Griffin: Towards Mixed Multi-Key Homomorphic Encryption

Thomas Schneider[a], Hossein Yalame[b] and Michael Yonli
ENCRYPTO Group, Technical University of Darmstadt, Darmstadt, Germany

Keywords: Secure Computation, Multi-Key Homomorphic Encryption.

Abstract: This paper presents Griffin, an extension of the mixed-scheme single-key homomorphic encryption framework Pegasus (Lu et al., IEEE S&P'21) to a Multi-Key Homomorphic Encryption (MKHE) scheme with applications to secure computation. MKHE is a generalized notion of Homomorphic Encryption (HE) that allows for operations on ciphertexts encrypted under different keys. However, an efficient approach to evaluate both polynomial and non-polynomial functions on encrypted data in MKHE has not yet been developed, hindering the deployment of HE to real-life applications. Griffin addresses this challenge by introducing a method for transforming between MKHE ciphertexts of different schemes. The practicality of Griffin is demonstrated through benchmarks with multiple applications, including the sorting of sixty four 45-bit fixed point numbers with a precision of 7 bits in 21 minutes, and evaluating arbitrary functions with a one-time setup communication of 1.4 GB per party and 2.34 MB per ciphertext. Moreover, Griffin could compute the maximum of two numbers in 3.2 seconds, a 2× improvement over existing MKHE approaches that rely on a single scheme.

1 INTRODUCTION

As data collection reaches unprecedented volumes and consumer awareness and apprehension regarding the usage of their personal information grow, the implementation of efficient privacy protection measures gets more and more important. Secure multi-party computation techniques (MPC) (Yao, 1986; Goldreich et al., 1987) allow to address this problem efficiently in a variety of real-world applications, including privacy-preserving machine learning (PPML) (Riazi et al., 2018; Boemer et al., 2020; Patra et al., 2021a; Brüggemann et al., 2023), clustering (Hegde et al., ; Keller et al., 2021), and federated learning (Nguyen et al., 2022; Fereidooni et al., 2021; Gehlhar et al., 2023). One promising approach is Homomorphic Encryption (HE), which allows for performing arbitrary computations over encrypted data. HE has been used as a building block in various applications such as PPML (Juvekar et al., 2018; Boemer et al., 2020; Mishra et al., 2020). Compared to traditional MPC methods, HE-based methods have the advantage of lower communication and round complexity (Chen et al., 2019b), but they come at the cost of higher computational complexity and runtime.

Existing HE schemes can be broadly categorized into two types: binary schemes such as FHEW (Ducas and Micciancio, 2015) and TFHE (Chillotti et al., 2020), and arithmetic schemes such as BFV (Fan and Vercauteren, 2012), BGV (Brakerski et al., 2014), and CKKS (Cheon et al., 2017). Arithmetic schemes typically support Single Instruction Multiple Data (SIMD) operations and are well-suited for evaluating functions which can be easily represented as a polynomial. However, evaluating non-linear functions, such as square roots or finding the minimum of two elements, requires numerical approximations, which can be very expensive to compute due to the need for deep circuits to achieve accurate results. On the other hand, binary schemes can easily evaluate binary circuits and even support lookup table (LUT) evaluation, making them more suitable for non-linear functions. However, they are less efficient for addition and multiplication circuits, as demonstrated in (Lou et al., 2020, Tab. 1). To overcome this limitation, Pegasus (Lu et al., 2021) recently proposed a framework that can switch between an arithmetic CKKS ciphertext and corresponding binary FHEW ciphertexts. This allows for the evaluation of arithmetic functions using CKKS and LUTs using FHEW. However, Pegasus only supports a single secret key, making it unsuitable for secure computation applications with multiple data providers, such as federated learning (McMahan et al., 2017; Fereidooni et al., 2021; Schneider et al., 2023).

[a] https://orcid.org/0000-0001-8090-1316
[b] https://orcid.org/0000-0001-6438-534X

Schneider, T., Yalame, H. and Yonli, M.
Griffin: Towards Mixed Multi-Key Homomorphic Encryption.
DOI: 10.5220/0012090200003555
In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*, pages 147-158
ISBN: 978-989-758-666-8; ISSN: 2184-7711

Table 1: Overview of existing homomorphic encryption protocols and compilers.

Name	**Functions**		**Conversion**	**MKHE**
	Arithmetic	**Boolean**	**Arithmetic ⟷ Boolean**	
Cingulata (Carpov et al., 2015)	✗	✓	✗	✗
FHEW (Ducas and Micciancio, 2015)	✗	✓	✗	✗
TFHE (Chillotti et al., 2020)	✗	✓	✗	✗
Transpiler (Gorantala et al., 2021)	✗	✓	✗	✗
Zama (Chillotti et al., 2021)	✗	✓	✗	✗
HElib (Halevi and Shoup, 2014)	✓	✗	✗	✗
HEAAN (Cheon et al., 2017)	✓	✗	✗	✗
CHET (Dathathri et al., 2019)	✓	✗	✗	✗
nGraph-HE (Boemer et al., 2019)	✓	✗	✗	✗
EVA (Dathathri et al., 2020)	✓	✗	✗	✗
SEAL (Microsoft Research, 2020)	✓	✗	✗	✗
E^3 (Chielle et al., 2018)	✓	✓	✗	✗
Lattigo (Mouchet et al., 2020)	✓	✓	✗	✗
MKHE-CKKS (Chen et al., 2019b)	✓	✗	✗	✓
MKHE-TFHE (Chen et al., 2019a)	✗	✓	✗	✓
OpenFHE (Al Badawi et al., 2022)	✓	✓	✗	✓
Chimera (Boura et al., 2020)	✓	✓	✓	✗
Pegasus (Lu et al., 2021)	✓	✓	✓	✗
Griffin (This work)	✓	✓	✓	✓

In an orthogonal line of research, Multi-Key Homomorphic Encryption (MKHE) schemes have been developed to support operations on ciphertexts encrypted using different keys (López-Alt et al., 2012; Chen et al., 2019a; Chen et al., 2019b). These schemes also provide on-the-fly MPC, where the circuit to be evaluated can be fixed after the data providers upload their encrypted data and the evaluation is non-interactive. However, to the best of our knowledge, no previous works can switch back and forth between different MKHE schemes. We introduce Griffin, the first hybrid MKHE framework that supports SIMD-style arithmetic operations while simultaneously supporting non-linear functions over binary circuits.

1.1 Our Contributions

The contributions of this paper are as follows:

- We introduce Griffin, the first MKHE scheme that can evaluate both arithmetic functions and LUTs on ciphertexts (§4).
- Griffin provides the on-the-fly MPC property, where the circuit to be evaluated can be fixed after the data providers upload their encrypted data.
- We introduce a new conversion similar to single-key Pegasus (Lu et al., 2021) allowing for transformation between MKHE ciphertexts of FHEW and CKKS (Chen et al., 2019b). This allows to compute the maximum of two numbers in 3.2 seconds, outperforming the existing solutions of (Chen et al., 2019b) by $\approx 2\times$ (§5).
- We implement Griffin on top of OpenPEGASUS[1], which is based on Microsoft SEAL[2], and benchmark it against other HE solutions, showing its benefits for different applications.

Tab. 1 provides a comparison of Griffin with previous works in terms of the features supported.

2 BACKGROUND

We define our parameters and notation in §2.1 and provide an overview of the FHE schemes used in Griffin in §2.2.

2.1 Parameters and Notation

$[k]$ refers to the set containing all non-negative integers smaller than k, i.e., $\{i | i \in \mathbb{N}_0, i < k\}$. We use $\mathbb{Z}_q$ to refer to $\mathbb{Z}/q\mathbb{Z}$. Let $\mathcal{R} := \mathbb{Z}[X]/(X^N + 1)$. We

[1] https://github.com/Alibaba-Gemini-Lab/OpenPEGASUS

[2] https://github.com/Microsoft/SEAL

use $\mathcal{R}_{\mathbb{A}}$ to refer to $A[X]/(X^N+1)$. If q is an integer, we define $\mathcal{R}_q := Z_q[X]/(X^N+1)$. $\mathcal{B}$ refers to the subset of polynomials with binary coefficients of $\mathcal{R}$. Similarly, $\mathbb{B}$ refers to the set $\{0,1\}$. For $a \in \mathcal{R}$ we use $[a]_q$ to refer to $a \bmod q$ where coefficients are reduced in $(q/2, q/2]$. We use $\langle \cdot,\cdot \rangle$ to denote the canonical inner product. Vectors are typically bold such as $\boldsymbol{a} \in \mathbb{B}^n$. We refer to the j-th element of a vector with a_j. We use the notation $\mathrm{ct} \in \mathrm{SCHEME}_s^p(m)$ to express that ct is a ciphertext obtained by encrypting the message m with key s in scheme SCHEME with parameters p. We omit s and p if the parameters and the key are clear from the context. If χ is a distribution, then $x \leftarrow \chi$ expresses that x is sampled from χ. If A is a set, then $x \leftarrow A$ denotes that x is uniformly randomly sampled from A. An extended RLWE ciphertext $\mathrm{ct} \in \widetilde{\mathrm{RLWE}}(m; \boldsymbol{g})$ of a message m is defined by $(c_i)_{i\in[d]}$ with $c_i \in \mathrm{RLWE}(\boldsymbol{g}[i]\cdot m)$ (Lu et al., 2021). Ecd refers to the CKKS encoding function.

2.2 FHE Schemes

We differ between Arithmetic schemes, which either perform arithmetic over a plaintext space such as $\mathbb{Z}/p\mathbb{Z}$ (Fan and Vercauteren, 2012) or fixed point complex vectors (Cheon et al., 2017), and Binary schemes, which can evaluate binary gates such as NAND on binary ciphertexts (Ducas and Micciancio, 2015). Binary schemes are often based on the Learning With Errors (LWE) assumption, whereas Arithmetic schemes use the Ring Learning With Errors (RLWE) assumption. **LWE Assumption (Regev, 2005).** Let n be the LWE dimension, q the ciphertext modulus, and $t \geq 2$ the plaintext modulus. A key vector $\boldsymbol{s} \in \mathbb{Z}_q^n$ is sampled from a given key distribution. Let $m \in \mathbb{Z}_t$ be the message that ought to be encrypted. For each encryption, a vector $\boldsymbol{a}$ is uniformly randomly sampled from $\mathbb{Z}_q^n$, and a random error e is sampled from $\mathbb{Z}_q$ according to the error distribution. A ciphertext consists of the elements $(\boldsymbol{a}, b)$, where $b = \langle \boldsymbol{a}, \boldsymbol{s} \rangle + e + \Delta m$, and Δ is the scaling factor that depends on the scheme. Commonly $\Delta = \lfloor \frac{q}{t} \rceil$ is used. The LWE assumption states that this tuple is indistinguishable from a tuple where b is sampled from $\mathbb{Z}_q$ uniformly at random.

RLWE Assumption (Stehlé et al., 2009; Lyubashevsky et al., 2010). Let M be an integer based on the ciphertext modulus and the security parameter. Let $\mathcal{R} := \mathbb{Z}[X]/(\Phi_M(X))$, where Φ_M is the M-th cyclotomic polynomial. Again, let q be the ciphertext modulus and t the plaintext modulus. The key $s \in \mathcal{R}_q$ is generated according to the key distribution. We encrypt a message $m \in \mathcal{R}_t$ by sampling an element $a \in \mathcal{R}_q$ uniformly at random and calculating $b = as + m + e$.

CKKS (Cheon et al., 2017). The CKKS idea is that each multiplication consumes a level, and the parameters are generated for a fixed level. So it is only possible to evaluate functions up to a given maximum multiplicative depth. A base p is fixed, and a modulus q_0 is chosen for the plaintext. We define $q_\ell = p^\ell \cdot q_0$ for $0 < \ell \leq L$. A ciphertext of level ℓ is a vector in $\mathcal{R}_{q_\ell}^k$ for fixed k. If binary operations are passed ciphertexts of different levels, then a rescaling procedure is executed to equalize the levels by lowering the level of the higher ciphertext.

HWT(h) is the set of signed binary vectors in $\{0,\pm 1\}^N$ with Hamming weight h. DG(σ^2) is a discrete Gaussian distribution on $\mathbb{Z}^N$ with standard deviation σ. ZO(p) is the distribution defined over $\{0,\pm 1\}^N$, where each element is -1 with probability $p/2$, +1 with probability $p/2$, and 0 with probability $1-p$. We sample $s \leftarrow \mathrm{HWT}(h)$, $a \leftarrow \mathcal{R}_{q_L}$ and $e \leftarrow \mathrm{DG}(\sigma^2)$ for the key generation. The secret key sk is set as $\mathrm{sk} = (1, s)$ while the public key pk is set to an encryption of 0. We also generate an evaluation key, which is the encryption of Ps^2 for an integer P that depends on the security parameter and modulus. In order to encrypt a message $m \in \mathcal{R}_{q_0}$, we sample $v \leftarrow \mathrm{ZO}(0.5)$ and $(e_0, e_1) \leftarrow \mathrm{DG}(\sigma^2)$. Then output $\mathrm{ct} = v \cdot \mathrm{pk} + (m + e_0, e_1) \pmod{q_L}$. Decryption consists of outputting $b + a \cdot s \bmod q_\ell$ for a given ciphertext (b, a).

FHEW (Ducas and Micciancio, 2015). In FHEW, $a \leftarrow \mathbb{Z}_q^n$ and s are chosen uniformly at random or as short vectors. Instead of using a separate error term, a randomized rounding function $\chi : \mathcal{R} \to \mathbb{Z}$ is used. Encryption is defined as:

$$\mathrm{Enc}_{\boldsymbol{s}}^{t/q}(m) = (\boldsymbol{a}, \chi(\langle \boldsymbol{a}, \boldsymbol{s} \rangle + mq/t) \mod q) \in \mathbb{Z}_q^{n+1}.$$

Decryption is defined as:

$$\mathrm{Dec}_{\boldsymbol{s}}^{t/q}(\mathrm{ct}) = \lfloor t(b - \langle \boldsymbol{a}, \boldsymbol{s} \rangle)/q \rceil \mod t \in \mathbb{Z}_t = m'.$$

Another technique that FHEW can use is key switching. Here, the components of the old key are decomposed along a basis, and each decomposed part is encrypted under the new key: $\boldsymbol{k}_{i,j,v} = \mathrm{Enc}_s^{q,q}(vz_iB_{ks}^j)$ for $i \in [n]$, $j \in [\lceil log_{B_{ks}} q \rceil]$ and $v \in [B_{ks}]$. A key switch is then performed by first calculating $a_{i,j}$ such that $a_i = \sum_j a_{i,j} B_{ks}^j$, and then calculating: $\mathrm{KeySwitch}((\boldsymbol{a}, b), \mathfrak{K}) = (\boldsymbol{0}, b) - \sum_{i,j} \boldsymbol{k}_{i,j,a_{i,j}}$, where $\mathfrak{K}$ is the set of all $\boldsymbol{k}_{i,j,v}$.

3 RELATED WORK

In this section, we summarize existing MKHE schemes and Pegasus (Lu et al., 2021), a conversion between single-key FHEW (Ducas and Micciancio, 2015) and CKKS (Cheon et al., 2017).

3.1 MKHE

MKHE schemes allow to perform operations on encrypted data under different keys. Thus, each party can encrypt their data with their key and publish the ciphertext. An evaluator can perform the desired operations on the ciphertexts, and finally, the involved parties can decrypt the ciphertext (Chen et al., 2019b). Many MKHE schemes have in common that they require the Common reference string (CRS) assumption (Chen et al., 2019b; Chen et al., 2019a). Traditional MPC approaches have communication that is proportional to the product of the complexity of the function and the number of parties (Chen et al., 2019b). In contrast, the communication in MKHE approaches is independent of the evaluated function. Another advantage is that the parties in Multi-Party Computation (MPC) need to stay online during the entire protocol execution. Moreover, MKHE approaches are more suitable for outsourcing. Some MPC protocols have the data owners secret-share their data with independent servers who perform the MPC to obtain the result (Mohassel and Zhang, 2017). However, they assume that the servers do not collude. Bootstrapping and key switching are also possible in MKHE schemes without interaction (Chen et al., 2019b; Chen et al., 2019a).

Relinearization. (Chen et al., 2019b) introduces two different relinearization techniques for MKHE-CKKS and MKHE-BFV. The first technique (Chen et al., 2019b, Alg. 1) has higher noise levels but has a faster scheme compared to the second variant (Chen et al., 2019b, Alg. 2). The second variant has lower noise levels and requires less storage. The relinearization techniques introduced in (Chen et al., 2019a) for MKHE-TFHE are very similar. We focus on the second variant since that variant offers higher accuracy and has been used for benchmarks in the literature (Chen et al., 2019b; Chen et al., 2019a).

Let $\boldsymbol{a} \leftarrow \mathcal{R}_q^d$ be the CRS for modulus q and decomposition degree d. k refers to the number of parties. The key generation of the public key is slightly modified. Instead of outputting a single zero encryption, we output d zero encryptions as the public key. Only a single zero encryption is needed for normal encryption. However, the other encryptions are needed for the relinearization.

$$\mathrm{KeyGen}(\mathrm{p}, s) := \boldsymbol{b} = -s \cdot \boldsymbol{a} + \boldsymbol{e} \pmod{q} \in \mathcal{R}_q^d,$$

where $\boldsymbol{e} \leftarrow \varphi^d$ for some error distribution φ and $\boldsymbol{a} \leftarrow \mathcal{R}_q^d$. The key s is drawn from the key distribution χ as in the normal single key schemes (Cheon et al., 2017; Fan and Vercauteren, 2012). This, in turn, allows to define a new encryption method that will be used to generate the new relinearization key:

$$\mathrm{UniEnc}(\mu; s) := \boldsymbol{d} = [\boldsymbol{d_0} | \boldsymbol{d_1} | \boldsymbol{d_2}] \in \mathcal{R}_q^{d \times 3}$$

1. $r \leftarrow \chi$, $\boldsymbol{e_1}, \boldsymbol{e_2} \leftarrow \varphi^d$ and $\boldsymbol{d_1} \leftarrow \mathcal{R}_q^d$
2. $\boldsymbol{d_0} = -s \cdot \boldsymbol{d_1} + \boldsymbol{e_1} + r \cdot \boldsymbol{g} \pmod{q}$
3. $\boldsymbol{d_2} = r \cdot \boldsymbol{a} + \boldsymbol{e_2} + \mu \cdot \boldsymbol{g} \pmod{q}$

where s is the secret key, q is the modulus and $\boldsymbol{g}$ is the decomposition vector. The decomposition vector represents a basis analogously to decomposition in BFV (Fan and Vercauteren, 2012) and TFHE (Chillotti et al., 2020). The relinearization key is then defined as $\boldsymbol{D} \leftarrow \mathrm{UniEnc}(s; s)$.

Arithmetic Operations. Generally, MKHE ciphertexts store a tuple of the ids of the parties whose secret key is required to decrypt the current ciphertext (Chen et al., 2019b). If a binary operation combines two ciphertexts whose id tuples are distinct, then the ciphertexts are expanded prior to the operation. Ciphertexts will be of the form $\mathrm{ct} \in \mathcal{R}_q^{k+1}$. Every id is associated with an index i such that $1 \leq i \leq k$ and $\mathrm{ct}[i]$ is multiplied with the private key of the party with id_i during decryption. The expansion consists of padding a ciphertext with zero polynomials at the indices corresponding to ids that are only present in the other ciphertext. This results in a change of the id to index mapping. An id might be associated with a new index afterward since both ciphertexts need to have the same mapping. The addition does not require any special treatment aside from a possible expansion. Multiplication also shares the first phase with the single key Fully Homomorphic Encryption (FHE) variant. However, the particular MKHE relinearization method is applied afterward.

Permutations and Rotations. Permutations use Galois Automorphisms, just like the single key variant of the schemes (Chen et al., 2019b, §5.1). However, since applying a Galois element to a ciphertext also changes the key of that ciphertext, a key switch to the original key is performed. Let $\tau_j : a(X) \mapsto (aX^j)$ be a Galois element. Then we define:

$$\mathrm{MKHE.GKGen}(j; s) := \mathrm{gk} = [\boldsymbol{h}_0 | \boldsymbol{h}_1] \in \mathcal{R}_q^{d \times 2},$$

where $\boldsymbol{h}_1 \leftarrow \mathcal{R}_q^d$ and $\boldsymbol{h}_0 = -s \cdot \boldsymbol{h}_1 + \boldsymbol{e'} + \tau_j(s) \cdot \boldsymbol{g} \pmod{q}$. To apply a Galois element, we then perform:

$$\mathrm{MKHE.EvalGal}(\mathrm{ct}; \{\mathrm{gk}_i\}_{1 \leq i \leq k}) := (c'_0, \ldots, c'_k)\text{ , where}$$

$$c'_0 = \tau_j(c_0) + \sum_{i=1}^{k} \langle \boldsymbol{g}^{-1}(\tau_j(c_i)), \boldsymbol{h}_{i,0} \rangle \pmod{q},$$

$c_i' = \langle \boldsymbol{g}^{-1}(\tau_j(c_i)), \boldsymbol{h}_{i,1} \rangle \pmod{q}$ for $1 \leq i \leq k$.

Distributed Decryption. For decryption, each party multiplies the index of the ciphertext associated with their id with their private key and adds noise from a distribution ϕ which has a larger variance than φ, reducing it $\bmod\ q$ and publishes the result (Chen et al., 2019b). The result can then be obtained by adding all these shares and by removing the scaling in the case of BFV (Fan and Vercauteren, 2012).

3.2 Pegasus (Lu et al., 2021)

Different FHE schemes have different strengths and advantages (Boura et al., 2020; Lu et al., 2021): on the one hand, CKKS (Cheon et al., 2017) and BFV (Fan and Vercauteren, 2012) excel at tasks that involve addition or multiplication and can be performed in a SIMD-like manner. On the other hand,TFHE (Chillotti et al., 2020) and FHEW (Ducas and Micciancio, 2015) might perform better when parallelization is not possible, the functions involve many non-linear operations, or when very deep circuits need to be evaluated. This motivated researchers to create conversions between different FHE schemes such as Pegasus (Lu et al., 2021) and Chimera (Boura et al., 2020). Pegasus (Lu et al., 2021) outperforms Chimera (Boura et al., 2020) with faster conversions and smaller keys. Indeed, benchmarks in (Lu et al., 2021) demonstrated a significant speedup compared to Chimera (Boura et al., 2020). This is the main reason why we provide a MKHE variant of Pegasus instead of Chimera in Griffin. The full Pegasus protocol merely allows for evaluating a look-up table with CKKS-ciphertexts. However, this is achieved by converting from CKKS to FHEW, performing the lookup, and then converting to CKKS. The functionality of Pegasus can be summarised as follows:

1. Apply the slots to coefficients transformation.
2. Extract all the slots of the ciphertext to obtain LWE ciphertexts.
3. Evaluate the LUT $T(x)$ on the LWE ciphertexts.
4. Repack the LWE ciphertexts into a RLWE ciphertext and apply the proper encoding.

Removing the two last steps results in a conversion from CKKS to FHEW, taking a tuple of LWE ciphertexts, and only applying the last step results in a conversion from FHEW to CKKS. Repacking involves homomorphically performing the partial LWE decryption, represented as a linear transformation, and then applying a modular reduction. The following procedures play an important role for Pegasus and we later translate them to the MKHE setting in Griffin:

LWE Key Switching. The key switching procedure allows changing the key of a given ciphertext. It is possible to change the lattice dimensionality of the LWE instance. Note that LWE ciphertexts are of the form $(b, \boldsymbol{a})$. Where $b = m + e - \langle \boldsymbol{a}, \boldsymbol{s} \rangle$. To obtain a new encryption, we use encryption of the old key $\boldsymbol{s}$ under the new key $\boldsymbol{s'}$ and homomorphically multiply it with $\boldsymbol{a}$ (Lu et al., 2021). This ciphertext can then be added to b to obtain encryption of m under $\boldsymbol{s'}$.

LUT Evaluation. The lookup table (LUT) function evenly samples a real function at n points in a bounded interval given by $[-q/4\Delta, q/4\Delta]$. These sampling points are used as the coefficients of a polynomial. The idea is to rotate the polynomial so that the constant part of the polynomial corresponds to the image of the function being evaluated under the given input. This image is then recovered by performing a slot extraction (Lu et al., 2021). This is achieved by first performing a modulus switch from q to $2n$. Then we perform an arithmetic operation such that $AC_{k,\underline{n}}$ is an encryption of $\hat{f} \cdot X^{\tilde{b}+\sum_{i\in[k]} \underline{s_i}\tilde{a}_i} = \hat{f} \cdot X^{\Delta m}$. This results in the desired value being in the constant part of the polynomial.

Linear Transformation (LT) Evaluation. Tiling transforms a rectangular matrix into a square matrix by repeating the matrix itself (Lu et al., 2021). The LT evaluation algorithm combines tiling with baby step and giant step to perform matrix multiplication and then adds the vector t to the result (Lu et al., 2021; Halevi and Shoup, 2014).

Eval Sine. EvalSine is used to approximate the modular reduction by q_0 as part of the repacking process. There is no need for a special MKHE variant. The algorithm is taken from (Bossuat et al., 2021).

Slots to Coefficients (S2C). The algorithm S2C takes a ciphertext in the slot encoding and converts it to a ciphertext in the coefficient encoding (Lu et al., 2021). While doing so, it consumes 1 or 2 slots. No special adjustment is needed to move this routine to the MKHE setting.

Slot Extraction. Given a RLWE ciphertext in the coefficient encoding, we can use this procedure to extract a given coefficient into a LWE ciphertext (Lu et al., 2021).

4 Griffin

Griffin is capable of switching between different MKHE schemes and also performing MPC. In Griffin, the dimensions of the ciphertexts increase due to the MKHE conversion and we require a switching, evaluation, repacking and rotation key of every party. We assume the CRS model and require the public keys of every party. Additionally, correctness for the prior multiplication method of the Pegasus LUT procedure does not hold with MKHE ciphertexts. As a result, Griffin uses a new multiplication method based on (Chen et al., 2019a). Most of the Pegasus (Lu et al., 2021) subroutines do not require any significant modifications to work with MKHE ciphertexts.

Griffin, Full Protocol. Alg. 1 shows the full protocol of Griffin. It is an extension of the full Pegasus protocol (Lu et al., 2021, Fig. 6). The main modifications are highlighted in gray.

Griffin LWE Key Switching. Adjusting the Key Switching (KS) routine from the single-key (Lu et al., 2021, Fig. 9) to the MKHE setting is fairly straightforward as shown in Alg. 2. We can simply perform parts of the routine for every party and sum the result. The operator $\diamond$ in Alg. 2 refers to a product between an extended RLWE ciphertext (the key switching key) and an arbitrary ring element. The ring element is decomposed along the same basis as the RLWE ciphertext, the two decomposed elements are then multiplied component wise, and all components of the result are then summed together: $r \diamond \mathrm{ct} := \sum_{i=0}^{d-1} \boldsymbol{g}^{-1}(r)[i] \cdot \mathrm{ct}[i]$.

Griffin LUT Evaluation. Our algorithm for LUT evaluation is shown in Alg. 3. This algorithm is an extension of the LUT evaluation in Pegasus (Lu et al., 2021, Fig. 2), however it has a wider loop range in line 5 and line 6 to obtain the correct evaluation key in Griffin.

Griffin Slot Extraction. Moving the slot extraction from the single key setting to the MKHE setting is shown in Alg. 4. It consists of applying the routine that is performed on a to all the a_i.

Griffin LT Evaluation. The LT algorithm of Griffin is given in Alg. 5 which is derived from the LT algorithm in Pegasus (Lu et al., 2021, Fig. 5).

Correctness of Griffin. In this section, we show the correctness of Griffin. Pegasus can be understood as a transformation between the single-key CKKS (Cheon et al., 2017) and FHEW (Ducas and Micciancio, 2015) encryption methods. Griffin allows for evaluating a look-up table with MKHE-FHEW. This is achieved by converting from MKHE-CKKS to MKHE-FHEW, performing the look-up, and then converting back to MKHE-CKKS. The functionality of Griffin is analogous to that of Pegasus (Lu et al., 2021) and can be described as follows: (a) Apply the slots to coefficients transformation, (b) extract all slots of the ciphertext to obtain LWE ciphertexts, (c) perform LUT evaluations on the LWE ciphertexts, and (d) repackage the LWE ciphertexts into an RLWE ciphertext and apply encoding. Omitting steps (c) and (d) results in a conversion from MKHE-CKKS to MKHE-FHEW, whereas performing only step (d) results in a conversion from MKHE-FHEW to MKHE-CKKS. Repacking involves homomorphically decrypting the LWE ciphertext, applying a linear transformation, and performing a modular reduction.

Algorithm 1: Griffin, Full Protocol (modified from (Lu et al., 2021, Fig. 6)).

```
Input:  • Ciphertext moduli $q_0, q_1, ..., q_{L-1}$, special modulus $q'$. Define $Q_i = \Pi_{l \in [i]} q_l$ for $1 \leq i \leq L$.
        • CRS: $\boldsymbol{a}$.
        • Public keys: $\{b_i\}_{i \in [k]}$.
        • Digit decomposition gadget vector $\boldsymbol{g}_{\text{digit}} = [1, B_{\text{ks}}, ..., B_{\text{ks}}^{d_{\text{ks}}}]$ for some $B_{\text{ks}}, d_{\text{ks}} > 0$ and $B_{\text{ks}}^{d_{\text{ks}}} \geq q_0$.
        • RNS decomposition gadget vector $\boldsymbol{g}_{\text{rns}} = [q' \bmod q_0]$.
        • Rescaling factors $0 < \Delta, \Delta_r, \Delta'_r < q_0$.
        • Switching keys $\mathrm{SwK}_{\overline{s_i} \to \underline{s_i}}$,
        • LWE evaluation keys $\mathrm{EK}_i$:
              $$\mathrm{EK}_{i,j,0} \in \mathrm{UniEnc}_{s_i}^{n,q'q_0}(I^+(\underline{s}_i[j])),$$
              $$\mathrm{EK}_{i,j,1} \in \mathrm{UniEnc}_{s_i}^{n,q'q_0}(I^-(\underline{s}_i[j]))$$
          for $j \in [\underline{n}]$, where $I^+(x) = 1\{x \geq 0\}$ and $I^-(x) = 1\{x \leq 0\}$ for $i \in [k]$.
        • Repacking keys $\mathrm{RK}_i \in \mathrm{RLWE}_{\overline{s}_i}^{\overline{n},Q_L}(\mathrm{Ecd}(\underline{s}_i, \Delta_r))$ for $i \in [k]$.
        • Rotation keys of the CKKS scheme $\mathrm{RotK}_i$ for $i \in [k]$.
        • A level-$l$ RLWE ciphertext ($l > 1$) of an encoded vector $\boldsymbol{v} \in \mathcal{R}^{\ell}$, i.e., $\mathrm{ct}_{\text{in}} \in \mathrm{RLWE}_{\overline{s}_{i \in [k]}}^{\overline{n},Q_l}(\mathrm{Ecd}(\boldsymbol{v}, \Delta))$ for $i \in [k]$.
        • A look-up table function $T(x) : \mathcal{R} \mapsto \mathcal{R}$.
Output: A RLWE ciphertext $\mathrm{ct}_{\text{out}} \in \mathrm{RLWE}_{\{\overline{s_i}\}_{i \in [k]}}^{\overline{n},Q_{L'}}(\mathrm{Ecd}(T(\boldsymbol{v}), \Delta))$, i.e., the evaluation of $T(x)$ on the elements of $\boldsymbol{v}$.
Slots to coefficients and drop moduli: $\mathrm{ct}' = \mathrm{S2C}(\mathrm{ct}_{\text{in}})$
Extract Coefficients: $\mathrm{ct}_l = \mathrm{Extract}^l(\mathrm{ct}')$ for each $l \in [\ell]$
for $l \in [\ell]$ do
    Switch from $\overline{n}$ to the smaller dimension $\underline{n}$. $\dot{\mathrm{ct}}_l = \mathrm{GRIFFIN.KS}(\mathrm{ct}_l, \{\mathrm{SwK}_{\overline{s_i} \to s_i}\}_{i \in [k]})$
    Evaluate the look-up table. $\ddot{\mathrm{ct}}_l = \mathrm{GRIFFIN.LUT}(\dot{\mathrm{ct}}_l, \{\mathrm{EK}_i\}_{i \in [k]}, \{b_i\}_{i \in [k]}, T(x))$
    Switch from $n$ to $\underline{n}$.
        $$\dddot{\mathrm{ct}}_l = (b_l, \boldsymbol{a}_{l,0}, ..., \boldsymbol{a}_{l,k-1}) =$$
        $$\mathrm{GRIFFIN.KS}(\ddot{\mathrm{ct}}_l, \{\mathrm{SwK}_{s_i \to \underline{s_i}}\}_{i \in [k]})$$
end for
Define $\boldsymbol{b} = b_0, ..., b_{\ell-1}$ and $\boldsymbol{A}_i \in \mathbb{Z}_{q_0}^{\ell \times \underline{n}}$, s.t. the $l$-th row of $\boldsymbol{A}_i$ is $\boldsymbol{a}_{l,i}$.
for $i \in [k]$ do
    Evaluate the linear transform $\tilde{\mathrm{ct}}_i = \mathrm{GRIFFIN.LT}(\mathrm{RK}_i, \mathrm{RotK}_i, \Delta'_r, \boldsymbol{A}_i, \boldsymbol{0})$
end for
$\tilde{\mathrm{ct}} = \mathrm{GRIFFIN.LT}(\mathrm{RK}_k, \mathrm{RotK}_k, \Delta'_r, \boldsymbol{A}_k, \boldsymbol{b}) + \sum_{i=0}^{k-1} \tilde{\mathrm{ct}}_i$
Evaluate the modulo $q_0$ on $\tilde{\mathrm{ct}}$ via $\mathcal{F}_{\text{mod}}$ and output the result as $\mathrm{ct}_{\text{out}}$.
```

Algorithm 2: Griffin LWE KS (modified from (Lu et al., 2021, Fig. 9)).

```
Input: A LWE ciphertext $(b, \{\boldsymbol{a}_i\}_{i\in[k]}) \in \mathrm{LWE}^{\overline{\boldsymbol{n}},q}_{\{\overline{\boldsymbol{s}_i}\}_{i\in[k]}}(m)$.
    The LWE switching key of every party:
    $$\mathrm{SwK}_{i,j} \in \widetilde{\mathrm{RLWE}}^{\underline{n},q}_{\{\underline{s_i}\}_{i\in[k]}}\left(\sum_{l\in[\underline{n}]} \overline{\boldsymbol{s}_i}[j\underline{n}+l]X^l; \boldsymbol{g}\right)$$
    for $j \in [\overline{n}/\underline{n}]$.
Output: A LWE ciphertext $\mathrm{ct}_{\mathrm{out}} \in \mathrm{LWE}^{\underline{n},q}_{\{\underline{\boldsymbol{s}_i}\}_{i\in[k]}}(m)$.
Define a set of polynomials $\{\hat{a}_{i,j}\}$ where $\hat{a}_{i,j} = \boldsymbol{a}_i[j\underline{n}] - \sum_{l=1}^{\underline{n}-1} \boldsymbol{a}_i[j\underline{n}+l]X^{\underline{n}-l}$for $j \in [\overline{n}/\underline{n}]$.
Compute $\tilde{\mathrm{ct}}_i = \sum_{j\in[\overline{n}/\underline{n}]} \hat{a}_{i,j} \diamond \mathrm{SwK}_{i,j} \in R^2_{\underline{n},q}$.
Output $(b, \mathbf{0}) + \sum_{i=0}^{k-1} \mathrm{Extract}^0(\tilde{\mathrm{ct}}_i)$ as $\mathrm{ct}_{\mathrm{out}}$.
```

Algorithm 3: Griffin LUT evaluation (modified from (Lu et al., 2021, Fig. 2)).

```
Input: $(b, \boldsymbol{a}) \in \mathrm{LWE}^{\underline{n},q}_{\{\underline{s_i}\}_{i\in[k]}}(\lfloor\Delta m\rceil)$ of a message $m \in \mathcal{R}$
    with a scaling factor $\Delta$ such that $|\lfloor\Delta m\rceil| < q/4$. Look-up table function $T(x) : \mathcal{R} \mapsto \mathcal{R}$. The evaluation keys
    $\{\mathrm{EK}_i\}_{i\in[k]}$ are a set of Uni-encryptions of the respective keys $\underline{\boldsymbol{s}}_i \in \{0, \pm 1\}^{\underline{n}}$. Public keys of the involved parties $\{b_i\}_{i\in[k]}$.
Output: A LWE ciphertext $\mathrm{ct}_{\mathrm{out}} \in \mathrm{LWE}^{n,q}_{\{s_i\}_{i\in[k]}}(\cdot)$.
Let $\eta_k = kq/(2n\Delta) \in \mathcal{R}$ for $1 \le k \le n/2$. Define a polynomial $\hat{f} \in \mathcal{R}_{n,q}$ whose coefficients are:
    $$f_j = \begin{cases} \lfloor\Delta T(0)\rceil & \text{if } j = 0 \\ \lfloor\Delta T(-\eta_j)\rceil & \text{if } 1 \le j \le n/2 \\ \lfloor-\Delta T(\eta_{n-j})\rceil & \text{if } n/2 < j < n \end{cases}$$
$\tilde{b} = \lfloor \frac{2n}{q} b\rceil$ and $\tilde{\boldsymbol{a}} = \lfloor \frac{2n}{q}\boldsymbol{a}\rceil$.
Initialize $AC_0 = (\hat{f} \cdot X^{\tilde{b} \bmod n}, 0) \in R^2_{n,q}$.
for $j \in [k\underline{n}]$ do
    $t_j = ((X^{\tilde{\boldsymbol{a}}[j] \bmod n} - 1) \cdot AC_j) \circ (\mathrm{EK}_{\lfloor j/\underline{n}\rfloor, j \bmod \underline{n}, 0}, \{b_i\}_{i\in[k]}) + \mathrm{AC}_j$
    $\mathrm{AC}_{j+1} = ((X^{-\tilde{\boldsymbol{a}}_i[j] \bmod n} - 1) \cdot t_j) \circ (\mathrm{EK}_{\lfloor j/\underline{n}\rfloor, j \bmod \underline{n}, 1}, \{b_i\}_{i\in[k]}) + t_j$
end for
$\mathrm{Extract}^0(\mathrm{AC}_{k\underline{n}})$ as $\mathrm{ct}_{\mathrm{out}}$.
```

To show the correctness of Griffin, we show the correctness of each of its subprocedures. Additionally, we present a lemma that applies to linear transformations generally, rather than the algorithm used in Pegasus only (Lu et al., 2021).

Lemma 1. Let c be a MKHE-CKKS ciphertext encrypting the polynomial m with noise level e and f be a linear transformation. There exists an efficient algorithm, using only public information, that calculates c', a MKHE-CKKS ciphertext encrypting the polynomial $f(m)$ with noise level $f(e) + e_{lt}$, where e_{lt} is the noise added by the evaluation of the linear transform.

Algorithm 4: Griffin Slot Extraction.

```
Input: A MKHE-RLWE ciphertext that uses the coefficient encoding: $\mathrm{ct} \in \mathrm{RLWE}^{n,Q_i}_s(m)$. The coefficient to extract $j$.
Output: A MKHE-LWE ciphertext $\mathrm{ct}' \in \mathrm{LWE}^{n,Q_i}_s(m_j)$.
Let $\mathrm{ct} = (\sum_{l=0}^{n-1} b_l X^l, \sum_{l=0}^{n-1} a_{0,l}X^l, \ldots, \sum_{l=0}^{n-1} a_{k-1,l}X^l)$.
return
    $$ct' = (b_j, a_{0,j}, a_{0,j-1}, \ldots, a_{0,0}, -a_{0,n-1}, \ldots, -a_{0,j+1},$$
    $$a_{1,j}, a_{1,j-1}, \ldots, a_{1,0}, -a_{1,n-1}, \ldots, -a_{1,j+1}, \ldots$$
    $$a_{k-1,j}, a_{k-1,j-1}, \ldots, a_{k-1,0}, -a_{k-1,n-1}, \ldots, -a_{k-1,j+1}).$$
```

Algorithm 5: Griffin LT evaluation (modified from (Lu et al., 2021, Fig. 5)).

```
Input: $\mathrm{ct}_{\mathrm{in}} \in \mathrm{RLWE}^{\overline{n},q}_{\{\overline{\boldsymbol{s}_i}\}_{i\in[k]}}(\mathrm{Ecd}(\boldsymbol{z}, \Delta_r))$. Rotation keys
    $\mathrm{RotK}_i$. A scaling factor $\Delta'_r > 0$. A plain matrix $\boldsymbol{M} \in \mathcal{R}^{\ell\times\underline{n}}$ such that $\ell, \underline{n} < \overline{n}$. A vector $\boldsymbol{t} \in \mathcal{R}^{\ell}$.
Output: A RLWE ciphertext $\mathrm{ct}_{\mathrm{out}} \in \mathrm{RLWE}^{\overline{n},q/\Delta_r}_{\{\overline{\boldsymbol{s}_i}\}_{i\in[k]}}(m)$.
Tiling and Diagonals. Let $\ddot{n} = \max(\ell, \underline{n})$ and $\tilde{n} = \min(\ell, \underline{n})$. Define $\tilde{n}$ vectors $\{\tilde{m}_j\}_{j=0}^{\tilde{n}-1}$ by going through the rows and columns of $\boldsymbol{M}$
    $$\tilde{\boldsymbol{m}}_j[r] = \boldsymbol{M}[r \bmod \ell, r + j \bmod \underline{n}] \text{ for } r \in [\ddot{n}].$$
Baby-Step. Let $\tilde{g} = \lceil\sqrt{\tilde{n}}\rceil$. For $g \in [\tilde{g}]$, compute $c_g = \mathrm{RotL}^g(\mathrm{ct}_{\mathrm{in}})$.
Giant-Step. Let $\tilde{b} = \lceil\tilde{n}/\tilde{g}\rceil$. Compute
    $$\tilde{\mathrm{ct}} = \sum_{b\in[\tilde{b}]} \mathrm{RotL}^{b\tilde{g}}\left(\sum_{g\in[\tilde{g}]} \mathrm{Ecd}(\tilde{\boldsymbol{m}}_{b\tilde{g}+g} \gg b\tilde{g}, \Delta'_r) \cdot c_g\right).$$
if $\ell \ge \underline{n}$ then
    Output $\mathrm{Rescale}(\tilde{\mathrm{ct}}, \Delta_r) + \mathrm{Ecd}(\boldsymbol{t}, \Delta'_r)$ as $\mathrm{ct}_{\mathrm{out}}$.
else
    Let $\gamma = \log(\underline{n}/\ell)$ and $\mathrm{ct}_0 = \tilde{\mathrm{ct}}$.
    Update iteratively for $1 \le j \le \gamma$
        $$\mathrm{ct}_j = \mathrm{RotL}^{\ell 2^j}(\mathrm{ct}_{j-1}) + \mathrm{ct}_{j-1}.$$
    Output $\mathrm{Rescale}(\mathrm{ct}_\gamma, \Delta_r) + \mathrm{Ecd}(\boldsymbol{t}, \Delta'_r)$ as $\mathrm{ct}_{\mathrm{out}}$.
end if
```

Proof. It has been established in previous research that rotations can be homomorphically evaluated on MKHE ciphertexts (Chen et al., 2019b). Additionally, it has been shown that linear transformations can be efficiently represented as a combination of rotations, constant products, and additions, as outlined in (Han et al.,). Given that all the basic operations required to evaluate a linear transformation can also be applied to an MKHE-ciphertext, it can be inferred that an algorithm for homomorphically evaluating linear transformations on MKHE-ciphertexts must exist. □

Corollary 1. The single key variant of the SlotsToCoefficients (S2C) transformation can be adapted to a MKHE variant by replacing all the basic operations with their MKHE equivalents. In other words, the S2C transformation can be transformed into a MKHE-variant by applying the corresponding MKHE counterparts of the primitive operations used in the single key variant.

From Lemma 1, we can deduce Corollary 1, which states that the SlotsToCoefficients (S2C) transformation corresponds to a linear transformation, as previously established in (Han et al.,). This is also the approach used for the implementation of MKHE variant of S2C. The single key variant of S2C was modified by replacing each primitive operation with its corresponding MKHE equivalent. As the effect of these operations on the encrypted message is identical, it follows that the resulting ciphertext utilizes a coefficient encoding rather than a slots encoding.

Lemma 2. Let c be a MKHE-CKKS ciphertext encrypting N coefficients, $v_0,...v_{N-1}$ under the keys $(s_1,...s_k)$. There exists an efficient algorithm that given an integer $p \in [N-1]$, produces a MKHE-FHEW ciphertext that encrypts v_p under the keys $(\boldsymbol{s}'_1,...,\boldsymbol{s}'_k)$, where $\boldsymbol{s}'_i$ is the coefficient vector corresponding to the polynomial s_i.

Proof. Alg. 4 is an example of such an algorithm. Let $c = (\{a_i\}_{i\in[k]}, b)$ be the input ciphertext and $c' = (\{\boldsymbol{a}'_i\}_{i\in[k]}, b')$. First, we use the p-th coefficient of the polynomial b to obtain b'. Just as in (Chillotti et al., 2020) we use antiperiodic indexes. Let N be the degree of the polynomial. $a'_{i,j}$ shall refer to the j-th coefficient of $\boldsymbol{a}'_i$. $a'_{i,j}$ is set to the $(p-j)$-th coefficient of a_i. Since we are using antiperiodic indexes, a negative index corresponds to the additive inverse of the respective coefficient. For example, -2 would correspond to the additive inverse of the $(N-2)$-th coefficient. This is a consequence of the assumption that we are using cyclotomic polynomials of the form X^N+1. Simple arithmetic shows that this method results in an $\boldsymbol{a}'_i = (a'_{i,0},...,a'_{i,n-1})$ such that $\langle \boldsymbol{a}'_i, \boldsymbol{s}'_i \rangle = (a_i s_i)_p$. That is, the inner product corresponds to the p-th coefficient of the product of the two polynomials. It follows that we obtain a correct MKHE-LWE encryption of v_p without any additional noise. □

Lemma 3. The MKHE variant of the LWE KS (Alg. 2) is a correct key switching procedure. That is, given a FHEW-MKHE ciphertext of dimension n, c encrypted by the keys s_i and a tuple of key switch keys $(\mathrm{SwK})_i$, the procedure outputs a FHEW-MKHE ciphertext c' of dimension n' encrypted by the keys s'_i for $i \in [k]$. We assume $\frac{n}{n'} > 1$.

Proof. First, we observe that the key switch key of party i is an extended RLWE encryption of the LWE secret key s_i. This RLWE instantiation uses a cyclotomic polynomial of degree n'. The elements of the LWE secret key vector are interpreted as coefficients of a polynomial. A single polynomial will not be able to contain all of the key elements, therefore there are $\lceil n/n' \rceil$-many. The LWE KS algorithm introduced in (Lu et al., 2021) homomorphically computes the inner product of a_i and s_i and adds the result to b of the input. This corresponds to a homomorphic partial decryption. A full decryption (i.e., bootstrapping) would be followed by a modular reduction. Our MKHE variant homomorphically computes this inner product for every party and then adds the sum of these ciphertexts to the b of the input. This again is a partial decryption, which follows from the definition of MKHE decryption. □

A crucial algorithm in Griffin is the LUT evaluation. In the single key case, this algorithm utilizes RGSW ciphertexts, which cannot be easily combined with MKHE ciphertexts. A possible solution is to replace the RGSW encryption of the evaluation key with the Uni-encryption method outlined in (Chen et al., 2019b). This allows for the use of one of the multiplication algorithms previously introduced, to evaluate the LUT table.

Lemma 4. Alg. 3 evaluates a LUT on a MKHE-LWE ciphertext.

Proof. First, we observe that the idea behind the single key LUT evaluation is to encode each possible image of the LUT as a coefficient in the polynomial $\hat{f}$. We then calculate $X^{\hat{b}+\hat{a}s} \approx X^{2nm/q}$. The polynomial coefficients of $\hat{f}$ are chosen such that the following slot extraction extracts the correct coefficient with the correct value. The reasoning behind the MKHE variant is similar. We calculate $X^{\hat{b}+\Sigma_{i=1}^{k}\hat{a}_i s_i} \approx X^{2nm/q}$. This multiplication of a MKHE ciphertext with a uni-encryption by a single party is identical to the usage of the algorithm in (Chen et al., 2019a). □

The correctness of the LT evaluation follows straightforwardly from Lemma 1. The final part is the modular reduction.

Lemma 5. MKHE-F-Mod correctly evaluates the modular reduction on a RLWE-MKHE ciphertext.

Proof. We obtain MKHE-F-Mod by replacing the primitive operations of F-Mod in (Bossuat et al., 2021) with their MKHE counterparts. Since the modifications of the encoded message are identical it follows that the reduction is applied properly. □

Theorem 4.1. Griffin is a correct MKHE instantiation of the Pegasus (Lu et al., 2021).

Proof. This follows from Corollary 1, Lemma 1, Lemma 2, Lemma 3, Lemma 4 and Lemma 5. □

Security of Griffin. The security of the Griffin scheme can be analyzed with relative ease due to the fact that the evaluator does not have access to any private information. The evaluator is only able to access public information, which simplifies the task of proving the security of the scheme to demonstrating that this public information does not reveal any sensitive information. To achieve this, we will demonstrate that all public information is indistinguishable from randomly generated data. Our scheme relies on the LWE, RLWE, CRS and circular security assumptions.

We will divide the analysis into two distinct phases:

- The setup phase, during which all keys and encrypted inputs are published.
- The decryption phase, during which ciphertexts are decrypted.

The setup phase in particular, involves public material from the MKHE-CKKS and the MKHE-FHEW schemes, as well as new material from the Griffin keys. We will only prove the security of the new material, as the security of the other material has already been established in (Chen et al., 2019b) and (Chen et al., 2019a). The decryption phase is equivalent to the decryption phase of the MKHE schemes and as such, does not require any special treatment. We only focus on the RKWE swtiching keys, needed for MKHE-LWE key switching, and repacking key, needed for the MKHE-repacking operation, as the rest are either present in MKHE-CKSS (Chen et al., 2019b) or are variants of (Chen et al., 2019a) present in MKHE-FHEW.

Lemma 6. The RLWE switching key is indistinguishable from a randomly generated ciphertext of the same size under the RLWE assumption and the circularity assumption.

Proof. We observe that the MKHE-RLWE switching key of one party is identical to the switching key in the single key instantiation. This in turn is a tuple of RLWE encryptions of the LWE key. This is secure if we combine the RLWE and circularity assumptions. □

Lemma 7. The repacking key is indistinguishable from a randomly generated ciphertext of the same size under the RLWE assumption and the circularity assumption.

Proof. The argument follows the same line of reasoning as in the previous Lemma 6. The repacking key is the collection of the party's single-key repacking key, which is a RLWE encryption of the LWE key. Combining the RLWE and the circularity assumption again results in the proof. □

Theorem 4.2. Griffin is secure against semi-honest adversaries under the LWE assumption.

Proof. This follows from combining the results of (Chen et al., 2019b), (Chen et al., 2019a) with Lemma 6 and Lemma 7. □

5 EVALUATION

In this section, we present several applications of Griffin. We primarily compare the runtime of the applications with single-key Pegasus (Lu et al., 2021) to show the overhead of our modifications.

Optimizations. Our implementation uses various optimisations, including the Residue Number System (RNS) representation (Chen et al., 2019b), the special modulus technique (Gentry et al., 2012), and seeded ciphertexts (Joye, 2022).

Selecting Parameters. For the benchmarks we use the same parameters as Pegasus (Lu et al., 2021). We use binary keys rather than trinary keys because the implementation only supports binary keys. Tab. 2 shows the parameters that are used in Pegasus (Lu et al., 2021). We use these parametes for the microbenchmarking in Griffin.

Table 2: Parameters used in Pegasus (Lu et al., 2021). n is the lattice dimensions, q' the special modulus, q_* a modulus prime used for the Residue Number System (RNS), σ_* the standard deviation of the noise distribution, B_{ks} and d_{ks} are parameters for the digit decomposition, Q is the modulus.

Encryption	Parameters
$\widetilde{\mathrm{RLWE}}_{\underline{s}}^{\underline{n},q_0}(\cdot)$, level 0	$\underline{n} = 2^{10}, q_0 \approx 2^{45}, \sigma_{\mathrm{ks}} = 2^{10}, B_{\mathrm{ks}} = 2^7, d_{\mathrm{ks}} = 7$
$\mathrm{RGSW}_{\boldsymbol{s}}^{n,q' q_0}(\cdot)$, level 1	$n = 2^{12}, q' \approx 2^{60}, \sigma_{\mathrm{lut}} = 2^{10}$
$\mathrm{RLWE}_{\overline{s}}^{\overline{n},*}(\cdot)$, level 2	$\overline{n} = 2^{16}, q_i \approx 2^{45}, \sigma_{\mathrm{ckks}} = 3.19, \log \overline{Q} = 795$

Communication Cost. Griffin requires communication only during the setup and the decryption phase. *Setup Phase:* The per party communication for the setup phase is independent of the number of parties. It also merely depends on the number of levels of the CKKS scheme and not on the depth of the circuit. The repacking procedure resets the number of levels and can thus be used to evaluate circuits whose multiplicative depth exceeds the number of levels. Generally, lower levels offer a better runtime for primitive operations, but this might come at the cost of more frequent repacking. Repacking itself also introduces a minimum number of levels, since it consumes levels. Our Griffin implementation requires 9 levels for repacking.

Decryption Phase: For decryption, each party receives part of the ciphertext from the evaluator. This part has a size of $\overline{n}\log_2(q_0)$ bits per party. Note that we are assuming that the ciphertext is at its lowest level. Even if the computation did not consume all levels, the evaluator can reduce the levels in order to reduce communication. This part is also unaffected by the seeded optimization. Afterwards, the party performs its computation and sends a polynomial with the same size back. The total communication per party for the decryption phase is thus $2\overline{n}\log_2(q_0)$ bits per ciphertext.

Since the communication costs only depend on the number of ciphertexts and the parameters, we can calculate them without a specific application in mind. Tab. 3 shows per party communication costs for Griffin and Pegasus considering Pegasus parameters (cf. Tab. 2). As Tab. 3 shows, communication per party for an application that requires each party to contribute a ciphertext is 3.86 GB and 2.08 GB with the seeded ciphertexts optimization (Joye, 2022). If the evaluator has downloaded the public material of the involved parties in the past, then this communication can be reduced to 12.42 MB or 6.21 MB with the seeded ciphertexts optimization. A single ciphertext can contain 2^{15} slots and thus the amortized ciphertext size (i.e., ciphertext size divided by number of slots) with the seeded ciphertexts optimization is 199 bytes. This corresponds to 35 ciphertext bits per plaintext bit with this optimization.

Table 3: Communication Costs for Pegasus Parameters in Griffin. SCO denotes seeded ciphertexts optimization (Joye, 2022). PK$_i$ is the public key of level i, SwK is a switching key, EK is the LUT evaluation key, RepackK is the repacking key, RotK is the rotation key, RelinK is the relinearization key of level 2, Input CT is the size of a fresh ciphertext, and output CT is the communication for decrypting one ciphertext. The column of seeded optimization is marked with ✗ if the optimisation cannot be applied.

Key	Size	Size with SCO
PK$_2$	105 MB	✗
PK$_1$	105 KB	✗
SwK$_{\bar{\boldsymbol{s}}\to\underline{\boldsymbol{s}}}$	4.92 MB	2.46 MB
SwK$_{\boldsymbol{s}\to\underline{\boldsymbol{s}}}$	315 KB	157.5 KB
EK	315 MB	210 MB
RepackK	12.42 MB	6.21 MB
RotK	3.11 GB	1.55 GB
RelinK	298.13 MB	198.75 MB
Input CT	12.42 MB	6.21 MB
Output CT	11.5 KB	✗
Total	3.86 GB	2.08 GB

Microbenchmarks. Tab. 4 shows the comparison of different operations of our MKHE framework Griffin with single-key Pegasus framework (Lu et al., 2021). Note that Pegasus only supports a single party.

Table 4: Runtime comparison of Griffin with Pegasus (Lu et al., 2021). LUT refers to a LUT evaluation and KS to a keyswitch. S2C is the slots to coefficients transformation, LT the linear transformation routine, and Mod the modular reduction.

	Pegasus (Lu et al., 2021)	Griffin (Ours)		
Operation	1 Party	1 Party	2 Parties	4 Parties
1 slot				
KS($\bar{s} \to \underline{s}$)	20ms	18ms	37ms	76ms
KS($s \to \underline{s}$)	1.5ms	1.3ms	2.5ms	4.9ms
LUT	0.93s	1.2s	2.96s	8.51s
256 slots				
S2C	0.78s	0.41s	0.76s	1.48s
LT	16.76s	15.4s	30.94s	62.55s
Mod	7.06s	13.6s	35.41s	108.26s
1024 slots				
S2C	1.28s	0.65s	1.21s	2.39s
LT	44.50s	42.9s	86.14s	174.00s
Mod	7.06s	13.35s	35.06s	107.72s
4096 slots				
S2C	2.02s	0.97s	1.88s	3.67s
LT	44.65s	43.32s	87.09s	175.84s
Mod	7.06s	13.43s	35.03s	107.57s

Sorting with Griffin. Similar to Pegasus (Lu et al., 2021), we use bitonic sorting (Batcher, 1968) which is data independent and can be parallelized easily. In order to implement this sorting, we approximate the min and max functions with LUTs.

$$\min(a,b) := 0.5(a+b) - 0.5|a-b|,$$

$$\max(a,b) := 0.5(a+b) + 0.5|a-b|.$$

Tab. 5 shows the performance results of Griffin in comparison with Pegasus (Lu et al., 2021).

Maximum of Two Numbers with Griffin. A pure MKHE-CKKS circuit that obtains the maximum of two numbers has 27 levels to reach an accuracy comparable to the MKHE-FHEW implementation (Cheon et al., 2019). The MKHE-CKKS approach to calculating the maximum is similar to the approach outlined above, except that the LUT for calculating $0.5|x|$ is replaced by sqrt(x^2). Most of the layers are consumed by approximating the sqrt function. Griffin replaces the LUT for $0.5|x|$ with a plaintext multiplication in CKKS. The algorithm halves the CKKS ciphertext, converts it to two FHEW ciphertext and then proceeds to use a single LUT for $|x|$. The parameters for the benchmark are slightly adjusted from Tab. 2.

Tab. 6 shows that Griffin has the lowest runtime compared to pure MKHE-CKKS (Chen et al., 2019b) and MKHE-FHEW. All benchmarks involved two parties. MKHE-CKKS uses (Cheon et al., 2019) with

Table 5: Runtimes for sorting (Batcher, 1968) with two parties in Griffin.

Elements	Threads	Pegasus (Lu et al., 2021) 1 party	**This work** 2 parties
32	20	34s	187s
	16	38s	186s
	8	73s	293s
	4	149s	476s
	1	493s	1441s
64	20	84s	458s
	16	104s	457s
	8	200s	748s
	4	409s	1277s
	1	1381s	4062s

a total depth of 15. MKHE-FHEW uses Pegasus (Lu et al., 2021) and Griffin combines MKHE-CKKS operations with MKHE-FHEW operations.

Table 6: Runtimes for calculating the maximum of two numbers among two parties.

Scheme	Time	Avg. Err. [%]	Max. Err. [%]
MKHE-CKKS (Chen et al., 2019b)	17.54s	2.00	4.28
MKHE-FHEW	5.91s	0.11	0.32
Griffin (Ours)	3.21s	0.17	0.50

6 CONCLUSION AND FUTURE WORK

We presented Griffin, a new MKHE scheme that can convert between different MKHE schemes depending on the function that needs to be evaluated. Griffin satisfies the on-the-fly property, i.e., it requires no interaction for converting from one MKHE scheme to another. To the best of our knowledge, this is the first time that a MKHE scheme with such properties has been constructed. As future work, we intend to expand the capabilities of Griffin by incorporating other MPC protocols like FLUTE (Brüggemann et al., 2023). Furthermore, we aim to explore the potential benefits of hardware acceleration (Münch et al., 2021) and synthesis tools (Patra et al., 2021b; Heldmann et al., 2021) in the context of Griffin.

ACKNOWLEDGEMENTS

This project received funding from the ERC under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230.

REFERENCES

Al Badawi, A., Bates, J., Bergamaschi, F., Cousins, D. B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., and Lee, Y. (2022). OpenFHE: Open-source fully homomorphic encryption library. In *WAHC*.

Batcher, K. E. (1968). Sorting Networks and Their Applications. In *American Federation of Information Processing Societies*.

Boemer, F., Cammarota, R., Demmler, D., Schneider, T., and Yalame, H. (2020). MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference. In *ARES*.

Boemer, F., Lao, Y., Cammarota, R., and Wierzynski, C. (2019). nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data. In *ACM International Conference on Computing Frontiers*.

Bossuat, J.-P., Mouchet, C., Troncoso-Pastoriza, J., and Hubaux, J.-P. (2021). Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-Sparse Keys. In *EUROCRYPT*.

Boura, C., Gama, N., Georgieva, M., and Jetchev, D. (2020). Chimera: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. *Journal of Mathematical Cryptology*.

Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2014). (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *ACM Transactions on Computation Theory*.

Brüggemann, A., Hundt, R., Schneider, T., Suresh, A., and Yalame, H. (2023). FLUTE: Fast and Secure Lookup Table Evaluations. In *IEEE S&P*.

Carpov, S., Dubrulle, P., and Sirdey, R. (2015). Armadillo: A Compilation Chain for Privacy Preserving Applications. In *International Workshop on Security in Cloud Computing*.

Chen, H., Chillotti, I., and Song, Y. (2019a). Multi-Key Homomorphic Encryption from TFHE. In *ASIACRYPT*.

Chen, H., Dai, W., Kim, M., and Song, Y. (2019b). Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference. In *CCS*.

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT*.

Cheon, J. H., Kim, D., Kim, D., Lee, H. H., and Lee, K. (2019). Numerical Method for Comparison on Homomorphically Encrypted Numbers. In *ASIACRYPT*.

Chielle, E., Mazonka, O., Gamil, H., Tsoutsos, N. G., and Maniatakos, M. (2018). E3: A Framework for Compiling C++ Programs with Encrypted Operands. *Cryptology ePrint Archive*.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2020). TFHE: Fast Fully Homomorphic Encryption over The Torus. *JoC*.

Chillotti, I., Joye, M., and Paillier, P. (2021). Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In *CSCML*.

Dathathri, R., Kostova, B., Saarikivi, O., Dai, W., Laine, K., and Musuvathi, M. (2020). EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *PLDI*.

Dathathri, R., Saarikivi, O., Chen, H., Laine, K., Lauter, K., Maleki, S., Musuvathi, M., and Mytkowicz, T. (2019). CHET: An optimizing compiler for fully-homomorphic neural-network inferencing. In *PLDI*.

Ducas, L. and Micciancio, D. (2015). FHEW: Bootstrapping Homomorphic Encryption in Less than A Second. In *EUROCRYPT*.

Fan, J. and Vercauteren, F. (2012). Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive. https://eprint.iacr.org/2012/144.

Fereidooni, H., Marchal, S., Miettinen, M., Mirhoseini, A., Möllering, H., Nguyen, T. D., Rieger, P., Sadeghi, A.-R., Schneider, T., Yalame, H., and Zeitouni, S. (2021). SAFELearn: Secure Aggregation for Private FEderated Learning. In *DLS@S&P*.

Gehlhar, T., Marx, F., Schneider, T., Suresh, A., Wehrle, T., and Yalame, H. (2023). SAFEFL: MPC-friendly Framework for Private and Robust Federated Learning. In *DLSP@S&P*.

Gentry, C., Halevi, S., and Smart, N. P. (2012). Homomorphic Evaluation of the AES Circuit. In *CRYPTO*.

Goldreich, O., Micali, S., and Wigderson, A. (1987). How to Play ANY Mental Game. In *STOC*.

Gorantala, S., Springer, R., Purser-Haskell, S., Lam, W., Wilson, R., Ali, A., Astor, E. P., Zukerman, I., Ruth, S., Dibak, C., Schoppmann, P., Kulankhina, S., and Forget, A. (2021). A General Purpose Transpiler for Fully Homomorphic Encryption. Cryptology ePrint Archive. https://eprint.iacr.org/2021/811.

Halevi, S. and Shoup, V. (2014). Algorithms in HElib. In *CRYPTO*.

Han, K., Hhan, M., and Cheon, J. H. Improved Homomorphic Discrete Fourier Transforms and FHE Bootstrapping. *IEEE Access*.

Hegde, A., Möllering, H., Schneider, T., and Yalame, H. SoK: Efficient Privacy-preserving Clustering. *PETS*, 2021.

Heldmann, T., Schneider, T., Tkachenko, O., Weinert, C., and Yalame, H. (2021). LLVM-based Circuit Compilation for Practical Secure Computation. In *ACNS*.

Joye, M. (2022). Guide to Fully Homomorphic Encryption over the [Discretized] Torus. In *TCHES*.

Juvekar, C., Vaikuntanathan, V., and Chandrakasan, A. (2018). GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security*.

Keller, H., Möllering, H., Schneider, T., and Yalame, H. (2021). Balancing Quality and Efficiency in Private Clustering with Affinity Propagation. In *SECRYPT*.

López-Alt, A., Tromer, E., and Vaikuntanathan, V. (2012). On-the-fly Multiparty Computation on The Cloud via Multi-Key Fully Homomorphic Encryption. In *STOC*.

Lou, Q., Feng, B., Fox, G. C., and Jiang, L. (2020). Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data. In *NeurIPS*.

Lu, W.-j., Huang, Z., Hong, C., Ma, Y., and Qu, H. (2021). Pegasus: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption. In *IEEE S&P*.

Lyubashevsky, V., Peikert, C., and Regev, O. (2010). On Ideal Lattices and Learning with Errors over Rings. In *EUROCRYPT*.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. (2017). Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS*.

Microsoft Research (2020). Microsoft SEAL. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., and Popa, R. A. (2020). Delphi: A Cryptographic Inference Service for Neural Networks. In *USENIX Security*.

Mohassel, P. and Zhang, Y. (2017). SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.

Mouchet, C., Bossuat, J.-P., Troncoso-Pastoriza, J., and Hubaux, J.-P. (2020). Lattigo: A Multiparty Homomorphic Encryption Library in Go. In *WAHC*.

Münch, J.-P., Schneider, T., and Yalame, H. (2021). VASA: Vector AES Instructions for Security Applications. In *ACSAC*.

Nguyen, T. D., Rieger, P., Chen, H., Yalame, H., Möllering, H., Fereidooni, H., Marchal, S., Miettinen, M., Mirhoseini, A., Zeitouni, S., Koushanfar, F., Sadeghi, A.-R., and Schneider, T. (2022). FLAME: Taming Backdoors in Federated Learning. In *USENIX Security*.

Patra, A., Schneider, T., Suresh, A., and Yalame, H. (2021a). ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*.

Patra, A., Schneider, T., Suresh, A., and Yalame, H. (2021b). SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation. In *IEEE HOST*.

Regev, O. (2005). On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *STOC*.

Riazi, M. S., Weinert, C., Tkachenko, O., Songhori, E. M., Schneider, T., and Koushanfar, F. (2018). Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *AsiaCCS*.

Schneider, T., Suresh, A., and Yalame, H. (2023). Comments on “Privacy-Enhanced Federated Learning Against Poisoning Adversaries. *IEEE TIFS*.

Stehlé, D., Steinfeld, R., Tanaka, K., and Xagawa, K. (2009). Efficient Public Key Encryption Based on Ideal Lattices. In *ASIACRYPT*.

Yao, A. C.-C. (1986). How to generate and exchange secrets. In *FOCS*.