use $N' = 2N = 2^{17}$. This choice is justified by the depth requirements of DKG with a large number of parties, and is also convenient implementation-wise, as it allows to handle N real slots in a single ciphertext.

In addition, we set the parameters to ensure that the bootstrapping failure probability is $\leq 2^{-128}$. Our algorithm performs bootstraps at two places, one in $\mathsf{RandBitGen}/\mathsf{RandGen}$ and one at the end of $\mathsf{SparseKeyGen}$ to allow for more depth in order to clean the bits of the secret keys. For these bootstrappings during DKG, we need to determine a bound on the value of $\mathcal{K}_n$, where n is the number of parties. We set $\mathcal{K}_{32} = 133$, $\mathcal{K}_{256} = 372$, $\mathcal{K}_{1024} = 742$ based on a probabilistic analysis which we report upon in App. H. With this value of $\mathcal{K}_n$, the modular reduction function for $\mathsf{EvalMod}$ step can be approximated with a degree-56 (resp. 62, resp. 126) even polynomial and 5 (resp. 6, resp. 6) iterations of double-angle formula [HK20]. Regarding the polynomials, we use the minimax approximation computed with the Remez algorithm. We note that other approximation methods could be used as well [HK20, LLL+21, LLK+22, MLS25].

To validate cleaning parameters, we measure the error at the end of the cleaning process. At each step of the iterative cleaning process, we use large rescaling in order to ensure that the rescaling error dominates other sources of error. When rescaling a ciphertext (a, b) under a key sk by Δ, the rescaling error in slot i is given by

$$\mathrm{RSerr}_i = \mathsf{Decode}\left(\lfloor a/\Delta \rceil\right)_i \odot \mathsf{Decode}\left(\mathsf{sk}\right)_i + \mathsf{Decode}\left(\lfloor b/\Delta \rceil\right)_i \ .$$

Assuming uniformity of the a and b parts, both real parts and imaginary parts of $\mathsf{Decode}(\lfloor a/\Delta \rceil)_i$ and $\mathsf{Decode}(\lfloor b/\Delta \rceil)_i$ sums N independent centered random variables. In view of the central limit theorem, we expect both real and imaginary parts of RSerr_i to behave as a centered Gaussian with variance $\mathsf{Decode}\left(\mathsf{sk}\right)_i^2 + 1$ up to a constant independent of i. Let σ be the corresponding standard deviation. In all our 256 experiments for each number of parties, we found the maximal error over all 65536 slots to be $\leq 2^{-105}$. As $(\mathsf{Decode}\left(\mathsf{sk}\right)_i^2 + 1)^{1/2} \geq 1$, we estimate that $\sigma \leq 2^{-105}$.

Under this Gaussian assumption, for the output to be 80-bit precise with overwhelming probability, it suffices to have, for all i,

$$14\sigma\sqrt{\mathsf{Decode}\left(\mathsf{sk}\right)_i^2 + 1} \leq 2^{-80} \ .$$

In view of $|\mathsf{Decode}\left(\mathsf{sk}\right)_i| \leq hn \leq 2^{17}$ in our experiments, this inequality reads as $14 \cdot 2^{-105} \cdot 2^{17.1} \leq 2^{-80}$, which holds. Under the Gaussian assumption, we thus estimate that the output of our experiments is 80-bit precise with probability $\geq 1 - 2^{-128}$.

Finally, we set the parameters for the cleaning after the main computation. We first perform bootstrapping on the ciphertext(s) output by the homomorphic sampling algorithm to recover modulus for cleaning, and clean using [CKSS25].

5.2 Estimate of Communication Costs

We estimate the communication costs per party in Tab. 5, based on the 4-round protocol description and the parameters in Tab. 4. In the table, we additionally include the sparse-key encapsulation keys and the automorphism keys required

N	$\log(PQ)$	dnum	$(h_{\mathsf{tmp}}, \widetilde{h}_{\mathsf{tmp}})$
2^{17}	≤ 2400	≤ 3	(128, 32)

	$\log Q$					$\log P$
n	Base	S2C	Mult	EvalMod	C2S	
32	32	26×3	32×23	$42 \times (6+5)$	32×3	61×12
256	34	28×3	34×23	$44 \times (6+6)$	34×3	61×13
1024	35	29×3	35×23	$48 \times (7+6)$	38×3	61×10

Table 4: FHE parameters used for the DKG protocol. Here h_{tmp} and $\widetilde{h}_{\mathsf{tmp}}$ denote the less-sparse and sparse Hamming weights of each party, respectively. The column $\log P$ refers to the auxiliary modulus for key switching. These parameters provide 128 bits of security, according to the lattice estimator fork available at `https://github.com/ludopulles/lattice-estimator`, which includes estimation of the cost of the new attacks exploiting the ring structure [HHO26].

for bootstrapping and sparse key sampling. We assume that a minimal set of automorphism keys is used; two keys each for baby and giant steps in the BSGS matrix multiplications, and 12 keys for one-hot vector generation. Our target parameter is the FGb parameter set of the HEaaN library [Cry22], and our estimation is based on the lattigo [MBTPH20] implementation of the protocol from [MTBH21]. We also consider a 10 Gbps LAN setting and a 200 Mbps WAN setting to estimate the communication latency.

Our estimation shows that the communication cost is dominated by the automorphism keys, which takes almost 85% of the communication cost. We note that this can be mitigated by deriving a complete key set from a small number of keys, using the methods described in [LLKN23, CKP26]. Nevertheless, our protocol requires communication of two full sets of keys; a temporary one for the homomorphic sampling, and a final one. The respective sizes of these two sets of keys depend on parameter choices for the temporary and the final FHE circuits. If all parameters are identical, our protocol thus doubles the total communication requirements.

5.3 Benchmark Results

We report the experimental results of our sparse key generation implementation, in Tab. 6. Run-times are averaged over 10 (CPU, single-threaded) or 100 (GPU) iterations.

Type	Round 1	Round 2	Round 3	Round 4
Public Key	35.0MB	-	12.7MB	-
SSE Key	36.5MB	-	13.7MB	-
Relin Key	280.0MB	140.0MB	127.4MB	63.7MB
Aut Key	2.80GB	-	1.27GB	-
Rand Enc	-	6.0MB	-	-
Share	-	-	32.8MB	-
Total	3.15GB	146.0MB	1.46GB	63.7MB
LAN	2.52s	0.12s	1.17s	0.05s
WAN	126.0s	5.84s	58.4s	2.55s

Table 5: Estimated communication cost of the DKG protocol from Fig. 6, per party. We use the parameters for 32 parties given in table 4. We assume 10Gbps of LAN and 200Mbps of WAN settings.

Phase		CPU time			GPU time		
		Num. parties					
		32	256	1024	32	256	1024
RandBitsGen		17.1	20.8	30.0	0.125	0.147	0.189
OneHotVecGen	(per 2 vectors)	4.69	5.00	5.92	0.021	0.023	0.025
	(total)	108	115	136	0.541	0.581	0.641
HomSampling	(per iter.)	3.20	3.30	3.75	0.015	0.016	0.017
	(total)	48.1	49.5	55.6	0.223	0.239	0.252
ShareGen		19.9	23.6	33.2	0.185	0.209	0.258
Total		193	217	256	1.08	1.18	1.34

Table 6: Performance of homomorphic sparse key sampling. This table only shows the on-the-fly homomorphic computation, and the key-generation timings or the communication costs are not included. RandBitsGen corresponds to the output part of Fig. 4; OneHotVecGen corresponds to Fig. 2; HomSampling corresponds to Fig. 3; ShareGen increases the precision of its ternary input to ≈ 105 bits. In OneHotVecGen, the unit cost is the generation of two vectors as we use complex numbers to handle two vectors at once during the linear algebra step. All timings are in seconds.

Acknowledgement

We thank the CCS reviewers for their thoughtful feedback and continued guidance throughout the revision process. We thank Seonghak Kim, Jonghoo Lee and Taeyeong Noh for their guidance and help regarding the implementation of our protocol.

Seonhong Min was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2023-00211649) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. RS-2024-00399491, Development of Privacy-Preserving Multiparty Computation Techniques for Secure Multiparty Data Integration).

References

ADE+23. Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Report 2023/1040, 2023.

ADK+23. Ehud Aharoni, Nir Drucker, Eyal Kushnir, Ramy Masalha, and Hayim Shaul. Generating one-hot maps under encryption. In *Cyber Security, Cryptology, and Machine Learning: 7th International Symposium, CSCML 2023, Be'er Sheva, Israel, June 29–30, 2023, Proceedings*, page 96–116, Berlin, Heidelberg, 2023. Springer-Verlag.

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 Berlin Heidelberg, Germany.

BBB+22. Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, 2022. Software available at `https://github.com/openfheorg/openfhe-development`.

BBG+21. Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part II*, volume 13043 of *Lecture Notes in Computer Science*, pages 129–161, Raleigh, NC, USA, November 8–11, 2021. Springer, Cham, Switzerland.

BCD+25. Carl Bootland, Kelong Cong, Daniel Demmler, Tore Kasper Frederiksen, Benoit Libert, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Samuel Tap, and Michael Walter. Threshold (fully) homomorphic encryption. Cryptology ePrint Archive, Report 2025/699, 2025.

BCKS24. Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 94–123, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

BEHG20. Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, page 12–24, New York, NY, USA, 2020. Association for Computing Machinery.

BEHZ16. Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*, volume 10532 of *Lecture Notes in Computer Science*, pages 423–442, St. John's, NL, Canada, August 10–12, 2016. Springer, Cham, Switzerland.

BFM+25. Zvika Brakerski, Offir Friedman, Avichai Marmor, Dolev Mutzari, Yuval Spiizer, and Ni Trieu. Threshold FHE with efficient asynchronous decryption. Cryptology ePrint Archive, Report 2025/712, 2025.

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, Cham, Switzerland.

BGGJ20. Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14:316–338, 08 2020.

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

BIP+18. Dan Boneh, Yuval Ishai, Alain Passelègue, Amit Sahai, and David J. Wu. Exploring crypto dark matter: New simple PRF candidates and their applications. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 699–729, Panaji, India, November 11–14, 2018. Springer, Cham, Switzerland.

BKSS24. Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology – ASIACRYPT 2024, Part I*, volume 15484 of *Lecture Notes in Computer Science*, pages 330–360, Kolkata, India, December 9–13, 2024. Springer, Singapore, Singapore.

BMM25. Jean-Philippe Bossuat, Christian Mouchet, and Janmajaya Mall. Compact 1-round threshold multiparty homomorphic encryption setup from public aggregatable transcripts, 2025. Contributed talk in the FHE.org conference, available at `https://www.youtube.com/watch?v=9Jsk6b4mFy0`.

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

BTH22. Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with

negligible failure-probability by using sparse-secret encapsulation. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 2022: 20th International Conference on Applied Cryptography and Network Security*, volume 13269 of *Lecture Notes in Computer Science*, pages 521–541, Rome, Italy, June 20–23, 2022. Springer, Cham, Switzerland.

CCK+25. Jung Hee Cheon, Hyeongmin Choe, Minsik Kang, Jaehyung Kim, Seonghak Kim, Johannes Mono, and Taeyeong Noh. Grafting: Decoupled scale factors and modulus in rns-ckks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 1098–1112, New York, NY, USA, 2025. Association for Computing Machinery.

CCP+24. Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. Attacks against the ind-cpad security of exact fhe schemes. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2505–2519, New York, NY, USA, 2024. Association for Computing Machinery.

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

CHK+18. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Cham, Switzerland.

CHK+19. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J.: Jacobson, Jr., editors, *SAC 2018: 25th Annual International Workshop on Selected Areas in Cryptography*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368, Calgary, AB, Canada, August 15–17, 2019. Springer, Cham, Switzerland.

CHK+21. Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021, Part III*, volume 13092 of *Lecture Notes in Computer Science*, pages 640–669, Singapore, December 6–10, 2021. Springer, Cham, Switzerland.

CHKS25. Jung Hee Cheon, Guillaume Hanrot, Jongmin Kim, and Damien Stehlé. SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025, Part III*, volume 15603 of *Lecture Notes in Computer Science*, pages 398–428, Madrid, Spain, May 4–8, 2025. Springer, Cham, Switzerland.

CKK20. Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages

221–256, Daejeon, South Korea, December 7–11, 2020. Springer, Cham, Switzerland.

CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland.

CKP26. Jung Hee Cheon, Minsik Kang, and Jai Hyun Park. Towards lightweight ckks: On client cost efficiency. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, ASIA CCS '26, page 1387–1398, New York, NY, USA, 2026. Association for Computing Machinery.

CKSS25. Hyeongmin Choe, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Leveraging discrete CKKS to bootstrap in high precision. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025: 32nd Conference on Computer and Communications Security*, pages 1083–1097, Taipei, Taiwan, October 13–17, 2025. ACM Press.

CMS+23. Sylvain Chatel, Christian Mouchet, Ali Utkan Sahin, Apostolos Pyrgelis, Carmela Troncoso, and Jean-Pierre Hubaux. PELTA - shielding multiparty-FHE against malicious adversaries. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 711–725, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

Cry22. CryptoLab. HEaaN library, 2022. Software available at `https://heaan.it/`.

DDEK+23. Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-fhe using noise flooding. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '23, page 35–46, New York, NY, USA, 2023. Association for Computing Machinery.

DMPS24. Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. BLEACH: Cleaning errors in discrete computations over CKKS. *Journal of Cryptology*, 37(1):3, January 2024.

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

EHL+23. Daniel Escudero, Cheng Hong, Hongqing Liu, Chaoping Xing, and Chen Yuan. Degree-D reverse multiplication-friendly embeddings: Constructions and applications. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 106–138, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore.

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

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

GHS12. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

GKS24. Kamil Doruk Gür, Jonathan Katz, and Tjerand Silde. Two-round threshold lattice-based signatures from threshold homomorphic encryption. In Markku-Juhani Saarinen and Daniel Smith-Tone, editors, *Post-Quantum Cryptography - 15th International Workshop, PQCrypto 2024, Part II*, pages 266–300, Oxford, UK, June 12–14, 2024. Springer, Cham, Switzerland.

HHO26. Jianhua Hou, Jiang Haodong, and Tabitha Ogilvie. Careful with the ring! concrete hardness gap between lwe and rlwe. In *CRYPTO*, 2026.

HK20. Kyoohyung Han and Dohyeong Ki. Better bootstrapping for approximate homomorphic encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, volume 12006 of *Lecture Notes in Computer Science*, pages 364–390, San Francisco, CA, USA, February 24–28, 2020. Springer, Cham, Switzerland.

HLSS25. Intak Hwang, Hyeonbum Lee, Jinyeong Seo, and Yongsoo Song. Practical zero-knowledge PIOP for maliciously secure multiparty homomorphic encryption. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025: 32nd Conference on Computer and Communications Security*, pages 4049–4063, Taipei, Taiwan, October 13–17, 2025. ACM Press.

HMSS25. Intak Hwang, Seonhong Min, Jinyeong Seo, and Yongsoo Song. On the security and privacy of CKKS-based homomorphic evaluation protocols. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025, Part VII*, volume 16251 of *Lecture Notes in Computer Science*, pages 296–330, Melbourne, VIC, Australia, December 8–12, 2025. Springer, Singapore, Singapore.

HS14. Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571, Santa Barbara, CA, USA, August 17–21, 2014. Springer Berlin Heidelberg, Germany.

JDP83. Kumar Joag-Dev and Frank Proschan. Negative association of random variables with applications. *The Annals of Statistics*, 11(1):286–295, 1983.

JVC18. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 1651–1669, Baltimore, MD, USA, August 15–17, 2018. USENIX Association.

KLSW24. Hyesun Kwak, Dongwon Lee, Yongsoo Song, and Sameer Wagh. A general framework of homomorphic encryption for multiple parties with non-interactive key-aggregation. In Christina Pöpper and Lejla Batina, editors, *ACNS 2024: 22nd International Conference on Applied Cryptography and*

Network Security, Part II, volume 14584 of *Lecture Notes in Computer Science*, pages 403–430, Abu Dhabi, UAE, March 5–8, 2024. Springer, Cham, Switzerland.

LLK+22. Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 551–580, Trondheim, Norway, May 30 – June 3, 2022. Springer, Cham, Switzerland.

LLKN23. Joon-Woo Lee, Eunsang Lee, Young-Sik Kim, and Jong-Seon No. Rotation key reduction for client-server systems of deep neural network on fully homomorphic encryption. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part VI*, volume 14443 of *Lecture Notes in Computer Science*, pages 36–68, Guangzhou, China, December 4–8, 2023. Springer, Singapore, Singapore.

LLL+21. Joon-Woo Lee, Eunsang Lee, Yongwoo Lee, Young-Sik Kim, and Jong-Seon No. High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021, Part I*, volume 12696 of *Lecture Notes in Computer Science*, pages 618–647, Zagreb, Croatia, October 17–21, 2021. Springer, Cham, Switzerland.

LMSS23. Changmin Lee, Seonhong Min, Jinyeong Seo, and Yongsoo Song. Faster TFHE bootstrapping with block binary keys. In Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik, editors, *ASIACCS 23: 18th ACM Symposium on Information, Computer and Communications Security*, pages 2–13, Melbourne, VIC, Australia, July 10–14, 2023. ACM Press.

LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer Berlin Heidelberg, Germany.

MBH23. Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. An efficient threshold access-structure for rlwe-based multiparty homomorphic encryption. *J. Cryptol.*, 36(2), March 2023.

MBTPH20. Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, pages 64–70, 2020.

MHWW24. Shihe Ma, Tairong Huang, Anyu Wang, and Xiaoyun Wang. Accelerating BGV bootstrapping for large p using null polynomials over $\mathbb{Z}_{p^e}$. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 403–432, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland.

MLS25. Seonhong Min, Joon-Woo Lee, and Yongsoo Song. Enhanced CKKS bootstrapping with generalized polynomial composites approximation. In *ASIACCS 25: 20th ACM Symposium on Information, Computer and Commu-*

nications Security, pages 1–12, Hanoi, Vietnam, August 25–29, 2025. ACM Press.

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

NHY+25. Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong, and Tao Wei. Xboot: Free-xor gates for ckks with applications to transciphering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):118–144, Sep. 2025.

PPS26. Jai Hyun Park, Alain Passelègue, and Damien Stehlé. THED: threshold dilithium from FHE. Cryptology ePrint Archive, Report 2026/638, 2026.

Ram88. M. V. Ramakrishna. An exact probability model for finite hash tables. In *Proceedings of the Fourth International Conference on Data Engineering*, page 362–368, USA, 1988. IEEE Computer Society.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore, MD, USA, May 22–24, 2005. ACM Press.

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.

SSTX09. Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 617–635, Tokyo, Japan, December 6–10, 2009. Springer Berlin Heidelberg, Germany.

STH+23. Yukimasa Sugizaki, Hikaru Tsuchida, Takuya Hayashi, Koji Nuida, Akira Nakashima, Toshiyuki Isshiki, and Kengo Mori. Threshold fully homomorphic encryption over the torus. In *ESORICS*, 2023.

Supplementary Material

A Reminders on CKKS Bootstrapping

We provide a concise description of each of the steps of CKKS bootstrapping algorithm below.

• Slots2Coeffs. Homomorphically perform a DFT (Discrete Fourier Transform) on the input ciphertext. This changes the encoding from slots mode to coefficients mode. In other words, it generates an encryption of a polynomial of which half the coefficients are the real parts of the message vector of the input ciphertext, and the other half are the imaginary parts, up to some ordering.

• ModRaise. Given an encryption $\mathsf{ct} \in R^2_{Q_0}$ of a cleartext $m \in \mathbb{R}[X]/(X^N+1)$ with scaling factor Q_0, increase the ciphertext modulus to the highest level $L-1$. This operation effectively raises the decryption identity (Eq. (1)) to the modulus Q_{L-1} under the form

$$c_0 + c_1 s = \mu + e + I \cdot Q_0 \pmod{Q_{L-1}} \ ,$$

where $I \in R$ is an integer polynomial with small-magnitude coefficients. Under the assumptions that the coefficients of $\mu+e$ are small compared to Q_0 and that s is ternary, expanding this identity shows that the coefficients of $I \cdot Q_0$ are rounded sums (up to ± 1 factors) of $h+1$ coefficients of c_0 and c_1. Heuristically assuming that the coefficients of c_0 and c_1 are uniformly distributed, this implies that, with some overwhelming probability $1-p$, the coefficients of I have absolute values bounded from above by a certain constant K_p. In the sequel, we set $p \leq 2^{-128}$. Otherwise, one can mount attacks against the security of threshold-CKKS (the attack described in [CCP+24, Sec. 3] readily extends from INDCPA-D security of FHE to security of ThFHE).

• Coeffs2Slots. Homomorphically perform an iDFT (inverse Discrete Fourier Transform) on the input ciphertext. This changes the encoding from coefficients mode to slots mode, up to some ordering. Note that Coeffs2Slots is essentially the inverse operation of Slots2Coeffs.

• EvalMod. Homomorphically evaluate the modular function $x \mapsto x \bmod Q_0$ over the input ciphertext, removing the term $I \cdot Q_0$ introduced during the ModRaise step.

B Additional Aspects and Extension of Homomorphic Key Sampling

B.1 One-hot Vector Generation from Aharoni et al

In [BEHG20], Boneh et al. proposed a one-hot vector generation algorithm at a high level. Later, their method was refined into a concrete solution by Aharoni et

al. [ADK+23]. Below, we detail how it can be implemented in the CKKS-inspired model of computation.

We take as input the vector

$$\mathbf{e} = (e_i) \ ,$$

where $e_{2^i} = b_i$ for $0 \leq i < k$, and $e_j = 0$ if j is not a power of 2.

We start by masking input to obtain $\mathbf{e}_i^{(0)}$, for $0 \leq i < k$, via

$$\begin{aligned}
\mathbf{e}_i &= \mathbf{e} \odot (\underbrace{0, \ldots, 0}_{2^i}, 1, \underbrace{0, \ldots, 0}_{N/2-2^i-1}) \ , \\
\mathbf{e}_i^{(0)} &= (1, \underbrace{0, \ldots, 0}_{N/2-1}) - \mathrm{Rot}_{-2^i}(\mathbf{e}_i) + \mathbf{e}_i \\
&= (1 - b_i, \underbrace{0, \ldots, 0}_{2^i-1}, b_i, \underbrace{0, \ldots, 0}_{N/2-2^i-1}) \ .
\end{aligned}$$

From these vectors, we build a binary tree of values $(\mathbf{e}_i^{(j)})$, for $0 \leq i < k/2^j$ and $0 \leq j \leq \log k$, where $\mathbf{e}_i^{(j)}$ has non-zero values in slots $2^{i2^j} \cdot \ell$, for $0 \leq \ell < 2^{2^j}$. This corresponds to tensor products (recall that the tensor product of $\mathbf{u} = (u_0, \ldots, u_{\ell-1})^T$ and $\mathbf{v} = (v_0, \ldots, v_{\ell-1})^T$ is the ℓ^2-dimensional vector obtained by concatenating the $v_i \cdot \mathbf{u}$'s). To build the tree, we use the following inductive definition, for $0 \leq i < k/2^j$ and $0 \leq j \leq \log k$:

$$\mathbf{e}_i^{(j+1)} = \left(\sum_{m=0}^{2^{2^j}-1} \mathrm{Rot}_{m2^{(i+1)2^j}}(\mathbf{e}_{2i}^{(j)}) \right) \odot \left(\sum_{m=0}^{2^{2^j}-1} \mathrm{Rot}_{m2^{i2^j}}(\mathbf{e}_{2i+1}^{(j)}) \right) .$$

The two inner sums can be computed using $2^j - 1$ rotations each through the classical rotate-and-sum algorithm; this yields $O(k)$ rotations per level and hence $O(k \log k)$ rotations overall, as claimed.

B.2 Proof of Lemma 1

The inductive definition of Chebyshev polynomials implies that $T_d(\cos(x)) = \cos(dx)$ holds for all $x \in \mathbb{R}$ and $d \geq 0$. In particular, for all real number y such that $|y| \leq 1$, there exists $z \in \mathbb{R}$ such that $y = \cos(z)$ and $|T_d(y)| = |\cos(dz)| \leq 1$. This proves that

$$|W_{B,d}(x)| \leq \frac{1}{T_d\left(\frac{B+1}{B-1}\right)} \ ,$$

for $x \in \{1, \ldots, B\}$. Further, from the inductive definition, we can deduce that, for $x \geq 1$ and $d \geq 0$:

$$\begin{aligned}
T_d(x) &= \frac{1}{2}\left((x + \sqrt{x^2-1})^d + (x - \sqrt{x^2-1})^d\right) \\
&\geq \frac{1}{2}(x + \sqrt{x^2-1})^d \ .
\end{aligned}$$

By plugging $x = (B+1)/(B-1)$, we obtain that

$$T_d\left(\frac{B+1}{B-1}\right) \geq \frac{1}{2}\left(\frac{(\sqrt{B}+1)^2}{B-1}\right)^d \ ,$$

from which the result follows.

For this quantity to be larger than ε^{-1}, it suffices that

$$\ln\left(\frac{2}{\varepsilon}\right) \leq d \ln\left(\frac{\sqrt{B}+1}{\sqrt{B}-1}\right) \ .$$

As

$$\ln\left(\frac{\sqrt{B}+1}{\sqrt{B}-1}\right) = -\ln\left(1-\frac{2}{\sqrt{B}+1}\right) \geq \frac{2}{\sqrt{B}+1} \ ,$$

it suffices to have

$$d \geq \frac{\sqrt{B}+1}{2} \ln\left(\frac{2}{\varepsilon}\right) \ ,$$

as claimed. □

B.3 Numerical Aspects of the Chebyshev-Based Indicator Function Evaluation

We discuss here a delicate numerical issue regarding the method of Sec. 3.1. As we use CKKS, the input of the approximate indicator function is itself imprecise. This has little impact for values in $\{1, \ldots, B\}$ as $W_{B,d}$ is guaranteed to be small even for real values in $[1, B]$. However, for x close to 0, the evaluation $W_{B,d}(x)$ could deviate significantly from 1. This deviation is related to the value $W'_{B,d}(0)$, which we estimate in the following lemma.

Lemma 5. *Let $n \geq 1$ and $B > 1$ be integers. Then we have*

$$\left|W'_{B,d}(0)\right| = \frac{d}{\sqrt{B}}\left(1 - 2^{-\Omega_B(d)}\right) \ .$$

Proof. Starting from $U_0 = 1$, $U_1 = 2x$ and the same recurrence relation as T_d yields the Chebyshev polynomials U_d of the second kind, which satisfy $T'_d(x) = dU_{d-1}(x)$. We have

$$|W'_{B,d}(x)| = \left|\frac{-2}{B-1}\frac{T'_d\left(\frac{B+1-2x}{B-1}\right)}{T_d\left(\frac{B+1}{B-1}\right)}\right| = \frac{2d}{B-1}\left|\frac{U_{d-1}\left(\frac{B+1-2x}{B-1}\right)}{T_d\left(\frac{B+1}{B-1}\right)}\right| \ ,$$

so that

$$|W'_{B,d}(0)| = \frac{2d}{B-1}\left|\frac{U_{d-1}\left(\frac{B+1}{B-1}\right)}{T_d\left(\frac{B+1}{B-1}\right)}\right| \ .$$

Using the identity

$$U_{d-1}(x) = \frac{1}{2\sqrt{x^2-1}} \left((x + \sqrt{x^2-1})^d - (x - \sqrt{x^2-1})^{-d} \right) \ ,$$

we obtain:

$$|W'_{B,d}(0)| = \frac{d}{\sqrt{B}} \frac{1 - z_B^d}{1 + z_B^d} \ ,$$

where $z_B = (\sqrt{B}-1)/(\sqrt{B}+1) < 1$.

This result implies that, for fixed B and increasing d, the accuracy of W improves on $[1, B]$ as $((B-1)/(\sqrt{B}+1)^2)^d$ but degrades at 0 as $2^{-p}d/\sqrt{B}$ where p is the input precision. A possible choice for d is derived by matching those two quantities.

B.4 Proof of Theorem 1

We first evaluate the matrix-vector product by relying on an extension of the diagonal method plus Baby-Step Giant-Step (BS-GS) algorithm (see [HS14]) to a $K \times k$ matrix for $K \gg k$. This extension is dual to the extension described in [JVC18, Sec. V.F] for matrices with fewer rows than columns.

We slice the matrix $\mathbf{M}$ into K/k, blocks $(\mathbf{M}_\ell)_{0 \le \ell < K/k}$ each of dimensions $k \times k$ and multiply each block with $\mathbf{b} = (b_0, \ldots, b_{k-1})^T$, using $2\sqrt{k}$ rotations. Now, instead of repeating this K/k times (once for each block), we first use $O(\log K)$ rotations to obtain a K-dimensional vector $\mathbf{b}_{\text{ext}}$ with K/k blocks $((\mathbf{b}_{\text{ext}})_\ell)_{0 \le \ell < K/k}$. Each block initially contains a copy of $(b_0, \ldots, b_{k-1})^T$. We then perform the K/k matrix-vector products of dimensions $k \times k$ in parallel, at a cost $O(\sqrt{k}) = O(\sqrt{\log K})$. The $O(\log K)$ initial rotations are the dominant component to the cost.

Once $\mathbf{w}$ is computed, we evaluate the function $\delta_{0,k}$ that maps $1, \ldots, k-1$ to 0 and 0 to 1. We are in the case of a boundary integer indicator function, hence this consumes $\frac{1}{2}\log k + O(1) = \frac{1}{2}\log\log K + O(1)$ multiplicative levels (for a fixed target precision) and only $O(\log\log K)$ multiplications. □

B.5 Batched Generation of One-hot Vectors

We describe how to batch the sampling of B one-hot vectors of dimension $K = 2^k$. For the sake of simplicity, we assume that $B \cdot k$ divides K. We assume that we process the random bits given as input, to obtain:

$$\mathbf{b} = (b_0, b_1, \ldots, b_{kB-1}, \ldots, b_0, b_1, \ldots, b_{kB-1})^T \in \{0,1\}^K \ ,$$

where the bits $b_0, b_1, \ldots, b_{kB-1}$ are repeated until they fill all K slots. This can be achieved using $\log(K/(kB))$ rotations. For α, β two integers such that $\alpha + \beta$ divides K, we let $\mathbf{c}_{\alpha,\beta}$ denote the vector

$$(\underbrace{1, 1, \ldots 1}_{\alpha}, \underbrace{0, 0, \ldots, 0}_{\beta}, \ldots, \underbrace{1, 1, \ldots 1}_{\alpha}, \underbrace{0, 0, \ldots, 0}_{\beta})^T \in \{0,1\}^K \ .$$

We consider the vectors

$$\mathbf{b}_i = \mathbf{b} \odot \mathrm{Rot}_{ik}\left(\mathbf{c}_{k,k(B-1)}\right) \ ,$$

for $0 \leq i < B$. We then use $\log B$ rotations on each vector $\mathbf{b}_i$ to obtain the vector

$$\widetilde{\mathbf{b}}_i = (b_{ik}, \ldots, b_{(i+1)k-1}, \ldots, b_{ik}, \ldots, b_{(i+1)k-1})^T \ ,$$

from which we deduce a one-hot vector using the algorithm from Fig. 2 without Steps 1-3, in $O(\sqrt{\log K})$ rotations. The amortized cost per one-hot vector is $O(\frac{\log(K/(kB))}{B} + \log B + \sqrt{\log K})$ rotations. For B between $\Omega(\sqrt{\log K})$ and $\exp(O(\sqrt{\log K}))$, this is $O(\sqrt{\log K})$, which is much less than the $O(\log K)$ rotations from the algorithm from Fig. 2.

Note that the analysis shows that the cost decreases and then increases as a function of B. This suggests that batches of large size may be handled by considering several smaller batches.

B.6 Estimate on H Satisfying Equation (4)

In order to bound H, we use the following classical Chernoff-type inequality: for $X_1, \ldots, X_h$ independent random variables, let $X = X_1 + \ldots + X_h$; then

$$\Pr(X \geq a) \leq \inf_{t>0} \left\{ \mathrm{e}^{-ta} \cdot \prod_{k=1}^{h} \mathbb{E}\left[\mathrm{e}^{tX_k}\right] \right\} \ .$$

It follows from the Markov inequality, applied to the random variable $\exp(tX)$. We let X_i be the random variable which counts the number of balls needed to fill a i-th bin when $i-1$ bins have already been filled. Then X_i follows a geometric law of parameter $(K-(i-1))/K$, so that

$$\mathbb{E}\left[\mathrm{e}^{tX_i}\right] = \frac{(K-i+1)\mathrm{e}^t}{K-(i-1)\mathrm{e}^t} \ .$$

Set $t = \ln(K/h)$. Then the denominator is positive, and the expectancy is bounded from above by $\exp(t)/(1-(i-1)/h)$. This yields:

$$\begin{aligned}
\Pr\left(X_1 + \cdots + X_h \geq H\right) &\leq \left(\frac{h}{K}\right)^{H-h} \cdot \prod_{i=1}^{h} \frac{1}{1-\frac{i-1}{h}} \\
&= \left(\frac{h}{K}\right)^{H-h} \frac{h^h}{h!} \\
&\leq \left(\frac{h}{K}\right)^{H-h} \mathrm{e}^h \ ,
\end{aligned}$$

where we used the inequality $h! \geq h^h \exp(-h)$. For this probability to be $\leq \varepsilon$, it thus suffices that

$$H \geq h\left(1 + \frac{1}{\ln(K/h)}\right) + \frac{\ln(1/\varepsilon)}{\ln(K/h)} \ .$$

B.7 Analysis of the Less-sparse Key Sampling

We analyze the approach of Sec. 3.4 in order to derive rigorous probabilistic bounds for the Hamming weight. The number of 0's and $\rho - 1$'s in $\mathbf{u}$ follows a binomial distribution with parameter $2/\rho$. Let $p_{h'}$ be the probability that the number of 0's and $\rho - 1$'s in $\mathbf{v}$ is less than h'. Using a standard tail bound for the binomial distribution, we have, for $\rho < 2N/h'$:

$$\begin{aligned} p_{h'} &\leq \left(e^{\rho h'/(2N)-1}(\rho h'/(2N))^{-\rho h'/(2N)}\right)^{2N/\rho} \\ &\leq e^{h'-2N/\rho}(\rho h'/(2N))^{-h'} \quad . \end{aligned}$$

For this to be less than p, it suffices that

$$\rho \leq \frac{2N}{h'(-W_{-1}(-p^{1/h'}/e))} \quad ,$$

where W_{-1} denotes the non-principal branch of the Lambert function with $W_{-1}(-1/e) = -1$. For such a choice, the vector $(\delta_{0,\rho-1}(u_i) - \delta_{\rho-1,\rho-1}(u_i))_{0\leq i<N}$ has Hamming weight $\geq h'$ with probability $\geq 1 - p$.

Symmetric tail bounds for the binomial distribution can be used to prove that, with probability $\geq 1 - p$, the Hamming weight is $\leq h'' = -(2N/\rho + \log p)/W_0(-(1 + \log(p^{\rho/(2N)}))/e)$, where W_0 denotes the principal branch of the Lambert function.

B.8 Homomorphic Block Key Generation

In recent works [LMSS23, CHKS25], a special key distribution called the *block key distribution* is leveraged for the sake of bootstrapping acceleration. It was first proposed by Lee et al. [LMSS23] to accelerate the blind rotation in TFHE bootstrapping. Briefly speaking, we see the secret key as a concatenation of some blocks, where each block is filled mostly with zeros, with at most one 1. The block key structure was later modified for a low-depth CKKS bootstrapping [CHKS25], by generalizing it to the ternary case, forcing a block to have exactly one nonzero element, and (optionally) allowing the blocks to overlap for enhanced security. Here, we do not consider the case of overlapping blocks, for simplicity.

By using this block distribution as the target distribution for the sparse key, the DKG protocol can be significantly accelerated, leading to replacing $H = 45$ by $H = 1$ in OneHotVecGen and removing the HomSampling step. We now outline the resulting algorithm.

Let the target Hamming weight be h and the key dimension be K. For simplicity, let us assume that $h < K$ are powers of 2: the number of blocks K/h is also a power of 2. The goal is essentially to uniformly sample a random entry inside each block and to set it either 1 or -1, with a probability of $1/2$ for each. This is identical to generating a one-hot vector inside the block and assigning the sign randomly. To realize this, we first run the one-hot vector algorithm for each block in a parallelized manner using SIMD packing, resulting in a block

key, but in binary. Then, similar to the ternary key generation algorithm given in Sec. 3.3, we multiply a random sign vector to the resulting binary key.

Let us first begin by describing the binary key generation algorithm. For conciseness, we let the block length be denoted $\ell := K/h$ and write $k := \log \ell$. Then, we can use hk bits $b_0, \ldots, b_{hk-1}$ to generate a binary block key with length K. Recall the matrix $\mathbf{M}$ from Sec. 3.2, with dimensions $\ell \times k$. Then, what we aim at is to compute the following:

$$\mathbf{u} := \frac{k}{2}\mathbf{1}_K - \frac{1}{2}\begin{bmatrix} \mathbf{M} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{M} & \ldots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \ldots & \mathbf{M} \end{bmatrix} \cdot \left(\mathbf{1}_{hk} - 2 \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{hk-1} \end{bmatrix} \right) .$$

We note that this matrix-vector multiplication can be computed similarly to the one-hot vector generation algorithm. To be specific, we slice the matrix in a block-wise manner, so that $\mathbf{M}$ can be represented with ℓ/k blocks $\mathbf{M}_1, \ldots, \mathbf{M}_{\ell/k} \in \mathbb{Z}^{k \times k}$, with each block multiplied by block-wise random vectors $(b_0, \ldots, b_{k-1})^T$, $(b_k, \ldots, b_{2k-1})^T, \ldots, (b_{(h-1)k}, \ldots, b_{hk-1})^T$. Next, with $O(\log(\ell/k))$ rotations, we obtain ℓ/k copies of $\mathbf{v}$, allowing us to perform matrix-vector multiplication in parallel, which requires only $O(\sqrt{k})$ rotations in total. Now, seeing this as a parallel execution of the matrix-vector multiplication from Sec. 3.2, it can be seen that each entry of the resulting vector $\mathbf{u}$ is in the interval $[0, k]$ and there exists only a single 0 inside each block of length ℓ. Therefore, it only remains to evaluate the indicator function $\delta_{k,0}(\mathbf{u})$ to generate a binary block key. Using our efficient Chebyshev-based method, this indicator function can be computed with only $\log k$ multiplications, consuming $\frac{1}{2}\log k + O(1)$ levels. Then, it only remains to multiply the result by a random sign vector, making the depth of this overall procedure $\frac{1}{2}\log k + O(1) = \frac{1}{2}\log\log(K/h) + O(1)$.

C Additional DKG Sub-protocols

C.1 Public Key Generation

We recall in Fig. 7 the folklore 1-round protocol for generating a common public key for a secret key sk shared between n parties $(P_i)_i$. It takes as input P_i's share sk_i of sk and returns a common public key pk associated to sk.

C.2 Relinearization Key Generation

We adapt to CKKS the 2-round protocol from [MTBH21] for generating a relinearization key for a secret key sk shared between n parties $(P_i)_i$. It is described in Fig. 8. It takes as input P_i's share sk_i of the secret key sk and returns a relinearization key for sk.

$\mathsf{PubKeyGen}(\mathsf{sk}_i)$

Round 1: Given sk_i:

1: sample a common $a \leftarrow \mathcal{H}$
2: $\mathsf{pk}_i \leftarrow -a \cdot \mathsf{sk}_i + e_i$
3: **store** a; **return** pk_i

Output: Given $(\mathsf{pk}_j)_{j\in[n]}$

1: $\mathsf{pk} \leftarrow \left(\sum_{1\le j\le n} \mathsf{pk}_j, a\right)$
2: **return** pk

Fig. 7: Public key generation.

$\mathsf{RelinKeyGen}(\mathsf{sk}_i)$

Round 1: Given sk_i:

1: sample a common $a \leftarrow \mathcal{H}$
2: sample $u_i \leftarrow \mathcal{D}_{\mathsf{tmp}}$ and $e_{i,0}, e_{i,1} \leftarrow \mathcal{D}_{\mathsf{err}}$
3: $h_{i,0} \leftarrow -a \cdot u_i + P \cdot \mathsf{sk}_i + e_{i,0}$
4: $h_{i,1} \leftarrow a \cdot \mathsf{sk}_i + e_{i,1}$
5: **return** $(h_{i,0}, h_{i,1})$

Round 2: Given $(\mathsf{sk}_i, (h_{j,0}, h_{j,1})_{j\in[n]})$

1: $h_b \leftarrow \sum_{1\le j\le n} h_{j,b}$, for $b \in \{0, 1\}$
2: sample $e_{i,2}, e_{i,3} \leftarrow \mathcal{D}_{\mathsf{err}}$
3: $k_{i,0} \leftarrow h_0 \cdot \mathsf{sk}_i + e_{i,3}$
4: $k_{i,1} \leftarrow h_1 \cdot (u_i - \mathsf{sk}_i) + e_{i,2}$
5: $k_i \leftarrow k_{i,0} + k_{i,1}$
6: **store** h_1; **return** k_i

Output: Given $(k_j)_{j\in[n]}$

7: $\mathsf{rlk} \leftarrow \left(\sum_{1\le j\le n} k_j, h_1\right)$
8: **return** rlk

Fig. 8: Relinearization key generation.

C.3 Randomness Generation

We generate randomness over $\mathbb{Z}_\rho$ using $\mathsf{RandGen}$, which is slightly different from $\mathsf{RandBitsGen}$ (Fig. 4). It is described in Fig. 9. In contrast to $\mathsf{RandBitsGen}$, it samples $r^{(i)}$ from $U(R_\rho)$ and does not perform a binary decomposition.

$\mathsf{RandGen}(1^n, \rho, \mathsf{pk})$

Round 1: Given pk

1 : sample $r^{(i)} \leftarrow U(R_\rho)$

2 : $\mathsf{ct}^{(i)} \leftarrow \mathtt{Enc}(\mathsf{pk}, r^{(i)})$

3 : **return** $\mathsf{ct}^{(i)}$

Output: Given $(\mathsf{ct}^{(j)})_{j\in[n]}$

1 : $\mathsf{ct} \leftarrow \mathsf{SI\text{-}BTS}(\sum_{1\le j\le n} \mathsf{ct}^{(j)})$

2 : **return** ct

Fig. 9: Generation of randomness.

C.4 Less-Sparse Sampler

We give in Fig. 10 a formal description of the less sparse generator outlined at the end of Sec. 3.4. For $\rho = 200$, this algorithm returns a ternary vector of dimension $N = 2^{16}$ with Hamming weight in $[192, 1024]$, with overwhelming probability.

$\mathsf{LessSparseSampler}(\mathbf{u}, \rho)$

1 : $\mathbf{u}_1 \leftarrow \delta_{\rho-1,0}(\mathbf{u})$

2 : $\mathbf{u}_2 \leftarrow \delta_{\rho-1,\rho-1}(\mathbf{u})$

3 : **return** $\mathbf{u}_1 - \mathbf{u}_2$.

Fig. 10: Less-sparse key sampler.

C.5 Key-Switching Key Generation

We finally provide in Fig. 11 a 1-round protocol to generate key-switching keys for two secret keys $\mathsf{sk}_0, \mathsf{sk}_1$ shared between the n parties. It takes as input the shares $\mathsf{sk}_{i,0}, \mathsf{sk}_{i,1}$ of P_i, and a modulus PQ_k and returns a key-switching key to switch from sk_0 to sk_1 at modulus PQ_k. To simplify the algorithm, it only

generates the Ring-LWE ciphertexts that the key-switching keys are made of. To obtain full key-switching keys, it may be run multiple times using appropriate scaling factors Δ. Note that sk_1 can be a linear function of sk_0 (for rotation and conjugation keys) but can be otherwise unrelated (for switching to and from the sparse key).

$\mathsf{KSKeyGen}(\mathsf{sk}_{i,0}, \mathsf{sk}_{i,1}, Q_k)$

Round 1: Given $(\mathsf{sk}_{i,0}, \mathsf{sk}_{i,1})$:

1: sample a common $a \leftarrow \mathcal{H}$ in R_{PQ_j};
2: sample $e_i \leftarrow \mathcal{D}_{\mathsf{err}}$
3: $\mathsf{ksk}^i_{\mathsf{sk}_0 \to \mathsf{sk}_1} \leftarrow -a \cdot \mathsf{sk}_{i,1} + \Delta \cdot \mathsf{sk}_{i,0} + e_i$
4: **store** a; **return** $\mathsf{ksk}^i_{\mathsf{sk}_0 \to \mathsf{sk}_1}$

Output: Given $\left(\mathsf{ksk}^j_{\mathsf{sk}_0 \to \mathsf{sk}_1}\right)_{j \in [n]}$

1: $\mathsf{ksk}_{\mathsf{sk}_0 \to \mathsf{sk}_1} \leftarrow \left(\sum_{1 \le j \le n} \mathsf{ksk}^j_{\mathsf{sk}_0 \to \mathsf{sk}_1}, a\right)$
2: **return** $\mathsf{ksk}_{\mathsf{sk}_0 \to \mathsf{sk}_1}$

Fig. 11: Key-switching key generation.

D Detailed Security Analysis

Let us first restate the claim regarding the security of our protocol.

Claim. Assuming Ring-LWE, perfect correctness of the ThFHE scheme, IND-CPA security of the underlying FHE scheme, as well as circular-security, our distributed key generation is simulation-secure.

Proof. Consider an adversary against the protocol, and let us assume without loss of generality that it corrupts all parties except P_1. We first describe the view of the adversary in the actual protocol. The adversary learns the public parameters pp and has access to the random oracle (or the CRS) used to compute common randomness. It further receives the following information from the honest party P_1 at the successive rounds:

- Round 1: $\mathsf{pk}^1_{tmp}, h^1_{tmp}$;
- Round 2: $\mathsf{ct}^1_{sp,r}, \mathsf{ct}^1_{lsp,r}, k^1_{tmp}$;
- Round 3: $p^1_{sp}, p^1_{lsp}, h^1_{lsp}$;
- Round 4: $\mathsf{pk}^1_{lsp}, k^1_{lsp}, \mathsf{ksk}^1_{sp \to lsp}, \mathsf{ksk}^1_{lsp \to sp}$.

As already mentioned, we ignore the material associated to evaluation keys for the analysis. To further simplify, we also ignore the sparse secret key and only consider elements related to the less-sparse secret key. This is without loss of generality, as both keys are independent and an identical analysis applies if we

consider both keys. Note in particular that much more information is available to the adversary about the less-sparse key, as we do not generate a public key nor an evaluation key for the sparse key. With these additional simplifications, the adversary's view can be reduced to:

- Round 1: $\mathsf{pk}^1_{tmp} = -a_1 \cdot \mathsf{sk}^1_{tmp} + e^1_1$, used to set pk_{tmp};
- Round 2: $\mathsf{ct}^1_{lsp,r} = \mathtt{Enc}(\mathsf{pk}_{tmp}, r_1)$, where r_1 is an additive share of the (encrypted) random bits that serve to homomorphically generate

$$\mathsf{ct}_{lsp,r} = \mathtt{Enc}\left(\mathsf{pk}_{tmp}, \sum_{1 \le j \le n} r_j \bmod \rho\right) \ ;$$

- Round 3: $p^1_{lsp} = a_2 \cdot \mathsf{sk}^1_{tmp} + \Delta \cdot \mathsf{fl}^1_{lsp} + e^1_2$, where a_2 is the uniform part of the ciphertext ct_{lsp} encrypting the key sk_{lsp}, homomorphically generated using $\sum_{1 \le j \le n} r_j \bmod \rho$ as randomness;
- Round 4: $\mathsf{pk}^1_{lsp} = -a_3 \cdot \mathsf{fl}^1_{lsp} + e^1_3$ which allows to recover pk_{lsp}.

We recall that $e^1_1, e^1_3 \leftarrow \mathcal{D}_{\mathsf{err}}$ are public key error terms, $e^1_2 \leftarrow \mathcal{D}_{\mathsf{efl}}$ is a flooding error for threshold partial decryption, $\mathsf{fl}^1_{lsp} \leftarrow \mathcal{D}_{\mathsf{sfl}}$ is a flooding term to mask the less-sparse secret key, r_1 are uniformly random bits, and a_1, a_2, a_3 are uniform ring elements known by the adversary. We slightly simplified pk^1_{lsp} by removing the $\frac{1}{n} \cdot \mathsf{m}$ term, as it is known to the adversary. The security analysis proceeds by a sequence of hybrid games.

Hyb$_0$. In this first hybrid game, the adversary's view is the above simplified view.

Hyb$_1$. In this game, we change how p^1_{lsp} is computed and define it as:

$$p^1_{lsp} = \Delta \cdot (\mathsf{sk}_{lsp} + \mathsf{fl}^1_{lsp}) - \left(b_2 + \sum_{1 < j \le n} a_2 \cdot \mathsf{sk}^j_{tmp}\right) + e_{\mathsf{sim}} \ ,$$

where $e_{\mathsf{sim}} \leftarrow \mathcal{D}_{\mathsf{efl}}$. We claim that the adversary's view in **Hyb**$_1$ is statistically close to that in **Hyb**$_0$. Indeed, correctness of decryption implies that $\mathsf{ct}_{lsp} = (b_2, a_2)$ satisfies $b_2 + a_2 \cdot \mathsf{sk}_{tmp} = \Delta \cdot \mathsf{sk}_{lsp} + e'$ for some small-magnitude error e', where $\mathsf{sk}_{tmp} = \sum_{1 \le i \le n} \mathsf{sk}^i_{tmp}$. Therefore, we have:

$$p^1_{lsp} = \Delta \cdot (\mathsf{sk}_{lsp} + \mathsf{fl}^1_{lsp}) + e' - \left(b_2 + \sum_{1 < j \le n} a_2 \cdot \mathsf{sk}^j_{tmp}\right) + e^1_2 \ .$$

Since e^1_2 is exponentially larger than e', by a standard noise flooding argument, we can define p^1_{lsp} as:

$$p^1_{lsp} = \Delta \cdot (\mathsf{sk}_{lsp} + \mathsf{fl}^1_{lsp}) + e' - \left(b_2 + \sum_{1 < j \le n} a_2 \cdot \mathsf{sk}^j_{tmp}\right) + e_{\mathsf{sim}} \ ,$$

where $e_{\mathsf{sim}} \leftarrow \mathcal{D}_{\mathsf{efl}}$, and where sk_{lsp} is defined as the key generated by running the less-sparse key generation algorithm (in plain) using $\sum_{1\leq j\leq n} r_j \bmod 2$ as random coins. $\mathbf{Hyb}_0$ and $\mathbf{Hyb}_1$ are statistically close thanks to flooding, and thanks to the perfect correctness of the decryption.

$\mathbf{Hyb}_2$. In this game, we replace $\mathsf{ct}^1_{lsp,r}$ by an encryption of 0's, but still use uniformly random coins u to generate the less sparse key. We recall that the less sparse key is generated in plain since $\mathbf{Hyb}_1$, so this does not affect the distribution of the adversary's view, except for $\mathsf{ct}^1_{lsp,r}$. $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_2$ are computationally indistinguishable, based on the IND-CPA security of the encryption scheme.

$\mathbf{Hyb}_3$. In this game, we generate the public key pk_{lsp} directly as $\mathsf{pk}_{lsp} := (b_3, a_3)$ with $b_3 \leftarrow -a_3 \cdot \mathsf{sk}_{lsp} + e_3$. We also now set $\mathsf{pk}^1_{lsp} := b_3 - \sum_{1<j\leq n} \mathsf{pk}^j_{lsp}$. This does not affect the view of the adversary and $\mathbf{Hyb}_2$ and $\mathbf{Hyb}_3$ are identically distributed.

$\mathbf{Hyb}_4$. In this game, we again change how p^1_{lsp} is computed and define it as:

$$p^1_{lsp} = \Delta \cdot \mathsf{fl}_{\mathsf{sim}} - \left(b_2 + \sum_{1<j\leq n} a_2 \cdot \mathsf{sk}^j_{tmp} \right) + e_{\mathsf{sim}} \ ,$$

where $\mathsf{fl}_{\mathsf{sim}} \leftarrow \mathcal{D}_{\mathsf{sfl}}$ and $e_{\mathsf{sim}} \leftarrow \mathcal{D}_{\mathsf{efl}}$. We again apply a standard noise flooding argument to claim that $\mathbf{Hyb}_4$ is statistically close to $\mathbf{Hyb}_3$ since fl^1_{lsp} is exponentially larger than sk_{lsp}.

$\mathbf{Hyb}_5$. In this game, we replace pk^1_{tmp} by a uniformly random ring element. $\mathbf{Hyb}_4$ and $\mathbf{Hyb}_5$ are computationally indistinguishable assuming Ring-LWE. We remind that $\mathsf{pk}_{tmp} = (b_{tmp}, a_i)$ where $b_{tmp} := \sum_{j\in[n]} \mathsf{pk}^j_{tmp}$.

$\mathbf{Hyb}_6$. In this last hybrid, we now define $\mathsf{pk}_{tmp} = (b_{tmp}, a_1)$ where b_{tmp} is a uniformly random ring element, and sample pk^1_{tmp} as $b_{tmp} - \sum_{1<j\leq n} \mathsf{pk}^j_{tmp}$. This does not change the adversary's view compared to $\mathbf{Hyb}_5$, but the adversary's view is now:

- Round 1: $\mathsf{pk}^1_{tmp} \leftarrow b_{tmp} - \sum_{1<j\leq n} \mathsf{pk}^j_{tmp}$, where b_{tmp} is a uniformly random ring element;
- Round 2: $\mathsf{ct}^1_{lsp,r} \leftarrow \mathtt{Enc}(\mathsf{pk}_{tmp}, 0)$;
- Round 3: $p^1_{lsp} \leftarrow \Delta \cdot \mathsf{fl}_{\mathsf{sim}} - (b_2 + \sum_{1<j\leq n} a_2 \cdot \mathsf{sk}^j_{tmp}) + e_{\mathsf{sim}}$;
- Round 4: $\mathsf{pk}^1_{lsp} \leftarrow \mathsf{pk}_{lsp} - \sum_{1<j\leq n} \mathsf{pk}^j_{lsp}$.

Overall, the adversary's view is sampleable given only pk_{lsp}, where pk_{lsp} is a public key associated with a less-sparse secret key obtained with non-threshold CKKS key generation. This completes the security analysis of our protocol.

E More Approaches for DKG

In this appendix, we discuss alternative approaches for DKG, using fewer rounds of communication.

E.1 Using Fewer Rounds

A solution with modified relinearization. The 4-round protocol consists of two main phases: (1) generate temporary keys and (2) generate the final keys with sparse and non-sparse secret keys. Each phase relies on [MTBH21] and consumes two communication rounds. Instead, we can use the modified relinearization from [KLSW24] to remove communication rounds. It relies on a special format of relinearization keys that can be generated with a single round of communication. The detailed protocol is given in App. E.2.

When using [KLSW24] for generating both temporary and final keys, one achieves a 2-round protocol. However, it outputs unconventional relinearization keys, which degrade the performance of the homomorphic computations.[7] Instead, one can rely on [KLSW24] only for the temporary keys and generate properly formed final keys using [MTBH21] in the second phase, for a total of three communication rounds. The parties can still generate the final sparse and non-sparse key sets using the algorithms in Sec. 3 and 4.1: indeed, in Sec. 3, we used homomorphic computations in a black-box manner, and it is possible to run the algorithms with the modified relinearization.

A two-round solution. To remove one round of communication in the second phase, it is possible to use more homomorphic computations to generate encrypted public and evaluation final keys from the encrypted final secret keys. Afterwards, each party obtains the public and evaluation keys via distributed decryption (with respect to the temporary keys).

A key-switching key from sk' to sk is a ciphertext $\mathtt{Enc}_{\mathsf{sk}}(P \cdot \mathsf{sk}')$ that decrypts to $P \cdot \mathsf{sk}'$ under sk. Note that P is an auxiliary prime (and recall that we focus on $\mathsf{dnum} = 1$ for simplicity). The relinearization key is a key switching key from sk^2 to sk, and the automorphism keys are the switching keys from $\phi(\mathsf{sk})$ to sk where ϕ is an automorphism. Our goal is to generate $\mathtt{Enc}_{\mathsf{sk}}(P \cdot \mathsf{sk}^2)$ and $\mathtt{Enc}_{\mathsf{sk}}(P \cdot \phi(\mathsf{sk}))$ from a given ciphertext $\mathsf{ct}_{\mathsf{sk}}$ that decrypts to sk.

We begin with the relinearization key. We sample the left part a of the key in clear using a hash function. We homomorphically generate the error $\mathsf{ct}_{\mathrm{err}}$ decrypting to err by homomorphically evaluating a PRF (such as [BIP+18]) and converting the bits to well-distributed Ring-LWE noise terms. Then, we compute

$$\mathsf{ct}_b = a \odot \mathsf{ct}_{\mathsf{sk}} + P \cdot (\mathsf{ct}_{\mathsf{sk}} \odot \mathsf{ct}_{\mathsf{sk}}) + \mathsf{ct}_{\mathrm{err}} \ .$$

The parties jointly decrypt ct_b and obtain the right part b of the relinearization key. By evaluation correctness, we have

$$b = a \cdot \mathsf{sk} + P \cdot \mathsf{sk} \cdot \mathsf{sk} + \mathrm{err} \ .$$

The plaintext arithmetic of the above homomorphic computations is a modular arithmetic over a large modulus PQ. To compute it efficiently, we consider P

[7] It is possible to convert the keys from [KLSW24] to conventional relinearization keys [BMM25]. However, this conversion consumes modulus, which reduces the achievable multiplicative depth between two consecutive bootstraps.

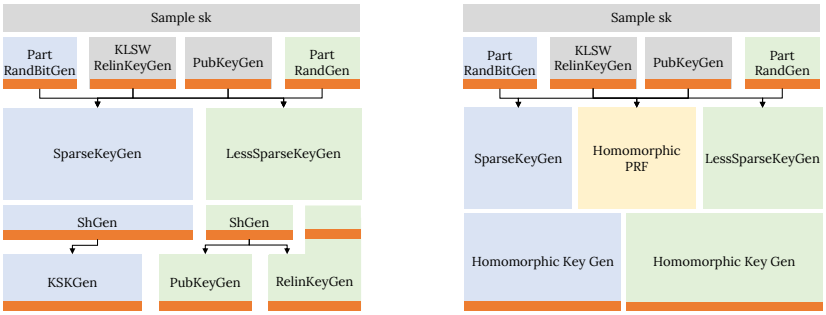
and Q that factor into small prime moduli p_i's.[8] Then, each homomorphic arithmetic modulo p_i can be implemented with BFV.

For the public key and automorphism keys, we compute

$$a \odot \mathsf{ct}_{\mathsf{sk}} + \mathsf{ct}_{\mathrm{err}} \quad \text{and} \quad a \odot \mathsf{ct}_{\mathsf{sk}} + P \cdot \phi(\mathsf{ct}_{\mathsf{sk}}) + \mathsf{ct}_{\mathrm{err}} \ .$$

Using this approach, one can sample properly-formed public and evaluation keys with a communication round in the second phase of the DKG protocol. Further, the noise terms in the keys are as in single-user CKKS and do not grow with the number of parties. The major cost is the homomorphic evaluation of PRF. To the best of our knowledge, each homomorphic PRF requires at least one bootstrap. Thus, to generate ℓ evaluation keys, it seems we need $\Omega(\ell)$ bootstraps, which makes it computationally more demanding than the previous approaches.

Fig. 12 visualizes the 3-round and 2-round DKG protocols.



(a) Visualization of the 3-round protocol

(b) Visualization of the 2-round protocol

Fig. 12: Visualizations of the 3-round and 2-round protocols. The gray rectangles represent temporary key generation, while the blue and green rectangles generate keys for sparse and less sparse secrets, respectively. The yellow rectangle denotes the homomorphic evaluation of a PRF to sample noise. The horizontal orange bars represent the communication steps between parties.

E.2 Modified Relinearization from [KLSW24]

We recall a modified relinearization procedure with modified relinearization keys introduced in [KLSW24]. The modified key generation protocol requires only a single round of communication. The key generation algorithm is described in Fig. 13, and the modified relinearization algorithm is described in Fig. 14.

[8] The modulus PQ is typically already a product of small primes for efficient ciphertext arithmetic based on the number-theoretic transform [CHK+19, CCK+25].

$\mathsf{KLSWRelinKeyGen}(\mathsf{sk}_i)$

Round 1: Given sk_i:

1: sample common $a, u \leftarrow \mathcal{H}$
2: sample $r_i \leftarrow \mathcal{D}_{\mathsf{sk}}$
3: sample $e_{i,0}, e_{i,1}, e_{i,2} \leftarrow \mathcal{D}_{\mathsf{err}}$
4: $h_{i,0} \leftarrow -\mathsf{sk}_i \cdot a + e_{i,0}$
5: $h_{i,1} \leftarrow -r_i \cdot a + P \cdot \mathsf{sk}_i + e_{i,1}$
6: $h_{i,2} \leftarrow -\mathsf{sk}_i \cdot u - P \cdot r_i + e_{i,2}$
7: **return** $(h_{i,0}, h_{i,1}, h_{i,2})$

Output: Given $(k_i)_{i\in[n]}$

1: $\mathsf{rlk} \leftarrow \left(\sum_{1\le i\le n} h_{i,0}, \sum_{1\le i\le n} h_{i,1}, \sum_{1\le i\le n} h_{i,2}\right)$
2: **return** rlk

Fig. 13: Key generation for modified relinearization from [KLSW24].

$\mathsf{KLSWRelin}(\mathsf{ct}_{\mathrm{mul}} = (c_0, c_1, c_2), \mathsf{rlk} = (h_0, h_1, h_2))$

1: $c_2' \leftarrow \lfloor \frac{1}{P} c_2 h_0 \rceil$
2: $\mathsf{ct}_0^* \leftarrow c_0 + \lfloor \frac{1}{P} c_2' h_2 \rceil$
3: $\mathsf{ct}_1^* \leftarrow c_1 + \lfloor \frac{1}{P} c_2 h_1 \rceil + \lfloor \frac{1}{P} c_2' u \rceil$
4: **return** $\mathsf{ct}_{\mathrm{out}} = (\mathsf{ct}_0^*, \mathsf{ct}_1^*)$

Fig. 14: Modified relinearization procedure from [KLSW24].

Based on the modified relinearization algorithms, one can construct a 3-round DKG protocol. For this purpose, we introduce in Fig. 15 a modified random bit generation sub-protocol which is slightly modified from Fig. 4. To put it all together, we obtain a 3-round DKG protocol, provided in Fig. 16. The parts in red correspond to the differences from the original 4-round DKG protocol (Fig. 6).

$\mathsf{PartRandBitsGen}(1^n, 1^\ell, \{\mathsf{sk}_i\}_{i\in[n]})$

Round 1: Given $\{\mathsf{sk}_i\}_{i\in[n]}$:

1: sample a common $a \leftarrow \mathcal{H}$
2: sample $r^{(i)} \leftarrow U(R_{2^\ell})$
3: sample $e^{(i)} \leftarrow \mathcal{D}_{\mathrm{err}}$
4: $\mathsf{ct}^{(i)} \leftarrow (a, a \cdot \mathsf{sk}_i + \Delta \cdot r^{(i)} + e^{(i)})$
5: **return** $\mathsf{ct}^{(i)}$

Output: Given $(\mathsf{ct}^{(j)})_{j\in[n]}$

1: $\mathsf{ct} \leftarrow \sum_{1\le j\le n} \mathsf{ct}^{(j)}$
2: $(\mathsf{ct}_0^k, \mathsf{ct}_1^k)_{k\in[\ell]} \leftarrow \mathsf{BitExtract}_\ell(\mathsf{SI\text{-}BTS}(\mathsf{ct}))$
3: **return** $(\mathsf{ct}_0^k, \mathsf{ct}_1^k)_{k\in[\ell]}$

Fig. 15: Generation of random bits. The algorithm without the last bit extraction step is $\mathsf{PartRandGen}(1^n, \rho, \{\mathsf{sk}_i\}_{i\in[n]})$, which outputs a slots-encoded CKKS encryption of uniform random vector drawn from $\mathbb{Z}_\rho^{N/2}$.

E.3 Estimating the Evaluation Efficiency Degradation Without Using the DKG Protocol

We now compare the EvalMod level consumption for $\mathcal{K} = 160$, compared to $\mathcal{K} = 16$ which we can use after running the DKG protocol. We recall that the function $x \mapsto x \bmod 1$ is approximated by turning x into x/γ for some large γ, and using the approximation $x \bmod 1 \approx \frac{\gamma}{2\pi}\sin(2\pi x/\gamma)$.

For $\mathcal{K} = 16$, if using $\gamma = 2^{-10}$ and a minimax approximation of degree-30 of $\cos(\pi x/4)$ over $[-16.25, 16.25]$ and three iterations of the double angle formula, we obtain an approximation of $\sin(2\pi x)/2\pi = \cos(2\pi(x-1/4))/2\pi$ over $[-16, 16]$ with ≈ 27 bits of precision, so an approximation of $\frac{\gamma}{2\pi}\sin(2\pi x/\gamma)$ with ≈ 17 bits of precision. This requires 8 multiplicative levels overall. For $\mathcal{K} = 160$, the most natural option available is to use a degree-58 approximation of $\cos(\pi x/16)$ and five iterations of the double angle formula, giving a precision of ≈ 18 bits.

For comparison purposes, we define the throughput as the number of remaining levels divided by the bootstrapping time. In a standard implementation with $N = 2^{16}$, using $\mathcal{K} = 16$ would leave us with 10 multiplicative levels available for homomorphic computations. The case $\mathcal{K} = 160$ requires 11 multiplicative levels; we thus are left with 7 levels of computation rather than 10. Further, as the

Round 1:

1: sample $\mathsf{sk}^i_{tmp} \leftarrow \mathcal{D}_{\mathsf{tmp}}$
2: $\mathsf{pk}^i_{tmp} \leftarrow \mathsf{PubKeyGen}_1(\mathsf{sk}^i_{tmp})$
3: $h^i_{tmp} \leftarrow \mathsf{KLSWRelinKeyGen}_1(\mathsf{sk}^i_{tmp})$
4: $\mathsf{ct}^i_{sp,r} \leftarrow \mathsf{PartRandBitsGen}_1(\mathsf{sk}^i_{tmp})$
5: $\mathsf{ct}^i_{lsp,r} \leftarrow \mathsf{PartRandGen}_1(\rho, \mathsf{sk}^i_{tmp})$
6: **broadcast** $(\mathsf{pk}^i_{tmp}, h^i_{tmp}, \mathsf{ct}^i_{sp,r}, \mathsf{ct}^i_{lsp,r})$

Round 2: Given $(\mathsf{sk}^i_{tmp}, (\mathsf{pk}^j_{tmp}, h^j_{tmp})_{j\in[n]})$:

1: $\mathsf{pk}_{tmp} \leftarrow \mathsf{PubKeyGen}_{\mathsf{out}}((\mathsf{pk}^j_{tmp})_{j\in[n]})$
2: $\mathsf{rlk}_{tmp} \leftarrow \mathsf{KLSWRelinKeyGen}_{\mathsf{out}}((h^j_{tmp})_{j\in[n]})$
3: $\mathsf{ct}_{sp,r} \leftarrow \mathsf{PartRandBitsGen}_{\mathsf{out}}((\mathsf{ct}^j_{sp,r})_{j\in[n]})$
4: $\mathsf{ct}_{lsp,r} \leftarrow \mathsf{PartBitGen}_{\mathsf{out}}(\rho, (\mathsf{ct}^j_{lsp,r})_{j\in[n]})$
5: $\mathsf{ct}_{sp} \leftarrow \mathsf{SparseKeySampler}(\mathsf{ct}_{sp,r})$
6: $\mathsf{ct}_{lsp} \leftarrow \mathsf{LessSparseKeySampler}(\mathsf{ct}_{lsp,r})$
7: $p^i_{sp} \leftarrow \mathsf{ShGen}_1(\mathsf{ct}_{sp}, \mathsf{sk}^i_{tmp})$, internally sampling a share fl^i_{sp}
8: $p^i_{lsp} \leftarrow \mathsf{ShGen}_1(\mathsf{ct}_{lsp}, \mathsf{sk}^i_{tmp})$, internally sampling a share fl^i_{lsp}
9: $h^i_{lsp} \leftarrow \mathsf{RelinKeyGen}_1(\mathsf{fl}^i_{lsp})$
10: **broadcast** $(p^i_{sp}, p^i_{lsp}, h^i_{lsp})$

Round 3: Given $(\mathsf{sk}_i, \mathsf{fl}^i_{sp}, \mathsf{fl}^i_{lsp}, (p^j_{sp}, p^j_{lsp}, h^j_{lsp})_{j\in[n]})$:

1: $(\mathsf{sk}^i_{sp}, \mathsf{m}_{sp}) \leftarrow \mathsf{ShGen}_{\mathsf{out}}(\mathsf{ct}_{sp}, \mathsf{sk}_i, \mathsf{fl}^i_{sp}, (p^j_{sp})_{j\in[n]})$
2: $(\mathsf{sk}^i_{lsp}, \mathsf{m}_{lsp}) \leftarrow \mathsf{ShGen}_{\mathsf{out}}(\mathsf{ct}_{lsp}, \mathsf{sk}_i, \mathsf{fl}^i_{lsp}, (p^j_{lsp})_{j\in[n]})$
3: $\mathsf{pk}^i_{lsp} \leftarrow \mathsf{PubKeyGen}_1(\mathsf{sk}^i_{lsp})$
4: $h^{n+1}_{lsp} \leftarrow \mathsf{RelinKeyGen}_1(\mathsf{m}_{lsp})$
5: $k^i_{lsp} \leftarrow \mathsf{RelinKeyGen}_2(\mathsf{fl}^i_{lsp}, (h^j_{lsp})_{j\in[n+1]})$
6: $k^{n+1}_{lsp} \leftarrow \mathsf{RelinKeyGen}_2(\mathsf{m}_{lsp}, (h^j_{lsp})_{j\in[n+1]})$
7: $\mathsf{ksk}^i_{sp\to lsp} \leftarrow \mathsf{KSKeyGen}_1(\mathsf{sk}^i_{sp}, \mathsf{sk}^i_{lsp}, PQ_L)$
8: $\mathsf{ksk}^i_{lsp\to sp} \leftarrow \mathsf{KSKeyGen}_1(\mathsf{sk}^i_{lsp}, \mathsf{sk}^i_{sp}, PQ_0)$
9: **broadcast** $(\mathsf{pk}^i_{lsp}, k^i_{lsp}, \mathsf{ksk}^i_{sp\to lsp}, \mathsf{ksk}^i_{lsp\to sp})$

Output: Given $(\mathsf{sk}^i_{lsp}, (\mathsf{pk}^j_{lsp}, k^j_{lsp}, \mathsf{ksk}^j_{sp\to lsp}, \mathsf{ksk}^j_{lsp\to sp})_{j\in[n]})$:

1: $\mathsf{pk}_{lsp} \leftarrow \mathsf{PubKeyGen}_{\mathsf{out}}((\mathsf{pk}^j_{lsp})_{j\in[n]})$
2: $\mathsf{rlk}_{lsp} \leftarrow \mathsf{RelinKeyGen}_{\mathsf{out}}((k^j_{lsp})_{j\in[n+1]})$
3: $\mathsf{ksk}_{sp\to lsp} \leftarrow \mathsf{KSKeyGen}_{\mathsf{out}}((\mathsf{ksk}^j_{sp\to lsp})_{j\in[n]})$
4: $\mathsf{ksk}_{lsp\to sp} \leftarrow \mathsf{KSKeyGen}_{\mathsf{out}}((\mathsf{ksk}^j_{lsp\to sp})_{j\in[n]})$
5: $\mathsf{pk} \leftarrow \mathsf{pk}_{lsp}$
6: $\mathsf{evk} \leftarrow (\mathsf{rlk}_{lsp}, \mathsf{ksk}_{lsp\to sp}, \mathsf{ksk}_{sp\to lsp})$
7: $\mathsf{sk}_i \leftarrow \mathsf{sk}^i_{lsp}$
8: **return** $(\mathsf{pk}, \mathsf{evk}, \mathsf{sk}_i)$

Fig. 16: 3-Round Distributed Key Generation protocol based on [KLSW24].

complexity of polynomial evaluation for a polynomial of degree d grows as $\sqrt{d}$, we expect an increase of 40% of EvalMod time, hence an increase of $\approx 20\%$ of total bootstrapping time. The throughput thus decreases by a factor $1.2/7 \cdot 10 \approx 1.71$.

We stress that this estimate is conservative; for once, the levels used in EvalMod use typically more modulus than ordinary multiplicative levels (so the saving of three EvalMod levels would rather correspond to four ordinary multiplicative levels); further, the increases in $\mathcal{K}$ and in the number of double angle formulas lead to stronger precision needs in EvalMod, requiring even larger scaling factors and likely consuming yet another multiplicative level.

F Delegated Key Generation Illustration

In Fig. 17, we provide an illustration of our delegated key generation approach, described in Sec. 4.5.

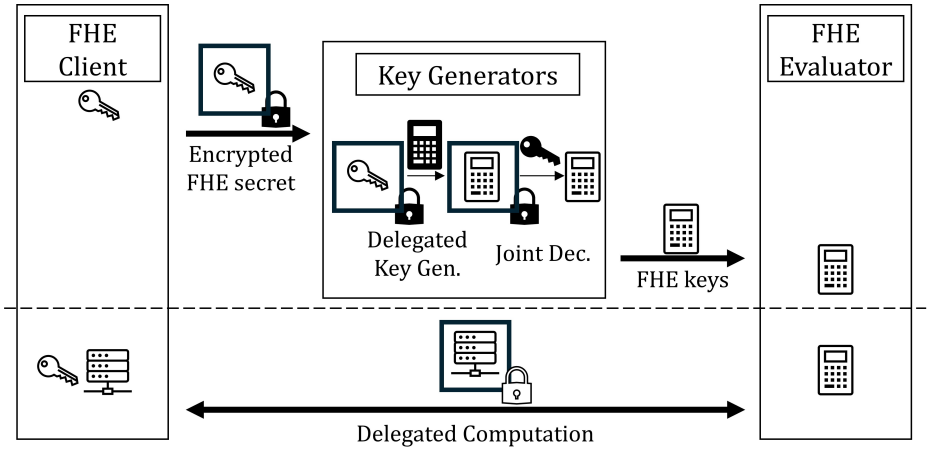


Fig. 17: Visualization of delegated key generation.

G Implementation Optimizations

Randomness generation. In the protocol given in Sec. 4.2, a sparse key and a less-sparse key are homomorphically generated. We choose to sample the less-sparse key as a dense key, i.e., uniform ternary. Its generation is equivalent to sampling a uniform vector over $\mathbb{Z}_3^N$, as discussed in Sec. 3.4. In our implementation, we generate this less-sparse key along with the random bits for the sparse key generation. By doing so, we can decrease the number of bootstraps required during the computation. For this, the i-th party (for $i \leq n$)

initially samples two degree-$(N/2)$ polynomials $m_i^1, m_i^2 \in R$, where the coefficients of m_i^1 are uniformly chosen from $\mathbb{Z}_8$, while those of m_i^2 are uniformly chosen from $\mathbb{Z}_9$. Then the i-th party discloses a public key encryption of the plaintext $\lfloor (Q_0/8)m_i^1 \rceil + \lfloor (Q_0/9)m_i^2 \rceil \in R_{Q_0}$. The modular addition of these messages is homomorphically obtained by summing these ciphertexts modulo Q_0. At this stage, the SI-BTS algorithm is employed to bootstrap this ciphertext and retrieve the randomnesses. Finally, we evaluate polynomials that extract the top, middle, and bottom bits from integers modulo 8, and the top and bottom trits (digits in base 3) from integers modulo 9. Then, three uniform binary vectors of dimension $N/2$ can be used for the sparse key generation, and two uniform ternary vectors of dimension $N/2$ can be used as less-sparse key.

Homomorphic cleaning. A strategy for handling binary data with CKKS was described in [DMPS24]. It consists in viewing bits as real numbers, and making them more accurate by homomorphically removing the noise introduced from the encryption and homomorphic operations. This process, which we will call cleaning, can be performed with a polynomial evaluation.

In the DKG protocol, we clean bits and trits. To clean a bit $x \in \{0, 1\}$ inside the one-hot vector generation algorithm, we evaluate the polynomial $p_1(x) = -2x^3 + 3x^2$, initially considered in [CKK20]. We use the variant from [CKSS25] to reduce modulus consumption.

In order to clean a vector in $\mathbf{u} \in \{-1, 0, 1\}^N$, we first separate -1 and 1 by applying the two maps $m_1 : x \mapsto (x^2 + x)/2$ and $m_2 : x \mapsto (x^2 - x)/2$ coordinate-wise to $\mathbf{u}$, to obtain $\mathbf{u}_1$ and $\mathbf{u}_2$ in $\{0, 1\}^N$. We apply the bits cleaning process to $\mathbf{u}_1$, $\mathbf{u}_2$ and recombine them via $\mathbf{u} = \mathbf{u}_1 - \mathbf{u}_2$.

In our implementation, cleaning is used twice. Initially, we clean the vector $\mathbf{v}_i$ just before rotate-and-sum in the sparse binary sampling algorithm of Fig. 3. Indeed, during rotate-and-sum, the noise can be amplified, possibly compromising correctness. We clean $\mathbf{v}_i$ by sequentially evaluating p_1 twice. Finally, the output ciphertexts of the sparse/dense keys are cleaned to create a gap of more than 80 bits between the message and the noise, to give space for noise flooding in the additive share generation. Overall, cleaning allows us to perform most of the computations at limited precision, which helps for efficiency (less modulus consumption leads to fewer bootstraps) and to increase the precision when needed.

Scaled Chebyshev polynomials. In the DKG protocol, several boundary integer indicator functions are evaluated. These are implemented with the algorithm given in Fig. 1. The initial plaintext-ciphertext multiplication by $-2c/(B-1)$ consumes one level. If the input can be scaled beforehand as part of prior computations, then we do not need to sacrifice this level. In our implementation, all boundary integer indicator functions are implemented in this way.

H Estimating the Number of Periods for the Bootstraps Inside the DKG Protocol

The DKG protocol from Fig. 6 and implemented as described in Sec. 5 performs two bootstraps. These are achieved using a temporary secret key built as the sum of N-dimensional shares generated independently by the n participants, each of those shares being a sparse ternary key with Hamming weight h.

To design an implementation of EvalMod with negligible failure probability for these bootstraps, we compute a bound $\mathcal{K}$ for the absolute value of the quantity I appearing in Eq. (2). For this purpose, we rely on the common assumption that, after rescaling by Q_0, the coefficients of c_0 and c_1 behave like independent uniform random variables in $Q_0^{-1}\{(1-Q_0)/2, \ldots, (Q_0-1)/2\}$.

Classically, the bound on $|I|$ is derived by obtaining an upper bound on $\|c_0 + c_1 \cdot \mathsf{sk}\|_\infty$, where sk is the secret key used during $\mathsf{ModRaise}$ and the computation $c_0 + c_1 \cdot \mathsf{sk}$ is performed over R. In this section, we assume that Q_0 is odd.

H.1 Approach Overview

Let $(\mathbf{v}^{(i)})_{0 \le i < n}$ be n vectors of $\{-1, 0, 1\}^N$ that are independently and uniformly sampled among those of Hamming weight h, and define $\mathbf{s} = (s_j)_{0 \le j < N} = \sum_{0 \le i < n} \mathbf{v}^{(i)} \in \mathbb{Z}^N$. For $\mathbf{u}$ a uniform vector in $\{(1-Q_0)/(2Q_0), \ldots, (Q_0 - 1)/(2Q_0)\}$ for some odd integer Q_0, we want to find a bound B such that

$$\Pr\left(|\langle \mathbf{s}, \mathbf{u} \rangle| \ge B\right) \le \frac{2^{-128}}{N} \ .$$

Here, the vector $\mathbf{u}$ models the coefficients of c_1 while $\mathbf{s}$ models the coefficients of sk; the inner product $\langle \mathbf{u}, \mathbf{s} \rangle$ models one coefficient of the product $c_1 \cdot \mathsf{sk}$. The factor N is to allow for the use of the union bound across N inner products and bound $\|c_1 \cdot \mathsf{sk}\|_\infty$ while keeping a probability below 2^{-128}.

For each $0 \le i < n, 0 \le j < N$, we write $v_j^{(i)} = \sigma_{ij} w_{ij}$, with σ_{ij} uniform and independent in $\{-1, 1\}$ and $w_{ij} \in \{0, 1\}$. We define $d_j = \sum_{0 \le i < n} w_{ij}$ for all $0 \le j < N$.

Our strategy is to establish a Chernoff-type bound by bounding the moment generating function $\tau \mapsto \mathbb{E}[\exp(\tau \langle \mathbf{s}, \mathbf{u} \rangle)]$. We evaluate this expectation in steps, first integrating with respect to the signs σ_{ij}, then with respect to the supports w_{ij}'s (through the d_j's), and finally with respect to the coordinates of $\mathbf{u}$. The key ingredient is the observation that the d_j's are *negatively associated* [JDP83], which allows us to separate them as if they were independent.

H.2 Integrating with Respect to the Signs

We have, for any $\tau > 0$:

$$\begin{aligned}\mathbb{E}\left[\exp(\tau\langle \mathbf{s}, \mathbf{u}\rangle)|\mathbf{u}, (w_{ij})_{ij}\right] &= \prod_{\substack{0\leq i<n\\0\leq j<N}} \mathbb{E}\left[\exp(\tau u_j w_{ij}\sigma_{ij})|\mathbf{u}, (w_{ij})_{ij}\right] \\ &= \prod_{\substack{0\leq i<n\\0\leq j<N}} \cosh(\tau u_j w_{ij}) \\ &= \prod_{0\leq j<N} (\cosh(\tau u_j))^{d_j} \quad .\end{aligned}$$

H.3 Negative Association, and Integrating with Respect to the Supports

Following [JDP83], we say that a family of random variables $(X_i)_{0\leq i<m}$ is Negatively Associated (NA) if for any disjoint $A, B \subseteq \{0, \ldots, m-1\}$ of cardinalities a and b, for any componentwise non-decreasing functions $f : \mathbb{R}^a \to \mathbb{R}$, $g : \mathbb{R}^b \to \mathbb{R}$, then $f((X_j)_{j\in A})$ and $g((X_j)_{j\in B})$ have non-positive covariance. By [JDP83, 3.1(c)], the random variables $(w_{ij})_{0\leq j<N}$ are NA for any fixed $0 \leq i < n$. By [JDP83, Property P7], we obtain that all w_{ij}'s are jointly NA. Finally, by [JDP83, Property P6], we have that the d_j's, for $0 \leq j < N$, are NA. We use this property to bound $\mathbb{E}[\exp(\tau\langle \mathbf{s}, \mathbf{u}\rangle)|\mathbf{u}]$, as follows.

$$\begin{aligned}&\mathbb{E}_{(\sigma_{ij})_{ij},(w_{ij})_{ij}}[\exp(\tau\langle \mathbf{s}, \mathbf{u}\rangle)|\mathbf{u}] \\ &\qquad = \mathbb{E}_{(w_{ij})_{ij}}\left[\mathbb{E}_{(\sigma_{ij})_{ij}}\left[\exp(\tau\langle \mathbf{s}, \mathbf{u}\rangle)\middle|\mathbf{u}, (w_{ij})_{ij}\right]\right] \\ &\qquad = \mathbb{E}\left[\prod_{0\leq j<N} (\cosh(\tau u_j))^{d_j}\,|\mathbf{u}\right] \\ &\qquad \leq \prod_{0\leq j<N} \mathbb{E}\left[(\cosh(\tau u_j))^{d_j}\,|\mathbf{u}\right] \quad .\end{aligned}$$

The last inequality comes from [JDP83, Property P2], using the fact that $d_j \mapsto (\cosh(\tau u_j))^{d_j}$ is a non-decreasing function. Now, note that d_j follows a binomial distribution with parameters $(n, h/N)$. This gives the following upper bound

$$\begin{aligned}&\mathbb{E}[\exp(\tau\langle \mathbf{s}, \mathbf{u}\rangle)|\mathbf{u}] \\ &\qquad \leq \prod_{0\leq j<N} \sum_{0\leq d_j\leq n} \binom{n}{d_j} \left(1-\frac{h}{N}\right)^{n-d_j} \left(\frac{h}{N}\right)^{d_j} \cosh(\tau u_j)^{d_j} \\ &\qquad = \prod_{0\leq j<N} \left(1 - \frac{h}{N} + \frac{h}{N}\cosh(\tau u_j)\right)^n .\end{aligned}$$

H.4 Last Integration

As the variables $(u_j)_{0\le j<N}$ are independent, we have

$$\begin{aligned}\mathbb{E}\left[\exp(\tau\langle \mathbf{s},\mathbf{u}\rangle)\right] &\le \mathbb{E}\left[\prod_{0\le j<N}\left(1-\frac{h}{N}+\frac{h}{N}\cosh(\tau u_j)\right)^n\right]\\ &= \left(\mathbb{E}\left[\left(1-\frac{h}{N}+\frac{h}{N}\cosh(\tau u_1)\right)^n\right]\right)^N\\ &= \left(Q_0^{-1}\cdot\sum_{u=(1-Q_0)/2}^{(Q_0-1)/2}\left(1-\frac{h}{N}+\frac{h}{N}\cosh\frac{\tau u}{Q_0}\right)^n\right)^N \quad .\end{aligned}$$

The inner sum corresponds to the midpoint rule approximation for a Q_0-step subdivision of the integral

$$\mathcal{I}(\tau)=\int_{-1/2}^{1/2}\left(1-\frac{h}{N}+\frac{h}{N}\cosh(\tau x)\right)^n \mathrm{d}x \ .$$

As the function which is integrated is convex, the midpoint rule gives an under-approximation. We deduce the following:

$$\mathbb{E}\left[\exp(\tau\langle \mathbf{s},\mathbf{u}\rangle)\right] \le \mathcal{I}(\tau)^N \ .$$

H.5 Tail Bound

Using Markov's inequality, we deduce the following tail bound, valid for all $t,\tau>0$:

$$\begin{aligned}\Pr\left[|\langle \mathbf{s},\mathbf{u}\rangle|>t\right] &= 2\cdot\Pr\left[\langle \mathbf{s},\mathbf{u}\rangle>t\right]\\ &= 2\cdot\Pr\left[\exp(\tau\langle \mathbf{s},\mathbf{u}\rangle)>\exp(\tau t)\right]\\ &\le 2\cdot\exp(-\tau t)\cdot\mathbb{E}\left[\exp(\tau\langle \mathbf{s},\mathbf{u}\rangle)\right]\\ &\le 2\cdot\exp(-\tau t)\cdot\mathcal{I}(\tau)^N \ .\end{aligned}$$

We note that the integral $I(\tau)$ can be rigorously computed by expanding the n-th power and using

$$\int_{-1/2}^{1/2}\cosh(\tau u)^k \mathrm{d}u = 2^{-k}\sum_{0\le\ell\le k}\binom{k}{\ell}\frac{\sinh\frac{\tau(2\ell-k)}{2}}{\frac{\tau(2\ell-k)}{2}} \ .$$

H.6 Numerical Evaluation

Our experiments in Sage show that the integral is accurately approximated by numerical evaluation, at least for our ranges of parameters. We obtain that

- for $n = 32$, it holds that $\Pr\left[|\langle \mathbf{u}, \mathbf{s}\rangle| > 132.9\right] < 2^{-144}$, using $\tau = 1.470$;
- for $n = 256$, it holds that $\Pr\left[|\langle \mathbf{u}, \mathbf{s}\rangle| > 371.3\right] < 2^{-144}$, using $\tau = 0.539$;
- for $n = 1024$, it holds that $\Pr\left[|\langle \mathbf{u}, \mathbf{s}\rangle| > 741.5\right] < 2^{-144}$, using $\tau = 0.271$.

By using the union bound over $N = 2^{16}$ variables $\langle \mathbf{u}, \mathbf{s}\rangle$ and adding $1/2$ for both the b-part and m, we obtain bounds $\mathcal{K}_n$ on I that hold with probability $\geq 1 - 2^{-128}$ for a specific choice of n. The above gives $\mathcal{K}_{32} = 133$, $\mathcal{K}_{256} = 372$ and $\mathcal{K}_{1024} = 742$.

H.7 Heuristic Closed-Form Estimate

Heuristically, the asymptotic expansion of the integral in powers of nh/N gives $\mathcal{I}(\tau) = 1 + (\tau^2 nh)/(24N) \cdot (1 + O(nh/N))$ when $nh = o(N)$. This suggests that Markov's inequality upper bound is $\approx 2\exp(-\tau t + \tau^2 nh/24)$. The latter quantity is minimized for $\tau^* \approx 12t/nh$, giving

$$\Pr(|\langle \mathbf{s}, \mathbf{u}\rangle| > t) \lessapprox 2\exp\left(-6\frac{t^2}{nh}\right) \ .$$

Hence, a tail bound with probability $1 - p$ corresponds to

$$t_p \approx \sqrt{\frac{nh}{6}\ln\frac{2}{p}} \ .$$

In our examples, this heuristic bound is a few units below the rigorously obtained values.