RESEARCH-ARTICLE

Lodia: Towards Optimal Sparse Matrix-Vector Multiplication for Batched Fully Homomorphic Encryption

JIPING YU, Tsinghua University, Beijing, China

KUN CHEN, Ant group, Hangzhou, Zhejiang, China

XIAOYU FAN, Tsinghua University, Beijing, China

YUNYI CHEN, Tsinghua University, Beijing, China

XIAOWEI ZHU, Ant group, Hangzhou, Zhejiang, China

WENGUANG CHEN, Tsinghua University, Beijing, China

Open Access Support provided by:

Ant group

Tsinghua University

Published: 19 November 2025

Citation in BibTeX format

CCS '25: ACM SIGSAC Conference on Computer and Communications Security
October 13 - 17, 2025
Taipei, Taiwan

Conference Sponsors:
SIGSAC

https://doi.org/10.1145/3719027.3765025
ISBN: 9798400715259

Lodia: Towards Optimal Sparse Matrix-Vector Multiplication for Batched Fully Homomorphic Encryption

Jiping Yu*
Tsinghua University
Beijing, China
Ant Group
Beijing, China
yjp19@mails.tsinghua.edu.cn

Kun Chen
Ant Group
Beijing, China
ck413941@antgroup.com

Xiaoyu Fan*
Tsinghua University
Beijing, China
Ant Group
Beijing, China
fanxy98@mail.tsinghua.edu.cn

Yunyi Chen*
Tsinghua University
Beijing, China
Ant Group
Beijing, China
cyy23@mails.tsinghua.edu.cn

Xiaowei Zhu
Ant Group
Beijing, China
robert.zxw@antgroup.com

Wenguang Chen
Tsinghua University
Department of Computer Science and SIGS
Beijing, China
Ant Group
Beijing, China
cwg@tsinghua.edu.cn

Abstract

Encrypted matrix-vector multiplication is a fundamental component of a variety of applications that involve data privacy concerns. Current algorithms utilizing fully homomorphic encryption (FHE) generally use *batching* to enhance computational efficiency while neglecting the *sparsity* of the matrices, a characteristic that exists naturally in many practical situations. Alternatively, porting plaintext algorithms that skip zero elements to address sparsity may fail to utilize batching and introduce additional privacy concerns.

We propose Lodia, an efficient outsourced sparse matrix-vector multiplication (SpMV) algorithm for batched FHE schemes without sacrificing privacy. It only requires $\Theta((n+m)\log(n+m)/s)$ FHE operations, where n is the number of rows/columns, m is the number of non-zero elements of the matrix, and s is the batch size of the FHE scheme. This is optimal for $m = \Omega(n)$ and $m = O(n^\rho)$ for some $\rho < 2$ (i.e., $an \le m \le bn^\rho$ asymptotically), covering most practical cases. To our knowledge, no method has been published with better than $\Theta(n^2/s)$ FHE operations, suitable for any sparse matrix, and without privacy concerns.

Lodia utilizes a novel *low-diagonal decomposition*, which decomposes a sparse matrix into a series of special matrices named low-diagonal matrices. Based on a conventional method encoding the matrix in diagonal order, each low-diagonal matrix can be efficiently multiplied by a vector. This results in an efficient SpMV method suitable for any sparse matrix. Experiments show that Lodia practically achieves a speedup of up to 96× compared to baselines that ignore matrix sparsity, and up to 3.6× compared to implementations even with fewer security guarantees. This is the first SpMV solution on encrypted data that can process a substantial matrix with over 8 million rows/columns and 125 million non-zero elements.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Sparse matrix-vector multiplication, Homomorphic encryption

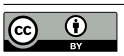
ACM Reference Format:
Jiping Yu, Kun Chen, Xiaoyu Fan, Yunyi Chen, Xiaowei Zhu, and Wenguang Chen. 2025. Lodia: Towards Optimal Sparse Matrix-Vector Multiplication for Batched Fully Homomorphic Encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765025

1 Introduction

Matrix-vector multiplication (MVM) has been recognized as a fundamental mathematical primitive and is widely used in numerous fields, e.g., quantitative economics [17, 30], neural networks [6, 22], information retrieval [41] and graph algorithms [32, 37]. In these domains, the importance of information security and privacy continues to grow, emphasizing the necessity for privacy-preserving MVM computation. *Fully homomorphic encryption (FHE)* is well suited for this setting, as it enables addition and multiplication operations on encrypted data that align with the needs of MVM. Thus, the data provider can encrypt the data and outsource the computation to a server, with FHE ensuring that the server does not learn sensitive information about the data throughout the computation.

As privacy-preserving MVM serves as a foundational element in many algorithms, its efficiency is crucial. For acceleration, one approach adopted by a series of existing studies [2, 10, 24, 26, 28, 34, 35] is to utilize the *single instruction multiple data (SIMD)* capabilities offered by certain FHE schemes. Such schemes like BFV [7, 18],

*Work is done during the research internship at Ant Group.



CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765025

BGV [8], and CKKS [11], allow simultaneous addition or multiplication of a *batch* of s encrypted values. If properly used in MVM, they can achieve better performance [13] compared to non-batched schemes like FHEW [16] or TFHE [12]. To multiply an $n \times n$ matrix with an $n \times 1$ vector, they require $\Theta(n^2/s)$ batched FHE operations.

Nevertheless, the majority of existing research on batched FHE has not thoroughly investigated another approach that is widely used and effective in plaintext MVM: exploiting the inherent sparsity of matrices. An $n \times n$ matrix is considered *sparse* if it includes only m non-zero elements, where $m \ll n^2$. Such matrices are prevalent in numerous real-world scenarios. For example, each constraint within a linear optimization problem might involve just a few variables, and each vertex in a graph might be linked to only a small subset of other vertices. In these instances, MVM becomes *sparse matrix-vector multiplication (SpMV)*. Leveraging this sparsity is essential for the development of efficient SpMV algorithms. For instance, a basic plaintext SpMV can bypass zero elements and conduct computations solely on the m non-zero elements, thus achieving a complexity of $\Theta(n+m)$, which eliminates the n^2 term.

Privacy-preserving SpMV based on batched FHE imposes unique challenges. The above $\Theta(n^2/s)$ methods, hereinafter called **dense baseline**, lead to inefficiency as they must handle all n^2 elements. Porting algorithms designed for plaintext SpMV can account for sparsity, yet it does not take advantage of batching and may lead to serious information leakage – by skipping zero elements, the locations of non-zero elements could be exposed, potentially revealing sensitive data, such as which vertex pairs are connected in a sparse graph. Hence, we will term it **sparse leaky baseline**. To our knowledge, although there exists an efficient algorithm specialized for permutation matrices [23], no SpMV algorithm for batched FHE has been published with better than $\Theta(n^2/s)$ FHE operations, suitable for any sparse matrix and without privacy concerns.

We propose Lodia, an efficient SpMV algorithm suitable for batched FHE schemes, which only requires $\Theta((n+m)\log(n+m)/s)$ FHE operations. This is optimal for $m = \Omega(n) \wedge m = O(n^\rho)$ for some $\rho < 2$ (i.e., $an \le m \le bn^\rho$ asymptotically), which is true for most real-world sparse matrices. The server is guaranteed to learn nothing about the matrix beyond basic information n and m.

We observed that an existing approach designed for dense matrices, namely the *diagonal order* method, can be used (after our modifications) to perform SpMV efficiently for particular matrices with a *low diagonal count*, meaning that all non-zero elements are confined to just a few diagonals. This inspires us to *decompose any sparse matrix as the product of multiple matrices, each having a low diagonal count* (~ 3). Afterwards, we only need to sequentially send each matrix to the (modified) diagonal order method, so that we get an efficient algorithm generally suitable for any sparse matrix.

To construct this **low diagonal decomposition (LDD)** for any matrix $\mathbf{A}$, we propose a novel method with a two-stage approach. **Coarse decomposition** decomposes $\mathbf{A}$ into four intermediate matrices $\mathbf{A} = \mathbf{GPDH}^\intercal$, where $\mathbf{G}$ and $\mathbf{H}$ are **ordered aggregation matrices** (Section 4.1), $\mathbf{P}$ is a permutation matrix and $\mathbf{D}$ is a diagonal matrix. **Fine decomposition** constructs an LDD for each intermediate matrix. Since $\mathbf{D}$ is already diagonal and the LDD of $\mathbf{P}$ is given by existing work [23], we propose new efficient algorithms to construct the LDD of $\mathbf{G}$ and $\mathbf{H}^\intercal$. Thus, we can construct an LDD of $\mathbf{A}$ by combining the decompositions of the intermediate matrices.

Based on LDD, we obtain an efficient SpMV algorithm suitable for any sparse matrix. Specifically, if $\mathbf{A} = \mathbf{A}^{(0)}\mathbf{A}^{(1)}\cdots\mathbf{A}^{(k-1)}$, we can rewrite $\mathbf{y} = \mathbf{Ax}$ as $\mathbf{y} = \mathbf{A}^{(0)}(\mathbf{A}^{(1)}(\cdots(\mathbf{A}^{(k-1)}\mathbf{x})\cdots))$ and perform each multiplication using the (modified) diagonal order approach. We name it Lodia as an abbreviation of **Lo**w **dia**gonal multiplication. We improve its practical performance by trading off between the computation depth and the number of FHE operations.

Our contributions are summarized as follows.

- We propose *low diagonal decomposition (LDD)*, to write a sparse matrix as the product of multiple matrices, whose non-zero elements occur only in a few (~ 3) diagonals. We propose a novel and efficient two-stage approach to construct the LDD for any sparse matrix. This may be of independent interest, e.g., help develop SpMV methods for other computation models.
- Based on LDD, we propose our SpMV algorithm named Lodia. Given the batch size s, as well as a matrix of n rows/columns and m non-zero elements, Lodia only needs $\Theta((n+m)\log(n+m)/s)$ operations, in contrast to previous $\Theta(n^2/s)$ methods, leaky approaches, or those only suitable for specific matrices. This is **optimal** for $m = \Omega(n) \wedge m = O(n^\rho)$ for some $\rho < 2$. For security, Lodia reveals nothing beyond n and m to the server.
- We propose optimizations to accommodate Lodia in a limited depth, enhancing its practical performance. Lodia achieves a speedup of up to 96× compared to *dense baseline* that do not utilize matrix sparsity and up to 3.6× compared to *sparse leaky baseline* with fewer security guarantees. This is the first SpMV solution on encrypted data that can handle a large matrix with over 8 million rows/columns and 125 million non-zero elements.

We hope that this work will inspire future research on sparse workloads and encourage interdisciplinary studies with other scientific fields. The remainder of this article is structured as follows. Section 2 provides background information on SpMV, batched FHE, and other related studies. Section 3 provides an overview of Lodia, including its formulation, associated challenges, and LDD. Section 4 introduces the mathematical perspectives of LDD. Section 5 presents the core design of Lodia. Section 6 introduces a practical optimization to fit Lodia to a limited depth. Section 7 evaluates the performance of Lodia. Finally, Section 8 concludes the paper.

2 Background and Related Work

In this section, we provide information on *sparse matrix-vector multiplication (SpMV)* and the batching operations of *fully homomorphic encryption (FHE)* schemes. Related work is also discussed.

2.1 Sparse Matrix-Vector Multiplication (SpMV)

Matrix-vector multiplication is a fundamental mathematical primitive, which is expressed as $\mathbf{y}_{n\times 1} = \mathbf{A}_{n\times n}\mathbf{x}_{n\times 1}$,[1] or $\mathbf{y}_j = \sum_{i=0}^{n-1} \mathbf{A}_{j,i}\mathbf{x}_i$, $\forall j \in \{0, 1, \ldots, n-1\}$. In various scenarios, the matrix can be *sparse*, that is, $\mathbf{A}$ contains only m non-zero elements and $m \ll n^2$, e.g., matrices from web graphs or social networks [1, 31], circuit simulation, analytics tasks, and optimization problems [14]. Thus, many existing studies [4, 9, 15, 29, 39, 42, 43] leverage this sparsity and propose efficient algorithms to perform (plaintext) sparse matrix-vector multiplication (SpMV). Their complexities are proportional

[1] To simplify descriptions, we assume that $\mathbf{A}$ is a square matrix here, while our algorithm can generally be applied to any non-square matrix.

to m rather than n^2, and thus outperform the methods designed for dense methods which require at least $\Theta(n^2)$ computation.

To investigate the sparsity of real-world matrices, we collect statistics on the ratio m/n from the SuiteSparse Matrix Collection [14], a widely recognized sparse matrix dataset. The median value of m/n is 7.9. A substantial portion (73.1%) of the matrices satisfy $m/n \leq 15$. Moreover, all matrices satisfy $m/n \geq 0.45$, and the vast majority (99.6%) satisfy $m/n \geq 1$. These indicate that, in most real-world sparse matrices, the number of nonzero elements m is typically only several times larger than the number of rows or columns n.

2.2 Batched Fully Homomorphic Encryption

Homomorphic encryption (HE) enables arithmetic operations on ciphertexts without leaking information about the corresponding plaintext. We use $[\![x]\!]$ to denote a ciphertext that decrypts to x. An HE scheme should guarantee *indistinguishability under chosen-plaintext attack (IND-CPA)*, which (informally) means that nothing about x should be learned from $[\![x]\!]$ without the corresponding secret key. An HE scheme also provides various homomorphic operators, e.g., $\boxplus$ such that $[\![x]\!] \boxplus [\![y]\!] \mapsto [\![x+y]\!]$ for any x, y. In this work, we focus on *fully homomorphic encryption (FHE)* schemes that support both addition and multiplication, in contrast to *partially homomorphic encryption (PHE)* that supports a single operation.

Various FHE schemes such as BFV [7, 18], BGV [8], and CKKS [11], can perform operations simultaneously on a batch of s values, which is known as *batching* or *single instruction multiple data (SIMD)*. For example, homomorphic addition $\boxplus$ can operate on two vectors, each with s elements, i.e. $[\![\mathbf{x}_{s\times 1}]\!] \boxplus [\![\mathbf{y}_{s\times 1}]\!] \mapsto [\![(\mathbf{x}+\mathbf{y})_{s\times 1}]\!]$. The s positions in a batch are called *slots*, and the value of s depends on the FHE parameters like ring dimension. For typical parameters, a batched arithmetic operation involving s slots, especially multiplication, is significantly faster than performing s independent operations in a non-batched FHE scheme like FHEW or TFHE [13]. Note that existing schemes operate in subtly different ways. BFV and BGV possess a constrained plaintext space, only working modulo a prime ideal. Although CKKS has a plaintext space of $\mathbb{C}$, its operations are inherently approximate, e.g., $[\![\mathbf{x}]\!] \boxplus [\![\mathbf{y}]\!] \mapsto [\![\mathbf{x}+\mathbf{y}+\mathbf{e}]\!]$ where $\mathbf{e}$ represents a minor perturbation.

To facilitate computation across different slots, they also provide *rotation* operations that effectively perform a circular shift on the corresponding plaintext. For CKKS, the rotation operator $\lll$ is provided so that $[\![\mathbf{x}_{s\times 1}]\!] \lll l \mapsto [\![\mathbf{x}_{l:s} \,||\, \mathbf{x}_{0:l}]\!]$, where $\mathbf{x}_{l:r}$ denotes a consecutive subvector $(\mathbf{x}_l, \mathbf{x}_{l+1}, \ldots, \mathbf{x}_{r-1})^\intercal$. For BFV/BGV, the operation abstracts rotating rows and/or columns of a $(s/2) \times 2$-matrix (see Section 3.1.2 for formal details).

2.3 Related Work

For small matrices. To utilize all slots, various existing studies [20, 21, 27, 36, 40, 46, 47] encode the entire matrix in the s slots to efficiently process small matrices. This requires $n^2 \leq s$, and the typical data size is only around $n \approx 64$. Ferguson et al. [19] leverage matrix sparsity in neural networks for efficient multiplication, but only consider small matrices up to $n = 64$.

For dense matrices. Various studies do not limit the matrix size, and the evaluations show that they support $n \approx 10^3$ [10, 26, 28, 38] or even $n \approx 10^4$ [2, 3, 24, 25, 34, 35]. However, these approaches are practically unable to process even larger matrices given reasonable hardware, because of a lower bound of n^2 for time and space complexity (and require $\Theta(n^2/s)$ FHE operations). Consider a matrix of $n = 8 \times 10^6$; stored in single precision without considering sparsity, it would require 256 TB even in an unencrypted form. An MVM would involve 6.4×10^{13} additions and multiplications of elements. Thus, they incur inefficiency when processing large sparse matrices. These techniques will be referred to as *dense baseline*.

In addition, some work [38] supports matrix multiplication of special shapes, e.g., $\mathbf{A}_{s\times n}\mathbf{B}_{n\times s}$ (where $1 \leq n \leq s$). So, it does not directly work on $\mathbf{A}_{n\times n}\mathbf{B}_{n\times 1}$ (i.e. matrix-vector multiplication).

For other HE schemes. There exist studies on SpMV but for other HE schemes, e.g., as a building block of decision tree algorithms [33, 44]. These methods are mainly limited by the versatility of the HE scheme. For example, Paillier-based methods are only applicable if the *plaintext* matrix can be revealed to the server.

For specific matrices. There are also methods of SpMV for batched FHE, but only suitable for specific matrices. Halevi and Shoup [23] propose an efficient method to multiply a permutation matrix with an encrypted vector based on Beneš networks [5], which is mathematically equivalent to constructing an LDD for a permutation matrix and serves as a building block in our LDD method.

However, to our knowledge, no efficient SpMV technique for batched FHE has been suggested that is generally applicable to any sparse matrix and without privacy concerns.

3 Motivation and Overview

This section provides motivation and an overview of this work. We formulate our task (Section 3.1) and discuss existing solutions (Section 3.2). Finally, we propose LDD and show its effect in developing an efficient SpMV algorithm for any sparse matrix (Section 3.3).

3.1 Problem Formulation

We formulate the task of a **generic SpMV algorithm for batched FHE schemes** as follows.

3.1.1 Setting. We identify a **matrix owner**, which holds the matrix $\mathbf{A}_{n\times n}$ with m non-zero elements,[2] and a **server**, which holds an encrypted vector $[\![\mathbf{x}_{n\times 1}]\!]$ and computes $[\![\mathbf{y}]\!]$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$. The server can only access $\mathbf{A}$ in an encrypted form, so that it learns nothing beyond n and m, from the encryption of $\mathbf{A}$.

We do not limit the source of $[\![\mathbf{x}]\!]$, e.g., is freshly encrypted by someone and sent to the server, or is the result of previous FHE computations on the server. Similarly, we do not limit how $[\![\mathbf{y}]\!]$ is further processed. The encryption of $\mathbf{x}$ may consist of multiple ciphertexts, e.g., $[\![\mathbf{x}_{n\times 1}]\!]$ is actually represented by $\lceil n/s \rceil$ ciphertexts $[\![\mathbf{x}_{0:s}]\!], [\![\mathbf{x}_{s:2s}]\!], \ldots$. This applies similarly to the result $[\![\mathbf{y}]\!]$.

3.1.2 Assumptions to the FHE Scheme. The algorithm is designed to be applicable to any FHE scheme that operates on a batch of s elements of a ring $(R, +, \times)$, where s is a power of 2. The scheme satisfies IND-CPA (indistinguishability under chosen-plaintext attack) and provides the following operations:

[2] We currently presume the existence of a matrix provider that has access to $\mathbf{A}$ in plaintext form. Future research is needed when $\mathbf{A}$ is only available as an encrypted matrix, because it would depend on the encryption format of $\mathbf{A}$ produced by previous computation. Fundamentally, our approach can be utilized if the preceding computational step results in an output that aligns with the format generated by EncryptMatrix.

- Encryption, $\text{Encrypt}(\mathbf{x}_{s\times 1}) \mapsto [\![\mathbf{x}]\!]$.
- Addition, $[\![\mathbf{x}_{s\times 1}]\!] \boxplus [\![\mathbf{y}_{s\times 1}]\!] \mapsto [\![\mathbf{x}+\mathbf{y}]\!]$, where $(\mathbf{x}+\mathbf{y})_i = \mathbf{x}_i + \mathbf{y}_i$.[3]
- Multiplication, $[\![\mathbf{x}_{s\times 1}]\!] \boxtimes [\![\mathbf{y}_{s\times 1}]\!] \mapsto [\![\mathbf{x}\circ\mathbf{y}]\!]$, where $(\mathbf{x}\circ\mathbf{y})_i = \mathbf{x}_i \times \mathbf{y}_i$.
- Rotation, $[\![\mathbf{x}_{s\times 1}]\!] \lll l \mapsto [\![\mathbf{y}_{s\times 1}]\!]$, where each element (1) $\mathbf{x}_i = \mathbf{y}_{(i+l) \bmod s}$ for CKKS-like schemes, or (2) $\mathbf{x}_i = \mathbf{y}_{(s/2)\cdot((\lfloor \frac{i}{s/2} \rfloor + \lfloor \frac{l}{s/2} \rfloor) \bmod 2) + ((i+l) \bmod (s/2))}$ for BFV/BGV.
- Decryption (also given the secret key), $\text{Decrypt}([\![\mathbf{x}_{s\times 1}]\!]) \mapsto \mathbf{x}$.

3.1.3 Functions. The algorithm should provide two functions named EncryptMatrix and SpMV. The matrix owner executes EncryptMatrix, which inputs $\mathbf{A}$ and outputs some information, including plaintext and an array of ciphertext, to be sent to the server. The server executes SpMV, which inputs anything produced by EncryptMatrix and a ciphertext $[\![\mathbf{x}]\!]$, and outputs $[\![\mathbf{y}]\!]$.

3.1.4 Security. It is assumed that both the matrix owner and the server are semi-honest, implying that they will not supply fake input or generate false computation results. The algorithm must guarantee that, from the result of EncryptMatrix, the server cannot distinguish two matrices with the same n and m. (Specifically, the server should not gain knowledge about the values of the non-zero elements or their positions.)

We choose the above security requirements, because when such an algorithm is used in a protocol, semi-honest and indistinguishably help prove the security of the protocol. Specifically, Appendix B provides a concrete example protocol, which use such indistinguishability to prove its security against semi-honest adversaries.[4]

3.2 Challenges and Existing Solutions

Now we analyze the challenge of the task formulated above by inspecting the defects of existing solutions.

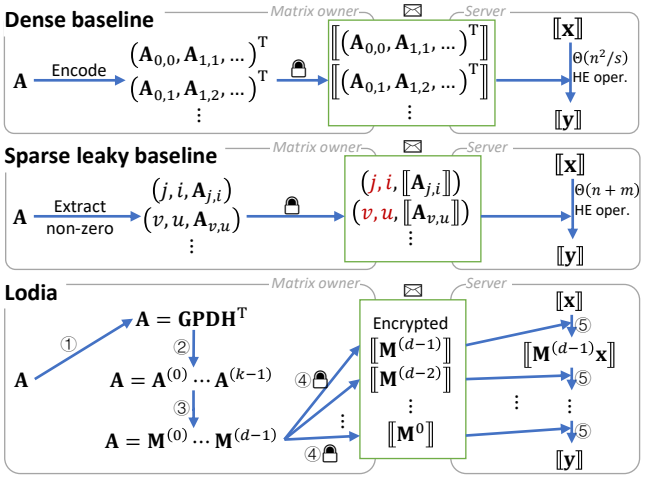
Dense baseline. Halevi and Shoup [23] summarized three types of order to encode a dense matrix, namely *column order*, *row order*, and *diagonal order*. These orders have also been recognized by the following studies until now [2, 10, 24, 26, 28, 34, 35]. Among these methods, the baby-step-giant-step (BSGS) technique [24], as an optimization based on the diagonal order method, is generally recognized as the state-of-the-art for scenarios where $n \geq s$ [3, 26, 28]. However, these methods are unaware of the sparsity and require $\Theta(n^2/s)$ FHE operations. As only m of the n^2 elements are useful, such dense methods incur suboptimal complexity. In our experiments, we will demonstrate that Lodia exhibits a growing advantage over the *dense baseline* with larger input matrices.

Sparse leaky baseline. Plaintext SpMV can be trivially implemented in $\Theta(n+m)$ complexity based on $\mathbf{y}_j = \sum_{i=0}^{n-1} \mathbf{A}_{j,i}\mathbf{x}_i$ (skipping zero elements to utilize sparsity). However, such a trivial method does not utilize batching and raises privacy concerns. For example, if the matrix provider encrypted $\mathbf{A}$ into a series of $(j, i, [\![\mathbf{A}_{j,i}]\!])$ tuples for $\mathbf{A}_{j,i} \neq 0$, the server would know the positions of the non-zero elements. This reveals significant details; for instance, by identifying positions within the adjacency matrix of a sparse graph, the server gains insight into which vertices are connected.

In addition, such leakage cannot be efficiently mitigated in a trivial way. For example, if the matrix provider also encrypted j and i (or in equivalent forms such as a $(0, 1)$-mask), it would degrade the complexity, e.g., to $\Theta(nm/s)$, making it even worse than the dense baseline (as long as $m \geq n$ for most cases). Even if such leakage in this baseline is kept, our method will still demonstrate superiority over this *sparse leaky baseline* in the experiments.

The details of *sparse leaky baseline* are elaborated in Appendix A. Note that we choose this as a baseline because this is what one could think of to utilize matrix sparsity, and there lacks published work about SpMV on batched FHE (Section 2.3).

Figure 1 illustrates the workflow of the baselines for reference.



① Coarse decomposition (§4.2)
② Fine decomposition (§4.3), $k = \Theta(\log(n+m))$
③ Depth optimization (§6)
④ Encrypt of modified diagonal order (§5.1)
⑤ SpMV of modified diagonal order (§5.1)

Figure 1: Workflow of Lodia and baselines

3.3 Overview of Lodia

An important observation is that with our specific modifications (Section 5.1), the diagonal order method operates efficiently on *low-diagonal* matrices. It further inspires us to decompose any sparse matrix into the product of multiple matrices, each of which can be efficiently processed in diagonal order. To begin with, we first define such low-diagonal matrices as follows.

Definition 1. The (non-zero) diagonal set of a matrix $\mathbf{A}$ is defined as $\text{DS}(\mathbf{A}) = \{d \mid \exists(j,i)(\mathbf{A}_{j,i} \neq 0 \wedge j - i = d)\}$.

$|\text{DS}(\mathbf{A})|$ reflects the (non-zero) diagonal count of $\mathbf{A}$. If it is bounded by a constant like 3, we refer to it as a low-diagonal matrix. We claim that, for such matrices,[5] the modified diagonal order method only requires $O(n/s)$ FHE operations to compute SpMV.

[3] While CKKS does not strictly adhere to this definition because of the extra perturbation term $\mathbf{e}$, this approximation does not influence algorithm design. Although a larger computation depth may result in reduced precision, our method retains at least 30-bit precision in the experiments. Therefore, CKKS is considered to perform effectively on the ring $(\mathbb{C}, +, \times)$ in this work.

[4] Note that we do not limit where $[\![\mathbf{x}]\!]$ comes from or where $[\![\mathbf{y}]\!]$ goes to, and the other parts of the protocol can be arbitrary, so we could not "enumerate all possible protocols" and prove that indistinguishably is enough for semi-honest security. Nevertheless, the proving strategy of other protocols would be similar to our example.

[5] A very formal expression would be "there exists a set $\tilde{D}$ computable from public information such that $\text{DS}(\mathbf{A}) \subseteq \tilde{D} \wedge |\tilde{D}| = O(1)$." See Theorem 5 for details.

The technical details of the (modified) diagonal order method will be given later in Section 5.1. For now, let us assume the existence of such an algorithm and continue with our intuition.

Our core idea is to decompose any sparse matrix $\mathbf{A}$ into the product of multiple low-diagonal matrices, so that each decomposed matrix can be efficiently handled. We call this process *low diagonal decomposition (LDD)*. To construct an LDD of $\mathbf{A}$, our method is characterized by two stages. The coarse decomposition writes $\mathbf{A}$ into four intermediate matrices $\mathbf{A} = \mathbf{GPDH}^\intercal$, where $\mathbf{G}$ and $\mathbf{H}$ are *ordered aggregation matrices* (Section 4.1), $\mathbf{P}$ is a permutation matrix and $\mathbf{D}$ is a diagonal matrix. Then, the fine decomposition finds LDD for each of them. This two-stage fashion effectively simplifies the problem – $\mathbf{D}$ does not need decomposition, the LDD of $\mathbf{P}$ is given by [23], and the decomposition of $\mathbf{H}^\intercal$ is symmetrical to that of $\mathbf{G}$. Thus, we only need a method to decompose $\mathbf{G}$, to complete the LDD of any sparse matrix.

After constructing LDD, we obtain an efficient SpMV algorithm. For example, if $\mathbf{A} = \mathbf{A}^{(0)}\mathbf{A}^{(1)}\cdots\mathbf{A}^{(k-1)}$ where $|\mathsf{DS}(\mathbf{A}^{(\cdot)}) \le 3|$,[6] we can rewrite $\mathbf{y} = \mathbf{Ax}$ as $\mathbf{y} = \mathbf{A}^{(0)}(\mathbf{A}^{(1)}(\cdots(\mathbf{A}^{(k-1)}\mathbf{x})\cdots))$. Thus, we can encrypt each decomposed matrix and perform the multiplication with the modified diagonal order method.

For complexity, our LDD construction will decompose $\mathbf{A}$ into $k = \Theta(\log(n+m))$ matrices, where each matrix has a diagonal count bounded by a constant 3, and the number of rows/columns is $\Theta(n+m)$. Based on the complexity of each invocation of the diagonal order method, our algorithm Lodia only requires $\Theta((n+m)\log(n+m)/s)$ FHE operations, which is proven to be optimal for $m = \Omega(n) \wedge m = O(n^\rho), \rho < 2$ (Section 5.4). We further optimize the depth of Lodia by reducing the number of decomposed matrices to a given depth budget d, so it can practically achieve better performance.

The complete workflow of Lodia is shown in Figure 1. Next, we will give technical details on LDD (Section 4). Based on this, we introduce the design of Lodia to realize the complete SpMV algorithm (Section 5). In addition, we propose practical optimizations of Lodia of limited depth (Section 6).

4 Low Diagonal Decomposition (LDD)

This section covers most of the mathematical foundations of LDD. Specifically, this section aims to establish the following Theorem 1.

Definition 2. The bit-diff set of a matrix $\mathbf{M}$ is defined as $\mathsf{BDS}(\mathbf{M}) = \{b \mid \exists(j,i)\ \mathbf{M}_{j,i} \neq 0 \wedge \lfloor (j \text{ xor } i)/2^b \rfloor \bmod 2 = 1\}$.

We choose the below Theorem 1 to focus on BDS rather than previously mentioned DS, because BDS would facilitate the optimizations in Section 6. Also, observe that if $|\mathsf{BDS}(\mathbf{M})| \le 1$, then $|\mathsf{DS}(\mathbf{M})| \le 3$, i.e. $\mathbf{M}$ is indeed a low-diagonal matrix.

Theorem 1 (Low Diagonal Decomposition). *Given a matrix $\mathbf{A}_{n\times n}$ of m non-zero elements,* FindLDD *(Algorithm 2) outputs k matrices $\mathbf{A}^{(0)}_{n\times\tilde m}, \mathbf{A}^{(1)}_{\tilde m\times\tilde m}, \mathbf{A}^{(2)}_{\tilde m\times\tilde m}, \cdots, \mathbf{A}^{(k-2)}_{\tilde m\times\tilde m}, \mathbf{A}^{(k-1)}_{\tilde m\times n}$, where $\tilde m = 2^{\lceil \log_2(n+m)\rceil}$ and $k = 4\log_2\tilde m$, such that*
(1) $\mathbf{A} = \mathbf{A}^{(0)}\mathbf{A}^{(1)}\cdots\mathbf{A}^{(k-1)}$.
(2) For each $l \in [k]$, $\mathsf{BDS}(\mathbf{A}^{(l)}) \subseteq F(l,\tilde m)$, where $F(l,\tilde m) = \begin{cases} \{\log_2\tilde m - 1 - l\} & (0 \le l \le \log_2\tilde m - 1), \\ \{\log_2\tilde m - 1 - |2\log_2\tilde m - 1 - l|\} & (\log_2\tilde m \le l \le 3\log_2\tilde m - 2), \\ \varnothing & (l = 3\log_2\tilde m - 1), \\ \{l - 3\log_2\tilde m\} & (3\log_2\tilde m \le l \le 4\log_2\tilde m - 1). \end{cases}$

We establish the theorem as follows. Initially, we provide a formal definition of the previously mentioned *ordered aggregation matrix (OAM)* (Section 4.1). We then introduce the strategy to perform coarse decomposition given any sparse matrix (Section 4.2). Subsequently, we suggest the technique of fine decomposition, especially for OAM (Section 4.3). Lastly, we integrate the coarse and fine decomposition stages to construct the complete FindLDD algorithm, thus concluding the proof of the theorem (Section 4.4).

4.1 Ordered Aggregation Matrix (OAM)

Definition 3. We say that a matrix $\mathbf{G}_{n\times\tilde m}$ is an *ordered aggregation matrix (OAM)*, iff it satisfies all the following:
(1) Each element of $\mathbf{G}$ is either 0 or 1.
(2) $\mathbf{1}_{1\times n}\cdot\mathbf{G} = \mathbf{1}_{1\times\tilde m}$, i.e. each column contains exactly one 1.
(3) $q_{i-1} \le q_i \le q_{i-1} + 1$ for $0 < i < \tilde m$, where q_i denotes the row index of 1 in the i-th column, i.e. $\mathbf{G}_{q_i,i} = 1$.

Traditionally, an "aggregation matrix" means a $(0,1)$-matrix with fewer rows than columns. We added "ordered" here to emphasize that the 1's should be in a specific order. The notation $\mathbf{G}$ is the second letter of a**g**gregation, avoiding conflicts with $\mathbf{A}$.

Examples of OAMs include $\mathbf{G}$ and $\mathbf{H}$ in Equation 1 below. In an OAM, the 1's move in a staircase pattern down the matrix, which would help us devise an effective fine decomposition method (Section 4.3). Intuitively, an OAM is a binary matrix that, when multiplied by a vector, aggregates a consecutive subset of the vector's components into one result. In contrast, its transpose duplicates one component in multiple positions. Specifically, we have $\mathbf{Gx} = (\mathbf{x}_0 + \mathbf{x}_1 + \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4 + \mathbf{x}_5, \mathbf{x}_6, \mathbf{x}_7)^\intercal$ and $\mathbf{H}^\intercal\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)^\intercal$.

4.2 Coarse Decomposition

Based on the above, we propose *coarse decomposition* as follows.

For now, we only consider matrices without zero rows or columns, while the general case is handled after the proof of this theorem.

Theorem 2 (Coarse Decomposition of Specific Matrices). *For a matrix $\tilde{\mathbf{A}}_{n\times n}$ of $\tilde m$ non-zero elements, if each row or column contains at least one non-zero element, we can construct matrices $\mathbf{G}_{n\times\tilde m}, \mathbf{P}_{\tilde m\times\tilde m}, \mathbf{D}_{\tilde m\times\tilde m}$ and $\mathbf{H}_{n\times\tilde m}$ such that $\tilde{\mathbf{A}} = \mathbf{GPDH}^\intercal$, where $\mathbf{G}$ and $\mathbf{H}$ are OAMs, $\mathbf{P}$ is a permutation matrix (i.e. each row or column contains an 1 while the others are 0), and $\mathbf{D}$ is a diagonal matrix.*

An example of coarse decomposition is shown below.

$$\tilde{\mathbf{A}} = \begin{pmatrix} 1&2&1&0&0\\ 0&0&1&0&0\\ 1&0&0&1&0\\ 0&0&0&0&1\\ 0&0&1&0&0 \end{pmatrix} = \mathbf{GPDH}^\intercal,\ \text{where } \mathbf{G} = \begin{pmatrix} 1&1&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix},$$
$$\mathbf{P} = \begin{pmatrix} 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&1&0&0&0&0\\ 1&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix}, \mathbf{D} = \begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&2&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix}, \mathbf{H} = \begin{pmatrix} 1&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&1&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix}. \quad (1)$$

[6] Similarly, a formal expression would be "we can give a set $\tilde D^{(i)}$ for each $i = 0, 1, \ldots, k-1$ such that $\mathsf{DS}(\mathbf{A}^{(i)}) \subseteq \tilde D^{(i)} \wedge |\tilde D^{(i)}| = O(1)$".

Intuitively, this comes from a "seemingly inefficient" SpMV algorithm of four steps, where each step can be realized by a matrix.

(1) Copy each element several times. Specifically, $\mathbf{x}_i$ is replicated according to the count of non-zero values in the i-th column of $\tilde{\mathbf{A}}$. Each value now corresponds to a non-zero element, and they are ordered by column index. This can be implemented by multiplying a transposed OAM $\mathbf{H}^\intercal$. For the example matrix in Equation 1 with 8 non-zero elements, we have $\mathbf{H}^\intercal\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)^\intercal$.
(2) Multiply each element by the corresponding weight, which can be realized by a diagonal matrix $\mathbf{D}$, e.g. $\mathbf{DH}^\intercal\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_0, 2\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)^\intercal$.
(3) Arrange the elements (from the order by column index) into the order by row index, with the help of a permutation matrix $\mathbf{P}$, e.g. $\mathbf{PDH}^\intercal\mathbf{x} = (\mathbf{x}_0, 2\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_0, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_2)^\intercal$.
(4) Sum up consecutive elements from the same row with an OAM $\mathbf{G}$, e.g. $\mathbf{GPDH}^\intercal\mathbf{x} = (\mathbf{x}_0 + 2\mathbf{x}_1 + \mathbf{x}_2, \mathbf{x}_2, \mathbf{x}_0 + \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_2)^\intercal = \tilde{\mathbf{A}}\mathbf{x}$.

As $\mathbf{x}$ can be arbitrary, we have $\tilde{\mathbf{A}} = \mathbf{GPDH}^\intercal$, where each matrix can be constructed according to the process of the above algorithm.

Formally, we prove Theorem 2 as follows.

Proof. This proof is given by the following three steps.

I Construct the matrices $\mathbf{G}$, $\mathbf{P}$, $\mathbf{D}$, and $\mathbf{H}$, given $\tilde{\mathbf{A}}$.
II Verify the properties of each matrix, e.g., $\mathbf{G}$ is indeed an OAM.
III Verify $\tilde{\mathbf{A}} = \mathbf{GPDH}^\intercal$.

Step I. We first define two auxiliary matrices, $\mathbf{L}_{\tilde{m}\times 3}$ and $\mathbf{L}'_{\tilde{m}\times 3}$:

- $\mathbf{L}$: Each row corresponds to a non-zero element of $\tilde{\mathbf{A}}$. Specifically, each non-zero $\tilde{\mathbf{A}}_{j,i}$ corresponds to a row $(j, i, \tilde{\mathbf{A}}_{j,i})$ in $\mathbf{L}$. The rows are sorted by i, i.e., the column with index 1.
- $\mathbf{L}'$: Mostly similar to $\mathbf{L}$, but the rows are sorted by j.

For the example $\tilde{\mathbf{A}}$ in Equation 1, $\mathbf{L}$ and $\mathbf{L}'$ are constructed as

$$\mathbf{L} = \begin{pmatrix} 2&0&1\\ 0&0&1\\ 0&1&2\\ 1&2&1\\ 0&2&1\\ 4&2&1\\ 2&3&1\\ 3&4&1 \end{pmatrix}, \mathbf{L}' = \begin{pmatrix} 0&0&1\\ 0&1&2\\ 0&2&1\\ 1&2&1\\ 2&0&1\\ 2&3&1\\ 3&4&1\\ 4&2&1 \end{pmatrix}.$$

Based on $\mathbf{L}$ and $\mathbf{L}'$, the four matrices are constructed as

- $\mathbf{G}_{n\times\tilde{m}}$: $\mathbf{G}_{j,i} = 1$ if $\mathbf{L}'_{i,0} = j$; otherwise $\mathbf{G}_{j,i} = 0$.
- $\mathbf{P}_{\tilde{m}\times\tilde{m}}$: $\mathbf{P}_{j,i} = 1$ if $\mathbf{L}_{i,0:2} = \mathbf{L}'_{j,0:2}$; otherwise $\mathbf{P}_{j,i} = 0$.
- $\mathbf{D}_{\tilde{m}\times\tilde{m}}$: $\mathbf{D}_{j,i} = \mathbf{L}_{i,2}$ if $j = i$; otherwise $\mathbf{D}_{j,i} = 0$.
- $\mathbf{H}_{n\times\tilde{m}}$: $\mathbf{H}_{j,i} = 1$ if $\mathbf{L}_{i,1} = j$; otherwise $\mathbf{H}_{j,i} = 0$.

Note that the results in Equation 1 are exactly constructed as above.

Step II. $\mathbf{G}$ is an OAM because it satisfies Definition 3:

(1) Each element is 0 or 1 according to the construction.
(2) Each column i column contains an 1 in row $\mathbf{L}'_{i,0}$ (i.e., $\mathbf{G}_{\mathbf{L}'_{i,0},i} = 1$), while the other elements in column i are 0.
(3) Note that q_i in Definition 3 is actually $q_i = \mathbf{L}'_{i,0}$ here, and $\mathbf{L}'_{i,0}$ is increasing with respect to i because $\mathbf{L}'$ is sorted by column 0. For $0 < i < \tilde{m}$, $q_{i-1} \le q_i$ because $\mathbf{L}'_{i,0}$ is increasing. On the other hand, $q_i \le q_{i-1}+1$ must hold; otherwise if $q_i > q_{i-1}+1$, the row $q_{i-1}+1$ would contain no non-zero elements.

Similarly, we can show that $\mathbf{H}$ is also an OAM:

(1) Similarly, trivial.
(2) Each column contains exactly one 1 because $\mathbf{H}_{\mathbf{L}_{i,1},i} = 1$ while the other elements in column i are 0.
(3) $q_i = \mathbf{L}_{i,1}$ here, which is increasing with respect to i because $\mathbf{L}$ is sorted by this column. Further demonstrations also hold because $\tilde{\mathbf{A}}$ does not contain any column of purely zeros.

$\mathbf{P}$ is a permutation matrix, because $\mathbf{L}$ and $\mathbf{L}'$ contain the same set of rows but only in different orders, i.e., for each i, we can find exactly one j satisfying $\mathbf{L}_{i,0:2} = \mathbf{L}'_{j,0:2}$ and vice versa. It is also obvious that $\mathbf{D}$ is a diagonal matrix.

Step III. We verify $\tilde{\mathbf{A}} = \mathbf{GPDH}^\intercal$ by considering each element:

$$\begin{aligned}
&(\mathbf{GPDH}^\intercal)_{j,i}\\
&= \textstyle\sum_{k_1=0}^{\tilde{m}-1}\sum_{k_2=0}^{\tilde{m}-1}\sum_{k_3=0}^{\tilde{m}-1}\mathbf{G}_{j,k_1}\mathbf{P}_{k_1,k_2}\mathbf{D}_{k_2,k_3}\mathbf{H}_{i,k_3}\\
&= \textstyle\sum_{k_1=0}^{\tilde{m}-1}\sum_{k_2=0}^{\tilde{m}-1}\mathbf{G}_{j,k_1}\mathbf{P}_{k_1,k_2}\mathbf{L}_{k_2,2}\mathbf{H}_{i,k_2} && (\mathbf{D}_{k_2,k_3} = \mathbf{L}_{k_2,2} \Leftrightarrow k_2 = k_3)\\
&= \textstyle\sum_{k_1=0}^{\tilde{m}-1}\sum_{k_2 \text{ s.t. } \mathbf{L}_{k_2,1}=i}\mathbf{G}_{j,k_1}\mathbf{P}_{k_1,k_2}\mathbf{L}_{k_2,2} && (\mathbf{H}_{i,k_2} = 1 \Leftrightarrow \mathbf{L}_{k_2,1=i})\\
&= \textstyle\sum_{k_1 \text{ s.t. } \mathbf{L}'_{k_1,0}=j}\sum_{k_2 \text{ s.t. } \mathbf{L}_{k_2,1}=i}\mathbf{P}_{k_1,k_2}\mathbf{L}_{k_2,2} && (\mathbf{G}_{j,k_1} = 1 \Leftrightarrow \mathbf{L}'_{k_1,0} = j)\\
&= \textstyle\sum_{k_2 \text{ s.t. } \mathbf{L}_{k_2,0}=j\wedge\mathbf{L}_{k_2,1}=i}\mathbf{L}_{k_2,2} && (\mathbf{P}_{k_1,k_2} = 1 \Leftrightarrow \mathbf{L}_{k_2,0:2} = \mathbf{L}'_{k_1,0:2})\\
&= \tilde{\mathbf{A}}_{j,i}. && \text{(definition of } \mathbf{L})
\end{aligned}$$

This completes the proof. □

Possibly, matrix $\mathbf{A}$ cannot be decomposed directly as above, because it may contain a row or a column of zeros. In such cases, we change some elements from 0 to ϵ so that it becomes a decomposable matrix $\tilde{\mathbf{A}}$ with $\tilde{m}$ non-zero elements, where ϵ is viewed as a non-zero element but computes like 0.[7] To minimize information leakage from the value $\tilde{m}$ about the number of zero rows or columns, we let $\tilde{m} = 2^{\lceil\log_2(n+m)\rceil}$. This is always feasible because $\tilde{m} \ge n+m$ and we need at most n additional entries of ϵ to eliminate all rows or columns full of zeros. This does not affect the correctness.

Note that we let $\tilde{m}$ be a power of 2 here, to simplify the following stage of fine decomposition. To avoid confusion, we use m to denote the number of non-zero elements in the (original) matrix $\mathbf{A}$, and use $\tilde{m}$ to denote that in the matrix after adding the elements of ϵ.

Corollary 2.1. *For a matrix $\mathbf{A}_{n\times n}$ of m non-zero elements, we can construct matrices $\mathbf{G}_{n\times\tilde{m}}$, $\mathbf{P}_{\tilde{m}\times\tilde{m}}$, $\mathbf{D}_{\tilde{m}\times\tilde{m}}$ and $\mathbf{H}_{n\times\tilde{m}}$ such that $\mathbf{A} = \mathbf{GPDH}^\intercal$, where $\tilde{m} = 2^{\lceil\log_2(n+m)\rceil}$, $\mathbf{G}$ and $\mathbf{H}$ are OAMs, $\mathbf{P}$ is a permutation matrix, and $\mathbf{D}$ is a diagonal matrix.*

4.3 Fine Decomposition

Based on the result of the above coarse decomposition, we now need to construct the LDD for each of the four intermediate matrices in this fine decomposition stage. In fact, this part focuses on the way to decompose the OAM $\mathbf{G}$, because, for the other three matrices:

- $\mathbf{D}$ does not need further decomposition.
- The LDD of $\mathbf{P}$ is given by previous work [23]. Specifically, $\mathbf{P} = \mathbf{P}^{(0)}\mathbf{P}^{(1)}\cdots\mathbf{P}^{(2\log_2\tilde{m}-2)}$, where each $\mathbf{P}^{(\cdot)}$ is a $\tilde{m}\times\tilde{m}$ matrix and $\mathrm{BDS}(\mathbf{P}^{(l)}) \subseteq \{\log_2\tilde{m} - 1 - |\log_2\tilde{m} - 1 - l|\}$.
- The LDD of the transposed OAM $\mathbf{H}^\intercal$ can be derived from that of $\mathbf{H}$. Specifically, if $\mathbf{H} = \mathbf{H}^{(\log_2\tilde{m}-1)}\mathbf{H}^{(\log_2\tilde{m}-2)}\cdots\mathbf{H}^{(0)}$ (which can be obtained by invoking the LDD of an OAM like $\mathbf{G}$), we obtain $\mathbf{H}^\intercal = (\mathbf{H}^{(0)})^\intercal(\mathbf{H}^{(1)})^\intercal\cdots(\mathbf{H}^{(\log_2\tilde{m}-1)})^\intercal$.

Thus, we only need to find a method to decompose $\mathbf{G}$.

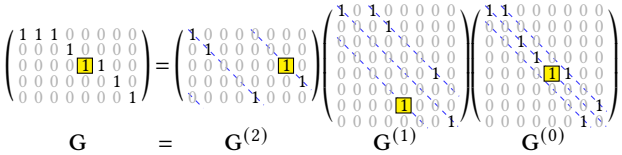
[7] Specifically, we have $\epsilon + 0 = \epsilon$, $\epsilon + a = a$, $\epsilon \times 0 = 0$, and $\epsilon \times a = \epsilon$ (for $a \ne 0$). ϵ is changed to 0 when sent to Encrypt.

Algorithm 1 FindLDD_G

Input An **ordered aggregation matrix** $\mathbf{G}_{n\times\tilde{m}}$ ($\tilde{m}$ is power of 2).
Output Matrices $\mathbf{G}^{(0)}, \mathbf{G}^{(1)}, \ldots, \mathbf{G}^{(\log_2 \tilde{m}-1)}$, such that $\mathbf{G} = \mathbf{G}^{(\log_2 \tilde{m}-1)}\mathbf{G}^{(\log_2 \tilde{m}-2)}\cdots\mathbf{G}^{(0)}$ (note the order).

1: $\mathbf{G}^{(0)}, \mathbf{G}^{(1)}, \ldots, \mathbf{G}^{(\log_2 \tilde{m}-2)} \leftarrow \mathbf{0}_{\tilde{m}\times\tilde{m}}$
2: $\mathbf{G}^{(\log_2 \tilde{m}-1)} \leftarrow \mathbf{0}_{n\times\tilde{m}}$
3: **for all** (j, i) such that $\mathbf{G}_{j,i} = 1$ **do**
4: $\quad k_0 \leftarrow i$
5: $\quad$ **for all** $b \in [\log_2 \tilde{m}]$ **do**
6: $\quad\quad$ **if** the b-th bit of j and i are the same **then**
7: $\quad\quad\quad k_{b+1} \leftarrow k_b$
8: $\quad\quad$ **else**
9: $\quad\quad\quad k_{b+1} \leftarrow k_b$ xor 2^b
10: $\quad\quad \mathbf{G}^{(b)}_{k_{b+1},k_b} \leftarrow 1$
11: **return** $\mathbf{G}^{(0)}, \mathbf{G}^{(1)}, \ldots, \mathbf{G}^{(\log_2 \tilde{m}-1)}$

We propose our method of constructing an LDD of an OAM, named FindLDD_G, in Algorithm 1. In the descriptions, we use $[n]$ to denote $\{0, 1, \ldots, n-1\}$. We start by initializing the $\log_2 \tilde{m}$ matrices with zeros. Then, we consider each non-zero element of $\mathbf{G}$. For each $\mathbf{G}_{j,i} = 1$ and for each bit b, we modify a specific element of $\mathbf{G}^{(b)}$, based on whether the b-th bit of j and i are the same or not (i.e. the binary representation of j xor i), where the b-th bit refers to that of the weight 2^b. After considering all such non-zero elements, we claim that we find an LDD of $\mathbf{G}$ as $\mathbf{G} = \mathbf{G}^{(\log_2 \tilde{m}-1)}\mathbf{G}^{(\log_2 \tilde{m}-2)}\cdots\mathbf{G}^{(0)}$. Note that the resulting matrices are multiplied in reverse. An example is presented below, which decomposes the matrix $\mathbf{G}$ in Equation 1.



$\mathbf{G}$ = $\mathbf{G}^{(2)}$ $\mathbf{G}^{(1)}$ $\mathbf{G}^{(0)}$

$\mathrm{BDS}(\cdot) \subseteq \{2\}$ $\mathrm{BDS}(\cdot) \subseteq \{1\}$ $\mathrm{BDS}(\cdot) \subseteq \{0\}$

$\mathrm{DS}(\cdot) \subseteq \{0, \pm 4\}$ $\mathrm{DS}(\cdot) \subseteq \{0, \pm 2\}$ $\mathrm{DS}(\cdot) \subseteq \{0, \pm 1\}$

In the example, $\mathbf{G}$ is decomposed to $\mathbf{G}^{(2)}\mathbf{G}^{(1)}\mathbf{G}^{(0)}$ such that each $\mathrm{BDS}(\mathbf{G}^{(b)}) \subseteq \{b\}$. We can verify that $\mathrm{DS}(\mathbf{G}^{(b)}) \subseteq \{0, \pm 2^b\}$, indicating that each decomposed matrix is a low-diagonal matrix (note that the possible non-zero diagonals are labeled with dashed lines).

For each 1 in $\mathbf{G}$, we modify an element to 1 in each result matrix, e.g., when the outer loop in FindLDD_G processes $\mathbf{G}_{2,4}$, it assigns $\mathbf{G}^{(2)}_{2,6} = \mathbf{G}^{(1)}_{6,4} = \mathbf{G}^{(0)}_{4,4} = 1$ (highlighted). This ensures that any (j, i) such that $\mathbf{G}_{j,i} = 1$ results in $(\mathbf{G}^{(2)}\mathbf{G}^{(1)}\mathbf{G}^{(0)})_{j,i} = 1$, and it is important to note that this is the only assignment approach compatible with the constraints of BDS, i.e., $k_2 = 6, k_1 = 4$ is the only solution of $\{2$ xor $k_2 =$ (either 0 or 4), k_2 xor $k_1 =$ (either 0 or 2), k_1 xor $4 =$ (either 0 or 1)$\}$. However, this does not establish the correctness of FindLDD_G; it is necessary to demonstrate that assigning 1 to the results will not "accidentally" introduce an extraneous entry of 1 into the product of the results that does not exist in $\mathbf{G}$, which is later proved based on the unique characteristics of OAM.

Formally, the properties of FindLDD_G are demonstrated below.

Theorem 3. *Each resulting matrix* $\mathbf{G}^{(b)}$ *of* FindLDD_G *satisfies* $\mathrm{BDS}(\mathbf{G}^{(b)}) \subseteq \{b\}$.

Algorithm 2 FindLDD

Input $\mathbf{A}_{n\times n}$ with m non-zero elements.
Output Matrices $\mathbf{A}^{(0)}_{n\times\tilde{m}}, \mathbf{A}^{(1)}_{\tilde{m}\times\tilde{m}}, \mathbf{A}^{(2)}_{\tilde{m}\times\tilde{m}}, \cdots, \mathbf{A}^{(k-2)}_{\tilde{m}\times\tilde{m}}, \mathbf{A}^{(k-1)}_{\tilde{m}\times n}$, where $\tilde{m} = 2^{\lceil \log_2(n+m)\rceil}$ and $k = 4\log_2 \tilde{m}$, such that $\mathbf{A} = \mathbf{A}^{(0)}\mathbf{A}^{(1)}\cdots\mathbf{A}^{(k-1)}$.

1: Coarse decomposition: find $\mathbf{G}$, $\mathbf{P}$, $\mathbf{D}$, and $\mathbf{H}$, such that $\mathbf{A} = \mathbf{GPDH}^\intercal$ (Section 4.2; use Corollary 2.1)
2: Fine decomposition:
3: $\quad \mathbf{G}^{(0)}, \mathbf{G}^{(1)}, \ldots, \mathbf{G}^{(\log_2 \tilde{m}-1)} \leftarrow$ FindLDD_G($\mathbf{G}$) (Algorithm 1)
4: $\quad \mathbf{P}^{(0)}, \mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(2\log_2 \tilde{m}-2)} \leftarrow$ FindLDD_P($\mathbf{P}$) ([23])
5: $\quad \mathbf{H}^{(0)}, \mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(\log_2 \tilde{m}-1)} \leftarrow$ FindLDD_G($\mathbf{H}$) (Algorithm 1)
$\quad$ *// Concatenate intermediate results*
6: $\mathbf{A}^{(0)}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(k-1)} \leftarrow \mathbf{G}^{(\log_2 \tilde{m}-1)}, \mathbf{G}^{(\log_2 \tilde{m}-2)}, \ldots, \mathbf{G}^{(0)}, \mathbf{P}^{(0)}, \mathbf{P}^{(1)}, \ldots, \mathbf{P}^{(2\log_2 \tilde{m}-2)}, \mathbf{D}, (\mathbf{H}^{(0)})^\intercal, (\mathbf{H}^{(1)})^\intercal, \ldots, (\mathbf{H}^{(\log_2 \tilde{m}-1)})^\intercal$
7: **return** $\mathbf{A}^{(0)}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(k-1)}$

Proof. Elements of $\mathbf{G}^{(b)}$ are only set to 1 in line 10. The possible values of (k_{b+1} xor k_b) are 0 and 2^b. According to the definition of $\mathrm{BDS}(\cdot)$, we obtain $\mathrm{BDS}(\mathbf{G}^{(b)}) \subseteq \{b\}$. □

Note that this indicates $\mathrm{DS}(\mathbf{G}^{(b)}) \subseteq \{0, 2^b, -2^b\}$, because xor operations can only add or subtract 2^b. Thus, $|\mathrm{DS}(\mathbf{G}^{(b)})| \leq 3$.

Theorem 4. FindLDD_G *produces correct results, i.e.,* $\mathbf{G} = \mathbf{G}^{(\log_2 \tilde{m}-1)}\mathbf{G}^{(\log_2 \tilde{m}-2)}\cdots\mathbf{G}^{(0)}$.

This can be demonstrated using the uniqueness of the binary decomposition of any integer, followed by employing a proof by contradiction to establish that each element on the left-hand side is equal to its counterpart on the right-hand side. Due to space constraints, the comprehensive proof is available in our technical report [45].

4.4 Putting Together

Combining the coarse and fine decomposition stages above, we present our complete algorithm FindLDD in Algorithm 2. To establish Theorem 1, we demonstrate that the characteristics of FindLDD essentially originate from those of the decomposed matrices.

Proof of Theorem 1. For (1), we have $\mathbf{A}^{(0)}\mathbf{A}^{(1)}\cdots\mathbf{A}^{(k-1)} = \mathbf{G}^{(\log_2 \tilde{m}-1)}\mathbf{G}^{(\log_2 \tilde{m}-2)}\cdots\mathbf{G}^{(0)}\mathbf{P}^{(0)}\mathbf{P}^{(1)}\cdots\mathbf{P}^{(2\log_2 \tilde{m}-2)}\mathbf{D}(\mathbf{H}^{(0)})^\intercal(\mathbf{H}^{(1)})^\intercal\cdots(\mathbf{H}^{(\log_2 \tilde{m}-1)})^\intercal = \mathbf{GPDH}^\intercal = \mathbf{A}$.

For (2), we have

- For $0 \leq l \leq \log_2 \tilde{m} - 1$, $\mathrm{BDS}(\mathbf{A}^{(l)}) = \mathrm{BDS}(\mathbf{G}^{(\log_2 \tilde{m}-1-l)}) \subseteq \{\log_2 \tilde{m} - 1 - l)\}$.
- For $\log_2 \tilde{m} \leq l \leq 3\log_2 \tilde{m} - 2$, $\mathrm{BDS}(\mathbf{A}^{(l)}) = \mathrm{BDS}(\mathbf{P}^{(l-\log_2 \tilde{m})}) \subseteq \{\log_2 \tilde{m} - 1 - |2\log_2 \tilde{m} - 1 - l|\}$.
- For $l = 3\log_2 \tilde{m} - 1$, $\mathrm{BDS}(\mathbf{A}^{(l)}) = \mathrm{BDS}(\mathbf{D}) = \varnothing$.
- For $3\log_2 \tilde{m} \leq l \leq 4\log_2 \tilde{m} - 1$, $\mathrm{BDS}(\mathbf{A}^{(l)}) = \mathrm{BDS}(\mathbf{H}^{(l-3\log_2 \tilde{m})}) = \{l - 3\log_2 \tilde{m}\}$.

This completes the proof of Theorem 1. □

5 Algorithm of Lodia

This section outlines the core algorithm of Lodia, derived from the earlier Theorem 1. Initially, we introduce our modified method for processing each low-diagonal matrix (Section 5.1) and subsequently showcase the full Lodia algorithm (Section 5.2). In addition, we

Algorithm 3 DiagEncryptMatrix

Input $\mathbf{A}_{n\times n}$ with m non-zero elements; a set $\tilde{D}$ satisfying $\mathrm{DS}(\mathbf{A}) \subseteq \tilde{D}$.
Output A list of ciphertexts, to be sent to DiagSpMV.

$E \leftarrow$ empty list of ciphertexts
for all $q \in [\lceil n/s \rceil]$ **do**
  for all $p \in [\lceil n/s \rceil]$ **do**
    for all $\hat{d} \in [s]$ **do**
      if $\neg(\exists k \in [s])$ such that $(qs+k)-(ps+\mathrm{Rot}(k,\hat{d})) \in \tilde{D}$ **then**
        continue
      $\mathbf{c}_{s\times 1} \leftarrow \{\text{each } \mathbf{c}_k = \mathbf{A}_{qs+k,ps+\mathrm{Rot}(k,\hat{d})}\}$
      Append $\mathrm{Encrypt}(\mathbf{c})$ to E
return E

// $\mathrm{Rot}(a,b)$ *denotes the target position if an element at position* a *is rotated by an offset* b*. Specifically,* $\mathrm{Rot}(a,b) = (a+b) \bmod s$*, if the FHE scheme provides CKKS-like rotation, or* $\mathrm{Rot}(a,b) = (s/2)\cdot((\lfloor \frac{a}{s/2} \rfloor + \lfloor \frac{b}{s/2} \rfloor) \bmod 2) + ((a+b) \bmod (s/2))$*, if it provides BFV/BGV-like rotation.*

Algorithm 4 DiagSpMV

Input E (output of DiagEncryptMatrix), $\tilde{D}$ (the same parameter sent to DiagEncryptMatrix), and $[\![\mathbf{x}_{n\times 1}]\!]$.
Output $[\![\mathbf{y}_{n\times 1}]\!]$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is the parameter sent to DiagEncryptMatrix.

for all $q \in [\lceil n/s \rceil]$ **do**
  for all $p \in [\lceil n/s \rceil]$ **do**
    for all $\hat{d} \in [s]$ **do**
      if $\neg(\exists k \in [s])$ such that $(qs+k)-(ps+\mathrm{Rot}(k,\hat{d})) \in \tilde{D}$ **then**
        continue
      $[\![\mathbf{c}]\!] \leftarrow$ pop the first element from E
      $[\![\mathbf{y}_{qs:(q+1)s}]\!] \overset{\boxplus}{\leftarrow} [\![\mathbf{c}]\!] \boxtimes ([\![\mathbf{x}_{ps:(p+1)s}]\!] \lll \hat{d})$
return $[\![\mathbf{y}]\!]$

examine its security characteristics (Section 5.3) and analyze the complexity, including optimality (Section 5.4).

5.1 Processing Each Low-Diagonal Matrix

After we complete the LDD for any sparse matrix, the remaining part is to efficiently multiply a low-diagonal matrix with a vector. Specifically, Algorithms 3 and 4 show our modified diagonal order method. In the descriptions, $[\![\mathbf{u}]\!] \overset{\boxplus}{\leftarrow} [\![\mathbf{v}]\!]$ means $[\![\mathbf{u}]\!] \leftarrow [\![\mathbf{u}]\!] \boxplus [\![\mathbf{v}]\!]$.

We use three nested loops to enumerate all the elements of $\mathbf{A}$. The first loop on q enumerates the row batch, so that the following only considers $\mathbf{A}_{j,i}$ for $\lfloor j/s \rfloor = q$. Similarly, the second loop over p is for the column batch, i.e. only considering $\mathbf{A}_{j,i}$ for $\lfloor i/s \rfloor = p$. The final loop on $\hat{d}$ enumerates the diagonal offset within the batch, i.e. only considering $\mathrm{Rot}(i,\hat{d}) = j$, where $\mathrm{Rot}(a,b)$ denotes the target position if an element at position a is rotated by an offset b.

Compared to the original diagonal order method [23], DiagSpMV needs an additional set $\tilde{D}$ as an parameter, which must satisfy $\mathrm{DS}(\mathbf{A}) \subseteq \tilde{D}$. The only modification lies in the if-statement, which determines whether any elements of $\mathbf{A}$ currently under consideration reside on any diagonal specified by $\tilde{D}$. If so, DiagEncryptMatrix encrypts these elements in a ciphertext. Otherwise, this ciphertext is skipped because we are sure that it contains only zeros based on the knowledge of $\mathrm{DS}(\mathbf{A}) \subseteq \tilde{D}$. Following this, DiagSpMV examines the data in the same sequence to complete the SpMV calculation.

Algorithm 5 LodiaEncryptMatrix

Input $\mathbf{A}_{n\times n}$ with m non-zero elements.
Output An encrypted result, to be sent to LodiaSpMV.

$\mathbf{A}^{(0)}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(k-1)} \leftarrow \mathrm{FindLDD}(\mathbf{A})$ (Algorithm 2)
$\tilde{m} \leftarrow 2^{\lceil \log_2(n+m) \rceil}$
$E \leftarrow$ an empty list
for all $l \in \{k-1, k-2, \ldots, 0\}$ **do**
  $\tilde{D}^{(l)} \leftarrow \mathrm{BDS2DS}(F(l,\tilde{m}))$ // *where* $F(l,\tilde{m})$ *is defined in Theorem 1*
  Concatenate $\mathrm{DiagEncryptMatrix}(\mathbf{A}^{(l)}, \tilde{D}^{(l)})$ to E
return $((n,\tilde{m}), E)$

Algorithm 6 LodiaSpMV

Input $((n,\tilde{m}), E)$ (output of LodiaEncryptMatrix); $[\![\mathbf{x}_{n\times 1}]\!]$.
Output $[\![\mathbf{y}_{n\times 1}]\!]$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is the parameter sent to LodiaEncryptMatrix.

$k \leftarrow 4\log_2 \tilde{m}$
$[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{x}]\!]$
for all $l \in \{k-1, k-2, \ldots, 0\}$ **do**
  $\tilde{D}^{(l)} \leftarrow \mathrm{BDS2DS}(F(l,\tilde{m}))$
  $[\![\mathbf{y}]\!] \leftarrow \mathrm{DiagSpMV}(E, \tilde{D}^{(l)}, [\![\mathbf{y}]\!])$ // *E is modified to be shorter*
return $[\![\mathbf{y}]\!]$

The above (modified) diagonal order method performs efficiently for low-diagonal matrices. Specifically,

Theorem 5. DiagEncryptMatrix *produces* $O(|\tilde{\mathrm{D}}| \cdot n/s)$ *FHE ciphertexts, and* DiagSpMV *requires* $O(|\tilde{\mathrm{D}}| \cdot n/s)$ *FHE operations.*

Proof. Consider the number of $(q,p,\hat{d})$ triples that cause the condition of if-statement to be evaluated as false (i.e. the loop body is not skipped), for each $d \in \tilde{D}$. The sum gives an upper bound of the number of FHE operations.

For the condition to be false, each $d \in \tilde{D}$ gives at most two choices of $\hat{d}$ (e.g. $\hat{d} = d \bmod s$ or $\hat{d} = (d + s/2) \bmod s$ for BFV/BGV-like rotation) and $O(n/s)$ choices of (q,p). Thus, for $|\tilde{D}|$ different values of d, at most $O(|\tilde{D}| \cdot n/s)$ triples of $(q,p,\hat{d})$ cause the if-statement to be false, each corresponding to $O(1)$ FHE ciphertexts and $O(1)$ FHE operations. Consequently, the total number of FHE ciphertexts and arithmetic operations is $O(|\tilde{D}| \cdot n/s)$. □

Thus, if $|\tilde{D}|$ is bounded by a small constant (e.g. 3), DiagSpMV only requires $O(n/s)$ FHE operations. In such cases, it achieves the optimal complexity, as even constructing the resulting $\lceil n/s \rceil$ ciphertexts would generally require at least n/s operations.[8]

Remarks. Although the above mainly assumes that $\mathbf{A}$ is a square matrix, this can be similarly applied to non-square matrices, which is also used as a building block of Lodia.

5.2 Full Algorithm

Now we can give the full algorithm of Lodia. The core idea is rather straightforward: the matrix provider decomposes $\mathbf{A}$ into multiple low-diagonal matrices and encrypts each with DiagEncryptMatrix, while the server only needs to feed each encrypted low-diagonal

[8]Note that the demonstration of Algorithms 3 and 4 is designed to be simple for explanatory purposes, but the actual implementation can bypass redundant loops and only require $O(|\tilde{D}| \cdot n)$ plaintext computation, instead of $O(n^2)$.

matrix into DiagSpMV. Observe that DiagEncryptMatrix requires the parameter $\tilde{D}$ as a superset of the diagonal set, while Theorem 1 only offers information about the bit-diff set, necessitating a transformation. For this, we define:

Definition 4. We define the conversion from a bit-diff set to a diagonal set as $\mathrm{BDS2DS}(B) = \{\sum_{x\in S} x \mid S \subseteq \bigcup_{b\in B}\{2^b, -2^b\}\}$.

This makes sense because we can prove the following:

Theorem 6. $\mathrm{BDS}(\mathbf{M}) \subseteq B \Rightarrow \mathrm{DS}(\mathbf{M}) \subseteq \mathrm{BDS2DS}(B)$.

Proof. Assume $\mathrm{BDS}(\mathbf{M}) \subseteq B$. We have $\mathrm{DS}(\mathbf{M})$

$$
\begin{aligned}
&= \{d \mid \exists (j,i)(\mathbf{M}_{j,i} \neq 0 \wedge j - i = d)\} \\
&= \bigcup_{(j,i):\mathbf{M}_{j,i}\neq 0} \{j-i\} \\
&\subseteq \bigcup_{(j,i):\mathbf{M}_{j,i}\neq 0} \{\textstyle\sum_{x\in S} x \mid S \subseteq \bigcup_{b:\lfloor (j \text{ xor } i)/2^b \rfloor \bmod 2=1} \{2^b, -2^b\}\} \\
&\subseteq \{\textstyle\sum_{x\in S} x \mid S \subseteq \bigcup_{(j,i):\mathbf{M}_{j,i}\neq 0} \bigcup_{b:\lfloor (j \text{ xor } i)/2^b \rfloor \bmod 2=1} \{2^b, -2^b\}\} \\
&= \{\textstyle\sum_{x\in S} x \mid S \subseteq \bigcup_{b:\exists (j,i)\ \mathbf{M}_{j,i}\neq 0 \wedge \lfloor (j \text{ xor } i)/2^b \rfloor \bmod 2=1} \{2^b, -2^b\}\} \\
&= \{\textstyle\sum_{x\in S} x \mid S \subseteq \bigcup_{b\in \mathrm{BDS}(\mathbf{M})} \{2^b, -2^b\}\} \\
&\subseteq \{\textstyle\sum_{x\in S} x \mid S \subseteq \bigcup_{b\in B} \{2^b, -2^b\}\} \\
&= \mathrm{BDS2DS}(B).
\end{aligned}
$$

□

From this, we present the encryption method of Lodia in Algorithm 5, along with the SpMV computation in Algorithm 6. The correctness of Lodia is obvious, given that of LDD and the diagonal order method.

5.3 Security Analysis

This section demonstrates the security characteristics of Lodia. Specifically, we provide proof that Lodia adheres to the security specifications presented in Section 3.1.4.

Theorem 7. *Given a matrix $\mathbf{A}_{n\times n}$ with m non-zero elements, the following information can be computed purely from n and $\tilde{m}$ (i.e. independent of the contents of the matrix), where $\tilde{m} = 2^{\lceil \log_2(n+m)\rceil}$: (1) the plaintext part of* LodiaEncryptMatrix($\mathbf{A}$)*, and (2) the number of ciphertexts produced by* LodiaEncryptMatrix($\mathbf{A}$).

Proof. (1) is trivial because the plaintext information produced by LodiaEncryptMatrix($\mathbf{A}$) is exactly $(n, \tilde{m})$.

For (2), the number of ciphertexts is contingent upon the frequency at which the if-statement within DiagSpMV evaluates to false (e.g. the loop body is executed), which is determined solely by n and $\tilde{D}$. Moreover, each $\tilde{D}$ is entirely dependent on n and $\tilde{m}$. As a result, the overall number of generated ciphertexts depends exclusively on n and $\tilde{m}$. □

No other information leakage is required for the server to be able to perform LodiaSpMV, because all the operations the server performs are on encrypted data in fixed patterns that do not depend on the content of $\mathbf{A}$ beyond size. By using a fixed computation circuit dependent only on n and $\tilde{m}$, we ensure that no information about which positions are non-zero is leaked; any two matrices of the same size and non-zero count will follow the exact same homomorphic operation sequence. This is a critical security feature that distinguishes Lodia from a naive, sparse implementation that would leak index information.

This security property can help develop a simulation-based proof when Lodia is integrated into a full multi-party protocol. See Appendix B for a specific example. Note that Lodia is indeed stronger than required in Section 3.1.4, because plaintext data consist only of n and $\tilde{m}$, so m cannot be precisely determined.

5.4 Complexity and Optimality

Now we analyze the complexity and the optimality of Lodia.

Theorem 8. *Given a matrix $\mathbf{A}_{n\times n}$ with m non-zero elements,* LodiaEncryptMatrix *generates $O((n+m)\log(n+m)/s)$ FHE ciphertexts, and* LodiaSpMV *requires $O((n+m)\log(n+m)/s)$ operations.*

Proof. First, we can obtain $|B| \leq 1 \Rightarrow |\mathrm{BDS2DS}(B)| \leq 3$. This can be proved by verifying $B = \emptyset \Rightarrow \mathrm{BDS2DS} = \{0\}$ and $B = \{b\} \Rightarrow \mathrm{BDS2DS} = \{0, 2^b, -2^b\}$.

Then, consider each $l \in [k]$ where $k = 4\log_2 \tilde{m}$. Theorem 1 gives $|F(l, \tilde{m})| \leq 1$, and thus $|\tilde{D}^{(l)}| = |\mathrm{BDS2DS}(F(l, \tilde{m}))| \leq 3$. Together with Theorem 5, each call to DiagEncryptMatrix produces $O(|\tilde{\mathrm{D}}| \cdot (n+m)/s) = O((n+m)/s)$ FHE ciphertexts and each call to DiagSpMV requires $O((n+m)/s)$ FHE operations. (Note that each matrix sent to DiagSpMV contains up to $\tilde{m} = O(n+m)$ rows/columns.) Thus, the total complexity over $k = O(\log(n+m))$ calls is $O((n+m)\log(n+m)/s)$. □

We claim that Lodia achieves the optimal complexity for a generic SpMV solution, for any matrix that is neither too sparse nor too dense ($m = \Omega(n)$ and $m = O(n^\rho)$ for some $\rho < 2$).[9] Specifically,

Theorem 9. *For a **generic SpMV algorithm for batched FHE schemes** (Section 3.1), if there exists $\rho < 2$ such that $m = O(n^\rho)$,* SpMV *requires a lower bound of $\Omega(m \log n/s)$ FHE operations.*

Since space is constrained, we present an outlined proof here, omitting the extensive details necessary for a full formal proof. In short, we assume a batched FHE scheme over $\mathbb{Z}_2$ and analyze the lower bound of the number of ciphertexts to encode a matrix given n and m. This would give a lower bound to the FHE operations required to process such ciphertexts. The full proof is available in our technical report [45].

Proof (outline). As the investigated SpMV algorithm does not assume a specific scheme, we assume a batched FHE scheme over the ring $(\mathbb{Z}_2, +, \times)$.[10] Let $l(\mathbf{A})$ denote the number of ciphertexts produced by EncryptMatrix($\mathbf{A}$). Consider two different $n \times n$ matrices $\mathbf{A}_1$ and $\mathbf{A}_2$, each with m non-zero elements. We claim that $l(\mathbf{A}_1) = l(\mathbf{A}_2)$; otherwise, the server could distinguish the two different matrices with the same n and m. Thus, we can let $l_{n,m}$ denote $l(\mathbf{A})$ where $\mathbf{A}$ is any $n \times n$ matrix with m non-zero elements.[11]

As the plaintext part of the output of EncryptMatrix, i.e. nothing beyond n and m, contains no useful information to distinguish the $C(n^2, m)$ possible matrices, this information can only be encoded

[9] In real-world scenarios, sparse matrices typically exhibit $m = O(n^\rho)$ for some $\rho < 2$. We justify the assumption $m = \Omega(n)$ as follows: According to the extensive SuiteSparse Matrix Collection [14], all matrices satisfy $m \geq 0.45n$, with 2880 of the 2893 matrices satisfying $m \geq n$. Furthermore, every matrix with at least 524,288 rows and columns also adheres to $m \geq n$. Therefore, assuming $m = \Omega(n)$ is reasonable.

[10] Actually, this proof outline can be applied to any FHE scheme assuming that the plaintext ring has a constant size (not necessarily 2).

[11] If EncryptMatrix produces a random count of ciphertexts, the distributions of $l(\mathbf{A}_1)$ and $l(\mathbf{A}_2)$ should be identical, and we let $l_{n,m}$ denote the minimal possible count.

in the produced ciphertexts. Thus, we have $2^{s \cdot l_{n,m}} \geq C(n^2, m)$ (otherwise, the algorithm would not be correct because it produces ciphertexts corresponding to the same encrypted value given two different matrices, while the vector to multiply can be arbitrary). For $m = O(n^\rho)$ where $\rho < 2$, this means $l_{n,m} = \Omega(m \log n/s)$.

Note that each produced ciphertext should take part in at least one FHE operation, i.e., all generated ciphertext must be useful.[12] Thus, the algorithm requires a lower bound of $\Omega(l(n,m))$ FHE operations to complete SpMV. As $l_{n,m} = \Omega(m \log n/s)$, the lower bound of the number of FHE operations becomes $\Omega(m \log n/s)$. □

For $m = \Omega(n)$ and $m = O(n^\rho)$, we have $\Theta((n+m)\log(n+m)/s) = \Theta(m \log n/s)$. Thus, Lodia achieves the optimal asymptotic complexity, given a fixed s. Note that Lodia leaks even less information than the assumption of n and m, because m cannot be uniquely determined based n and $\tilde{m} = 2^{\lceil \log_2(n+m) \rceil}$.

Remarks on the complexity of plaintext processing. The plaintext part of Lodia is insignificant compared to ciphertext operations. Theoretically, coarse decomposition requires $\Theta(n+m)$ computation (following Step I in proof of Theorem 2, Section 4.2), and fine decomposition requires $\Theta(\tilde{m} \log \tilde{m}) = \Theta((n+m)\log(n+m))$ (outer loop is linear and inner loop is logarithm in Algorithm 1, Section 4.3). Practically, this is insignificant compared to ciphertext operations, e.g., decomposing the largest matrix with $n + m = 2^{27}$ only requires a few seconds.

6 Lodia with Optimized Depth

Although the above achieves excellent complexity regarding the number of ciphertexts and arithmetic operations, it may require a large multiplicative depth. Since each DiagSpMV call consumes one depth, the total multiplicative depth is $k = 4 \log_2 \tilde{m}$. This could practically introduce additional overhead concerning the depth, as the depth impacts the cost per operation in current FHE schemes.

Thus, to practically improve speed performance, we can trade off between the amount of work and the depth. Our intuition is to merge back a consecutive range of decomposed matrices so that the decomposition consists of fewer matrices, saving the multiplication depth. This is advantageous because the amount of work might not increase rapidly. As an example, if we know that $\mathsf{BDS}(\mathbf{A}^{(i)}) \subseteq \{1\}$ and $\mathsf{BDS}(\mathbf{A}^{(i+1)}) \subseteq \{2\}$, we have $\mathsf{BDS}(\mathbf{A}^{(i)}\mathbf{A}^{(i+1)}) \subseteq \{1,2\}$ (see Theorem 10 below). If we call DiagSpMV with (encrypted) $\mathbf{A}^{(i+1)}$ then with $\mathbf{A}^{(i)}$, we consume two depths and need $3+3=6$ units of FHE operations (as $|\mathsf{BDS2DS}(\{1\})| = |\mathsf{BDS2DS}(\{2\})| = 3$). On the other hand, if we send the product $\mathbf{A}^{(i)}\mathbf{A}^{(i+1)}$ to DiagSpMV, we consume only one depth and need 7 units of operations (as $|\mathsf{BDS2DS}(\{1,2\})| = 7$). For this example, we can reduce the depth by half, only with 1/6 additional computation.

[12] If not all ciphertexts are useful, we can instead inspect the "optimized" version without such "useless" ciphertext. As long as the optimized version follows this lower bound, the original version should also follow. Details: The plaintext produced by EncryptMatrix and any other public information is the only information source that the server can utilize to decide whether to completely ignore a certain ciphertext, which is also known by the matrix provider. Thus, we can move this decision to EncryptMatrix (note that this decision is uniform for all $C(n^2, m)$ possible matrices), so that it would not produce such a "totally useless" ciphertext. Afterwards, we continue to analyze this version so that even the optimized version follows this lower bound. Thus, the original unoptimized version should also follow.

6.1 Design of Depth-Optimized Lodia

To extend the previous instance so that this optimization can be utilized effectively for any input matrix, we need the following two steps: (1) determine which decomposed matrices should be combined again, and (2) estimate (a superset of) the diagonal set of each combined matrix, serving as the parameter $\tilde{D}$ passed to the modified diagonal order approach. We will first introduce (2), since the cost model of (1) depends on the approach of (2).

6.1.1 Diagonal Set of Combined Matrices. In general, the bit-diff set for the product of two matrices can be estimated by combining the two bit-diff supersets. Formally,

Theorem 10. *If* $\mathsf{BDS}(\mathbf{A}^{(1)}) \subseteq B^{(1)}$ *and* $\mathsf{BDS}(\mathbf{A}^{(2)}) \subseteq B^{(2)}$, *we have* $\mathsf{BDS}(\mathbf{A}^{(1)}\mathbf{A}^{(2)}) \subseteq B^{(1)} \cup B^{(2)}$.

Proof. $\mathsf{BDS}(\mathbf{A}^{(1)}\mathbf{A}^{(2)})$

$=\{b \mid \exists(j,i)\ (\mathbf{A}^{(1)}\mathbf{A}^{(2)})_{j,i} \neq 0 \wedge \lfloor (j \text{ xor } i)/2^b \rfloor \bmod 2 = 1\}$

$=\{b \mid \exists(j,k,i)\ \mathbf{A}^{(1)}_{j,k} \neq 0 \wedge \mathbf{A}^{(2)}_{k,i} \neq 0 \wedge \lfloor (j \text{ xor } i)/2^b \rfloor \bmod 2 = 1\}$

$\subseteq\{b \mid \exists(j,k,i)\ \mathbf{A}^{(1)}_{j,k} \neq 0 \wedge \mathbf{A}^{(2)}_{k,i} \neq 0$
$\wedge (\lfloor (j \text{ xor } k)/2^b \rfloor \bmod 2 = 1 \vee \lfloor (k \text{ xor } i)/2^b \rfloor \bmod 2 = 1)\}$

$\subseteq\{b \mid \exists(j,k,i)\ (\mathbf{A}^{(1)}_{j,k} \neq 0 \wedge \lfloor (j \text{ xor } k)/2^b \rfloor \bmod 2 = 1)$
$\vee (\mathbf{A}^{(2)}_{k,i} \neq 0 \wedge \lfloor (k \text{ xor } i)/2^b \rfloor \bmod 2 = 1)\}$

$\subseteq\{b \mid \exists(j,k)\ \mathbf{A}^{(1)}_{j,k} \neq 0 \wedge \lfloor (j \text{ xor } k)/2^b \rfloor \bmod 2 = 1)\}$
$\cup \{b \mid \exists(k,i)\ \mathbf{A}^{(2)}_{k,i} \neq 0 \wedge \lfloor (k \text{ xor } i)/2^b \rfloor \bmod 2 = 1\}$

$\subseteq \mathsf{BDS}(\mathbf{A}^{(1)}) \cup \mathsf{BDS}(\mathbf{A}^{(2)})$

$\subseteq B^{(1)} \cup B^{(2)}$. □

Based on this, we can estimate a superset of the diagonal set of a combined matrix. For example, consider combining the matrices $\mathbf{A}^{(l)}, \mathbf{A}^{(l+1)}, \ldots, \mathbf{A}^{(r)}$. Given that $\mathsf{BDS}(\mathbf{A}^{(i)}) \subseteq F(i,\tilde{m})$ holds for any i (as stated in Theorem 1), it follows that $\mathsf{BDS}(\mathbf{A}^{(l)}\mathbf{A}^{(l+1)}\cdots\mathbf{A}^{(r)}) \subseteq \bigcup_{i=l}^{r} F(i,\tilde{m})$. Thus, $\mathsf{DS}(\mathbf{A}^{(l)}\mathbf{A}^{(l+1)}\cdots\mathbf{A}^{(r)}) \subseteq \mathsf{BDS2DS}(\bigcup_{i=l}^{r} F(i,\tilde{m}))$, so that we can pass $\mathsf{BDS2DS}(\bigcup_{i=l}^{r} F(i,\tilde{m}))$ as the parameter $\tilde{D}$ to the modified diagonal order method.

6.1.2 Determine the Combining Strategy. The approach for determining which matrices should be merged is derived from the cost model outlined in the preceding section. For example, when the encrypted form of $\mathbf{A}^{(l)}\mathbf{A}^{(l+1)}\cdots\mathbf{A}^{(r)}$ is sent to DiagSpMV, it requires $|\mathsf{BDS2DS}(\bigcup_{i=l}^{r} F(i,\tilde{m}))|$ units of FHE operations.

We assume that a *depth budget* d is given, which means that Lodia is allowed to consume d multiplicative depths, i.e., we need to rewrite $\mathbf{A} = \mathbf{A}^{(0)}\mathbf{A}^{(1)}\cdots\mathbf{A}^{(k-1)}$ as the product of d matrices. The method of choosing this d in practice is given in Section 6.2.3. Given a specific d, we adopt dynamic programming to find the optimal merging plan, which is generalized from [23] that targets a permutation matrix. The minimal amount of work if we merge the first $\hat{k}$ decomposed matrices (i.e. $\mathbf{A}^{(0)}$ to $\mathbf{A}^{(\hat{k}-1)}$) to fit in a depth

Algorithm 7 FindCombineStrategy

Input $\tilde{m}$; the depth budget d.
Output An array C of $d+1$ elements, with $C[0]=0$ and $C[d]=4\log_2 \tilde{m}$, meaning that the matrices $\mathbf{A}^{(C[i])}, \mathbf{A}^{(C[i]+1)}, \ldots, \mathbf{A}^{(C[i+1]-1)}$ should be merged together, for each $i \in [d]$.

$k \leftarrow 4\log_2 \tilde{m}$
for all $\hat{k} \in [k+1]$ **do**
  for all $\hat{d} \in [d+1]$ **do**
    if $\hat{k}=0 \wedge \hat{d}=0$ **then**
      $f(\hat{k},\hat{d}) \leftarrow 0$
      $G(\hat{k},\hat{d}) = \{0\}$
    else if $\hat{d}=0$ **then**
      $f(\hat{k},\hat{d}) \leftarrow \infty$
    else
      $f(\hat{k},\hat{d}) \leftarrow \min_{k' \in [\hat{k}+1]} \left\{ f(k',\hat{d}-1) + \left| \mathsf{BDS2DS}\left(\bigcup_{j=k'}^{\hat{k}-1} F(j,\tilde{m})\right)\right| \right\}$
      $k' \leftarrow \operatorname{argmin}_{k' \in [\hat{k}+1]} \left\{ f(k',\hat{d}-1) + \left| \mathsf{BDS2DS}\left(\bigcup_{j=k'}^{\hat{k}-1} F(j,\tilde{m})\right)\right| \right\}$
    // In case of a tie, choose the smallest k' that minimizes the expression
      $G(\hat{k},\hat{d}) = \mathsf{Concat}(G(k',\hat{d}-1), \{k'\})$
$C \leftarrow \mathsf{Concat}(G(k,d), \{k\})$
return C

Algorithm 8 LodiaOptEncryptMatrix

Input $\mathbf{A}_{n\times n}$ with m non-zero elements; the depth budget d.
Output An encrypted result, to be sent to LodiaSpMV.

$\tilde{m} \leftarrow 2^{\lceil \log_2(n+m)\rceil}$
$C \leftarrow \mathsf{FindCombineStrategy}(\tilde{m}, d)$ (Algorithm 7)
$\mathbf{A}^{(0)}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(k-1)} \leftarrow \mathsf{FindLDD}(\mathbf{A})$ (Algorithm 2)
$E \leftarrow$ an empty list
for all $i \in \{d-1, d-2, \ldots, 0\}$ **do**
  $\mathbf{M}^{(i)} \leftarrow \mathbf{A}^{(C[i])}\mathbf{A}^{(C[i]+1)} \cdots \mathbf{A}^{(C[i+1]-1)}$
  $\tilde{D}^{(i)} \leftarrow \mathsf{BDS2DS}\left(\bigcup_{j=C[i]}^{C[i+1]-1} F(j,\tilde{m})\right)$
  Concatenate $\mathsf{DiagEncryptMatrix}(\mathbf{M}^{(i)}, \tilde{D}^{(i)})$ to E
return $((n,\tilde{m}), E)$

budget of $\hat{d}$ can be computed as

$$f(\hat{k},\hat{d}) = \min_{k' \in [\hat{k}+1]} \left\{ f(k',\hat{d}-1) + \left| \mathsf{BDS2DS}\left(\bigcup_{j=k'}^{\hat{k}-1} F(j,\tilde{m})\right)\right| \right\}$$

with $f(0,0)=0$ and $f(\hat{k},0)=\infty$ (for $\hat{k}>0$). Thus, the required value is $f(k,d)$, and the strategy can be determined by identifying which k' provides the minimum value.

Algorithm 7 illustrates the implementation of this dynamic programming method, providing an exact strategy for combining the d resulting matrices. The complexity of deriving this strategy is only $\Theta(k^2 d) = \Theta(\log^3(n+m))$ (note that $d \le k = 4\log\tilde{m}$), which is insignificant compared to the other computational parts. Also note that the combination strategy does not depend on the specific content of the input matrix $\mathbf{A}$ or any decomposed matrix $\mathbf{A}^{(\cdot)}$. Instead, it can be determined exclusively by $\tilde{m}$ and d.

6.1.3 Putting Together. Using the aforementioned combination technique from dynamic programming, we introduce the depth-optimized Lodia encryption method in Algorithm 8, along with the corresponding SpMV method in Algorithm 9. In

Algorithm 9 LodiaOptSpMV

Input $((n,\tilde{m}),E)$ (output of LodiaEncryptMatrix); $[\![\mathbf{x}_{n\times 1}]\!]$; d.
Output $[\![\mathbf{y}_{n\times 1}]\!]$ such that $\mathbf{y}=\mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is the parameter sent to LodiaOptEncryptMatrix.

$C \leftarrow \mathsf{FindCombineStrategy}(\tilde{m}, d)$ (Algorithm 7)
$[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{x}]\!]$
for all $l \in \{d-1, d-2, \ldots, 0\}$ **do**
  $\tilde{D}^{(i)} \leftarrow \mathsf{BDS2DS}\left(\bigcup_{j=C[i]}^{C[i+1]-1} F(j,\tilde{m})\right)$
  $[\![\mathbf{y}]\!] \leftarrow \mathsf{DiagSpMV}(E, \tilde{D}^{(i)}, [\![\mathbf{y}]\!])$ *// E is modified to be shorter*
return $[\![\mathbf{y}]\!]$

LodiaOptEncryptMatirx, the matrix owner computes the combining strategy, applies it to produce d merged matrices, and feeds each of them to the modified diagonal order, where the diagonal set is estimated as described in Section 6.1.1. Within LodiaOptSpMV, the computation server can similarly reconstruct the combining strategy and estimate the diagonal sets, allowing execution of the diagonal order method with the corresponding parameters.

6.2 Analysis

We now conduct a brief analysis of Lodia after this optimization.

6.2.1 Security. The security analysis of the depth-optimized Lodia is mostly similar to that of the core algorithm of Lodia presented in Section 5.3. For example, similar to Theorem 7, the security attributes of the optimized version can be demonstrated as follows.

Theorem 11. *Given a matrix $\mathbf{A}_{n\times n}$ with m non-zero elements, the following information can be computed purely from n, $\tilde{m}$ and the depth budget d, where $\tilde{m} = 2^{\lceil \log_2(n+m)\rceil}$: (1) the plaintext part of* LodiaOptEncryptMatrix($\mathbf{A}$)*, and (2) the number of ciphertexts produced by* LodiaOptEncryptMatrix($\mathbf{A}$).

Proof. (1) is trivial. For (2), we can employ the same proof strategy as in Theorem 7, because each $\tilde{D}$ given to DiagEncryptMatrix remains exclusively determined by n, $\tilde{m}$, and d. □

Consequently, we are able to provide simulation-based security proofs when the depth-optimized Lodia is incorporated into two-party protocols (see Appendix B for details).

6.2.2 Complexity. We claim that Lodia retains the optimal complexity of $O((n+m)\log(n+m)/s)$ with a depth budget of $d = \Theta(\log(n+m))$. The proof is provided in our technical report [45].

6.2.3 Choosing Depth Budget d. Although the above provides an asymptotically optimal selection, determining the precise depth budget d when dealing with a matrix in practice remains necessary. We are not providing a universal strategy for selecting the optimal d, because of various optimization goals (a parameter that runs quickly may result in a larger encrypted size) and differences between encryption schemes (e.g., for the same matrix, the fastest depth budget d can be different for BFV and for CKKS).

However, in practice, users can accurately calculate the encryption size of Lodia for any given d, and anticipate the execution time of SpMV by considering the time taken for each FHE operation and the number of operations that Lodia requires for a chosen depth budget d. This can be performed rapidly, without the need for any

Table 1: Number of slots s, generated by OpenFHE

Implementation		Baselines	Lodia					
			$d = 5$	9	13	17	21	25
Batch size s	BFV[1]	8192	16384		32768		65536	
	BGV[1]	8192	16384	32768	65536			
	CKKS[2]	8192	16384		32768			65536

[1]Ring dimension equals s. [2]Ring dimension equals $2s$.

ciphertext computation, encryption, or even examining the matrix, since it relies only on n, m, and d. Consequently, one can quickly determine the encryption size and the SpMV execution time for every d, allowing the selection of a suitable option.

7 Evaluation

This section presents the experimental results. Following a description of the setup (Section 7.1), key comparative results with baseline methods are detailed (Section 7.2). Finally, we assess the effectiveness of Lodia in matrices with different densities (Section 7.3).

7.1 Experimental Setup

7.1.1 Compared Solutions. The solutions we aim to compare include *dense baseline* and *sparse leaky baseline*. In particular, we select the baby-step-giant-step (BSGS) method [24].[13] The *sparse leaky baseline* ports the plaintext SpMV algorithm, which does not exploit batching capabilities, necessitates $\Theta(n+m)$ FHE operations, and discloses the positions of non-zero elements to the server.[14] For Lodia, the depth budget d is selected to achieve the optimal SpMV execution speed, as in Section 6.2.3.[15]

7.1.2 Environment. The experiments were carried out on a server equipped with 2× Intel Xeon Platinum 8380 CPUs, which collectively offer 80 cores. In addition to a RAM capacity of 1 TB DDR4-3200, an array comprising 23× Intel Optane SSD DC P5800X 1.6 TB drives is utilized to increase the scale of data processing; otherwise, baseline solutions are limited to processing only small matrices. To guarantee an equitable comparison, every experiment is required to interact with external disks, even if the data could be accommodated within the RAM. For software, we employ OpenFHE v1.2.3 as the FHE library to facilitate BFV, BGV, and CKKS schemes.

7.1.3 Encryption Parameters. To ensure an unbiased evaluation, most parameters follow the default settings suggested by OpenFHE. We only specify the multiplicative depth necessary for the SpMV algorithm, i.e., 1 for the baselines and d for Lodia with a depth budget of d. The plaintext modulus is set to 786433 for BFV and BGV. Following this, OpenFHE's parameter finder assumes a 60-bit length for each prime modulus and estimates the minimal ring dimension required to maintain 128-bit security. Specifically, Table 1 presents the batch size s for each algorithm with each FHE scheme.[16] We verify the correctness to ensure that the results are exact for BFV or BGV, and achieve at least 30-bit precision for CKKS.

7.2 Performance Comparison with Baselines

Now we present the main comparison results of Lodia against the baseline methods. We evaluated the three approaches using matrices of varying sizes, recording the encryption size and the SpMV execution time. The average density m/n is set at 15, representing a typical case for sparse matrices.[17] All matrices are randomly generated, as their contents do not affect the number of FHE operations for the three methods being compared. Although this part primarily displays results for the BFV scheme, experiments were also conducted for the BGV and CKKS schemes, which demonstrated similar trends (see our technical report [45] for detailed plots).

The results are presented in Figure 2. In instances where a baseline cannot complete execution due to excessive space requirements that exceed available disk capacity, we predict execution time by extrapolating from smaller datasets (using a linear approach for *sparse leaky baseline* and a quadratic approach for *dense baseline*). This method is justified because the results for smaller datasets align well with this trend. Note that the size of the encryption can still be accurately computed, even if the execution does not succeed. Both the encryption size (Figure 2a) and the execution time (Figure 2b) exhibit analogous patterns, since the computational complexity with respect to time and space is identical for any method.

7.2.1 Comparison to Dense Baseline. Given that Lodia presents better complexity, the space-saving and speed-up rates of Lodia compared to *dense baseline* can increase consistently for larger datasets. At scale 23, where *dense baseline* can complete execution, Lodia achieves a speedup of 8.4× and needs only 6.3% of the encrypted size. For the maximum scale 27, Lodia requires merely 0.55% encrypted space, with a predicted speedup of 96×. Note that in very small instances, *dense baseline* might be faster than Lodia since Lodia incorporates a larger constant factor behind its complexity. However, for matrix scales of at least 20, Lodia outperforms this baseline in terms of encryption size and execution time.

Remarks. For $m/n = 7$ instead of 15, Lodia can achieve a better speedup (up to 380× [45]). This is still typical since the median ratio m/n in the SuiteSparse Matrix Collection [14] is 7.9.

7.2.2 Comparison to Sparse Leaky Baseline. Compared to this baseline, Lodia consistently shows advantages in terms of both encryption size and computation time in all scenarios evaluated. Note that the benefits of Lodia can decrease with larger datasets, as *sparse leaky baseline* scales linearly, while Lodia is superlinear. Despite this, even in the most extensive test case involving more than 8 million rows/columns with more than 125 million non-zero elements,

[13]Refer to our technical report [45] for detailed experimental results, which also cover less optimal dense implementations such as row-order and column-order. Certain dense methods in Section 2.3 are excluded from the comparison due to their focus on different contexts: some [25] require the server to know the plaintext matrix, while others [2, 10, 26, 34, 35] focus on matrix-matrix multiplication instead of MVM. Also, for a large square matrix ($n \geq s$), some studies [3, 26, 28] are equivalent to the diagonal order with BSGS.

[14]Note that addressing this privacy issue with a $\Theta(nm/s)$ approach would be inefficient and worse than *dense baseline*. In this study, we exclude the scenarios where the arrangement of non-zero elements follows a specific pattern that could make *sparse leaky baseline* advantageous. We focus on the most challenging scenario for this method, where no such special arrangement exists, necessitating the encryption of each of the m non-zero elements as separate ciphertexts.

[15]We ignore any d that leads to $(n+m)/s$ being less than the CPU core count, which would result in inefficiency due to an insufficient number of parallel tasks.

[16]Note that the parameter increase of Lodia would not bring communication overhead: Lodia actually saves communication because of smaller encrypted matrix (Figure 2a). Also, the increase of key size is ∼10 GB even for the largest parameter, which is insignificant compared to matrix encryption size (and the key can be reused).

[17]This is justified because 73.1% of the matrices in the SuiteSparse Matrix Collection [14] satisfy $m/n \leq 15$. Furthermore, Lodia performs better with smaller m/n (Section 7.3), so this selection does not benefit Lodia.

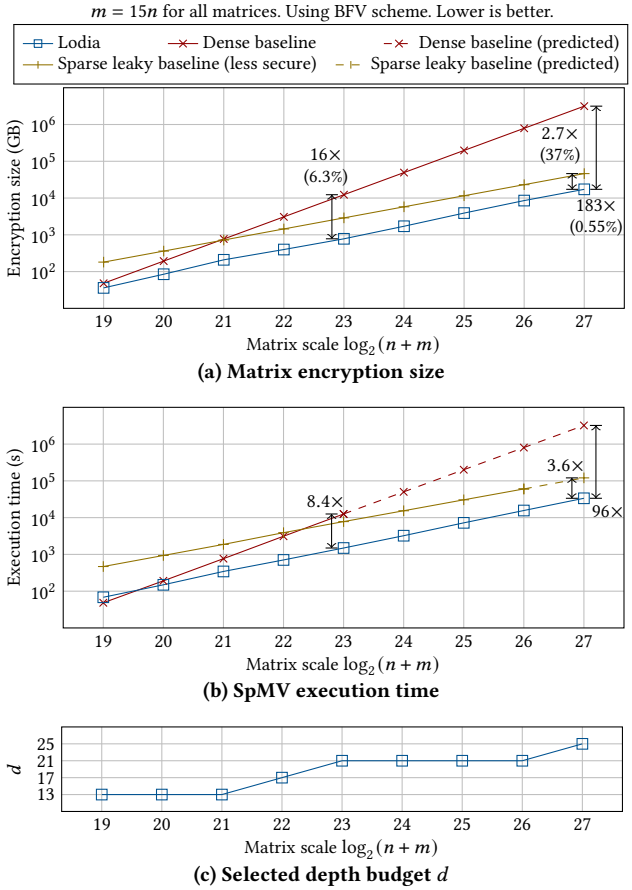


Figure 2: Performance comparison of Lodia and baselines

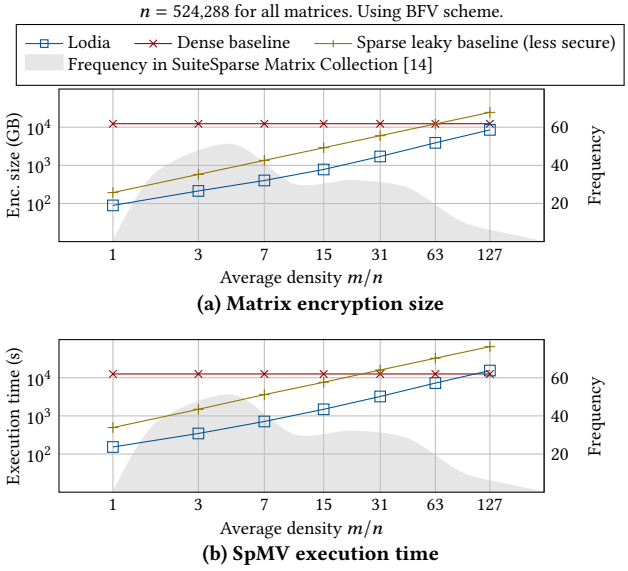


Figure 3: Performance of varying density

Lodia maintains an encryption size of just 37%, and the execution speed for SpMV is enhanced by a factor of 3.6×. Furthermore, this baseline is considerably less secure than Lodia, due to the requirement of disclosing all positions of the non-zero elements to the server, whereas Lodia limits exposure to merely n and $\tilde{m}$.

7.2.3 Selected Depth Budget for Lodia. In Figure 2c, we present the actual depth budget d selected. This is consistent with the understanding that larger matrices require a higher d, with the optimal depth budget approximately on the order of $\Theta(\log(n+m))$. This also demonstrates the efficacy of the optimizations presented in Section 6, since the chosen d is considerably less than $k = 4\lceil\log(n+m)\rceil$, which represents the depth of Lodia without this optimization.

7.3 Impact of Varying Matrix Density

While the preceding section demonstrates the efficacy of Lodia for a representative case of $m/n = 15$, it is also important to assess its performance under different matrix densities. For this purpose, we maintain a constant number of rows $n = 524{,}288$ while varying the average density m/n. We still run the three methods for comparison, continuing to record the encryption and SpMV execution times. Similarly, the experiments are repeated for BGV and CKKS, which reveal similar trends (see our technical report [45] for detailed plots).

The results are illustrated in Figure 3. The performance of *dense baseline* depends solely on n but not on m, resulting in a horizontal line. For the other two methods, the encryption size and execution time increase with larger values of m. Since Lodia consistently surpasses *sparse leaky baseline*, the focus is on comparing Lodia with *dense baseline*. As shown in the figure, Lodia performs better than *dense baseline* when $m/n \leq 63$; they are comparable when $m/n = 127$ (Lodia takes slightly more time but reduces the encryption size), and Lodia is not expected to perform better for $m/n > 127$.

To understand the average density m/n of real-world matrices, we count the number of matrices within each density interval, using data from the widely recognized SuiteSparse Matrix Collection [14], focusing on matrices that have at least 524,288 rows and columns. The frequency distribution is represented in gray. Among the 190 matrices examined, 177 (93.2%) have an average density $m/n \leq 63$, and 187 (98.4%) satisfy $m/n \leq 127$, suggesting that Lodia can improve performance for most large-scale sparse matrices.

8 Conclusion

We propose Lodia, an efficient sparse matrix-vector multiplication (SpMV) algorithm for batched fully homomorphic encryption. The core idea is to decompose any sparse matrix into the product of multiple matrices, each with a low diagonal count, so that we can leverage an efficient SpMV algorithm for low-diagonal matrices. Lodia achieves excellent complexity of $\Theta((n+m)\log(n+m)/s)$ FHE operations, which is optimal for $m = \Omega(n)$ and $m = O(n^\rho)$ for some $\rho < 2$. The evaluation shows that Lodia practically outperforms the baselines by 3.6× ∼ 96×, and is able to process a large matrix with over 8 million rows/columns and 125 million non-zero elements.

Acknowledgments

The authors thank the anonymous reviewers for their valuable feedback on this work, including prior submissions. This work was supported by Ant Group Research Intern Program. The corresponding authors are Kun Chen and Wenguang Chen.

References

[1] 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[2] Youngjin Bae, Jung Hee Cheon, Guillaume Hanrot, Jai Hyun Park, and Damien Stehlé. 2024. Plaintext-Ciphertext Matrix Multiplication and FHE Bootstrapping: Fast and Fused. In *Annual International Cryptology Conference*. Springer, 387–421.

[3] Shashank Balla and Farinaz Koushanfar. 2023. HELiKs: HE Linear Algebra Kernels for Secure Inference. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2306–2320.

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[5] Václav E Beneš. 1965. *Mathematical theory of connecting networks and telephone traffic*. Academic press.

[6] Chris M Bishop. 1994. Neural networks and their applications. *Review of scientific instruments* 65, 6 (1994), 1803–1832.

[7] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual cryptology conference*. Springer, 868–886.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[9] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[10] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. 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 26*. Springer, 31–59.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 409–437.

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

[13] Pierre-Emmanuel Clet, Oana Stan, and Martin Zuber. 2021. BFV, CKKS, TFHE: Which one is the best for a secure neural network evaluation in the cloud?. In *Applied Cryptography and Network Security Workshops: ACNS 2021 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, and SiMLA, Kamakura, Japan, June 21–24, 2021, Proceedings*. Springer, 279–300.

[14] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[15] Christopher De Sa, Albert Cu, Rohan Puttagunta, Christopher Ré, and Atri Rudra. 2018. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1060–1079.

[16] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 617–640.

[17] Frank J Fabozzi, Harry M Markowitz, and Francis Gupta. 2011. Portfolio selection. *The Theory and Practice of Investment Management* (2011), 45–78.

[18] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).

[19] Aidan Ferguson, Perry Gibson, Lara D'Agata, Parker McLeod, Ferhat Yaman, Amitabh Das, Ian Colbert, and José Cano. 2025. Exploiting Unstructured Sparsity in Fully Homomorphic Encrypted DNNs. In *Proceedings of the 5th Workshop on Machine Learning and Systems*. 31–38.

[20] Jianxin Gao and Ying Gao. 2024. GMS: an efficient fully homomorphic encryption scheme for secure outsourced matrix multiplication. *The Journal of Supercomputing* (2024), 1–27.

[21] Yang Gao, Gang Quan, Soamar Homsi, Wujie Wen, and Liqiang Wang. 2024. Secure and efficient general matrix multiplication on cloud using homomorphic encryption. *The Journal of Supercomputing* (2024), 1–41.

[22] Kevin Gurney. 2018. *An introduction to neural networks*. CRC press.

[23] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 554–571.

[24] Shai Halevi and Victor Shoup. 2018. Faster homomorphic linear transformations in HElib. In *Annual International Cryptology Conference*. Springer, 93–120.

[25] Jiaxing He, Kang Yang, Guofeng Tang, Zhangjie Huang, Li Lin, Changzheng Wei, Ying Yan, and Wei Wang. 2024. Rhombus: Fast homomorphic matrix-vector multiplication for secure two-party inference. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2490–2504.

[26] Zhicong Huang, Cheng Hong, Chenkai Weng, Wen-jie Lu, and Hunter Qu. 2021. More efficient secure matrix multiplication for unbalanced recommender systems. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2021), 551–562.

[27] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. 2018. Secure outsourced matrix computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 1209–1222.

[28] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX security symposium (USENIX security 18)*. 1651–1669.

[29] Kasper Green Larsen and Ryan Williams. 2017. Faster online matrix-vector multiplication. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2182–2189.

[30] W Leontief. 1986. Input-Output Economics.

[31] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[32] Dong Liu, Hong-Yu Bai, Hui-Jia Li, and Wen-Jun Wang. 2014. Semi-supervised community detection using label propagation. *International Journal of Modern Physics B* (2014).

[33] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. 2023. Squirrel: A Scalable Secure Two-Party Computation Framework for Training Gradient Boosting Decision Tree. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6435–6451.

[34] Xirong Ma, Chuan Ma, Yali Jiang, and Chunpeng Ge. 2024. Improved privacy-preserving PCA using optimized homomorphic matrix multiplication. *Computers & Security* 138 (2024), 103658.

[35] Pradeep Kumar Mishra. 2018. Faster Secure Matrix Computations over Homomorphic Encryption. (2018).

[36] Pradeep Kumar Mishra, Deevashwer Rathee, Dung Hoang Duong, and Masaya Yasuda. 2021. Fast secure matrix multiplications over ring-based homomorphic encryption. *Information Security Journal: A Global Perspective* 30, 4 (2021), 219–234.

[37] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd, et al. 1999. The pagerank citation ranking: Bringing order to the web. (1999).

[38] Jai Hyun Park. 2025. Ciphertext-ciphertext matrix multiplication: Fast for large matrices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 153–180.

[39] Richard Peng and Santosh Vempala. 2021. Solving sparse linear systems faster than matrix multiplication. In *Proceedings of the 2021 ACM-SIAM symposium on discrete algorithms (SODA)*. SIAM, 504–521.

[40] Panagiotis Rizomiliotis and Aikaterini Triakosia. 2022. On matrix multiplication with homomorphic encryption. In *Proceedings of the 2022 on Cloud Computing Security Workshop*. 53–61.

[41] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

[42] Richard W Vuduc and Hyun-Jin Moon. 2005. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications: First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005. Proceedings 1*. Springer, 807–816.

[43] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. 1–12.

[44] Yuncheng Wu, Shaofeng Cai, Xiaokui Xiao, Gang Chen, and Beng Chin Ooi. 2020. Privacy preserving vertical federated learning for tree-based models. *arXiv preprint arXiv:2008.06170* (2020).

[45] Jiping Yu, Kun Chen, Xiaoyu Fan, Yunyi Chen, Xiaowei Zhu, and Wenguang Chen. 2025. Lodia: Towards Optimal Sparse Matrix-Vector Multiplication for Batched Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2025/1425. https://eprint.iacr.org/2025/1425

[46] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. GALA: Greedy computation for linear algebra in privacy-preserved neural networks. *arXiv preprint arXiv:2105.01827* (2021).

[47] Xiaopeng Zheng, Hongbo Li, and Dingkang Wang. 2023. A New Framework for Fast Homomorphic Matrix Multiplication. *Cryptology ePrint Archive* (2023).

Algorithm 10 PortPlainEncryptMatrix

Input $\mathbf{A}_{n\times n}$ with m non-zero elements.
Output An encrypted result E, to be sent to PortPlainSpMV.

$E \leftarrow$ empty list of ciphertexts
for all (j, i) such that $\mathbf{A}_{j,i} \neq 0$ **do**
  $\mathbf{c} \leftarrow \mathbf{0}_{s\times 1}$
  $\mathbf{c}_{j \bmod s} \leftarrow \mathbf{A}_{j,i}$
  Append $(j, i, \mathsf{Encrypt}(\mathbf{c}))$ to E
return E

Algorithm 11 PortPlainSpMV

Input E (output of PortPlainEncryptMatrix); $[\![\mathbf{x}_{n\times 1}]\!]$.
Output $[\![\mathbf{y}_{n\times 1}]\!]$ such that $\mathbf{y} = \mathbf{A}\mathbf{x}$, where $\mathbf{A}$ is the parameter sent to PortPlainEncryptMatrix.

$[\![\mathbf{y}]\!] \leftarrow [\![\mathbf{0}_{n\times 1}]\!]$
for all $(j, i, [\![\mathbf{c}]\!]) \in E$ **do**
  $[\![\mathbf{y}_{\lfloor j/s\rfloor s:(\lfloor j/s\rfloor+1)s}]\!] \xleftarrow{\boxplus} [\![\mathbf{c}]\!] \boxtimes ([\![\mathbf{x}_{\lfloor i/s\rfloor s:(\lfloor i/s\rfloor+1)s}]\!] \lll \mathsf{Rot}(j, -i))$
return $[\![\mathbf{y}]\!]$

Algorithm 12 An example two-party protocol using Lodia

Roles Two parties, involving a client P_C and a server P_S.
Input P_C provides $\mathbf{A}_{n\times n}$ with m non-zero elements, and $\mathbf{x}_{n\times 1}$.
Output P_C finally learns $\mathbf{y}$, satisfying $\mathbf{y} = \mathbf{A}\mathbf{x}$.
Public information n and $\tilde{m}$, where $\tilde{m} = 2^{\lceil \log_2(n+m)\rceil}$.

P_C encrypts $\mathbf{x}$ into $[\![\mathbf{x}]\!]$, and sends it to P_S
P_C computes $L \leftarrow \mathsf{LodiaEncryptMatrix}(\mathbf{A})$, and sends it to P_S
P_S computes $[\![\mathbf{y}]\!] \leftarrow \mathsf{LodiaSpMV}(L, [\![\mathbf{x}]\!])$, and sends it to P_C
P_C decrypts $[\![\mathbf{y}]\!]$ into $\mathbf{y}$

Algorithm 13 Simulator of P_S of the protocol in Algorithm 12

Input Public information n and $\tilde{m}$.
Output A simulated view of the received messages and generated random numbers by server P_S.

Independently randomly generate a matrix $\mathbf{A}_{n\times n}$ with $(\tilde{m} - n)$ non-zero elements, and a vector $\mathbf{x}_{n\times 1}$
Execute the protocol in Algorithm 12 with P_C inputting $\mathbf{A}$ and $\mathbf{x}$, recording the received messages and generated random numbers by P_S
return anything recorded in the above step

A Details of Sparse Leaky Baseline

For reference, the details of *sparse leaky baseline* are introduced here. Specifically, Algorithm 10 shows the encryption method, while Algorithm 11 demonstrate the SpMV process.

Its intention is to rewrite basic plaintext SpMV algorithm in homomorphic forms. Similar to plaintext algorithm, each non-zero element is still handled with multiplication and addition. However, since $\mathbf{x}_i$ and $\mathbf{y}_j$ may lie in different slots, an additional rotation is needed when operating with batched FHE.

B Security Analysis with an Example Protocol

The security property of Lodia, as outlined in Section 5.3, can help develop a simulation-based proof when Lodia is integrated into a full multi-party protocol. To demonstrate this, we present an example protocol in Algorithm 12 involving a client and a server. Note that this does not suggest that Lodia is restricted to this particular protocol. Rather, this protocol acts as a demonstrative instance and the approach used in this security proof can be applied to other protocols incorporating SpMV as a part of the computation.

In this protocol, the client provides the input data $\mathbf{A}$ and $\mathbf{x}$ and outsources the SpMV task to the server. The client can achieve this by encrypting both $\mathbf{x}$ and $\mathbf{A}$, then sending the encrypted data to the server. Thus, the server is able to execute LodiaSpMV to carry out the SpMV operation. Finally, the client obtains the encrypted result and proceeds to decrypt it.

Now we use Theorem 7 to help develop a simulation-based proof, to prove the security of Algorithm 12.

Theorem 12. *The protocol in Algorithm 12 is secure against semi-honest adversaries.*

Proof. The protocol is trivially secure if the client is corrupted, because the server does not input sensitive data to the protocol. We only need to demonstrate that it is secure when the server is corrupted, by giving a simulator S for P_S satisfying

$$\{S()\}_{\mathbf{A},\mathbf{x}} \overset{c}{\equiv} \{\mathrm{View}_S^{\mathrm{Alg.12}}(\mathbf{A}, \mathbf{x})\}, \tag{2}$$

where $\{\cdot\}$ denotes a probability ensemble, $\overset{c}{\equiv}$ denotes computational indistinguishability (in polynomial time), and $\mathrm{View}_S^{\mathrm{Alg.12}}(\mathbf{A}, \mathbf{x})$ denotes the view (including the received messages and generated random bits) of P_S during an execution of Algorithm 12 with P_C providing $\mathbf{A}$ and $\mathbf{x}$ as input. Note that S is not supplied with the actual input and output in the protocol of P_S, because P_S does not input or output anything. Thus, S can only use public information.

Algorithm 13 shows a possible implementation of S. It just generates random input data $\mathbf{A}$ and $\mathbf{x}$, simulates the execution of the protocol, and records the view generated during this execution.

Theorem 7 states that the plaintext components and the number of ciphertexts between $\{S()\}_{\mathbf{A},\mathbf{x}}$ and $\{\mathrm{View}_S^{\mathrm{Alg.12}}(\mathbf{A}, \mathbf{x})\}$ are required to be (deterministically) identical. The only scenario in which S could contravene Equation 2 is if the ciphertext distributions are computationally distinguishable, which is not possible. Such a situation would provide counterexamples that refute the IND-CPA property of the FHE scheme. Consequently, Equation 2 must hold. □

Note that the same analysis can be applied to depth-optimized Lodia, as long as the depth budget d appears as public information.

C Further Details

Our technical report [45] contains further details about this work, including:

- Proof of Theorem 4, i.e. the correctness of fine decomposition of an OAM.
- Complexity analysis of depth-optimized Lodia.
- Full proof of Theorem 9, i.e. the optimality of Lodia.
- Detailed experimental results, including:
  - Results on BGV and CKKS schemes, in addition to the results reported in the main body on BFV scheme.
  - Results of suboptimal dense methods, namely row order and column order methods.