Implementing and Optimizing Matrix Triples with Homomorphic Encryption

Johannes Mono ●
johannes.mono@rub.de
0000-0002-0839-058X

Tim Güneysu ●●
tim.gueneysu@rub.de
0000-0002-3293-4989

● Ruhr University Bochum, Bochum, Germany
● DFKI GmbH, Bremen, Germany

In today's interconnected world, data has become a valuable asset, leading to a growing interest in protecting it through techniques such as privacy-preserving computation. Two well-known approaches are multi-party computation and homomorphic encryption with use cases such as privacy-preserving machine learning evaluating or training neural networks. For multi-party computation, one of the fundamental arithmetic operations is the secure multiplication in the malicious security model and by extension the multiplication of matrices which is expensive to compute in the malicious model. Transferring the problem of secure matrix multiplication to the homomorphic domain enables savings in communication complexity, reducing the main bottleneck.

In this work, we implement and optimize the homomorphic generation of matrix triples. We provide an open-source implementation for the leveled BGV (Brakerski Gentry Vaikuntanathan) scheme supporting plaintext moduli of arbitrary size using state-of-the-art implementation techniques. We also provide a new, use-case specific approach to parameter generation for leveled BGV-like schemes heuristically optimizing for computation time and taking into account architecture-specific constraints. Finally, we provide an in-depth analysis of the homomorphic circuit enabling the re-use of key switching keys and eliminating constant multiplications, combining our results in an implementation to generate homomorphic matrix triples for arbitrary plaintext moduli.

Our implementation is publicly available and up to 2.1× faster compared to previous work while also providing new time-memory trade-offs for different computing environments. Furthermore, we implement and evaluate additional, use-case specific optimization opportunities such as matrix slicing for the matrix triple generation.

Introduction
Section 1

The sensitive nature of data in our interconnected world is leading to a growing interest in protecting it through techniques such as privacy-preserving computation. One approach to privacy-preserving computation is Multi-Party Computation (MPC). MPC involves multiple parties that want to compute a function together without revealing their inputs to each other. A well-known example is Yao's millionaire problem, where two millionaires want to determine who is wealthier without disclosing their wealth.

MPC protocols use so-called secret sharing where each party holds a share of a given variable during the function evaluation. Most commonly, additive secret sharing is used: during function evaluation, each party holds a share of a given variable and its actual value can only be reconstructed by adding up all values.

Computing simple operations such as addition or subtraction is rather straightforward as each party performs these operations on their shares locally. More sophisticated operations such as multiplications however require the parties to communicate, increasing the complexity of MPC protocols. Even worse, this complexity grows with more powerful, yet more realistic adversary models.

There are two security models to consider when designing protocols for MPC: the semi-honest model, where corrupted parties follow the protocol specification but try to discover information about the inputs of honest parties, and the malicious model, where corrupted parties are allowed to deviate from the protocol and collude to break security. In general, the semi-honest model provides weaker security guarantees in real-world settings as it does not account for the possibility of corrupted parties deviating from the protocol in order to break security.

One approach to multiplication of two values a and b in the semi-honest model is to locally multiply the shares $a_i \cdot b_i$, afterward communicating with the other parties and re-randomizing their sum $\sum_i a_i \cdot b_i$. However, this approach does not transfer to the malicious model. Here, we make use of so-called Beaver triples, a secret-shared tuple (a, b, c) satisfying $a \cdot b = c$. Using this triple, the parties can randomize their local multiplication and perform a secure multiplication even in the malicious security model.

Generating these triples using only MPC requires expensive public key operations and using other privacy-preserving techniques such as Homomorphic Encryption (HE) results in more efficient MPC protocols. HE enables the computation on encrypted data by operating on the ciphertext. Most commonly, it uses a client-server model where the clients

encrypt their data and send it to the server. The server can then perform arbitrary computations such as additions or multiplications on the encrypted data and return the encrypted result to the clients.

When used in conjunction with MPC, we operate in a slightly different model where the idea is as follows: each party encrypts random values a_i and b_i, broadcasting the encryption to the other parties. Then, using the homomorphic properties of the encryption, the random shares are added up and multiplied homomorphically such that each party holds an encrypted triple $(a, b, a \cdot b)$. Finally, the triple is decrypted in a distributed fashion resulting in a shared triple ready for use in the MPC protocol.

This idea can be extended to more complex forms of multiplication such as matrix multiplication, a pivotal operation in data analysis such as neural networks. Here, each party generates random matrices A_i and B_i and applying the same process leads to a shared matrix triple $(A, B, A \cdot B)$.

There are multiple challenges associated with the generation of matrix triples such as an efficient approach to homomorphic matrix multiplication or an HE implementation supporting the large parameters that are required for the security of the MPC protocol.

For the former, Jiang et al. [15] provide a state-of-the-art method to compute matrix multiplication homomorphically. Building upon their work, Chen et al. [6] propose and implement a protocol to compute matrix triples in the malicious model. However, their implementation is not available as their research focused on mathematically improving the Zero-Knowledge Proof (ZKP) used in the matrix triple protocol.

Thus, to the best of our knowledge, there currently is no HE library for either the Brakerski Gentry Vaikuntanathan (BGV) or Brakerski Fan Vercauteren (BFV) scheme, the two most relevant schemes to homomorphic matrix multiplication, that supports the large plaintext moduli that are required for the secure generation of matrix triples. Also, there is no additional research further analyzing and optimizing the homomorphic computations, the main bottleneck of the matrix triple generation.

Hence, our goal in this work is to improve the current state-of-the-art in generating matrix triples and to provide an efficient, publicly-available implementation for the research community. Overall, our work provides the following contributions:

- We analyze homomorphic matrix multiplication and, as a consequence, eliminate constant multiplications and enable the re-use of key switching keys. We also implement the matrix triple protocol, improving performance compared to previous work, and evaluate the implementation for different time-memory trade-offs.
- We provide a new approach to parameter generation for leveled BGV-like schemes heuristically optimizing for compu-

tation time. Our algorithm is applicable to HE uses cases in general and takes into account architecture-specific constraints such as the native integer size of the computing environment.

- We provide an open-source implementation for the leveled BGV scheme supporting plaintext moduli of arbitrary size using state-of-the-art implementation techniques.

Outline

In Section 2, we provide the theoretical background for our contributions focusing on BGV and the existing protocol for matrix triple generation. In Section 3, we introduce our new approach to parameter generation and our analysis results of the square matrix multiplication before we describe our implementation and present results on benchmarking and profiling in Section 4. We conclude our work in Section 6 by summarizing our results and discussing possible future work.

Preliminaries
Section 2

In the following, we provide the theoretical background of our work. We first describe the underlying ring arithmetic and continue with the BGV scheme. Then, we describe a homomorphic approach to square matrix multiplication and the matrix triple generation in the malicious adversary model.

2.1 DCRT Representation

Current state-of-the-art HE schemes are based on the Learning with Errors over Rings (RLWE) hardness assumption [16]. Software and hardware implementations of these schemes usually operate on elements in the ring $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$ using Chinese Remainder Theorem (CRT) representations (we assume n as power-of-two). For a given polynomial in coefficient representation, we apply two variants: the Residue Number System (RNS) to operate on native integer sizes (for example 64 bit) and the Number Theoretic Transform (NTT) for faster polynomial multiplication in time $\mathcal{O}(n \log n)$. The Double CRT (DCRT) representation combines both variants.

Residue Number System

Given a composite modulus $m = \prod m_i$, the RNS representation of a polynomial α is $[\alpha]_{m_i}$ where $[\cdot]_{m_i}$ denotes modular reduction of each coefficient to the integer set $\mathbb{Z}_{m_i} = [-m_i/2, m_i/2)$. The inverse of the RNS is the CRT over $\mathbb{Z}_m$ and reconstructs the multi-precision representation of α.

Number Theoretic Transform

Given the ring $\mathcal{R}_q$ with a primitive $2n$-th root of unity ξ, the NTT representation of a polynomial α is the factorization $\prod_{i \in \mathbb{Z}_{2n}^*}(x - \xi^i) \bmod q$ into linear terms [11] where $\mathbb{Z}_{2n}^*$ is the multiplicative group over $\mathbb{Z}_{2n}$.

The NTT and its inverse, the Inverse NTT (INTT), can be computed using a Fast Fourier Transform in time $\mathcal{O}(n \log n)$. Modular polynomial multiplication corresponds to coefficient-wise multiplication in the NTT representation.

2.2 Canonical Embedding Norm

Evaluating a polynomial $\alpha \in \mathcal{R}$ at all primitive $2n$-th roots of unity ζ^i is also known as canonical embedding. The canonical embedding norm is then defined as $\|\alpha\|_{\mathsf{can}} = \max_{i \in \mathbb{Z}_{2n}^*} \|\alpha(\zeta^i)\|_\infty$. It has the useful property that $\|\alpha \cdot \beta\|_{\mathsf{can}} = \|\alpha\|_{\mathsf{can}} \cdot \|\beta\|_{\mathsf{can}}$ for any $\alpha, \beta \in \mathcal{R}$. For a power-of-two degree, $\|\alpha\|_\infty \leq \|\alpha\|_{\mathsf{can}}$ [8].

2.3 BGV Scheme

The BGV scheme [4] operates on elements in $\mathcal{R}$ for a power-of-two n. For a plaintext modulus t and a ciphertext modulus q, a ciphertext $(c_0, c_1) \in \mathcal{R}_q \times \mathcal{R}_q$ encrypts a message $m \in \mathcal{R}_t$ if

$$c_0 + c_1 \cdot s \equiv m + te \pmod{q},$$

given a secret key s and a small error e.

The error grows with each homomorphic operation and decryption is correct as long as the error e does not wrap around modulo q. For a given n, a larger q allows for more homomorphic operations while a smaller q increases the security level λ. A plaintext polynomial can encode two vectors of size $n/2$ using packing such that operations are applied element-wise [20].

The leveled BGV scheme supports the following operations on the underlying plaintext vector: element-wise addition, constant multiplication, element-wise multiplication and rotation. All these operations, excluding rotation, influence the error. Other operations influencing the error are modulus switching and key switching.

Mono et al. [17] describe the leveled BGV scheme and analyze the associated bound $B = \|[c_0 + c_1 \cdot s]_q\|_{\mathsf{can}}$ for each operation in the DCRT representation. We simplify their formulas in Subsection 4.3 based on use-case and implementation-specific parameter choices.

Hoisting

Hoisting is an optimization technique applied to multiple rotations on the same ciphertext. For a fixed input ciphertext, the default process is usually as follows:

1. Rotate the ciphertext (computationally cheap).
2. Compute the so-called base extension of each rotation (computationally expensive).
3. Perform the dot-product with the key switching key and apply modulus switching.

For hoisting, we can switch the order of operations [14]:

1. Compute the base extension on the initial ciphertext (computationally expensive).
2. Rotate the extended ciphertext (computationally cheap).
3. Perform the dot-product with the key switching key and apply modulus switching.

Thus, instead of computing the base extension for each rotation, we only perform this computationally expensive operation once.

2.4 Homomorphic Matrix Multiplication

Jiang et al. [15] present an efficient version of homomorphic matrix multiplication. For a square matrices $A, B \in \mathbb{Z}^{d \times d}$, their idea is roughly the following:

1. Encode the matrix as vector and use homomorphic packing techniques to encrypt one matrix in one ciphertext.
2. Rewrite matrix multiplication based on matrix permutations and component-wise multiplication.
3. Apply the permutations using linear transformations on the packed vector using matrix-vector multiplication.
4. Use a diagonal-based matrix-vector multiplication to reduce the number of constant multiplications.

Using these techniques, the homomorphic matrix multiplication takes $\mathcal{O}(d)$ operations with one multiplication and two constant multiplications. In the following, we will take a closer look at each step.

Encoding

The encoding of a matrix A with row i and column j to a vector a is simply a concatenation of rows. Thus, we define the decoding isomorphism $\iota : \mathbb{Z}^{d^2} \to \mathbb{Z}^{d \times d}$ as

$$\iota : a \mapsto A = (a_{i \cdot d + j})_{0 \leq i,j < d}.$$

Jiang et al. [15] also propose an interleaved encoding packing m matrices into one vector that reduces the complexity of the matrix multiplication from $\mathcal{O}(d)$ to $\mathcal{O}(d/m)$. For the interleaved encoding, we define the

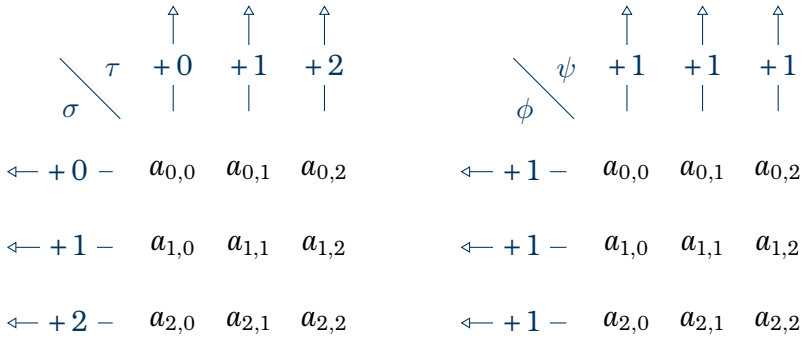

Figure 1: Matrix permutations σ, τ, ϕ and ψ for $d = 3$.

decoding isomorpishm as

$$\begin{aligned}\iota : \left(\mathbb{Z}^{d^2}\right)^m &\to \mathbb{Z}^{m\times d\times d}\\ a &\mapsto A^m = \left((a_{(i\cdot d+j)\cdot m+l})_{0\le i,j<d}\right)_{0\le l<m}.\end{aligned}$$

Permutation-based multiplication

First, we define four permutations σ, τ, ϕ and ψ (see also Figure 1):

$$\begin{aligned}\sigma(A)_{i,j} &= A_{i,j+i}\ \tau(A)_{i,j} = A_{i+j,j}\\ \phi(A)_{i,j} &= A_{i,j+1}\psi(A)_{i,j} = A_{i+1,j}\end{aligned}$$

Then, for matrices $A, B \in \mathcal{R}^{d\times d}$, we define matrix multiplication as

$$A \cdot B = \sum_{k=0}^{d-1} (\phi^k \circ \sigma(A)) \cdot (\psi^k \circ \tau(B)).$$

Correctness follows by computing each resulting component, we refer the interested reader to the original paper for the details [15].

Linear transformations

Each matrix permutation can also be expressed as a linear transformation on the encoded vector a. Hence, we define transformations U_σ, U_τ, U_ϕ and U_ψ corresponding to the permutations as

$$\begin{aligned}\left(U_\sigma\right)_{i\cdot d+j,\ell} &= \begin{cases}1, & \text{if } \ell = i\cdot d + [j+i]_d\\ 0, & \text{else}\end{cases}\\ \left(U_\tau\right)_{i\cdot d+j,\ell} &= \begin{cases}1, & \text{if } \ell = [i+j]_d\cdot d + j\\ 0, & \text{else}\end{cases}\\ \left(U_\phi^k\right)_{i\cdot d+j,\ell} &= \begin{cases}1, & \text{if } \ell = i\cdot d + [j+k]_d\\ 0, & \text{else}\end{cases}\\ \left(U_\psi^k\right)_{i\cdot d+j,\ell} &= \begin{cases}1, & \text{if } \ell = [i+k]_d\cdot d + j\\ 0, & \text{else}\end{cases}\end{aligned}$$

for $0 \leq i, j < d$ and $0 \leq \ell < d^2$. When counting the non-zero diagonals for each matrix, U_σ has $(2d-1)$, U_τ has d, U_ϕ has two and U_ψ has one. For more details, we again refer the interested reader to the original paper [15].

Diagonal matrix-vector multiplication

Halevi and Shoup [13] introduce the idea of diagonal matrix-vector multiplication in the context of HE for a linear transformation $L : \mathbb{Z}^d \to \mathbb{Z}^d$, defined as $L : a \mapsto U \cdot a$ for a vector $a = (a_0, \ldots, a_{d-1})$ and a matrix $U \in \mathbb{Z}^{d \times d}$.

We sum the element-wise multiplications of all rotations

$$\rho_\ell(a) = (a_\ell, \ldots, a_{d-1}, a_0, \ldots, a_{\ell-1}) \in \mathbb{Z}^d$$

with all ℓ-th diagonal vectors

$$u_\ell = (U_{0,\ell}, \ldots, U_{d-\ell-1,d-1}, U_{d-\ell,0}, \ldots, U_{d-1,\ell-1}) \in \mathbb{Z}^d$$

such that

$$U \cdot a = \sum_{\ell=0}^{d-1} (u_\ell \odot \rho_\ell(a)).$$

2.5 Block Matrix Multiplication

For two matrices A and B with dimensions $k_1 d \times k_2 d$ and $k_2 d \times k_3 d$, the matrices can be partitioned into blocks of size $d \times d$. Then, we can compute the matrix product $A \cdot B$ with dimension $k_1 d \times k_3 d$ using block matrix multiplication:

$$(AB)_{k_1,k_3} = \sum_{k=1}^{k_2} A_{k_1,k} B_{k,k_3}.$$

2.6 Matrix Triple Protocol

Chen et al. [6] provide an MPC protocol for matrix triples $(A, B, A \cdot B)$ in the malicious security model with a dishonest majority based on HE. Each party encrypts and broadcasts their matrix shares A^i and B^i. Additionally, each party invokes a ZKP to prove that the encryption is valid.

Then, all parties reconstruct the matrices homomorphically and compute c_{AB} using the square matrix multiplication. As dishonest majority settings require authenticated triples, the three ciphertexts c_A, c_B and c_{AB} are authenticated with the encrypted Message Authentication Code (MAC) key ct_α and each party runs the distributed decryption receiving a MAC output share $(A \cdot B)^i$.

Rotaru et al. [18] show how to create shared keys for BGV that keep the same properties as a non-shared key with respect to the error analysis. Additionally, Baum, Cozzo, and Smart [3] provide a ZKP for the BGV

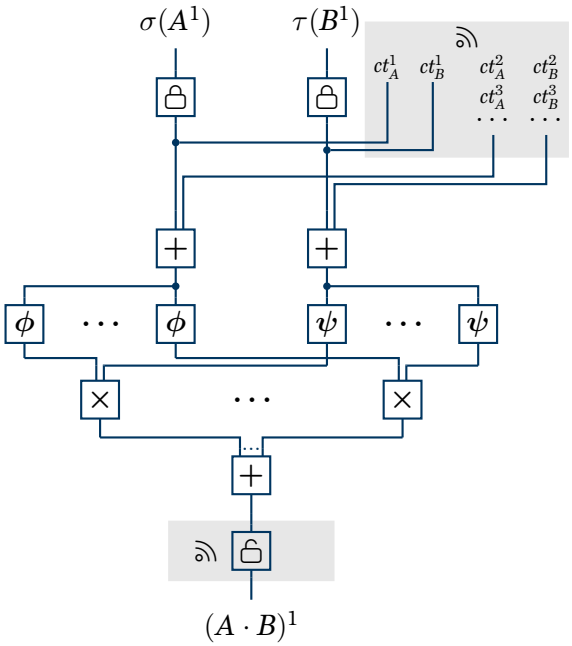

Figure 2: Matrix Triple Generation for Party 1.

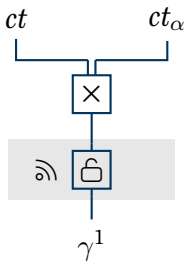

Figure 3: MAC Computation for Party 1.

scheme integrated in the SPDZ framework[1] and one of its successors, SCALE-MAMBA[2].

For parameters, Chen et al. [6] use $\lambda = 128$, $n = 2^{15}$, $\log q = 720$, $\log t = 128$, $\sigma \approx 3.2$ and $\lambda' = 80$. For encoding, Chen et al. [6] encode two vectors of size $n/2 = 2^{14}$ in each plaintext, thus multiplying two matrices with $d = 128$. In addition, they refer to the interleaved packing of Jiang et al. [15] as optimization if smaller matrices are desired.

In Figure 2, we display the high-level homomorphic computation of each party as circuit with MAC computation separated out to Figure 3. The parts of the circuit requiring communication such as the distributed decryption are highlighted with a gray background.

1 https://github.com/bristolcrypto/SPDZ-2

2 https://github.com/KULeuven-COSIC/SCALE-MAMBA

In the following, we improve the squared matrix multiplication of Jiang et al. [15] to enable the re-use of key-switching keys and we eliminate constant multiplications for both variants. We also integrate the former with an optimization known as *hoisting* and provide additional mathematical context and analysis to the matrix packing for squared matrix multiplication. Finally, we describe our new approach to parameter generation and its application to the use case of matrix triple generation.

3.1 Re-Use of Key Switching Keys

Common patterns in the rotations of a homomorphic circuit can help to save on key switching keys. For the sake of simplicity, we assume that the plaintext vector contains one encoded matrix with dimension d, however, our approach scales to more complex packing scenarios (see Subsection 3.2). Our proposed adjustments are based on the following observations:

1. σ and τ can be applied before encryption, thus performing homomorphic constant multiplications with U_σ and U_τ can be avoided [6].
2. As the encoding is a concatenation of rows, rotating the encrypted vector by $k \cdot d$ corresponds to the transformation ψ^k. Since the rotation itself already corresponds to the transformation, we do not need to apply the multiplication with the non-zero diagonal of U_ψ. Additionally, when iterating over k, we can continuously rotate by $i_B = d$ and reuse the key switching key.
3. For the transformation ϕ^k, the two non-zero diagonals are at index k and $k - d$. If we encrypt $\rho_{i_0}(\sigma(A))$ for $i_0 = -i_B$ and apply an initial rotation of i_B to a copy, we can work in parallel on two encrypted vectors with "initial positions" of 0 and $-i_B$. Then, when iterating over k, we can continuously rotate the encrypted vector by $i_A = 1$ and reuse the key switching key.

Overall, we need three key switching keys: two keys to realize the matrix transformations and one additional key for the homomorphic multiplication. This compares to the naïve approach of key switching that requires $2d$ keys for matrix A and d keys for matrix B. Additionally, we eliminate $2d$ constant multiplications. The changes to the original circuit are depicted in Figure 4.

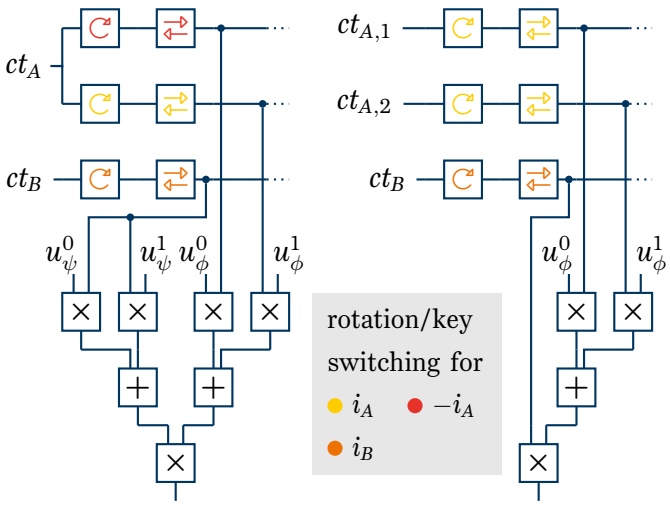

Figure 4: Comparison of the original (left) and improved (right) circuit for the inner loop of squared matrix multiplication.

$$a_{0,0}\, c_{0,0}\, a_{0,1}\, c_{0,1}\, a_{0,2}\, c_{0,2}\, a_{1,0}\, c_{1,0}\, a_{1,1}\, c_{1,1}\, a_{1,2}\, c_{1,2}\, a_{2,0}\, c_{2,0}\, a_{2,1}\, c_{2,1}\, a_{2,2}\, c_{2,2}$$
$$b_{0,0}\, d_{0,0}\, b_{0,1}\, d_{0,1}\, b_{0,2}\, d_{0,2}\, b_{1,0}\, d_{1,0}\, b_{1,1}\, d_{1,1}\, b_{1,2}\, d_{1,2}\, b_{2,0}\, d_{2,0}\, b_{2,1}\, d_{2,1}\, b_{2,2}\, d_{2,2}$$

Figure 5: Packing of matrices A, B, C and D for $d = 3$.

Integration with Hoisting

Contrary to hoisting, the approach of reusing key switching keys is an iterative algorithm and thus cannot be easily parallelized. However, both approaches can be combined depending on the computing environment.

As an example, consider a squared matrix dimension $d = 128$ and a server with 16 threads. We generate $2 \cdot 16$ key switching keys for

$$\{i_A, 2i_A, \ldots, 16i_A\} \text{ and } \{i_B, 2i_B, \ldots, 16i_B\}$$

and make full use of our threads to apply these rotations. Afterward, we base extend the new ciphertexts rotated by $16i_A$ and $16i_B$, respectively, only slightly increasing computation time and significantly reducing the amount of memory required.

In Subsection 4.5, we compare each method as well as the integrated approach with respect to the running time and memory usage.

3.2 Matrix Packing Enhancements

For matrix packing, we combine the natural encoding of BGV with two row vectors with the interleaved encoding applied to each row vector with $n = 2 \cdot m \cdot d^2$ (see also Figure 5). Instead of using U_ϕ as is, we diagonally compose two copies to transform both vectors in parallel. Since we compose diagonally, the resulting matrix still has two non-zero diagonals.

The previous observations on key switching keys can be adapted to our new encoding and we can also re-use the key switching keys with

packed ciphertexts. More specifically, we adjust our previous observations as follows:

1. We can apply σ and τ before the encoding on each matrix and thus before encryption.
2. The interleaved encoding is still a concatenation of rows and rotating by $k \cdot m \cdot d$ corresponds to the transformation ψ^k. Thus, we can continuously rotate by $i_B = m \cdot d$ and reuse the key switching key.
3. The definition of U_ϕ has to be adjusted for the interleaved encoding to

$$\left(U_\phi^k\right)_{(i \cdot d + j) \cdot m + \mu, \ell} = \begin{cases} 1, & \text{if } \ell = (i \cdot d + [j+k]_d) \cdot m + \mu \\ 0, & \text{else} \end{cases}$$

   for $0 \leq i, j < d$, $0 \leq \mu < m$ and $0 \leq \ell < m \cdot d^2$. U_ϕ still has two non-zero diagonals, one at index $k \cdot m$ and one at index $(k-d) \cdot m$. Thus, we can continuously rotate by $i_A = m$ and again reuse the key switching key.

Instead of generating U_ϕ, composing it with itself and extracting the non-zero diagonals, we generate the diagonals itself. Then, we can use the diagonals directly for the constant multiplication:

$$u^k_{\phi,(k-d)\cdot m} = \begin{cases} 1, & \text{if } [\ell]_{m \cdot d} < m \cdot (d-k) \\ 0, & \text{else} \end{cases}$$

$$u^k_{\phi,k\cdot m} = \begin{cases} 0, & \text{if } [\ell]_{m \cdot d} < m \cdot (d-k) \\ 1, & \text{else} \end{cases}$$

For completeness, we also define the other linear transformation matrices

$$\left(U_\sigma\right)_{(i \cdot d + j) \cdot m + \mu, \ell} = \begin{cases} 1, & \text{if } \ell = (i \cdot d + [j+i]_d) \cdot m + \mu \\ 0, & \text{else} \end{cases}$$

$$\left(U_\tau\right)_{(i \cdot d + j) \cdot m + \mu, \ell} = \begin{cases} 1, & \text{if } \ell = ([i+j]_d \cdot d + j) \cdot m + \mu \\ 0, & \text{else} \end{cases}$$

$$\left(U^k_\psi\right)_{(i \cdot d + j) \cdot m + \mu, \ell} = \begin{cases} 1, & \text{if } \ell = ([i+k]_d \cdot d + j) \cdot m + \mu \\ 0, & \text{else} \end{cases}$$

$$0 \leq i, j, k < d \quad 0 \leq \mu < m \quad 0 \leq \ell < m \cdot d^2$$

as well as the vector formulas for U_ψ

$$u^k_{\psi,k\cdot d\cdot m} = \begin{cases} 1, & \text{if } [\ell]_{n/2} < m \cdot d \cdot (d-k) \\ 0, & \text{else} \end{cases}$$

$$u^k_{\psi,n/2+k\cdot d\cdot m} = \begin{cases} 0, & \text{if } [\ell]_{n/2} < m \cdot d \cdot (d-k) \\ 1, & \text{else} \end{cases}$$

$$0 \leq k < d \quad 0 \leq \ell < n.$$

Matrix Slicing

Combining the interleaved encoding with block matrix multiplication can result in better performance depending on the desired output dimensions. If the desired dimension is for example 192×192, then using $d = 64$ results in much better performance than $d = 128$. Similar to bit slicing, this slices the matrix in smaller chunks on purpose to improve performance. For the previously mentioned example, we present benchmark numbers in Subsection 4.5.

3.3 Use-Case Specific Parameters

When generating parameters for the leveled BGV scheme, the most important parameters are the polynomial degree n, the ciphertext modulus q and the key switching modulus P. Currently, there are two main approaches to generate these: dynamic generation or static generation.

In dynamic parameter generation, the HE library tracks an associated error and automatically applies modulus switching once a certain threshold has been reached. The process usually follows the following steps [12]:

1. Implement the homomorphic circuit with the chosen HE library.
2. Choose a starting bit length for the ciphertext modulus q.
3. Execute the implementation and see if it fails: if yes, increase q, else decrease it.
4. Repeat until a comfortable error margin is reached.
5. Check the security with the lattice estimator [2] and then choose your degree accordingly.

The advantage of this approach is that the user does not have to worry about modulus switching making implementation more accessible. The disadvantage is that the exact error margin is not easy to compute and depends on the number of circuit evaluations, thus this method quickly gets computationally expensive.

In static parameter generation, the homomorphic circuit is considered in levels separated by modulus switching applied directly before or after each multiplication. Here, the process of generating parameters usually is as follows [17]:

1. Split up the homomorphic circuit into levels and determine the level with the longest path.

2. Calculate an error threshold and the error growth for the level with the biggest growth.
3. Based on this, calculate the level modulus size and set the ciphertext modulus accordingly.
4. Check the security with the lattice estimator, then choose your degree accordingly.

The advantages of this approach are compile-time known parameters and a known error margin. As for disadvantages, modulus switching is applied regardless of its necessity and the prime size does not necessarly match the native integer size of implementations resulting in more RNS moduli used than necessary.

Our new approach combines multiple of these advantages. We dynamically generate the parameters evaluating only the noise growth of the homomorphic circuit ahead of time, providing compile-time known parameters, and heuristically adjust the ciphertext modulus providing a known error margin. For modulus switching, the user inserts the candidates into the circuit design and the algorithm heuristically determines the number of primes to switch taking into account the native integer size.

More specifically, our algorithm evaluates the following steps:

1. We set our initial bit width b to the native integer size and fix an initial degree n, for example $b = 64$ and $n = 2^{10}$.
2. Then, we compute and heuristically optimize the number of RNS moduli q_i and P_i with our core algorithm.
3. Afterward, we reduce the bit width of our native integer until the number of moduli increases again, thus increasing security by reducing the overall ciphertext modulus without increasing computation costs.
4. Finally, we increase or decrease the degree as required for the desired security estimate and stop once our parameters do not change anymore.

The main idea of our core algorithm is to automatically determine and reduce the number of RNS moduli by looking ahead multiple levels and is based on the following observations:

- If increasing the number of moduli P_i for key switching decreases the number of ciphertext moduli q_i by at least the same amount, this reduces ciphertext size and the number of primes to compute on.
- If modulus switching reduces the number of ciphertext moduli after the look-ahead, then we need to compute on less moduli in the future and thus applying the switch is good.

Table 1: Comparison of HE Parameters

	Our work	Chen et al. [6]
λ	128	128
n	32768	32768
q	600 bit	712 bit
P	240 bit	
t	128 bit	128 bit
sw	$0, 1, 1$	

- If the error after these optimizations is still larger than the maximum decryption error, we need to increase the ciphertext modulus.

We provide pseudocode for the core part of our algorithm in Algorithm 1. Here, $genprime(n, b)$ is a function returning a unique prime p of bit size b with $p \equiv 1 \bmod 2n$ and $L_i(\cdot)$ evaluates all levels down to level i and returns the corresponding error estimate.

Note that this approach to parameter generation extends to BGV-like schemes in general as we do not rely on BGV-specific mechanisms.

Generating Parameters for Squared Matrix Multiplication

For the squared matrix multiplication, we have to insert modulus switching candidates and fix the look-ahead parameter x. For the former, we insert three candidates:

1. after the addition of all encrypted shares;
2. before each multiplication of transformed ciphertexts; and
3. after the addition and key switching of all these products.

For the latter, we fix $x = 1$ due to the small depth of our circuit. Figure 6 depicts the full homomorphic circuit including the modulus switching candidates and the MAC key computation.

The resulting parameters are displayed in Table 1 where we also compare them to the BFV parameters of Chen et al. [6] (they do not provide specific information on modulus switching or the key-switching modulus). The algorithm determines that the first modulus switching candidate does not reduce the error enough and that removing one prime from the ciphertext modulus results in good error development for the other two candidates.

Algorithm 1: Core Algorithm for Parameter Generation

Require: degree n, bit size b, look-ahead x, levels $L_i(\cdot)$
Ensure: ciphertext moduli q, switch moduli P, switch count sw

$q \leftarrow \{\mathsf{genprime}(n, b)\}$
$P \leftarrow \{\}$
$sw \leftarrow (0, 0, \ldots, 0)$
$q_{\mathsf{new}} \leftarrow q$
$i \leftarrow \ell$
while $i > 0$ do
  $j \leftarrow \ell$
  while $j \geq i$ and $j - x \geq 0$ do
    $p \leftarrow \{\mathsf{genprime}(n, b)\}$
    $B_{\mathsf{cur}} \leftarrow L_j(q, P, sw)$

    # heuristically adjust P
    $B_{\mathsf{new}} \leftarrow L_{j-x}(q, P \cup p, sw)$
    if $\lceil B_{\mathsf{new}}/b \rceil < \lceil B_{\mathsf{cur}}/b \rceil$ then
      $P \leftarrow P \cup p$
      $j \leftarrow \ell$
      continue
    end if

    # heuristically adjust sw
    $sw_{\mathsf{new}} \leftarrow sw$
    $sw_{\mathsf{new},j} \leftarrow sw_{\mathsf{new},j} + 1$
    $B_{\mathsf{new}} \leftarrow L_{j-x}(q, P, sw_{\mathsf{new}})$
    if $\lceil B_{\mathsf{new}}/b \rceil + sw_j < \lceil B_{\mathsf{cur}}/b \rceil$ then
      $sw \leftarrow sw_{\mathsf{new}}$
      $j \leftarrow \ell$
      continue
    end if

    # check whether we can decrypt
    if not $2B_{\mathsf{cur}} < \prod q$ then
      $q_{\mathsf{new}} \leftarrow q \cup \{\mathsf{genprime}(n, b)\}$
      break
    end if
    $j \leftarrow j - 1$
  end while
  if $q = q_{\mathsf{new}}$ then
    $i \leftarrow i - 1$
  end if
end while

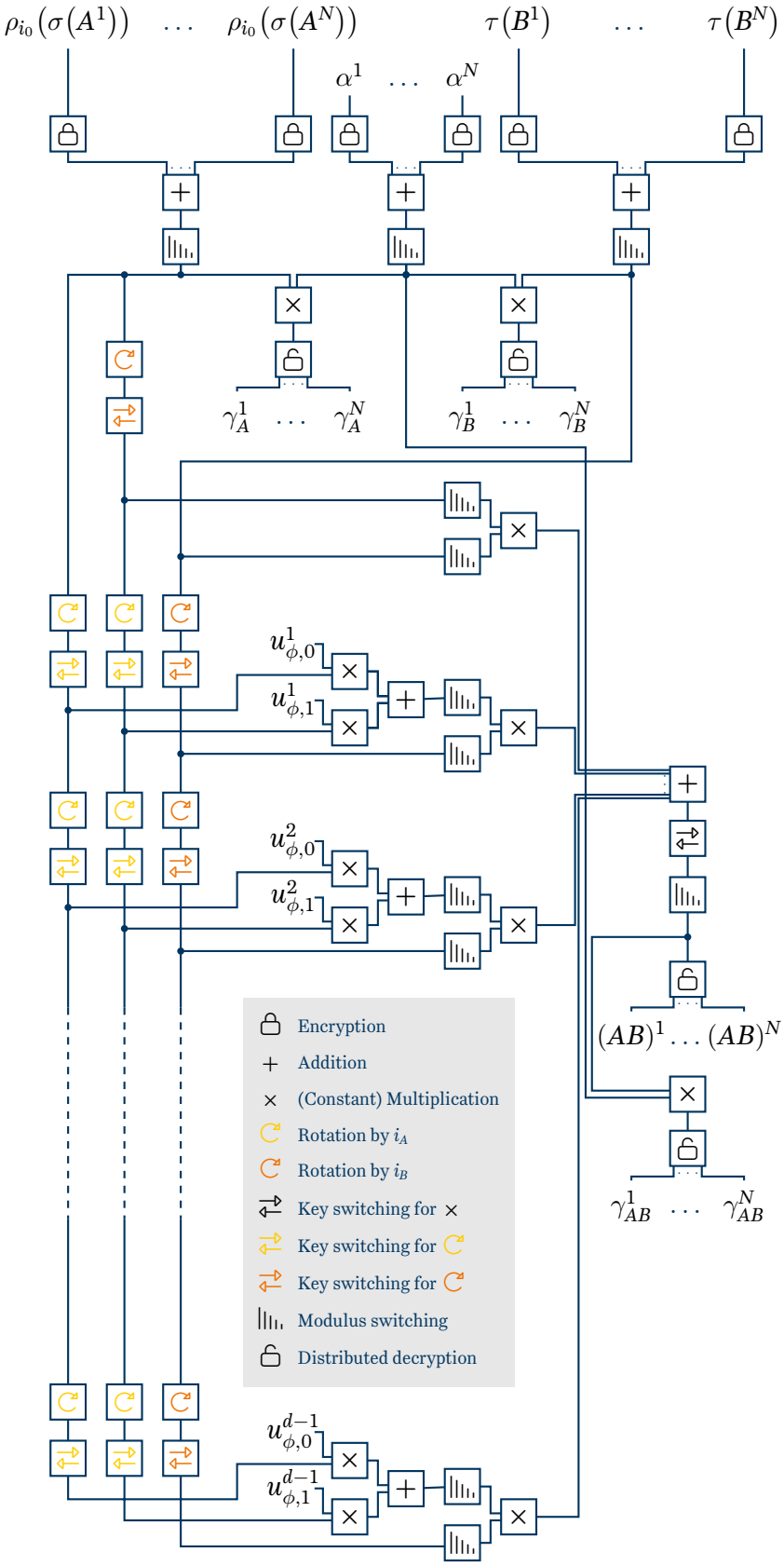

Figure 6: HE circuit of the Squared Matrix Multiplication

In the following, we describe our implementation of the leveled BGV scheme and the matrix triples including the parameter generation, publicly available at GitHub[3]. We start by describing the arithmetic layer for ciphertext operations, continuing with the message layer for arbitrary plaintext moduli and the parameter generation. Finally, we provide benchmarks for our implementation.

4.1 Ciphertext Arithmetic

For ciphertext arithmetic, we implement two variants for the RNS primes q_i: a 64 bit signed variant and a 64 bit unsigned variant.

Signed variant

The signed variant uses signed Barrett reduction for addition and subtraction and signed Montgomery multiplication [19]. For the NTT, we follow the pseudo-code from Seiler [19]; the same NTT is used as reference code in the Kyber implementation[4]. The largest supported bit size for each modulus is $b = 62$ as one bit is reserved for the sign bit and one bit for additions.

Unsigned variant

The unsigned variant is based on the HE Acceleration Library (HEXL) library by Intel[5] and supports a modulus size of up to $b = 63$ as no signed bit is needed. As this implementation is optimized and thus faster, we use it as default choice in our build system.

4.2 Plaintext Arithmetic

For plaintext arithmetic, we implement the arithmetic for an arbitrary large plaintext modulus t. Here, we face two main challenges: the representation of t in the RNS and correcting the modulus switching error.

Representation of the plaintext modulus

For the RNS, we precompute $t \bmod q_i$ and store it with the ciphertext modulus. For key generation and encryption, we need multiplication by t and for modulus and key switching, we need the inversion of t. Both operations also work in the RNS representation and hence we are able to compute on native integers. However, for the error generation, we need to pass a one-time seed with each RNS prime instead of generating the

3 https://github.com/Crypto-TII/mat3

4 https://github.com/pq-crystals/kyber

5 https://github.com/intel/hexl

polynomial once as the multiplication by t can wrap around each individual modulus.

Modulus switching error

During modulus switching, we multiply the ciphertext by $\frac{1}{q_i}$. For $q_i = 1 \bmod t$, we do not need to correct the plaintext m, the common approach in state-of-the-art libraries. For $q_i \neq 1 \bmod t$ such as with arbitrarily large t, we receive the factor $\frac{1}{q_i} m \bmod t$ and need to multiply the plaintext by q_i at some point. These factors can accumulate across operations and need to be corrected either before encryption, after encryption, or integrated with a constant multiplication.

In our library, we implement a semi-automated approach to handle this factor. When modulus switching, a user chooses how often this factor should be corrected for, a process which is then automatically performed during decryption.

4.3 Parameter Generation

Parameters are generated with a Python script. A bounds class contains the computed BGV bounds for each operation, thus only small changes are required to support other BGV-like schemes. Then, each level (that is each part separated by the modulus switching candidates) is implemented as function and passed to the implementation of the core algorithm. The script also has an option to set the flooding error during distributed decryption, automatically taking it into account when optimizing the parameters.

Computing the Bounds

In our implementation, we already make specific choices and therefore can simplify some of the error bounds. For the original bounds and the details on notation, we refer the interested reader to the original paper [17].

We set $D = 6$ for a failure probability of roughly 2^{-55}. Additionally, we sample the coefficients of the secret and the error from a centered binomial distribution in $[-1, 1]$ and $[-21, 21]$ with variances $V_s = 0.5$ and $V_e = 10.5$, respectively. Note that the Homomorphic Encryption Standard [1] recommends the standard deviation $\sigma = 3.2$ resulting in the variance $\sigma^2 = 10.24$.

For key switching, k and k' are the number of primes q_i and P_j, respectively, and the q_i are evenly distributed in ω products. We denote the maximum of these products as $\tilde{q}$.

Encryption

$$B_{\mathsf{enc}} \leq Dt\sqrt{n\left(\frac{1}{12} + 2nV_eV_s + V_e\right)}$$

$$\leq t\sqrt{3n(126n+127)}$$

Addition & Multiplication

$$B_{\mathsf{add}} \leq B + B'$$
$$B_{\mathsf{mul}} \leq B \cdot B'$$

Constant Multiplication

$$\begin{aligned} B_{\mathsf{const}} &\leq Dt\sqrt{\frac{n}{12}} \cdot B \\ &\leq t\sqrt{3n} \cdot B \end{aligned}$$

Modulus Switching

$$\begin{aligned} B_{\mathsf{ms}} &\leq \frac{B}{q_i} + Dt\sqrt{\frac{n}{12}(1+nV_s)} \\ &\leq \frac{B}{q_i} + \frac{t}{2}\sqrt{3n(2n+4)} \end{aligned}$$

Key Switching

$$\begin{aligned} B_{\mathsf{ks}} &\leq B + \frac{\tilde{q}}{P}Dtn\sqrt{\frac{k+k'+\omega-1}{12}V_e} + Dt\sqrt{\frac{nk'}{12}(1+nV_s)} \\ &\leq B + \frac{\tilde{q}}{P}tn\sqrt{63\frac{k+k'+\omega-1}{2}} + \frac{t}{2}\sqrt{3nk'(2n+4)} \end{aligned}$$

4.4 Matrix Triples

For the triples implementation, we use our BGV implementation and and add multi-threading capabilities using OpenMP[6]. A user can decide to turn the hoisted version on or off and decide whether to pre-rotate all blocks or save on memory and rotate them each time the block is multiplied.

Lazy key switching

As suggested by Chen et al. [6], we also implement lazy key switching as shown in Figure 6. This means, that we first add up the ciphertexts after the first multiplication, hence lazily deferring the key switching, and only need to apply it once on the final sum. Note that the following modulus switching can be combined with this key switching for a slightly better error growth.

Distributed decryption

In our implementation, we currently assume pre-generated shared keys [18] and only simulate the flooding error added during distributed decryption. This ensures our tests are working correctly. For a more detailed discussion on the integration with existing work, we refer the interested reader to Subsection 5.1.

6 https://www.openmp.org/

4.5 Benchmarking

We run our benchmarks on Ubuntu 18.04.1 with Linux kernel 5.4.0-87-generic. The Central Processing Unit (CPU) is an AMD EPYC 7742 CPU at 2.25 GHz featuring 64 cores and 2 TiB of available memory, a L1 cache of size 32 KiB, a L2 cache of size 512 KiB and a L3 cache of size 16 MiB. For comparison, Chen et al. [6] use an Intel Xeon Platinum 8168 CPU at 2.7 GHz featuring 16 cores and SEAL version 3.3.

Here, we investigate four different scenarios:

- comparison of our work with previous work;
- evaluating the time-memory trade-off for key re-use;
- evaluating the time-memory trade-off for pre-rotation; and
- measuring the performance increase with matrix slicing.

For benchmarking itself, we use the Google benchmark library[7].

Comparison with Previous Work

We summarize our comparison with Chen et al. [6] in Table 2. Here, we set $d = 128$ and run our implementation with 16 threads, but only using multi-threading for rotation and multiplication as in their original work. Note that all times are amortized, that is divided by $m = 2$. Additionally, we enable hoisting and pre-rotation to match their implementation as close as possible.

Since we do not have access to the original code including the benchmarks, we do not exactly know how the benchmarks were composed, thus making comparison a difficult task. The most notable difference is in the addition of the MAC key where our implementation is significantly faster. We suppose that two things are influencing the result here: first, the BFV multiplication requires a scaling option which might influence running times. Second, our decryption routine supports ciphertexts with three polynomials, thus we do not need to apply key switching down to polynomials, an operation that might still be applied in the implementation by Chen et al. [6].

Our implementation of rotations is slightly slower compared to Chen et al. [6], the core part of the square matrix multiplication on the other hand is much faster. Thus, combining the times of rotation, multiplication/block composition and MAC addition is up to $2.1\times$ faster. As we only simulate distributed decryption (see also Subsection 4.4), we cannot make conclusions about the different numbers here.

Time-Memory Trade-Offs

The results for the time-memory trade-offs for key re-use with $dim = 128$ and pre-rotation enabled are summarized in Table 3 and Table 4 with a

7 https://github.com/google/benchmark

Table 2: Comparison of our work with previous work.

d	Encrypt	RotA	RotB	Multiply	AddMAC	DDec
			Chen et al. [6]			
128	0.10 s	1.8 s	0.9 s	1.4 s	0.6 s	1 s
256	0.38 s	5.6 s	2.3 s	10.1 s	2.4 s	4 s
384	0.86 s	12.8 s	4.9 s	34.0 s	5.4 s	9 s
512	1.52 s	21.8 s	8.0 s	79.6 s	9.6 s	16 s
1024	6.08 s	79.6 s	32.9 s	648.0 s	38.4 s	64 s
			Our Work			
128	0.09 s	1.50 s	0.85 s	0.56 s	0.00 s	0.09 s
256	0.38 s	6.25 s	3.20 s	3.31 s	0.01 s	0.34 s
384	0.85 s	14.15 s	6.95 s	10.85 s	0.03 s	0.77 s
512	1.51 s	24.80 s	12.30 s	27.35 s	0.05 s	1.38 s
1024	6.10 s	98.00 s	49.35 s	236.00 s	0.20 s	5.55 s

single thread and 16 threads, respectively. Here, the latter version with compares full hoisting to hoisting combined with key re-use as described in Subsection 3.1. In Table 5, we compare memory usage and running time for pre-rotation with $d = 128$ for a single thread.

For all benchmarks, we check the total running time and the total amount of memory. For the latter, we use the command line tool `time`, that is the memory allocated by the program on a whole. This for example also includes the memory required by the benchmarking library, giving us only a rough estimate of the memory usage of the actual triple generation.

For the hoisting/re-use time-memory trade-off, the single-threaded results show that we roughly save 4 GiB of memory for a 20 % increase in the running time. This is especially interesting for hardware implementations that are limited in their memory resources, but can easily offset the performance overhead 20 %. In the multi-threaded setting, we only save roughly 3.5 GiB of memory, but the overhead to the running time is only 5–10 %, getting less with an increasing amount of blocks.

The impact of pre-rotation on the other hand is much larger. Without pre-rotation, the memory usage stays almost the same (the slight increase is due to the unencrypted input and output matrices). Here, the overhead as expected roughly corresponds to a multiple of the running time of the rotations: for input matrices with two blocks by a factor of one, with three blocks by two and with four blocks by three.

Matrix Slicing

For matrix slicing, we run the benchmarks with 128 threads, enabling hoisting and pre-rotation for $d = 32$, $d = 64$ and $d = 128$. Note that even

Table 3: Evaluation of hoisting and key re-use (one thread).

	128	256	384	512
		Time in s		
Hoisting	74.8	325	792	1535
Key Re-Use	90.8	392	952	1839
		Memory in GiB		
Hoisting	9.08	20.94	40.96	68.35
Key Re-Use	5.30	17.18	36.99	64.72

Table 4: Evaluation of hoisting and key re-use (16 threads).

	128	256	384	512
		Time in s		
Hoisting	6.48	32.3	80.4	154
Key Re-Use	7.72	35.3	86.2	163
		Memory in GiB		
Hoisting	9.84	21.02	40.77	68.43
Key Re-Use	5.83	17.68	37.44	65.10

Table 5: Evaluation of pre-rotation and re-computation.

	128	256	384	512
		Time in s		
Pre-Rotation	74.8	325	792	1535
Re-Computation	75.3	597	2024	4741
		Memory in GiB		
Pre-Rotation	9.08	20.94	40.96	68.35
Re-Computation	4.49	4.84	5.42	6.23

Table 6: Comparison of sliced versus non-sliced triples.

d	m	Blocks	Time
32	32	6	4.53 s
64	8	3	3.81 s
128	2	2	7.10 s

though we run all the benchmarks with 128 threads, the implementation only makes use of d threads. A summary of our results with amortized timing is presented in Table 6. The table also contains the amount of packed matrices m for each dimension and how many blocks are needed to construct the output matrix of size 192×192.

Here, it clearly shows that using the smaller dimension of 64 bit for the square matrix blocks results in a better performance compared to the 128 bit blocks. However, using the even smaller 32 bit blocks results in a worse amortized performance compared to the 64 bit blocks. In general, we recommend to benchmark the performance difference for the desired output dimensions, thus making sure to choose the best configuration possible.

Discussion
Section 5

Before concluding our work, we will discuss some possibilities for improvement and continuation for our work and reflect on general learnings for implementing HE use cases. We will start by discussing the selection process for the HE scheme, talk about further possibilities for advancing parameter generation and the challenges memory presents for HE use cases in general.

5.1 Choosing the HE Scheme

BGV-like schemes in general perform well for highly parallelizable tasks based on addition and multiplication. As matrix multiplication is such a task, using BGV-like schemes is a natural fit.

For the matrix triples, we are interested in integer arithmetic, for which there currently are two state-of-the-art schemes: BGV and BFV. For the matrix triple implementation, we believe that BGV is the superior choice for two reasons: parameter generation and the current ecosystem.

Previous work shows that BGV performs better for large plaintext moduli compared to BFV [7]. Thus, a BGV library and a BGV implementation of the matrix triples also requires less memory. Additionally, due to the smaller parameters, we also reduce runtime costs since we need to

compute on less RNS primes.

Additionally, the well known SCALE-MAMBA framework uses BGV as its homomorphic scheme using a custom version of HElib[8]. It also includes the ZKP implementation of TopGear [3]. Using our BGV backend instead, this provides a full solution to generate maliciously secure matrix triples and we currently work on providing this solution to the research community.

5.2 Advancing Parameter Generation

Our work is a next step in generating parameters for BGV-like schemes and opens up further possibilities for future research.

The most obvious possibility is automatically determining places for modulus switching candidates for a given circuit. Optimally, this also would take into account further opportunities for optimization such as combining the modulus switching with key switching or lazy modulus switching where, for example, modulus switching is applied after multiple results are accumulated and thus less computations have to be performed.

Extending this idea to larger circuits, another possibility is automatically determining where bootstrapping has to take place in any given circuit. Finding solutions for both approaches and combining it with our newly proposed approach would lead to automatic parameter generation for arbitrary BGV circuits, not only leveled circuits.

Other future research could look at modulus switching when $q_i \neq 1 \bmod t$, needed for arbitrarily large plaintext moduli, as it is no longer possible to compute between ciphertexts with different scaling factors. Automatically determining where scaling factors could be applied pre-encryption and post-encryption and where only a constant multiplication, which then influences error growth again, is needed is still an open research question.

5.3 Dealing with Memory Costs

As in previous work [5], we noticed two main bottlenecks for implementing the HE use case: the NTT computation and memory operations. For the former, there are multiple interesting approaches to accelerate these in hardware [9, 10] and reduce the NTT as computational bottleneck to a minimum.

For the latter, there are still open challenges such as reducing the amount of temporary buffers needed or optimizing the data layout depending on the circuit to be computed. However, we believe it to be a necessity to find solutions dealing with the memory wall to bring HE forward.

8 https://github.com/homenc/HElib

5.4 Designing the API

As discussed in Subsection 5.3, there are still large computing and memory bottlenecks when implementing HE use cases. With the design of our Application Programming Interface (API) for the BGV library, we put the focus on enabling the full use of available computing resources if suitable for the specific use case. This includes transparent data structures, access to lower-level API functions for experts and the possibility to apply multi-threading across the individual RNS moduli.

For ease of use, we also handle the conversion of ciphertexts from and to the NTT domain automatically as manual handling proved to be to error prone for library users. One of the things we would like to improve on in a future version of the library is the control over memory resources of the library as we currently need temporary allocations for certain subroutines.

Although this is the same approach other libraries currently take, we believe that due to the memory bottlenecks in HE implementations, this aspect should receive more attention from the research and developer community.

Conclusion
Section 6

In this work, we implement and optimize the homomorphic generation of matrix triples. We provide an open-source implementation for the leveled BGV scheme supporting plaintext moduli of arbitrary size using state-of-the-art implementation techniques. We also provide a new, use-case specific approach to parameter generation for leveled BGV-like schemes heuristically optimizing for computation time and taking into account architecture-specific constraints.

Additionally, we provide an in-depth analysis of the homomorphic circuit enabling the re-use of key switching keys and eliminating constant multiplications, combining our results in the first publicly available implementation to generate homomorphic matrix triples for arbitrary plaintext moduli.

Furthermore, we benchmark and evaluate our work compared to the previous implementation of Chen et al. [6] and evaluate different time-memory tradeoffs as well as matrix slicing for certain output dimensions. We also highlight additional optimization opportunities such as matrix slicing.

Our implementation is publicly available and up to 2.1× faster compared to previous work and provides additional time-memory trade-offs for different computing environments.

6.1 Acknowledgements

The work described in this paper has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972 and by the Cryptography Research Center at Technology Innovation Institute (TII), Abu Dhabi.

References

[1] Martin R. Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin E. Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. "Homomorphic Encryption Standard". In: IACR Cryptol. ePrint Arch. 2019.939 (2019). URL: https://eprint.iacr.org/2019/939.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. "On the concrete hardness of Learning with Errors". In: J. Math. Cryptol. 9.3 (2015), pp. 169–203. URL: http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml.

[3] Carsten Baum, Daniele Cozzo, and Nigel P. Smart. "Using TopGear in Overdrive: A More Efficient ZKPoK for SPDZ". In: Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers. Ed. by Kenneth G. Paterson and Douglas Stebila. Vol. 11959. Lecture Notes in Computer Science. Springer, 2019, pp. 274–302. DOI: 10.1007/978-3-030-38471-5_12. URL: https://doi.org/10.1007/978-3-030-38471-5_12.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: ACM Trans. Comput. Theory 6.3 (2014), 13:1–13:36. DOI: 10.1145/2633600. URL: https://doi.org/10.1145/2633600.

[5] Leo de Castro, Rashmi Agrawal, Rabia Tugce Yazicigil, Anantha P. Chandrakasan, Vinod Vaikuntanathan, Chiraag Juvekar, and Ajay Joshi. "Does Fully Homomorphic Encryption Need Compute Acceleration?" In: IACR Cryptol. ePrint Arch. 2021.1636 (2021). URL: https://eprint.iacr.org/2021/1636.

[6] Hao Chen, Miran Kim, Ilya P. Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. "Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning". In: Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12493. Lecture Notes in Computer Science. Springer, 2020, pp. 31–59. DOI: 10.1007/978-3-030-64840-4_2. URL: https://doi.org/10.1007/978-3-030-64840-4_2.

[7] Ana Costache and Nigel P. Smart. “Which Ring Based Somewhat Homomorphic Encryption Scheme is Best?” In: Topics in Cryptology - CT-RSA 2016 - The Cryptographers’ Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings. Ed. by Kazue Sako. Vol. 9610. Lecture Notes in Computer Science. Springer, 2016, pp. 325–340. DOI: 10.1007/978-3-319-29485-8_19. URL: https://doi.org/10.1007/978-3-319-29485-8_19.

[8] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 643–662. DOI: 10.1007/978-3-642-32009-5_38. URL: https://doi.org/10.1007/978-3-642-32009-5_38.

[9] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. “F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption”. In: MICRO ’21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021. ACM, 2021, pp. 238–252. DOI: 10.1145/3466752.3480070. URL: https://doi.org/10.1145/3466752.3480070.

[10] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. “BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption”. In: IACR Cryptol. ePrint Arch. 2022.657 (2022). URL: https://eprint.iacr.org/2022/657.

[11] Craig Gentry, Shai Halevi, and Nigel P. Smart. “Homomorphic Evaluation of the AES Circuit”. In: Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Springer, 2012, pp. 850–867. DOI: 10.1007/978-3-642-32009-5_49. URL: https://doi.org/10.1007/978-3-642-32009-5_49.

[12] Charles Gouert, Rishi Khan, and Nektarios Georgios Tsoutsos. “Optimizing Homomorphic Encryption Parameters for Arbitrary Applications”. In: IACR Cryptol. ePrint Arch. 2022.575 (2022). URL: https://eprint.iacr.org/2022/575.

[13] Shai Halevi and Victor Shoup. “Algorithms in HElib”. In: Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. Lecture Notes in Computer Science. Springer, 2014, pp. 554–571. DOI: 10.1007/978-3-662-44371-2_31. URL: https://doi.org/10.1007/978-3-662-44371-2_31.

[14] Shai Halevi and Victor Shoup. “Faster Homomorphic Linear Transformations in HElib”. In: Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I. Ed. by Hovav Shacham and Alexandra Boldyreva.

Vol. 10991. Lecture Notes in Computer Science. Springer, 2018, pp. 93–120. DOI: 10.1007/978-3-319-96884-1_4. URL: https://doi.org/10.1007/978-3-319-96884-1_4.

[15] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. "Secure Outsourced Matrix Computation and Application to Neural Networks". In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1209–1222. DOI: 10.1145/3243734.3243837. URL: https://doi.org/10.1145/3243734.3243837.

[16] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: J. ACM 60.6 (2013), 43:1–43:35. DOI: 10.1145/2535925. URL: https://doi.org/10.1145/2535925.

[17] Johannes Mono, Chiara Marcolla, Georg Land, Tim Güneysu, and Najwa Aaraj. "Finding and Evaluating Parameters for BGV". In: IACR Cryptol. ePrint Arch. 2022.706 (2022). URL: https://eprint.iacr.org/2022/706.

[18] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. "Actively Secure Setup for SPDZ". In: J. Cryptol. 35.1 (2022), p. 5. DOI: 10.1007/s00145-021-09416-w. URL: https://doi.org/10.1007/s00145-021-09416-w.

[19] Gregor Seiler. "Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography". In: IACR Cryptol. ePrint Arch. 2018.39 (2018). URL: http://eprint.iacr.org/2018/039.

[20] Nigel P. Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations". In: Des. Codes Cryptogr. 71.1 (2014), pp. 57–81. DOI: 10.1007/s10623-012-9720-4. URL: https://doi.org/10.1007/s10623-012-9720-4.