HELiKs: HE Linear Algebra Kernels for Secure Inference

Shashank Balla
University of California San Diego
San Diego, California, USA
sballa@ucsd.edu

Farinaz Koushanfar
University of California San Diego
San Diego, California, USA
farinaz@ucsd.edu

ABSTRACT

We introduce HELiKs, a groundbreaking framework for fast and secure matrix multiplication and 3D convolutions, tailored for privacy-preserving machine learning. Leveraging Homomorphic Encryption (HE) and Additive Secret Sharing, HELiKs enables secure matrix and vector computations while ensuring end-to-end data privacy for all parties. Key innovations of the proposed framework include an efficient multiply-accumulate (MAC) design that significantly reduces HE error growth, a partial sum accumulation strategy that cuts the number of HE rotations by a logarithmic factor, and a novel matrix encoding that facilitates faster online HE multiplications with one-time pre-computation. Furthermore, HELiKs substantially reduces the number of keys used for HE computation, leading to lower bandwidth usage during the setup phase. In our evaluation, HELiKs shows considerable performance improvements in terms of runtime and communication overheads when compared to existing secure computation methods. With our proof-of-work implementation[1], we demonstrate state-of-the-art performance with up to 32× speedup for matrix multiplication and 27× speedup for 3D convolution when compared to prior art. HELiKs also reduces communication overheads by 1.5× for matrix multiplication and 29× for 3D convolution over prior works, thereby improving the efficiency of data transfer.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; • **Computing methodologies → Machine learning**.

KEYWORDS

Privacy-preserving inference; deep neural networks; secure two-party computation

ACM Reference Format:
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 (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623136

1 INTRODUCTION

The prevalence of data analytics in modern society has been greatly amplified by the era of big data, the ever-increasing computational power and connectivity, and consequently the now ubiquitous cloud computation. This has resulted in increased productivity and informed decision-making for individuals and organizations all over. On the other hand, the popularity of outsourcing data analytics to a cloud server has also sparked severe privacy concerns, as sensitive information can be vulnerable to exposure or misuse. This has moved the industry and the research community at large to push for the adoption of privacy-enhancing technologies. Homomorphic Encryption (HE) is one such technology that allows for computing on encrypted data without requiring any decryption. HE offers true end-to-end encryption where no one but the owner of the data sees the data in the clear. HE has been used to enable several critical privacy-preserving applications like secure genome analysis [3], private information retrieval [32, 35] and private set intersection [11]. HE has also seen commercial use with the password monitor feature for the Microsoft Edge web browser [29].

Most modern data analytics applications involve processing Deep Neural Networks (DNNs), which is much more demanding compared to the aforementioned applications of HE. With the proliferation of cloud-based tools that use inference on trained DNNs for data analytics, a huge community of researchers has been focused on applying HE to enable secure (privacy-preserving) inference on DNNs. A cloud-based DNN inference scenario, also known as Machine Learning-as-a-Service (MLaaS), involves a server holding the trained DNN model and clients sending their data for inference. A HE-based solution enables clients to send encryptions of their data and securely outsource all the DNN computation to the server without revealing their inputs. As linear algebra operations on high-dimensional data are integral to most data analytics procedures, the ability to perform fast and lightweight secure computation of these operations can greatly improve the practicality of a wide range of privacy-preserving applications on today's computers.

In this paper, we present HELiKs, a novel HE-based framework for fast and secure matrix multiplications and 3D convolutions in privacy-preserving DNN inference. The key contributions of HELiKs are:

- A new MAC-core (multiply-accumulate) design and a matrix encoding scheme that involves a one-time offline pre-computation process. The new algorithm cuts the number of NTT operations required for HE multiplications by 66.67%.
- A partial sum accumulation strategy that reduces the number of rotations from $O(n \log n)$ to $O(n)$ and the number of keys used in HE computation from $O(\log n)$ to just $O(1)$. This leads to a substantial reduction in bandwidth usage during the setup phase.
- A novel use of Symmetric-key HE that effectively halves the total amount of encrypted data sent from the client to the server. This also enables us to drop the public key to further lower the bandwidth usage during the setup phase.

[1]Access HELiKs on GitHub: https://github.com/shashankballa/HELiKs

CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623136

- We show a proof-of-work implementation that is faster than prior art by up to 31× for Matrix multiplication and 27× for 3D Convolution, while utilizing 1.4× and 29× lower bandwidth for communication, respectively.

In the rest of the paper, we detail the HELiKs framework and demonstrate its potential for a wide range of applications requiring secure matrix multiplication. The remainder of the paper is organized as follows: Section 2 presents the background on secure computation, homomorphic encryption, and other relevant concepts as well as the related work. Section 3 presents the high-level design of the HELiKs. Section 4 gives the implementation details of secure matrix multiplication in HELiKs and Section 5 explains the implementation process of 3D convolutions in the HELiKs. In Section 6 we evaluate the HELiKs Framework and compare it against prior art. Finally, Section 7 concludes the paper and highlights future directions.

2 BACKGROUND

2.1 Secure Computation and Homomorphic Encryption

Preserving privacy is paramount, especially when handling confidential data in collaborative environments. The realm of cryptography offers secure computation as a solution to this challenge. This field enables multiple parties to carry out calculations on their private data while ensuring the confidentiality of the inputs. Early pioneering protocols, such as Oblivious Transfers (OT) [37], Yao's garbled circuits (GC) [45], and GMW [20], were the cornerstones for secure two-party computation (2PC). Over the years, these seminal protocols have been refined and improved for a range of applications [6, 23, 25, 42]. Modern applications frequently employ a fusion of these traditional methodologies with the latest optimizations, designed to cater to specific computational needs [26, 38, 39]. Notably, the data exchanged between parties during these protocols usually increases in tandem with the total multiplication operations needed. While GC achieves this exchange in a constant number of communication rounds, OT and GMW necessitate a round of communication for every multiplication. A primary challenge with classical 2PC is the near-equal computational effort demanded from both participating parties.

On the other hand, Homomorphic Encryption (HE) is a cutting-edge technique that facilitates operations on encrypted data without the necessity for decryption keys or the original data [18]. In an HE setup, clients encrypt and forward their data to servers. Servers then perform calculations on this encrypted dataset and return the processed data. Unique to HE is its ability to separate communication volume from the computational load. The data exchange is governed only by the dimensions of the input and output. The computational burden predominantly rests with the server, with clients primarily responsible for generating ciphertexts. This involves transforming data into HE plaintexts, encrypting them into ciphertexts, and finally, decrypting and decoding them back into original data. Servers shoulder the weighty task of performing complex operations on these encrypted datasets. Given its distinct advantages, HE emerges as an optimal solution for two-party secure computation in the client-server model. This makes it a compelling alternative to conventional protocols that expect both parties to have comparable computational capacities.

Consider the case of securely computing a (M, N) matrix-vector product. In an OT-based approach, the communication overhead hinges on the product value, $M \times N$, because the computation involves so many multiplications. However, with HE, one only has to communicate the input and output across and the overhead depends only on their sum, $M+N$. Taking the instance of COINN [26], a contemporary OT-driven framework designed for secure inferences, the communication needed for evaluating the linear layers of ResNet50 exceeds 0.5 TB. In contrast, our HE-based methodology necessitates less than 0.5 GB for the same computation.

2.2 Fully Homomorphic Encryption and Leveled Variants

A Fully Homomorphic Encryption (FHE) scheme that allows for arbitrary computations on encrypted data was first presented by Gentry [18]. FHE schemes achieve unlimited computation capacity through a compute-intensive operation referred to as the bootstrapping procedure. TFHE [10] and FHEW [16] are a few popular FHE schemes that are primarily designed around the promise of a fast bootstrapping procedure. These FHE schemes eliminate the dependency of communication on the computation to be performed; however, they introduce significant overhead in computation time because of the bootstrapping procedure.

In contrast, leveled variants of HE schemes propose a different construct with larger HE parameters that enable only a fixed amount of computation without requiring bootstrapping. This results in significantly faster processing at the expense of increased communication arising from a larger ciphertext expansion due to the larger HE parameters. The communication complexity of Leveled HE schemes is contingent upon the multiplicative depth of the computation, that is, the number of nested or sequential multiplications on the encrypted data. This presents a substantial improvement for applications that involve the multiplication of large matrices which involves a large number of scalar multiplications (one for each element) but the total multiplicative depth of the computation is only 1.

2.3 HE from Ring Learning With Errors

Modern HE schemes are based on the Ring-Learning With Errors (RLWE) [31] hard problem that involves finding a small error in a polynomial equation. RLWE-HE schemes operate on polynomial elements, i.e., the plaintexts or ciphertexts in these schemes are either a single polynomial or a vector of 2 polynomials, respectively. The polynomial data structures enable packing a vector of cleartext values into a single plaintext/ciphertext. This enables a SIMD style of execution in HE and gives a significant boost in the throughput of the HE computation [44]. These HE schemes are primarily parameterized by the ring they operate on, $\mathbb{R}_Q^n = \mathbb{Z}_Q[X]/(X^n + 1)$, which is the set of all polynomials of degree less than n with integer coefficients from $\mathbb{Z}_Q$ (finite field of integers modulo Q). Processing on cleartext data that has been encrypted using RLWE-HE schemes requires complex operations on polynomials from $\mathbb{R}_Q^n$ to realize a specific arithmetic operation on the underlying cleartext data. BGV[7], BFV[8, 17] and CKKS[9] are the most popular

RLWE-HE schemes. While BGV and BFV are tailored for integer data and offer exact computation, the CKKS scheme is designed for approximate computation on real data. These schemes are implemented in most of the software libraries for HE [2, 22, 43] which also provide functions that perform the necessary computation on ciphertext/plaintext inputs to realize an addition, Hadamard multiplication (multiplication), and a cyclic permutation (rotation) of the encrypted vectors.

In RLWE-HE schemes, a ciphertext $\overline{\mathbf{ct}}$ is represented as a vector containing two polynomial elements from $\mathbb{R}_Q^n$, denoted as $\overline{\mathbf{ct}} \in [\mathbb{R}_Q^n]^2$. For the BFV and BGV schemes, a plaintext is an element from $\mathbb{R}_P^n$, where P ($< Q$) is the plaintext modulus that dictates the dynamic range of encrypted data. In the CKKS scheme, which is designed for approximate computation, the plaintext is an element from $\mathbb{R}_Q^n$ and does not include a plaintext modulus. The encryption process for a vector of secret values $\mathbf{m}$ involves encoding it into a polynomial to produce a plaintext $\overline{pt}_\mathbf{m}$ and then encrypting this plaintext to obtain a ciphertext $\overline{\mathbf{ct}}_\mathbf{m} = (\overline{a}, \overline{b})$. Here, the first polynomial element $\overline{a}$ is chosen uniformly at random from $\mathbb{R}_Q^n$. The second polynomial element, $\overline{b}$, is computed as $\overline{b} = \overline{a} \cdot \overline{sk} + \delta \cdot \overline{pt}_\mathbf{m} + \epsilon \cdot \overline{e}$, where $\overline{sk}$ is the secret key, $\overline{e}$ is a small error, δ is a scaling factor for the plaintext message, and ϵ is a scaling factor for the error. For the BGV and CKKS schemes, $\delta = 1$, while for the BFV scheme, $\delta = \lfloor Q/P \rceil$. Whereas, $\epsilon = 1$ for the BFV and CKKS schemes and $\epsilon = P$ for the BGV scheme. The method presented in this paper is designed for RLWE-HE schemes and thus, for the remainder of this paper, we will refer to RLWE-HE as simply HE.

2.4 Computing on Encrypted Data with HE

In Figure 1 we show the runtimes of all ciphertext computation operations for the CKKS, BFV and BGV HE schemes in the SEAL library [43] for various choices of the polynomial modulus degree (and no variation of other HE parameters). The numbers depicted in this chart are averages of 1000 trials. Here, *Add plain* and *Multiply plain* are optimized variants of the arithmetic operations when only one of the operands is encrypted and the other is in plaintext. Additions are the fastest operations in HE: the corresponding polynomials in the ciphertexts/plaintexts are added together to obtain the sum of the underlying cleartext data. Multiplications are more complex, involving a convolution operation for multiplying polynomials in the ciphertexts/plaintexts to generate polynomials for the product ciphertext that encrypts the Hadamard product of the cleartext data encrypted in the inputs. It has become common practice to speed up the convolutions for polynomial multiplication with the Number Theoretic Transform (NTT) [30]. NTT is the finite-field equivalent of the Fourier transforms; it follows the same structure of computation but with arithmetic operations and twiddle factors for the respective finite field. NTTs bring the complexity of one polynomial multiplication down from $O(n^2)$ to $O(n \log n)$.

Rotations are the most time-consuming operations, involving a very expensive key-switching procedure that uses a special evaluation key $\overline{\mathbf{ek}}$ to change the key the input ciphertext has been encrypted with in such a way that when the resulting ciphertext is decrypted with the original secret key, it outputs a vector of the same cleartext data but with elements cyclically rotated with

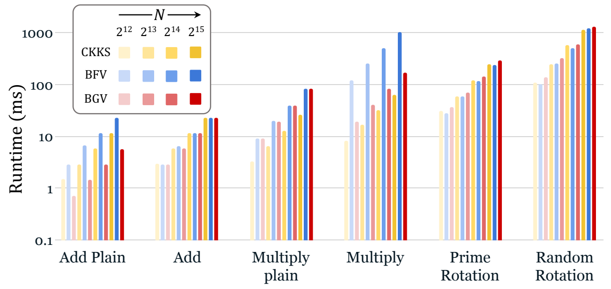


Figure 1: Runtimes for ciphertext computation operations.

the desired shift. These evaluation keys are very large (larger than ciphertexts), and one key only works for a specific rotation. The polynomials that have been standardized [1] to meet 128-bit security are at least of degree $n = 2048$. Such large polynomials, with $n \geq 2048$, can encode vectors with n elements in the case of BFV and BGV, or $n/2$ elements in the case of CKKS. This makes for a huge number of total possible unique shifts (up to n) and an impractical number of evaluation keys to be generated and stored to enable all these rotations in one shot. The aforementioned libraries assuage this issue by only generating $\log n$ evaluation keys for rotations involving power-of-2 shifts, which we will refer to as *prime rotations*. Arbitrary rotations are performed as a composition of multiple *prime rotations*, involving as many *prime rotations* as the hamming weight of the binary decomposition of the desired arbitrary shift amount.

2.5 Error Growth from HE Computation

In RLWE-HE schemes, *error* lies at the heart of security. The encryption process in these schemes perturbs the plaintext polynomial by adding a small random error sampled from a discrete Gaussian distribution. This error coexists with the underlying message in the ciphertext, meaning that any computation on these ciphertexts affects both the underlying data and the error, causing the error to grow. As a result, ciphertexts can only handle a limited amount of error; if the error exceeds a set budget, the underlying data becomes permanently corrupted, leading to decryption failure. The total available error budget is determined by the coefficient modulus Q of the ring $\mathbb{R}_Q^n$, while the rate at which the error grows after a specific operation also depends on the polynomial modulus degree n [36] among other parameters.

Regrettably, the coefficient modulus Q affects the security level of the HE scheme in an inverse manner: a higher Q immediately results in a higher error budget but also significantly lowers the security level. Moreover, higher parameters lead to a larger ciphertext expansion factor due to the use of larger polynomials, resulting in slower operations. Therefore, for a fixed computation, error growth determines the size of the ciphertexts, and accurately estimating error growth helps in choosing the optimal HE parameters for the best performance.

In Figure 2, we show the error growth due to computation in BGV and BFV schemes. The same trends apply to the CKKS scheme as well but since the SEAL library doesn't provide an API for investigating the error in the CKKS scheme we do not show results

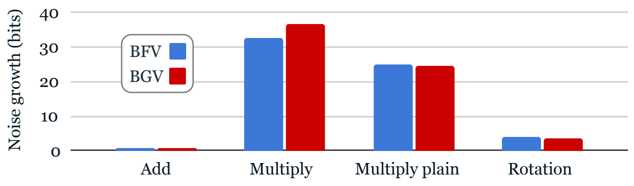


Figure 2: Error growth from computation in BFV and BGV.

for it. We refer the readers to Costache et al. [12] for a detailed discussion on error growth in the CKKS scheme. A freshly encrypted ciphertext has the least amount of error necessary to achieve a predetermined security level. While the addition of a ciphertext with a plaintext is free of error growth, adding two ciphertexts leads to error growth. The error in the sum-ciphertext is the sum of the errors in the input ciphertexts and the error growth is additive e_{add}. Rotation results in more significant error growth than additions, with the error increasing by an additive factor e_{rot}. Multiplication has the highest error growth of all operations, and the error grows by a multiplicative factor e_{mul}. Consequently, careful placement of multiplications in the computation graph is crucial for curbing error growth in HE computation and selecting lower HE parameters.

2.6 Related Work

CyptoNets [15] was the first system for secure neural network inference on data encrypted with HE. They use a variant of HE that is only capable of linear operations and hence the neural network they consider replaced non-linear activation functions with HE-friendly polynomial activations. Moreover, the HE scheme used also did not support packing and the algorithm implemented for the linear operations was very inefficient. Halevi and Shoup [21] first introduced an efficient algorithm for securely computing a matrix product inside a single ciphertext using HE, which primarily consists of two steps. For simplicity, we consider the case of a matrix-vector product where the matrix is locally held by the server as its private input, and the vector is the client's private input. The client encrypts the vector into contiguous slots of an HE ciphertext and sends it to the server for computation. The first step is *diagonal encoding*, where the server extracts the diagonals of its 2D matrix and encodes them into HE plaintexts. These diagonals, extracted from the server's matrix, are *output-aligned*, meaning they begin from the first row of the server's matrix and proceed diagonally downwards. The second step, *iterative MAC* (multiply-accumulate), involves the server initializing an accumulator with a zero ciphertext using the public key. At each iteration, the server rotates the ciphertext containing the client's vector, multiplies it with the corresponding diagonal, and accumulates the product into the accumulator. After the final iteration, the server generates an additive secret sharing of the final result by sampling a random vector and subtracting it from the accumulator's final value. The server then sends the ciphertext holding the difference back to the client and retains the random vector as its share of the final result. The client decrypts the ciphertext with its secret key and obtains its share of the final result.

Gazelle [27] was the first mixed-protocol framework for secure neural network inference that employed HE for linear layers, GC for non-linear layers and secret-sharing to communicate intermediate results. *Gazelle* implemented Halevi and Shoup's algorithm for matrix multiplication and optimized it to efficiently compute on wide matrices (number of columns > number of rows), which are common in neural networks. Most importantly, *Gazelle* extended this optimized matrix multiplication algorithm to compute 3D convolution, the most prevalent operation in convolutional neural networks. Several subsequent works [33, 40] on secure inference utilized *Gazelle*'s algorithm implemented with the BFV HE scheme for processing the neural network's linear layers. In a later work, Reagen et al. [41] proposed a reordering of operations in *Gazelle* that leads to a lower error growth which subsequently enabled them to work with lower HE parameters to gain a performance boost. Zhang et al.[46] observed that *Gazelle*'s optimization for wide matrices contained few redundant HE rotations which can instead be performed in cleartext without any loss in security. In their implementation, *GALA* [46], they skip these expensive HE rotations leading to faster computation times. Another line of work [4, 5] proposed packing values from different inputs along the batch axis into a single ciphertext. This mitigated the requirement for special algorithms to compute linear algebra operations inside a single ciphertext and the requirement for HE rotations since the values in a single ciphertext correspond to different inputs. This approach is not very useful in many Machine-Learning-as-a-Service scenarios where each user typically queries a single input and not a batch.

Cheetah [24], however, adopted a completely new approach, presenting a new matrix encoding scheme that leverages the implicit convolution computation on the coefficients of polynomials when multiplying two elements from $\mathbb{R}_Q^n$. Consequently, the new algorithms for matrix multiplication and 3D convolution do not require any costly rotation operations. However, this approach generates the final result in multiple ciphertexts, each holding only a few elements of the final result. To reduce communication costs, the authors propose an extraction scheme that breaks up the RLWE ciphertexts into their individual LWE components and returns only the LWE components that individually hold an element of the final result. Although *Cheetah* is implemented with the BFV HE scheme, its plaintext encoding and extraction steps diverge from how BFV is natively implemented in popular HE libraries. As a result, extensive patches to the HE library are necessary for its implementation. Furthermore, the algorithm also implements an optimization specific to the BFV scheme's decryption structure, enabling the truncation of the ciphertext's least significant bits to further decrease the amount of communicated data. Due to this, the *Cheetah* algorithm does not guarantee equivalent performance with the BGV scheme. Additionally, the encoding scheme requires placing secret messages in the coefficient space of the polynomials in the ciphertext, rendering it inapplicable to the CKKS HE scheme.

2.7 Threat Model

All previously discussed secure inference approaches from Section 2.6 operate within a two-party semi-honest threat model. Within this setting, a remote server hosts a model and provides Machine Learning services, while a client sends a request for inference on a specific input. While the trained parameters of the ML model remain confidential to the server and the input is private to the client, both the model architecture and hyper-parameters are openly available to both parties. Under the semi-honest model, each party is expected to rigidly follow the protocol but might attempt to glean information about the other party's data.

3 HELiKs FRAMEWORK

3.1 Design and Achievements

The design of the HELiKs Framework is centered on four key principles:

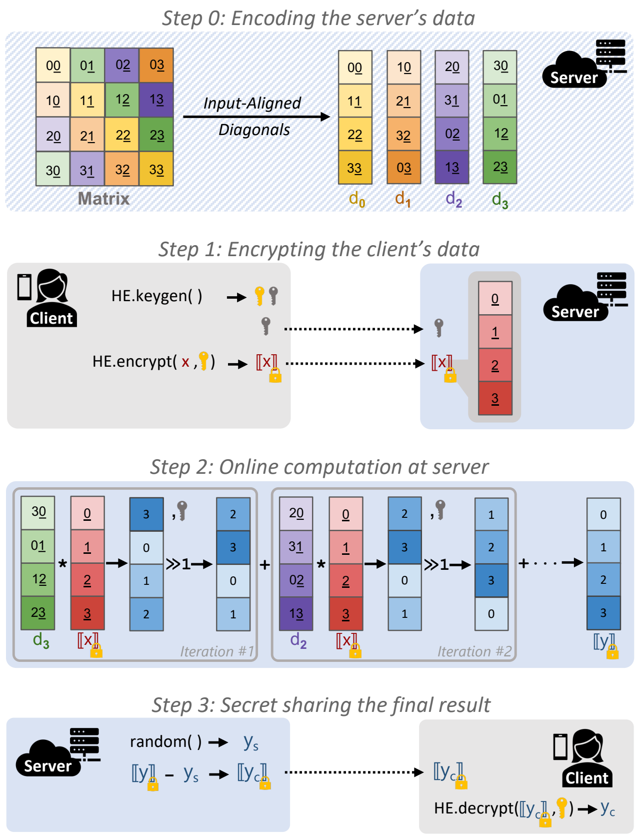
- **Consistency:** Ensuring that the encoding format remains unchanged from input to output, allowing seamless composition of multiple HE operations without the need for client-side repacking between operations.
- **Integrity:** Adhering to scheme definitions for ciphertexts, which enables compatibility with any HE library without necessitating modifications.
- **Compatibility:** Supporting all popular RLWE-HE schemes (BGV, BFV, and CKKS) for a versatile, plug-and-play module that can be incorporated into any HE application, regardless of the underlying scheme.
- **Performance:** Implementing a streamlined and efficient approach approach to minimize overheads, ultimately achieving industry-leading running times and bandwidth usage.

To realize these principles, HELiKs builds upon the Halevi-Shoup method by integrating various optimizations, resulting in substantial improvements in computation complexity, processing time, and overall bandwidth usage. A key innovation is the *MAC-core* design, which reduces the number of NTTs needed during multiplications by 33%. In addition, HELiKs introduces a novel *matrix encoding* that works in tandem with the MAC-core design to further accelerate multiplications. This encoding includes a one-time offline pre-computation process that cuts the number of NTTs performed during online computation by an extra 50%.

HELiKs also incorporates a restructured computation graph for partial sums, which effectively controls error growth from HE computation without compromising correctness. The *partial sum accumulation* strategy in HELiKs lowers the number of rotations from $O(n \log n)$ to $O(n)$, substantially reducing the dependency on the number of evaluation keys from $O(\log n)$ to just $O(1)$. This leads to a significant decrease in the number of evaluation keys sent to the server. To further optimize bandwidth usage, HELiKs reworks the computation to eliminate the public key dependency and adapts the algorithm for symmetric key variants of HE. This optimization halves the data transmitted between the client and server.

3.2 High-level Overview

HELiKs framework comprises efficient protocols for securely computing matrix products and 3D convolution. The primary optimizations stem from the core MAC computation in a matrix product all of which carry over to the protocol 3D convolution. In this section, we give a high-level overview of the protocol for securely computing a matrix-vector product detailing the steps involved in server-side data encoding, client-side encryption, online computation, and secret sharing of the final result. This protocol is illustrated in Figure 3.



Step 0: Encoding the server's data
Matrix
Input-Aligned Diagonals
d_0 d_1 d_2 d_3
Server
Step 1: Encrypting the client's data
Client
HE.keygen( )
HE.encrypt(x ,)
[[x]]
Step 2: Online computation at server
≫1
Iteration #1
Iteration #2
[[y]]
Step 3: Secret sharing the final result
random( ) → y_s
[[y]] − y_s → [[y_c]]
HE.decrypt([[y_c]],) → y_c

Figure 3: Protocol for securely computing a matrix-vector product in the HELiKs framework.

3.2.1 *Step 0: Encoding the Server's Data*.

The server starts by preprocessing its secret matrix to reduce the online computation time. It extracts diagonals from the matrix and creates a set of one-dimensional vectors. Each diagonal consists of elements from the matrix in a diagonal pattern, with the first diagonal starting at the first index of the first column and continuing diagonally downwards. The server then converts these vectors (matrix diagonals) into plaintexts for the chosen RLWE-based HE scheme, with polynomials represented in the NTT domain.

3.2.2 *Step 1: Encrypting the Client's Data*.

The client creates keys for a symmetric-key version of the RLWE-based HE scheme and encrypts its secret vector, generating an input-ciphertext. This input-ciphertext and a key-switching key are sent to the server. The key-switching key is necessary for rotating encrypted data in a ciphertext one slot to the right.

3.2.3 *Step 2: Online Computation at the Server*.

Once the server receives the input-ciphertext and key-switching key, it transforms the polynomials in the input-ciphertext to the NTT domain. Following this, the server multiplies the input-ciphertext by the plaintext containing the last diagonal and transforms the polynomials in the resulting product-ciphertext from the NTT domain to the coefficient domain. This product-ciphertext is used to initialize an accumulator-ciphertext. The server then performs a series of iterative computations, involving rotation, multiplication, and addition operations. In each iteration, the server uses four inputs: the input-ciphertext containing the client's secret vector, the key-switching key for rotation, the accumulator-ciphertext, and the plaintext containing the corresponding diagonal of the server's matrix for that iteration.

The server processes the diagonals of its matrix in descending order, with the first iteration using the last diagonal, and the final iteration using the first diagonal. During each iteration, the server multiplies the input-ciphertext with the plaintext containing the corresponding diagonal. The polynomials in the resulting product-ciphertext are then transformed back to the coefficient representations from the NTT domain. Next, the accumulator-ciphertext is rotated, shifting the underlying data one slot to the left, and then incremented by adding the product-ciphertext. After completing the iterations, the server applies modulus-switching to the accumulator-ciphertext, storing it as the result-ciphertext.

3.2.4 *Step 3: Secret Sharing the Final Result*.

The server generates a random vector, encodes it into plaintext, adds it to the result-ciphertext to obtain the output-ciphertext, and sends the output-ciphertext back to the client. The server negates the random vector and retains it as its share of the result of the secure matrix-vector product. The client decrypts the output-ciphertext to obtain its share of the result of the secure matrix-vector product.

4 SECURE MATRIX MULTIPLICATION

In this section, we present the implementation details of the specific algorithm in the HELiKs framework for secure matrix multiplication with HE.

4.1 Encoding 2D Matrices

Recall from Section 2.3 that RLWE-HE schemes can only represent fixed-length vectors, particularly with a scheme instantiated on $\mathbb{R}_Q^n$ we can only pack a vector of n ($n/2$ for CKKS) elements in a single ciphertext/plaintext. Further, using a polynomial modulus degree n that is a power of 2 enables highly efficient implementations of NTT. Therefore, most practical HE applications, including our approach, set n to a power of 2. As a result, the number of slots in an HE ciphertext is also a power of 2. To efficiently pack a 2D matrix in the available slots, it is first padded to the nearest powers of 2 of the original dimensions.

The basic case of packing a square matrix, as discussed in Section 3.2.1 and illustrated in *Step 0* of Figure 3, is applicable only when using the CKKS scheme on $\mathbb{R}_Q^n$ with matrix dimension $d = n/2$. This is because the CKKS scheme supports a full range of cyclic rotations across the $n/2$ available slots, allowing for rotations of any size between 1 and $n/2-1$. For BFV and BGV schemes on $\mathbb{R}_Q^n$, which support only the group of half rotations, we implement a modified version of *Gazelle*'s Hybrid approach as presented in *GALA* [46]. Packed diagonals are input-aligned, and the final *rotate-and-sum* computation to accumulate the two final partial sums in the two halves of the ciphertext is performed on the cleartext secret shares.

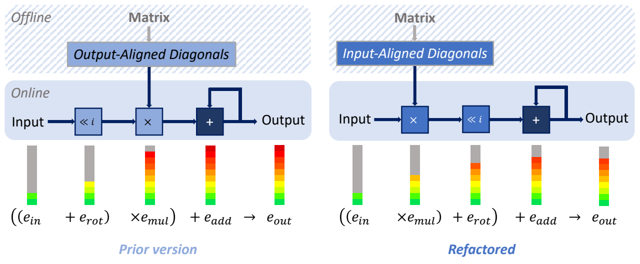

Offline
Matrix
Output-Aligned Diagonals
Online
Input
$\ll i$
$\times$
$+$
Output
$((e_{in} \quad + e_{rot}) \quad \times e_{mul}) \quad + e_{add} \rightarrow e_{out}$
Prior version
Matrix
Input-Aligned Diagonals
Input
$\times$
$\ll i$
$+$
Output
$((e_{in} \quad \times e_{mul}) \quad + e_{rot}) \quad + e_{add} \rightarrow e_{out}$
Refactored

Figure 4: Refactored partial sum computation.

In scenarios involving square matrices of size $d \leq n/4$, we pack $\frac{n}{d}$ ($\frac{n}{2d}$ for CKKS) unique diagonals in each plaintext on the server-side and encrypt $\frac{n}{d}$ ($\frac{n}{2d}$ for CKKS) copies of the input vector in the input ciphertext on the client side. For rectangular matrices with dimensions (d_o, d_i) with either d_o or $d_i \leq n/4$, we pack input-aligned extended diagonals that wrap around the smaller dimension. For cases where $d_o \leq n/4$, we perform the final rotate-and-sum computation on the cleartext secret shares similar to *GALA* [46], bypassing the need for costly ciphertext computation in HE.

4.2 Partial Sum Computation

We start with the Halevi-Shoup algorithm as our baseline, the details of which were presented in Section 2.6. The computation involved in the algorithm is iterative where in each iteration the input ciphertext is rotated and then multiplied by the corresponding diagonal for the iteration and finally accumulated with the result of the previous iteration. This is illustrated in the left half of Figure 4. In our implementation, the operations are reordered such that the input ciphertext is first multiplied by the diagonal, then rotated to align the product with the output, and finally accumulated into the result of the previous iteration. To ensure correctness, the encoding scheme for the matrix is tweaked to produce *input-aligned* diagonals i.e., the diagonals now begin from the first column of the matrix and proceed diagonally downwards.

A refactored version is illustrated in the right half of Figure 4. Though the total counts of each operation remain the same, the error growth is now much less than what it was, leaving some available error budget unused. This allows us to shave off these unused bits from the coefficient modulus and work with ciphertexts that have polynomials with much smaller coefficients while maintaining the dynamic range (plaintext modulus) and security level for the encrypted data. The drop in the size of the ciphertexts linearly reduces the processing times of all HE operations across the board as well as lowers bandwidth usage during communication.

4.3 Partial Sum Accumulation

Recall from Section 2.4 that arbitrary rotations in HE are composed of multiple *prime rotations* for which the evaluation keys have been generated. For a square matrix of dimension d, the partial sum computation in our current version is comprised of d iterations where the i^{th} iteration involves a rotation of i steps to the left. This would require a total of about $(d-1) \times (\log d)/2$ *prime rotations*[2]

[2]The average hamming weight of all positive integers less than d is $\frac{\log d}{2}$.

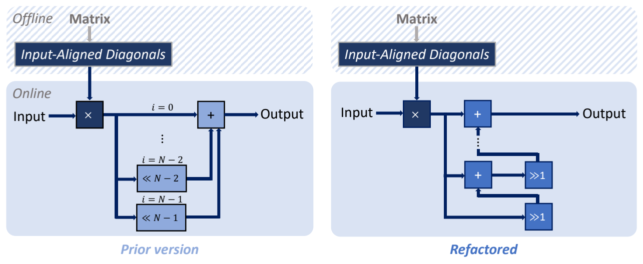


Figure 5: Refactored partial sum aggregation.

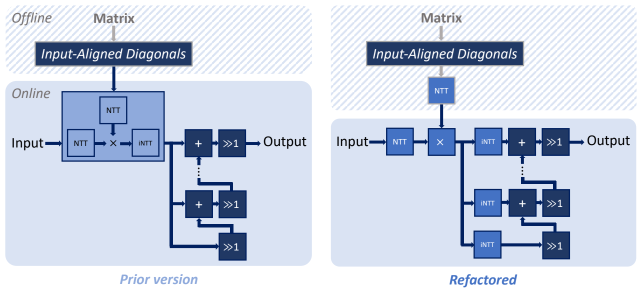


Figure 6: Refactored multiplications with fewer NTTs.

with $\log d$ evaluation keys. Figure 5 illustrates this in the schematic on the left-hand side.

A refactored version is shown in the right half of Figure 5. It features a restructured partial sum accumulation strategy that consumes the diagonals in descending order. In the first iteration, the input ciphertext is first multiplied with the plaintext holding the last diagonal, the product is then rotated 1 step to the right and used to initialize an accumulator. In the subsequent iterations, after the input ciphertext is multiplied with the corresponding plaintext[3] it is added into the accumulator and then the value stored in the accumulator is rotated 1 step to the right. The refactored version lowers the number of *prime rotations* by a factor of $(\log d)/2$. Moreover, since all the *prime rotations* involve a fixed shift of 1 step to the right the computation now requires only one evaluation key.

4.4 NTT pre-computation

Recall from Section 2.4 that all contemporary HE libraries implement NTT and inverse-NTT (iNTT) to speed up the computation involved in polynomial multiplications. For every multiplication, both operands first undergo an NTT, then a Hadamard product of the respective NTT representations of the operands is computed, and finally, this Hadamard product is transformed back to its native coefficient representation through an iNTT to obtain the final result. The left half of Figure 6 illustrates the intrinsic calls to NTT and iNTT by the multiplication operator in popular HE libraries when both operands are not explicitly represented in the NTT domain. This involves a total of $2d$ calls to NTT and d calls to iNTT for d multiplications for a matrix of dimension d.

In the right half of Figure 6, we show the refactored algorithm which reduces the total calls made to (i)NTT by 66.7%. We first remove all the redundant NTTs for the input ciphertext by limiting it to just one. Further, we observe that the NTTs for the plaintext diagonals are also wastefully re-calculated for subsequent queries to the protocol involving the same matrix. To assuage this issue, we integrate these NTT computations into the offline matrix encoding process. Though this results in a longer encoding time for the matrix, it leads to a much faster online processing time. Moreover, the computation cost involved in the new encoding algorithm amortizes over subsequent queries involving the same matrix.

[3]In the i^{th} iteration the input is multiplied with the $(d-i)^{\text{th}}$ diagonal.

4.5 Tiling for Large Matrices

To efficiently compute on large matrices with dimensions $(d_o, d_i) > n/2$, we partition the matrix into $n_o \times n_i$-many square tiles with a dimension of $n/2$ and encrypt the input vector of d_i elements using n_i ciphertexts. The final result is produced in n_o ciphertexts, and we compute one output ciphertext at a time following an iterative process, which is largely similar to the descriptions provided in earlier sections. During each iteration, we first compute n_i products of the n_i input ciphertexts with the corresponding diagonal from the n_i tiles. We then combine these n_i products into a single ciphertext and perform the iNTT, accumulation with the result of the previous iteration, and rotation of one step to the right. This method reduces the total number of iNNTs and rotations by a factor of n_i compared to a naive approach of computing $n_o \times n_i$ matrix products.

4.6 Symmetric Key HE

HE can also be instantiated with a symmetric key variant where a single key is used for both encryption and decryption. When a ciphertext is generated with symmetric key encryption, the first polynomial element $\overline{a}$ is created randomly. This enables the possibility of transmitting a short seed in place of the full polynomial. The receiving party can then use this seed to locally expand and reconstruct the polynomial with a pseudorandom generator (PRG). Although the second polynomial element $\overline{b}$ must still be transmitted in its entirety, this approach significantly reduces the communication cost compared to public key encryption methods. To achieve this reduction in communication cost, the algorithm is modified to remove all dependencies on the public key. This alteration also ensures that the server is no longer capable of encrypting any data, further enhancing the security and efficiency of the HELiKs framework.

4.7 Secret Sharing

For BFV and BGV schemes, the plaintext modulus P is typically selected as a Pseudo-Mersenne prime in the form of $P = 2kn + 1$ [7, 19, 44], where n represents the polynomial modulus degree and k is a positive integer. When P takes this form, the polynomial ring $\mathbb{R}_P^n$ has a multiplicative group order of 2^n. The Chinese Remainder Theorem (CRT) representation of these polynomial rings enables the packing of up to n messages from $\mathbb{Z}_P$ into a single polynomial, allowing for efficient SIMD operations on encrypted data.

The BFV and BGV schemes can only process data with values from $\mathbb{Z}_P$ and support arithmetic operations in $\mathbb{Z}_P$ for encrypted computation. Therefore, we employ the same finite field for secret sharing in the BFV and BGV schemes. After the server completes the final computation (excluding the *rotate-and-sum* operation), it generates a random vector $\mathbf{y'_s}$ of n elements uniformly sampled from $\mathbb{Z}_P$ for each ciphertext $\overline{\mathbf{ct}}_{\mathbf{y'}}$ that encrypts a portion $\mathbf{y'}$ of the final result. The server encodes this vector into a plaintext $\overline{pt}_{\mathbf{y'_s}}$, subtracts it from $\overline{\mathbf{ct}}_{\mathbf{y'}}$ to obtain $\overline{\mathbf{ct}}_{\mathbf{y'_c}} = \overline{\mathbf{ct}}_{\mathbf{y'}} - \overline{pt}_{\mathbf{y'_s}}$, ensuring that $\mathbf{y'_c} + \mathbf{y'_s} \equiv y' \mod P$, and returns $\overline{\mathbf{ct}}_{\mathbf{y'_c}}$ to the client. The client then decrypts $\overline{\mathbf{ct}}_{\mathbf{y'_c}}$ to obtain $\mathbf{y'_c}$ and performs the remaining *rotate-and-sum* operation on $\mathbf{y'_c}$ in $\mathbb{Z}_P$ to acquire its secret share of the result, $\mathbf{y_c}$. Simultaneously, the server carries out the *rotate-and-sum* operation on $\mathbf{y'_s}$ to obtain its final secret share, $\mathbf{y_s}$.

For the CKKS scheme, the ciphertext modulus Q is a product of l primes, enabling l levels of computation i.e., $Q = \prod_{i=0}^{l-1} q_i$. The HE computation utilizes $l-1$ levels, and after the computation, the ciphertexts holding the result are *rescaled* down to q_0 such that each ciphertext $\overline{\mathbf{ct}}_{\mathbf{y'}}$ is in $[\mathbb{R}^n_{q_0}]^2$. Secret sharing in CKKS can be implemented using the conversion scheme described in the *MP2ML* framework [4]. Following this approach, the server ends up with a secret share $\mathbf{y_s}$ that has uniformly random elements from $\mathbb{Z}_{q_0}$, while the client obtains $\mathbf{y_c}$ with elements from $\mathbb{Z}_Q$ such that $\mathbf{y_c} + \mathbf{y_s} \equiv y \mod Q$. This asymmetry in the finite fields arises from the fact that the plaintext space of CKKS is $\mathbb{R}^n_Q$ and though the ciphertexts are in $[\mathbb{R}^n_{q_0}]^2$ the decryption process results in a plaintext from $\mathbb{R}^n_Q$.

5 SECURE 3D CONVOLUTION

HELiKs also includes an optimized implementation for 3-dimensional convolution, which is the most common operation in many popular DNN models. Gazelle [27] proposed an encoding scheme for 4D filters and an algorithm to evaluate 3D convolutions based on a series of rotate, multiply and accumulate operations. The algorithm presented in *Gazelle* [27] has been reused in many popular privacy-preserving DNN frameworks to date [33, 40]. The algorithm for Secure 3D Convolution in HELiKs builds on the specific implementation of *Gazelle* in *CrypTFlow2* [34, 40] with insights gained from developing the protocol for matrix multiplication.

First, we briefly recall the convolution operation in Neural Networks. Convolution of a (h, w) 2D input with a (f_h, f_w) 2D filter involves scanning the filter across the entire input and storing a weighted sum of the overlapping (f_h, f_w) portion of input with respective filter elements at corresponding output locations to generate a (h, w) 2D output. The convolution of a (c_i, h, w) 3D input with a (c_i, f_h, f_w) 3D filter involves the same process as above, with the weighted sum now involving a (c_i, f_h, f_w) 3D overlapped region, and results in a (h, w) 2D output. The 3D convolution operation repeats this process c_o times for (c_o, c_i, f_h, f_w) 4D filter to generate a (c_o, h, w) 3D output.

This transformation of a (c_i, h, w) input to a (c_o, h, w) output using a (c_o, c_i, f_h, f_w) 4D filter can be reinterpreted as a subtle variation of a matrix-vector product of a $(c_i, 1)$ input with a (c_o, c_i) matrix to generate a $(c_o, 1)$ output. In a matrix-vector product, we multiply scalar elements in the input vector with scalar elements of the matrix, at the lowest level, and then aggregate the scalar products that correspond to the same output locations to generate the output. The 3D convolution operation is very similar where in place of the scalar multiplications we perform 2D convolutions of 2D channels of input with 2D sections of the filter, at the lowest level, and then aggregate results of these 2D convolutions that correspond to the same output channel to generate the final output.

5.1 Offline Filter Encoding

We implement a modified version of Gazelle's encoding scheme for 4D filters which allows us to first multiply the filter with the input ciphertext and then perform the rotation. The encoded filter returned by the encoding scheme is a (c_o, c'_i, f) 3D array of plaintexts, $c'_i = c_i/c_n$ is the number of input ciphertexts, $c_n = n/(h \times w)$ is the number of channels encoded in a single ciphertext and $f = f_h \times f_w$ is the number of filter elements. In the inner-most dimension, the encoded filter contains f plaintexts pertaining to two 2D convolutions of two input channels with the corresponding (f_h, f_w) sections of the filter for the respective output channels. To generate these f plaintexts for the 2D convolution, we mostly follow the `HE_preprocess_filters_OP()` method implemented in *CrypTFlow2* [34]. The only change is that each plaintext generated using their method was designed to be multiplied with a particular rotated version of the input channel, which we undo by rotating the plaintexts by the same amount in the opposite direction and saving these rotation amounts for later use. As mentioned earlier in Section 4.2, performing multiplication before rotation significantly reduces the error growth in the HE computation. Finally, all plaintexts are transformed to their NTT representations which allows us to perform much faster multiplications during the online process as mentioned in Section 4.4. This encoding of the 4D filters is a one-time offline pre-computation that is reused for all subsequent convolutions involving the same filter.

5.2 Online Convolution Processing

The online process begins with the client sending c'_i HE ciphertexts to the server for its (c_i, h, w) input image generated with symmetric key encryption. We first transform all the input ciphertexts to their NTT representations and then proceed with an iterative algorithm to compute partial sums for the convolution as illustrated in Algorithm 1. This algorithm generates c_o ciphertexts that together hold c_n partial sums for each channel in the final output image. The process involves 3 nested loops where the outermost loop iterates over the output channels, the next loop iterates over all filter elements, and the innermost loop iterates over all input ciphertexts.

First, we carry over the *tiling* approach from Section 4.5 to minimize the total number of rotations. This is achieved by computing the products for all input ciphertexts with the corresponding plaintext filters for the corresponding output channel and filter element and aggregating them into a single ciphertext which is then rotated. Next, we restructure the accumulation of these intermediate ciphertexts in order to minimize the total number of rotations, iNNTs, and evaluation keys used for the computation as described in Section 4.3. For each output channel, we stream through the filter elements backward starting with the last one. For each segment of f_w filter elements, intermediate ciphertexts for the first $f_w - 1$ filter elements are rotated 1 step to the right and the next intermediate ciphertext is rotated w steps to the right. This way the computation requires only 2 evaluation keys with all rotations performed in one shot.

Algorithm 1: 3D Convolution in HELiKs

```
Input: Ciphertexts of input image $\overline{\mathbf{ct}}_{\mathbf{i},1}, \overline{\mathbf{ct}}_{\mathbf{i},2}, .., \overline{\mathbf{ct}}_{\mathbf{i},c_i'}$
       Encoded filters $\overline{\mathrm{F}}$ with $(c_o \times c_i' \times f)$ plaintexts ;
       Evaluation keys for rotations $\overline{\mathbf{ek}}_1$ and $\overline{\mathbf{ek}}_{\mathrm{w}}$
Output: Ciphertexts of partial results $\overline{\mathbf{ct}}_{\mathbf{o}',1}, \overline{\mathbf{ct}}_{\mathbf{o}',2}, .., \overline{\mathbf{ct}}_{\mathbf{o}',c_o}$
/* $c_n = n/(h \times w)$ */
/* $c_i' = c_i/c_n$ */
/* $f = f_h \times f_w$ filter elements */
for $p = 1$ to $c_i'$ do
    $\overline{\mathbf{ct}}_{\mathbf{i},p} = \mathrm{NTT}(\overline{\mathbf{ct}}_{\mathbf{i},\mathbf{p}})$
end
for $q = 1$ to $c_o$ do
    /* Start from the last filter element */
    $\overline{pt}_{\mathrm{fil},f} = \overline{\mathrm{F}}[q, 1, f]$
    $\overline{\mathbf{ct}}_{\mathbf{o}',q} = \mathrm{multiply}(\overline{\mathbf{ct}}_{\mathbf{i},1}, \overline{pt}_{\mathrm{fil},f})$
    for $s = 2$ to $c_i'$ do
        $\overline{pt}_{\mathrm{fil},f} = \overline{\mathrm{F}}[q, s, f]$
        $\overline{\mathbf{ct}}_{\mathbf{o}',q} = \overline{\mathbf{ct}}_{\mathbf{o}',q} + \mathrm{multiply}(\overline{\mathbf{ct}}_{\mathbf{i},s}, \overline{pt}_{\mathrm{fil},r})$
    end
    $\overline{\mathbf{ct}}_{\mathbf{o}',q} = \mathrm{iNTT}(\overline{\mathbf{ct}}_{\mathbf{o}',q})$
    for $r = f - 1$ to $1$ do
        if $r \mod f_w = 0$ then
            $\overline{\mathbf{ct}}_{\mathbf{o}',q} = \text{rotate-right}(\overline{\mathbf{ct}}_{\mathbf{o}',q}, \overline{\mathbf{ek}}_{\mathrm{w}})$
        else
            $\overline{\mathbf{ct}}_{\mathbf{o}',q} = \text{rotate-right}(\overline{\mathbf{ct}}_{\mathbf{o}',q}, \overline{\mathbf{ek}}_1)$
        end
        $\overline{pt}_{\mathrm{fil},r} = \overline{\mathrm{F}}[q, 1, r]$
        $\overline{\mathbf{ct}}_{\mathbf{partial}} = \mathrm{multiply}(\overline{\mathbf{ct}}_{\mathbf{i},1}, \overline{pt}_{\mathrm{fil},r})$
        for $s = 2$ to $c_i'$ do
            $\overline{pt}_{\mathrm{fil},r} = \overline{\mathrm{F}}[q, s, r]$
            $\overline{\mathbf{ct}}_{\mathbf{partial}} = \overline{\mathbf{ct}}_{\mathbf{partial}} + \mathrm{multiply}(\overline{\mathbf{ct}}_{\mathbf{i},s}, \overline{pt}_{\mathrm{fil},r})$
        end
        $\overline{\mathbf{ct}}_{\mathbf{partial}} = \mathrm{iNTT}(\overline{\mathbf{ct}}_{\mathbf{partial}})$
        $\overline{\mathbf{ct}}_{\mathbf{o}',q} = \overline{\mathbf{ct}}_{\mathbf{o}',q} + \overline{\mathbf{ct}}_{\mathbf{partial}}$
    end
end
```

Finally, we refactor the algorithm to avoid using zero ciphertexts and thus remove all dependency on the public key.

After generating the partial sums using the algorithm described above, c_n partial sums for each output channel are aggregated using an approach similar to the `HE_output_rotations()` method implemented in *CrypTFlow2* [34]. We modify this method as illustrated in Algorithm 2, to restructure the computation and accumulation of intermediate ciphertexts such that the computation only requires 2 evaluation keys with all rotations performed in one shot. On a high level, this process involves accumulating all channels in one half of a partial sum with $c_n/2$ rotations of $h \times w$ steps to the right and adding up the accumulated results in the two halves of the partial sum by rotation of the columns of the partial sum. Our implementation requires only 2 evaluation keys and does not require a public key for the computation.

Algorithm 2: Output Rotations for 3D Convolution

```
Input: Partial results of convolution $\overline{\mathbf{ct}}_{\mathbf{o}',1}, \overline{\mathbf{ct}}_{\mathbf{o}',2}, .., \overline{\mathbf{ct}}_{\mathbf{o}',c_o}$
       Evaluation keys for rotation $\overline{\mathbf{ek}}_{h \times w}$ and $\overline{\mathbf{ek}}_{\mathrm{col}}$
Output: Ciphertexts of output image $\overline{\mathbf{ct}}_{\mathbf{o},1}, \overline{\mathbf{ct}}_{\mathbf{o},2}, .., \overline{\mathbf{ct}}_{\mathbf{o},c_o'}$
/* $c_n = n/(h \times w)$ */
/* $c_o' = c_i/c_n$ */
out-idx-flags ← Boolean vector of $c_o'$ false values
for $s = 0$ to $s < 2c_o/c_n$ with increments of 2 do
    $q' = s \times c_n/2 + 1$
    $q'' = q'' + c_n/2$
    $\overline{\mathbf{ct}}_{\mathbf{partial},s+1} = \overline{\mathbf{ct}}_{\mathbf{o}',q'}$
    $\overline{\mathbf{ct}}_{\mathbf{partial},s+2} = \overline{\mathbf{ct}}_{\mathbf{o}',q''}$
    for $t = 1$ to $t < c_n/2$ do
        $q' = s \times c_n/2 + t + 1$
        $q'' = q' + c_n/2$
        $\overline{\mathbf{ct}}_{\mathbf{partial},s+1} = \overline{\mathbf{ct}}_{\mathbf{partial},s+1} + \overline{\mathbf{ct}}_{\mathbf{o}',q'}$
        $\overline{\mathbf{ct}}_{\mathbf{partial},s+2} = \overline{\mathbf{ct}}_{\mathbf{partial},s+2} + \overline{\mathbf{ct}}_{\mathbf{o}',q''}$
        $\overline{\mathbf{ct}}_{\mathbf{partial},s+1} = \text{rotate-right}(\overline{\mathbf{ct}}_{\mathbf{partial},s+1}, \overline{\mathbf{ek}}_{h \times w})$
        $\overline{\mathbf{ct}}_{\mathbf{partial},s+1} = \text{rotate-right}(\overline{\mathbf{ct}}_{\mathbf{partial},s+2}, \overline{\mathbf{ek}}_{h \times w})$
    end
    out-idx = $(s/2 \mod c_o) + 1$
    if $s = 0$ then
        if out-idx-flags[out-idx] is false then
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{partial},s+1}$
            out-idx-flags[out-idx] = true
        else
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} + \overline{\mathbf{ct}}_{\mathbf{partial},s+1}$
        end
        if $2c_o/c_n = 1$ and $2c_i/c_n > 1$ then
            $\overline{\mathbf{ct}}_{\mathbf{partial},s+1} =$
              rotate-columns$(\overline{\mathbf{ct}}_{\mathbf{partial},s+1}, \overline{\mathbf{ek}}_{\mathrm{col}})$
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} + \overline{\mathbf{ct}}_{\mathbf{partial},s+1}$
        end
        if $2c_o/c_n > 1$ then
            $\overline{\mathbf{ct}}_{\mathbf{partial},s+2} =$
              rotate-columns$(\overline{\mathbf{ct}}_{\mathbf{partial},s+2}, \overline{\mathbf{ek}}_{\mathrm{col}})$
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} + \overline{\mathbf{ct}}_{\mathbf{partial},s+2}$
        end
    else
        if out-idx-flags[out-idx] is false then
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{partial},s+1}$
            out-idx-flags[out-idx] = true
        else
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} + \overline{\mathbf{ct}}_{\mathbf{partial},s+1}$
        end
        if $2c_o/c_n > 1$ then
            $\overline{\mathbf{ct}}_{\mathbf{partial},s+2} = \text{rotate-columns}(\overline{\mathbf{ct}}_{\mathbf{partial},s+2})$
            $\overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} = \overline{\mathbf{ct}}_{\mathbf{o},\text{out-idx}} + \overline{\mathbf{ct}}_{\mathbf{partial},s+2}$
        end
    end
end
```

Table 1: Total operation counts for *prime rotations* (refer Section 2.4), NTTs and iNTTs, along with the total error growth and number of keys utilized for computing a convolution with the BFV scheme using *Gazelle* [27], *CrypTFlow2* [40], *Cheetah* [24] and HELiKs. The values shown in this table pertain to a (c_i, h, w) input image and a (c_o, c_i, f_h, f_w) filter. The quantity $c_n = n/(h \cdot w)$ denotes the number of image channels packed in a ciphertext with n slots and $f = f_h \cdot f_w$ is the total number of filter elements.

Convolution	*Prime Rotations*	NTTs	iNTTs	Error Growth	Keys Utilized	# Out cts
Gazelle	$\frac{c_i \cdot c_o \cdot (c_n - 1)}{c_n^2} \times \frac{\log c_n}{2} + \frac{c_i \cdot (f-1)}{c_n} \times \frac{\log f}{2}$	$\frac{2 \cdot c_o \cdot c_i \cdot f}{c_n}$	$\frac{c_o \cdot c_i \cdot f}{c_n}$	$\frac{c_i \cdot (f-1)}{c_n} \cdot (e_{in} + e_{rot}) \cdot e_{mul}$ $+ \frac{c_i \cdot (f-1)}{c_n} \cdot (c_n - 1) \cdot f \cdot e_{rot}$	$\overline{\mathbf{pk}} + \log n\, \overline{\mathbf{ek}}$s	$\frac{c_o}{c_n}$
CrypTFlow2	$\frac{c_o \cdot (c_n - 1)}{c_n} \times \frac{\log c_n}{2} + \frac{c_i \cdot (f-1)}{c_n} \times \frac{\log f}{2}$	$\frac{2 \cdot c_o \cdot c_i \cdot f}{c_n}$	$\frac{c_o \cdot c_i \cdot f}{c_n}$	$\frac{c_i \cdot (f-1)}{c_n} \cdot (e_{in} + e_{rot}) \cdot e_{mul}$ $+ (c_n - 1) \cdot f \cdot e_{rot}$	$\overline{\mathbf{pk}} + \log n\, \overline{\mathbf{ek}}$s	$\frac{c_o}{c_n}$
Cheetah	0	$\frac{2 \cdot c_o \cdot c_i \cdot f}{c_n}$	$\frac{c_o \cdot c_i \cdot f}{c_n}$	$e_{in} \cdot e'_{mul}$	$\overline{\mathbf{pk}}$	$\frac{c_o \cdot c_i \cdot f}{c_n}$
HELiKs (Our work)	$\frac{c_o \cdot (c_n - 1)}{c_n} + c_o \cdot (f - 1)$	$\frac{c_i}{c_n}$	$c_o \cdot f$	$\frac{c_i \cdot (f-1)}{c_n} \cdot e_{in} \cdot e_{mul}$ $+ (c_n - 1) \cdot f \cdot e_{rot}$	up to 4 $\overline{\mathbf{ek}}$s	$\frac{c_o}{c_n}$

5.3 Operation Counts and Error Growth

We summarize the total operation counts of *prime rotations*, NTTs and iNTTs for convolution with *Gazelle*, *CrypTFlow2* and our implementation in Table 1. The authors of *Gazelle* proposed two strategies for performing the rotations to accumulate partial sums, *input-rotations* and *output-rotations*, and showed that the *output-rotations* variant performs better. The *output-rotations* strategy involves $f - 1$ ($f = f_h \times f_w$) rotations for each input ciphertext requiring $\log f/2$ *prime rotations* on average for each rotation. The rotated input ciphertexts are multiplied with the corresponding plaintexts and accumulated to generate $c_i \cdot c_o/c_n$ intermediate ciphertexts. This process involves $c_i \cdot c_o \cdot f/c_n$ multiplications with each requiring two NTTs to transform the operands to their NTT representations and one iNTT to transform the product back to the coefficient representation for the rest of the computation. The intermediate ciphertexts are then grouped into c_o/c_n sets corresponding to the output ciphertexts they should be accumulated into. Every set contains c_i ciphertexts, and each ciphertext in the set is rotated $c_n - 1$ times to accumulate all channels it packs. Each rotation here requires an average of $\log c_n/2$ prime rotations.

The authors of *CrypTFlow2* made the observation that the *output-rotations* strategy utilized a factor of c_i/c_n more rotations than required. The algorithm implemented in *CrypTFlow2*, first accumulates all intermediate ciphertexts corresponding to the same rotation amount and then performs rotations and the subsequent accumulations to generate the final result. Since our implementation in HELiKs builds on this algorithm, we gain the same improvements for the outer rotations when aggregating the intermediate ciphertexts. The number of inner rotations performed to generate these intermediate ciphertexts in our implementation differs from *CrypTFlow2* because we reorder the operations to perform the multiplications first on the inputs before any other operation. As shown in Table 1, this results in an error growth that is $e_{rot} \cdot e_{mul} \cdot c_i \cdot (f-1)/c_n$ lower than all prior methods enabling us to work with much smaller HE parameters. Further, because of the NTT pre-computation we are able to significantly lower the number of (i)NTTs performed during the online processing and gain significant speedups in online runtimes.

The authors of *Cheetah* [24] introduce a distinct approach, rooted in the concept that polynomial multiplications implicitly evaluate inner products over their coefficients. To harness this, *Cheetah* devises an encoding technique in which the plaintext is positioned in the coefficients of the polynomial, differing from the conventional Canonical embedding in the polynomial's evaluation space. This deviation from scheme definitions necessitates engineered solutions in the software libraries for seamless operation. When data is encoded using this method, the core computation is quite direct, necessitating only $c_i \cdot c_o/c_n$ HE multiplications[4], with each involving 2 NTTs and one iNTT. The absence of rotations negates the need for any evaluation keys. However, there's a caveat: this technique disperses the end result across $c_i \cdot c_o/c_n$ ciphertexts, which calls for further adjustments in the software library to solely extract the segment of the RLWE ciphertexts harboring the actual result. This unconventional encoding also brings additional overhead to the error growth. Since the plaintexts reside in the coefficient domain, its comparable magnitude in the Canonical (evaluation) space magnifies by a factor of up to $6\sqrt{n}$ [13], denoted as $e'_{mul} \leq 6\sqrt{n} \cdot e_{mul}$.

6 EVALUATION

This section provides a performance evaluation for the algorithms implemented in HELiKs and compares them to prior work.

[4]Assuming $(W_W, H_W) = (W, H)$ [24, Appendix D]

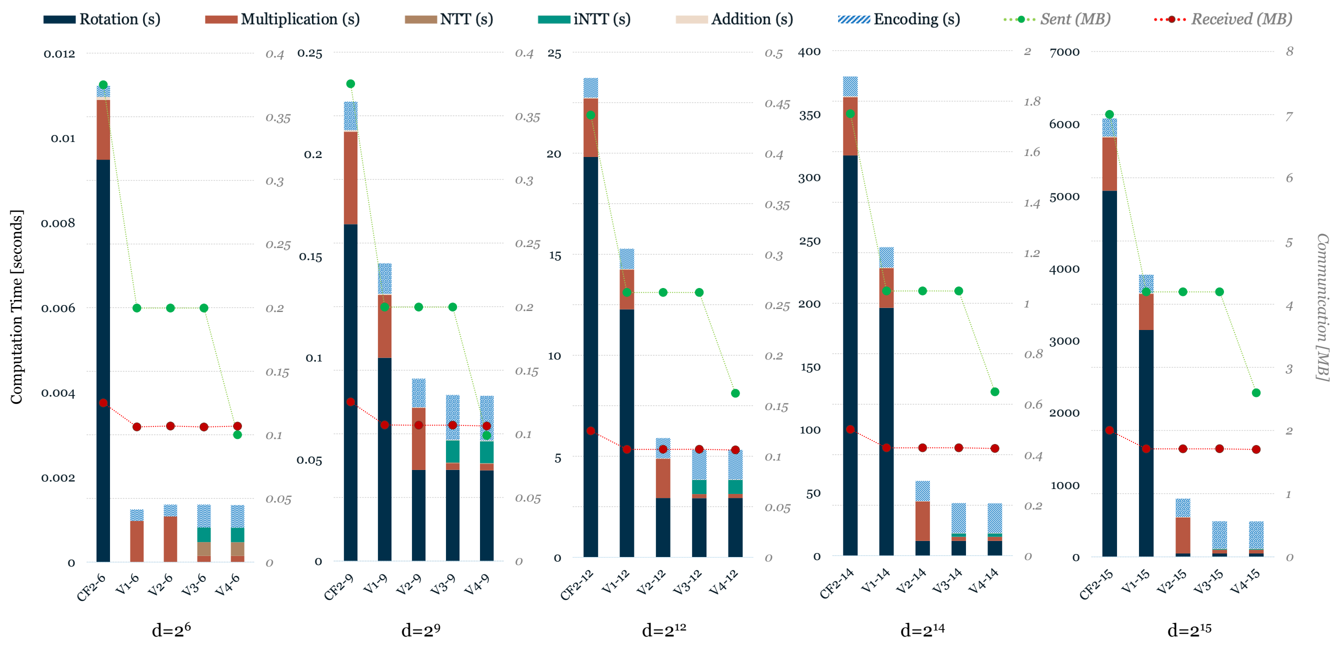


Figure 7: Stacked computation times for all operations and communication measurements for Matrix multiplication using HELiKs for square matrices of dimension $d = 2^6, 2^9, 2^{12}, 2^{14}$ and 2^{15}. The computation times shown for each matrix of dimension pertain to the baseline *CrypTFlow2* and each optimization included in HELiKs, where V4 is the final version implemented in HELiKs. The communication measurements are for the input data sent by the client to the server and the output data received by the client from the server.

Experimental Setup: We ran evaluation experiments on an Intel i7-7700K CPU with 32GB of memory, with a default bandwidth of 100 MB/s. Reported numbers are the average of 100 runs. For each experiment, we pick parameters that achieve similar precision and accuracy to the works against which we compare our performance. Since all the relevant prior work uses the BFV HE scheme, all experiments shown below are performed with the BFV scheme in the SEAL Library [43] to ensure fairness. The computation performed in all experiments is verified for correctness (no decryption failure).

6.1 Matrix Multiplication

For all experiments in this section, the polynomial modulus is set to $n = 8192$ and the plaintext modulus is set to $P = 4293918721$ which is a 32-bit prime. In Figure 7, we present the benchmarks for matrix multiplications with the baseline *Gazelle* algorithm as implemented in *CrypTFlow2* (CF2) and with each of the optimizations implemented in HELiKs:

- V1: Performing multiplications first (refer Section 4.2).
- V2: Performing fixed rotations (refer Section 4.3)
- V3: Pre-computing NTTs (refer Section 4.4)
- V4: Using symmetric key encryption (refer Section 4.6)

We include the *tiling* optimization from Section 4.5 in *V2*, *V3* and *V4* only. Since the optimizations proposed in *GALA* [46] are similar to *V1*, we do not include tiling for *V1*, so it can be used as a point of comparison with *GALA*.

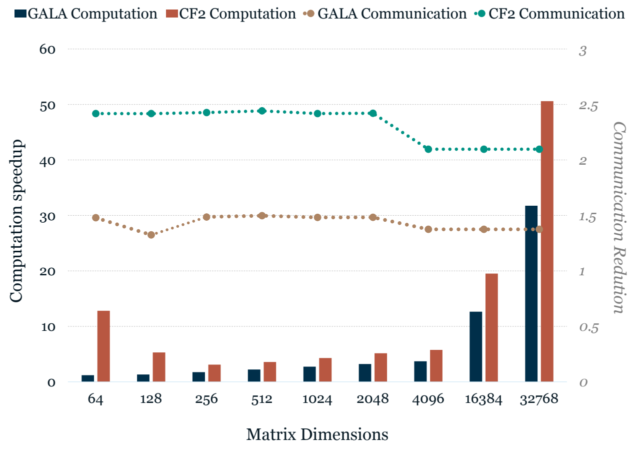


Figure 8: Matrix multiplication performance gains using HELiKs over *CrypTFlow2* and *GALA*.

The values shown in Figure 7 are for square matrices of dimensions $d = 2^6, 2^9, 2^{12}, 2^{14}, 2^{15}$ where Va-b pertains to version a for $d = 2^b$. Since CF2 has a much higher error growth it requires a large ciphertext modulus of 218 bits in total while *V1*, *V2*, *V3* and *V3* only require a ciphertext modulus of only 133 bits. As shown in the figure, rotations dominate the computation in most scenarios.

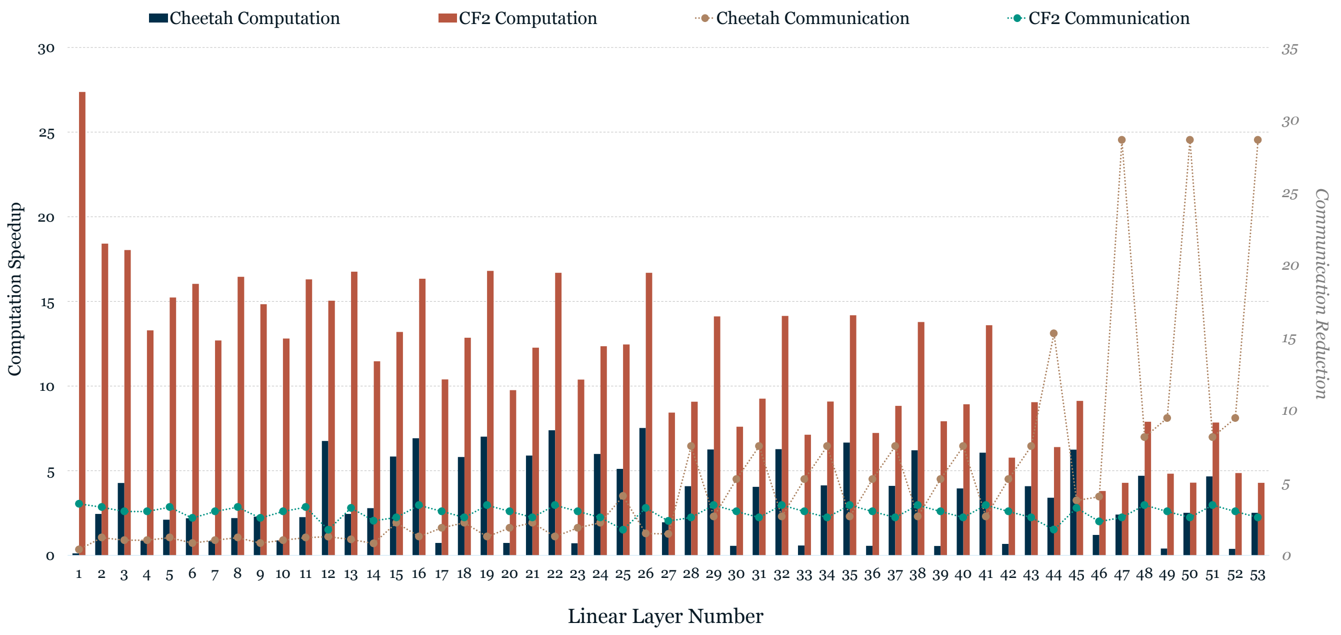


Figure 9: Convolution performance boost with HELiKs over *CrypTFlow2* and *Cheetah* for linear layers in ResNet50.

For the matrix with $d = 2^6$, the computation only involves a single multiplication. For large matrices, the computation is dominated by multiplications and we see significant gains from the NTT pre-computation. Finally, with the use of symmetric key HE we are able to reduce the bandwidth used during communication.

In Figure 8, we show the speedup and communication reduction of HELiKs over *CrypTFlow2* and *GALA*. HELiKs achieves a speedup of up to 51× over *CrypTFlow2* and up to 32× over *GALA*. The total amount of data sent and received during the computation with HELiKs is 2.4× lower than *CrypTFlow2* and 1.5× lower than *GALA*.

6.2 3D Convolution

In this section we evaluate the performance of 3D convolutions in HELiKs against the implementations of *CrypTFlow2* [40] and *Cheetah* [24]. Again, the polynomial modulus is set to $n = 8192$ and the plaintext modulus is set to $P = 4293918721$ for HELiKs and *CrypTFlow2* to achieve a precision of 15 bits for the whole operation. For HELiKs, we use a ciphertext modulus of $Q = 133$ bits and for *CrypTFlow2* we use a ciphertext modulus of 218 bits. *Cheetah* is run with the default parameters reported in the paper. All experiments in this section are run with 4 threads on the system used for the experiments. We do not compare with *GALA* since the algorithm presented in the paper is exactly the same as in *CrypTFlow2*. Moreover, the HE parameters reportedly used in *GALA* for their experiments, polynomial modulus degree $n = 2048$ and 20-bit plaintext modulus, are inapplicable with the algorithm presented in the paper [46] since it requires that the ciphertext has at least $2 \times h \cdot w$ slots. This condition does not hold for the models reported as well as the individual convolution benchmarks reported in the paper. In light of these inconsistencies, we do not consider the numbers reported in the paper [46] for our evaluation of HELiKs.

In Figure 9, we show the computation speedup as well as the reduction in the total bandwidth used for the communication for all the linear layers in the ResNet50 model with HELiKs over *CrypTFlow2* and *Cheetah*. With respect to *CrypTFlow2*, HELiKs achieves a speedup of up to 28× with a minimum speedup of at least 4×. Most of the gains for HELiKs come from the initial layers where the size of the input image is high and consequently, $c_n = h \cdot w/n$ is very low, leading to a much lower number of total prime rotations (refer Table 1) performed during the computations. Though our gains for the later layers is low, HELiKs achieves a cumulative speedup of 7.5× over *CrypTFlow2* for all the layers combined. HELiKs takes just 60 seconds to run whereas *CrypTFlow2* take almost 445 seconds for all the layers in ResNet50. In terms of bandwidth used during communication, HELiKs is 1.8−3.6× better than *CrypTFlow2*. In total, HELiKs requires communication of only 482MB whereas *CrypTFlow2* requires 1.472 GB of data for all layers in ResNet50.

With respect to *Cheetah*, HELiKs is almost 7.52× faster for some layers but for a few layers, it takes almost 8.64× longer. On the whole, HELiKs is still almost 2× faster than *Cheetah* which takes 117 seconds for all the linear layers in ResNet50. In terms of communication, HELiKs is almost 29× lower than *Cheetah* for some layers but for a few layers it is 2.5× higher. For the complete model, HELiKs still requires 2.2× lower amount of data to be communicated compared to *Cheetah* which communicates about 1.052 GB of data. This shows that in spite of the *Cheetah* framework requiring extensive patches to the SEAL Library, as discussed in Section 2.6, HELiKs outperforms it in terms of both computation as well as communication.

6.3 End-to-End Inference

In Figure 10, we illustrate the comprehensive runtimes and communication overheads when executing secure inferences using a ResNet50 DNN. We evaluate three frameworks: *CrypTFlow2*, *Cheetah*, and our proposed system, HELiKs, which integrates *CrypTFlow2*'s protocols, especially for secure non-linear operations over the ring $\mathbb{Z}_P$.

***CrypTFlow2* Performance:** *CrypTFlow2* completes the inference in a total of 570 seconds, involving a communication volume of 29.79 GB. Notably, the HE computation for the linear layers dominates the runtime, accounting for 78% of it. This contrasts with the non-linear layers which are comparatively faster, benefiting from legacy cryptographic methods that have seen significant optimizations over the past three decades. The computation's fixed-point arithmetic nature makes truncation critical to control data expansion. Emphasizing accurate truncation, *CrypTFlow2* incurs substantial communication costs, with 17.75 GB of data transmission dedicated solely to truncation during the secure inference on ResNet50. The encrypted results produced by the *CrypTFlow2* framework precisely match the computations performed on the plaintexts in the clear.

***Cheetah*'s Innovations:** *Cheetah* introduces novel HE kernels for linear layers, achieving a reduced runtime of 273 seconds. Their unique encoding strategy frees them from the natural plaintext limitations of RLWE schemes, the ring $\mathbb{Z}_P$ ($\mathbb{R}$ in the case of CKKS), thus enabling more efficient protocols for non-linear operations over l-bit integers ($\mathbb{Z}_{2^l}$). The real standout in *Cheetah*'s strategy is the marked reduction in communication. They leverage DNNs' inherent resilience to noise and employ a lightweight approximate truncation by removing the constraint of correcting the error in the least significant bit. Prior work [14] had already shown that this error does not affect the accuracy of the DNN, and due to this, *Cheetah* demonstrates significant savings in communication costs with approximate truncation protocol only requiring a mere 345.4 MB. Furthermore, their decision to employ VOLE style OTs rooted in the silent-OT extension [6], as opposed to the IKNP-based OT extension [28] utilized by *CrypTFlow2*, further bolsters communication efficiency. Silent-OT extension only requires transmission of short seeds which are expanded locally to yield the correlated randomness. While this cuts the communication cost logarithmically, it involves much more local computation.

HELiKs + *CrypTFlow2*: Our system, HELiKs, was constructed within the Secure and Correct Inference (SCI) Library, embedded in the *CrypTFlow* framework [34]. Our prime focus is on linear layers, allowing HELiKs to substantially expedite the HE computation time and overall runtime. Using HELiKs, the secure inference on a ResNet50 model is accomplished in 183 seconds. This performance is notably 3.1× swifter than *CrypTFlow2* and 1.5× faster than *Cheetah*. However, in terms of communication, HELiKs closely mirrors *CrypTFlow2*, as it also shoulders the faithful truncation's communication cost.

7 CONCLUSION

Overall, HELiKs significantly reduces runtime and communication overheads for a wide range of applications requiring secure matrix multiplication and convolutions. Adhering to the design principles

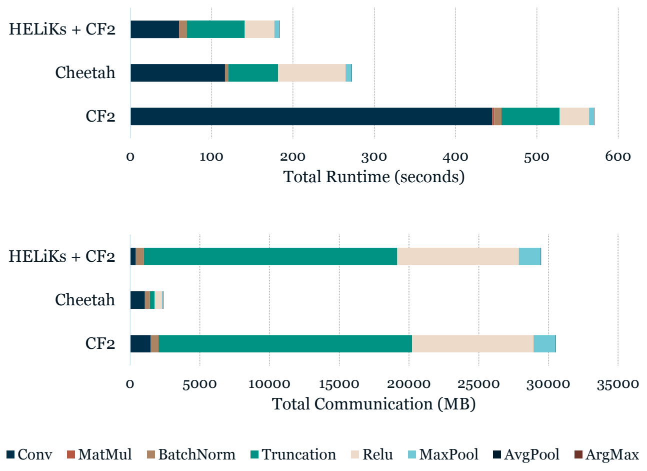


Figure 10: Runtimes and communication overheads for privacy-preserving inference on a pre-trained ResNet50 DNN using *CrypTFlow2*, *Cheetah*, and HELiKs with *CrypTFlow2*.

of, *Consistency*, *Integrity*, *Compatibility* and *Performance*, HELiKs is able to offer state-of-the-art performance with plug-and-play kernels for any HE scheme that do not require any modifications to existing libraries. By providing efficient and secure computation, HELiKs can be a valuable tool for preserving privacy in various collaborative scenarios. For future work, we intend to focus on DNN computation and apply and extend HELiKs to a diverse set of model architectures.

REFERENCES

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[2] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915 https://eprint.iacr.org/2022/915.

[3] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. 2020. Secure large-scale genome-wide association studies using homomorphic encryption. Cryptology ePrint Archive, Paper 2020/563. https://doi.org/10.1073/pnas.1918257117 https://eprint.iacr.org/2020/563.

[4] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. 2020. MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference. Cryptology ePrint Archive, Paper 2020/721. https://eprint.iacr.org/2020/721 https://eprint.iacr.org/2020/721.

[5] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. Cryptology ePrint Archive, Paper 2019/947. https://eprint.iacr.org/2019/947 https://eprint.iacr.org/2019/947.

[6] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. 2019. Efficient Pseudorandom Correlation Generators: Silent OT Extension and More. In *Advances in Cryptology – CRYPTO 2019 - 39th Annual International Cryptology Conference, Proceedings (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*, Daniele Micciancio and Alexandra Boldyreva (Eds.). Springer Verlag, Germany, 489–518. https://doi.org/10.1007/978-3-030-26954-8_16

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference.* 309–325. https://eprint.iacr.org/2011/277.pdf

[8] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science*. 97–106. https://eprint.iacr.org/2011/344.pdf

[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*. 409–437. https://eprint.iacr.org/2016/421.pdf

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2018. TFHE: Fast Fully Homomorphic Encryption over the Torus. Cryptology ePrint Archive, Paper 2018/421. https://eprint.iacr.org/2018/421 https://eprint.iacr.org/2018/421.

[11] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication. Cryptology ePrint Archive, Paper 2021/1116. https://eprint.iacr.org/2021/1116 https://eprint.iacr.org/2021/1116.

[12] Anamaria Costache, Benjamin R. Curtis, Erin Hales, Sean Murphy, Tabitha Ogilvie, and Rachel Player. 2022. On the precision loss in approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2022/162. https://eprint.iacr.org/2022/162 https://eprint.iacr.org/2022/162.

[13] Anamaria Costache, Kim Laine, and Rachel Player. 2019. Evaluating the effectiveness of heuristic worst-case noise analysis in FHE. Cryptology ePrint Archive, Paper 2019/493. https://eprint.iacr.org/2019/493 https://eprint.iacr.org/2019/493.

[14] Anders Dalskov, Daniel Escudero, and Marcel Keller. 2019. Secure Evaluation of Quantized Neural Networks. Cryptology ePrint Archive, Paper 2019/131. https://eprint.iacr.org/2019/131 https://eprint.iacr.org/2019/131.

[15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. Technical Report MSR-TR-2016-3. https://www.microsoft.com/en-us/research/publication/cryptonets-applying-neural-networks-to-encrypted-data-with-high-throughput-and-accuracy/

[16] Léo Ducas and Daniele Micciancio. 2014. FHEW: Bootstrapping Homomorphic Encryption in less than a second. Cryptology ePrint Archive, Paper 2014/816. https://eprint.iacr.org/2014/816 https://eprint.iacr.org/2014/816.

[17] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144 https://eprint.iacr.org/2012/144.

[18] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph. D. Dissertation. Stanford University. https://crypto.stanford.edu/craig/

[19] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2011. Fully Homomorphic Encryption with Polylog Overhead. Cryptology ePrint Archive, Paper 2011/566. https://eprint.iacr.org/2011/566 https://eprint.iacr.org/2011/566.

[20] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (New York, New York, USA) *(STOC '87)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[21] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. Cryptology ePrint Archive, Paper 2014/106. https://eprint.iacr.org/2014/106 https://eprint.iacr.org/2014/106.

[22] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481 https://eprint.iacr.org/2020/1481.

[23] David Heath and Vladimir Kolesnikov. 2021. One Hot Garbling. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 574–593. https://doi.org/10.1145/3460120.3484764

[24] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 809–826. https://www.usenix.org/conference/usenixsecurity22/presentation/huang-zhicong

[25] Siam Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. 2020. TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit. Cryptology ePrint Archive, Report 2020/1181. https://eprint.iacr.org/2020/1181.

[26] Siam Umar Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. 2021. COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3266–3281. https://doi.org/10.1145/3460120.3484797

[27] 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)*. USENIX Association, Baltimore, MD, 1651–1669. https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[28] Vladimir Kolesnikov and Ranjit Kumaresan. 2013. Improved OT Extension for Transferring Short Secrets. Cryptology ePrint Archive, Paper 2013/491. https://eprint.iacr.org/2013/491 https://eprint.iacr.org/2013/491.

[29] Kristin Lauter, Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. 2021. Password Monitor: Safeguarding passwords in Microsoft Edge. Microsoft Research Blog. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/

[30] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. 2008. SWIFFT: A Modest Proposal for FFT Hashing. In *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5086)*. Springer, 54–72. https://doi.org/10.1007/978-3-540-71039-4_4

[31] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2012. On Ideal Lattices and Learning with Errors Over Rings. Cryptology ePrint Archive, Paper 2012/230. https://eprint.iacr.org/2012/230 https://eprint.iacr.org/2012/230.

[32] Samir Jordan Menon and David J. Wu. 2022. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy (SP)*. 930–947. https://doi.org/10.1109/SP46214.2022.9833700

[33] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference System for Neural Networks. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice* (Virtual Event, USA) *(PPMLP'20)*. Association for Computing Machinery, New York, NY, USA, 27–30. https://doi.org/10.1145/3411501.3419418

[34] Microsoft Research India (mpc msri). 2022. CrypTFlow: An End-to-end System for Secure TensorFlow Inference. https://github.com/mpc-msri/EzPC/tree/52c7a779db0b92ea68e57095972c5edf7c6262e3.

[35] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. Cryptology ePrint Archive, Paper 2021/1081. https://eprint.iacr.org/2021/1081 https://eprint.iacr.org/2021/1081.

[36] Sean Murphy and Rachel Player. 2019. A Central Limit Framework for Ring-LWE Decryption. Cryptology ePrint Archive, Paper 2019/452. https://eprint.iacr.org/2019/452 https://eprint.iacr.org/2019/452.

[37] Michael O. Rabin. 2005. How To Exchange Secrets with Oblivious Transfer. Cryptology ePrint Archive, Paper 2005/187. https://eprint.iacr.org/2005/187 https://eprint.iacr.org/2005/187.

[38] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. 2022. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. In *2022 IEEE Symposium on Security and Privacy (SP)*. 576–595. https://doi.org/10.1109/SP46214.2022.9833697

[39] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SIRNN: A Math Library for Secure RNN Inference. Cryptology ePrint Archive, Paper 2021/459. https://eprint.iacr.org/2021/459 https://eprint.iacr.org/2021/459.

[40] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 325–342. https://doi.org/10.1145/3372297.3417274

[41] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 26–39. https://doi.org/10.1109/HPCA51647.2021.00013

[42] Lawrence Roy. 2022. SoftSpokenOT: Communication-Computation Tradeoffs in OT Extension. *IACR Cryptol. ePrint Arch.* 2022 (2022), 192.

[43] SEAL 2023. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA..

[44] N. P. Smart and F. Vercauteren. 2011. Fully Homomorphic SIMD Operations. Cryptology ePrint Archive, Paper 2011/133. https://eprint.iacr.org/2011/133 https://eprint.iacr.org/2011/133.

[45] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 162–167. https://doi.org/10.1109/SFCS.1986.25

[46] Qiao Zhang, Chunsheng Xin, and Hongyi Wu. 2021. GALA: Greedy ComputAtion for Linear Algebra in Privacy-Preserved Neural Networks. *CoRR* abs/2105.01827 (2021). arXiv:2105.01827 https://arxiv.org/abs/2105.01827

A HELiKs IN OPENFHE

We incorporated HELiKs into the OpenFHE Library [2], with the BGV scheme. OpenFHE offers a broad spectrum of HE schemes and corresponding operations, including multiplication and key-switching. One challenge, however, is its limited transparency regarding manual adjustments of all HE parameters, especially the coefficient modulus—unlike what SEAL offers. Instead, OpenFHE

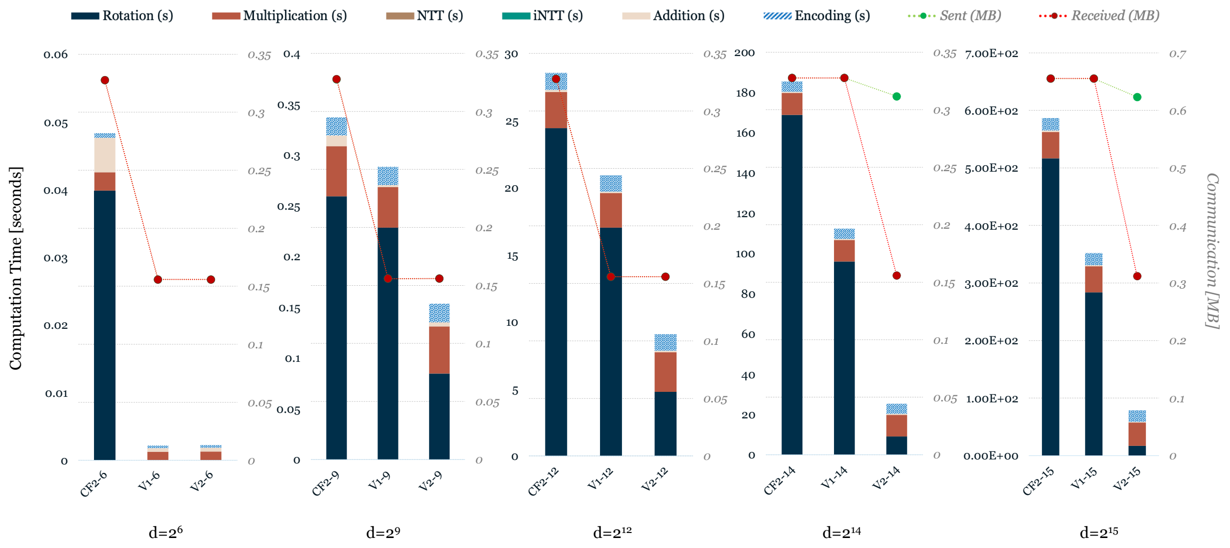
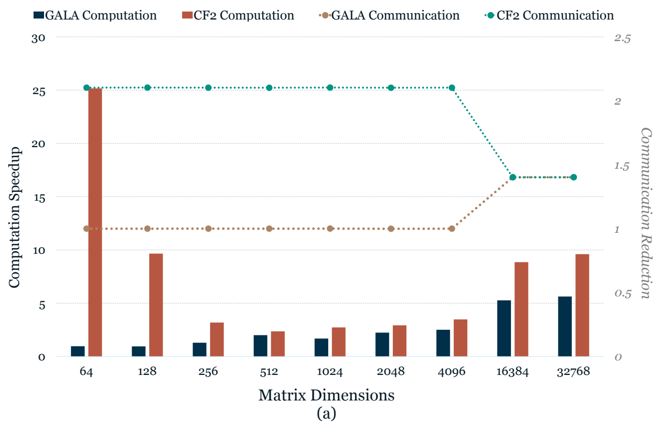


Figure 11: Stacked computation times for all operations and communication measurements for Matrix multiplication using HELiKs (V2) for square matrices of dimension d. The communication metrics represent the client's sent input and received output. All algorithms, *CrypTFlow2* (CF2), *GALA* (V1) and HELiKs (V2), are implemented in OpenFHE with BGV scheme.



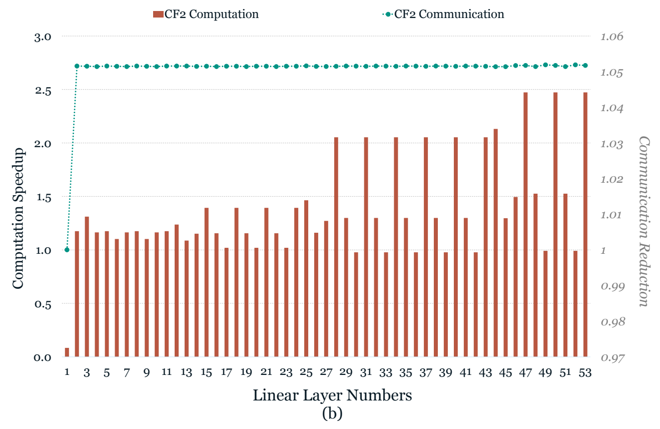


Figure 12: (a) Matrix multiplication efficiency with computation speedups and reduced communication using HELiKs in the OpenFHE Framework with BGV scheme, comparing *CrypTFlow2* and *GALA*. (b) Computation speedup and communication reduction of HELiKs over *CrypTFlow2* for all linear layers of ResNet50 in OpenFHE.

enables users to specify a multiplicative depth and plaintext modulus, from which it automatically determines the coefficient modulus. This auto-selection is grounded in aggressive noise estimation, which often results in choosing parameters much larger than necessary for the computation at hand. Recent research on HE noise estimation [12, 13, 36] indicates a significant discrepancy between noise estimates and empirical observations. In response to this, we've selected a plaintext modulus of $P = 65537$ and multiplicative depths of 2 for *CrypTFlow2* algorithms and 1 for HELiKs. For the sake of security, the smallest polynomial ring size, ensuring 128-bit security for the provided plaintext modulus and multiplicative depth in OpenFHE, was selected: $n = 16384$ for *CrypTFlow2* and $n = 8192$ for HELiKs. Another limitation of OpenFHE is its absence of simple routines to transfer a ciphertext to the NTT domain or for symmetric encryption due to which it is impossible to incorporate the enhancements highlighted in Sections 4.4 and 4.6.

In Figure 11, we delineate the runtime specifics and communication overhead for matrix multiplication over various matrix dimensions, using *CrypTFlow2* (CF2), *GALA* (V1), and HELiKs (V2) without the NTT and Symmetric Key upgrades. All the methodologies are framed within the BGV scheme, and our observations are consistent with the discussion in Section 6.1. Figure 12 (a) illustrates the tangible performance boosts of secure matrix multiplication with HELiKs when compared to *CrypTFlow2* and *GALA* in both computation and communication facets. In Figure 12 (b) we show the performance improvements for secure convolution using HELiKs over *CrypTFlow2* for all the linear layers in a pre-trained ResNet50 DNN model. Given that the convolution computation primarily hinges on multiplications (approximately $10 - 20\times$ the rotations), and considering the absence of NTT pre-computation, multiplication performance enhancements remain unattainable.