FLASH: Towards a High-performance Hardware Acceleration Architecture for Cross-silo Federated Learning

Junxue Zhang[1,2], Xiaodian Cheng[1,2], Wei Wang[2], Liu Yang[1,2], Jinbin Hu[1], Kai Chen[1]
[1]*iSINGLab @ Hong Kong University of Science and Technology* [2]*Clustar*

Abstract

Cross-silo federated learning (FL) adopts various cryptographic operations to preserve data privacy, which introduces significant performance overhead. In this paper, we identify nine widely-used cryptographic operations and design an efficient hardware architecture to accelerate them. However, directly offloading them on hardware statically leads to (1) inadequate hardware acceleration due to the limited resources allocated to each operation; (2) insufficient resource utilization, since different operations are used at different times. To address these challenges, we propose FLASH, a high-performance hardware acceleration architecture for cross-silo FL systems. At its heart, FLASH extracts two basic operators—modular exponentiation and multiplication—behind the nine cryptographic operations and implements them as highly-performant engines to achieve adequate acceleration. Furthermore, it leverages a dataflow scheduling scheme to dynamically compose different cryptographic operations based on these basic engines to obtain sufficient resource utilization. We have implemented a fully-functional FLASH prototype with Xilinx VU13P FPGA and integrated it with FATE, the most widely-adopted cross-silo FL framework. Experimental results show that, for the nine cryptographic operations, FLASH achieves up to 14.0× and 3.4× acceleration over CPU and GPU, translating to up to 6.8× and 2.0× speedup for realistic FL applications, respectively. We finally evaluate the FLASH design as an ASIC, and it achieves 23.6× performance improvement upon the FPGA prototype.

1 Introduction

Training a high-quality machine learning model requires massive data, which is likely to be distributed across different institutions or companies in the real world. However, the increasing concern about data privacy and emerging regulations and lawsuits restrict these data from being collected together in one place for centralized training. To solve this problem, federated learning (FL) has been proposed to enable distributed learning among these *data silos* by performing local computation within a data silo and securely aggregating the intermediate results (*e.g.*, gradients/parameters) to generate a global model without revealing any original data to the outside world [44, 48, 89].

To ensure the security of cross-silo FL, various cryptographic techniques have been used. For example, partially homomorphic encryptions (PHE), *e.g.*, Paillier, have been used to enable parameter computation/aggregation directly on ciphertexts [73]. RSA is used to build the blind signature-based Private Set Intersections (PSI) for sample alignment [45]. In this paper, we perform a comprehensive analysis of existing cross-silo FL applications and identify nine widely-used cryptographic operations, such as encryption/decryption, computation over ciphertexts, *etc*. (more details in §3.1). While preserving privacy, these cryptographic operations significantly degrade the performance (§3.2). For example, our experiments show that these operations cause up to 60.74× performance degradation. The reasons are two-fold: (1) they are of high computational complexity, *e.g.*, Paillier encryption has a $O(2^N)$[1] time complexity; (2) they introduce large number calculations, *e.g.*, additively HE and RSA encryption generate 2048-bit ciphertexts which need to be broken down to multiple 64-bit numbers and executed with limited parallelism on current CPU architecture.

In this paper, we ask: *can we offload these cryptographic operations to dedicated hardware to accelerate cross-silo FL?* Towards answering this question, our first attempt went with GPU. However, as we will reveal in §3.3, the cryptographic operations used in cross-silo FL involve complicated computation stages and dramatically inflate the data, making them inappropriate for GPUs. While GPU is ideal for performing data parallelism over tensors with short numbers, *e.g.*, single-precision floats, it fails to provide efficient pipeline execution for cryptographic operations with large numbers, *e.g.*, 2048-bit integers. The reason is that the limited size of the shared memory within one Streaming Multiprocessor (SM) causes frequent data exchange between external and on-chip shared memory during the pipeline execution, significantly compromising the performance. While it might be feasible to work around the limitation of GPUs, it requires complex mechanisms such as a complex memory orchestration system. Our paper does not take this direction (§6).

To further accelerate cross-silo FL, we seek a more efficient hardware acceleration architecture beyond the existing GPU architecture. To this end, we choose to use FPGA as a prototype and further explore an Application-specific Integrated Circuit (ASIC). We believe such customized hardware architecture will exhibit several desired properties for our purpose. First, it is possible to tailor a hardware architecture for efficient cross-silo FL by customizing the hardware circuits from scratch, so that we can design an optimized fine-grained pipelining with flexible bit-width support for accelerating cryptographic operations. Second, the customized hardware architecture allows us to provide sufficient on-chip memory for storing large numbers used in the processing pipeline for superior performance. However, while promising, we identify

[1]N is the bit-width of the exponent n, and n is the public key in Paillier encryption.

that directly offloading the nine cryptographic operations to the hardware statically will pose two key challenges (§3.3):

- **Inadequate hardware acceleration due to limited resources.** To achieve high performance, one operation may need multiple hardware instances for high parallelism. However, as the hardware resource of a chip is limited, directly offloading all these nice operations to the hardware causes inadequate resources to speed up each operation, leading to suboptimal performance. Our implementation with this approach on a Xilinx VU13P FPGA [23] chip shows that each operation only achieves $\sim 50\%$ acceleration on average.
- **Insufficient resource utilization due to static offloading.** Different FL applications use different cryptographic operations, and within each application, different operations are used at different times. Consequently, statically offloading *all* the operations as a whole results in resource under-utilization because not all the operations are used at all times simultaneously.

To address the challenges, we take a closer look at these nine cryptographic operations and observe that almost all of them build upon two basic operators: *modular exponentiation and modular multiplication.* Based on this observation, we propose FLASH, a high-performance hardware acceleration architecture for cross-silo federated learning (§4). At its core, FLASH uses the majority of hardware resources to implement the two basic operators as high-performance engines to achieve adequate hardware acceleration. We also design fine-grained pipelines with sufficient on-chip memory to improve both the intra- and inter-engine execution efficiency for superior performance. Furthermore, based on these basic engines, FLASH adopts a dataflow scheduling module to dynamically compose these engines into different cryptographic operations on-demand to achieve high resource utilization.

We have provided a down-scale but full functional implementation of FLASH with Xilinx VU13P FPGA [23][2] for prototyping purpose, integrated it with FATE [2]—the most widely-adopted cross-silo FL framework—and evaluated it extensively with real-world FL applications. We compare the performance of FLASH with (1) the vanilla FATE, which uses GMP [19] to implement these cryptographic operations with CPU. GMP provides a highly-optimized implementation for modular multiplication and exponentiation operations, which uses many optimization algorithms, including but not limited to the two mentioned in our paper. We choose Intel Xeon Silver 4114 CPU similar to prior works [76]; (2) the FATE where the cryptographic operations are accelerated by NVIDIA P4[3] GPU [11, 43]. Here we use P4 GPU because it has the closest INT8 TOPS (although $\sim 2\times$ better) as FLASH (we typically use INT8 TOPS to denote the general computation power of a chip). P4 has ~ 20 INT8 TOPS [11]. VU13P has ~ 38.3 INT8 DSP TOPS while reaching peak 891MHz operation frequency [23, 24]. As FLASH uses 300MHz, it achieves ~ 12.9 DSP INT8 TOPS. Moreover, both of them are built with 16nm technology. Finally, with the standard Synopsys software tools (*e.g.*, Design Compiler [14], VCS [16] and Prime Time [15]), we further evaluate the performance of FLASH if implemented as an ASIC.

[2]We use FPGA for prototyping purposes, so we do not consider the price advantages/disadvantages of the VU13P FPGA chip.

[3]We note that latest NVIDIA A100 [9]/H100 [10] may have a better performance than P4/VU13P, however, they are much more expensive while still sharing the architectural deficiency of GPU in general as discussed in §3.3. The focus of FLASH is to pursue a more efficient hardware acceleration architecture for cross-silo FL.

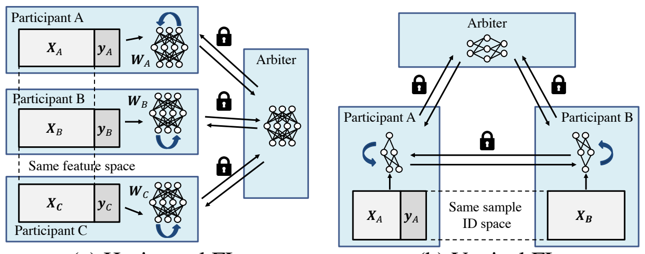


Figure 1: Two paradigms of cross-silo FL.

Overall, some of our key results are as follows:

- Across the nine concrete cryptographic operations (§6.2), FLASH (with FPGA implementation) outperforms CPU and GPU by $10.4\times - 14.0\times$ and $1.4\times - 3.4\times$, respectively.
- Over the nine realistic FL applications (§6.3), FLASH (with FPGA implementation) can consistently outperform CPU and GPU by up to $6.8\times$ and $2.0\times$, respectively.
- Our software evaluation of FLASH as an ASIC with 12nm and 28nm fabrication techniques (§6.5) shows that it can achieve $23.6\times$ and $7.1\times$ *additional* performance improvement upon the FPGA implementation, respectively.

As a final note, we are fully aware that there exist various other privacy-preserving techniques [27, 71, 81, 85]. However, in current industry-level deployments, Paillier and RSA schemes built with the nine cryptographic operations we investigated in this work are, to date, the most widely adopted approach in cross-silo FL systems [2, 41, 49], primarily due to the reason that they can achieve relatively better performance and are easier to use compared to other privacy-preserving schemes. Our goal is to provide plug-and-play acceleration capability for these industry-level cross-silo FL systems.

2 Cross-silo Federated Learning

FL was first proposed by Google to train a language model for keyboard input prediction from massive Android devices without leaking privacy-sensitive data [36, 90]. Recently, FL has evolved from the above cross-device scenarios to collaboratively train machine learning models across different data silos, *i.e.*, cross-silo FL [44, 48, 89]. A data silo is a repository or collection of data under the control of a single entity (*e.g.*, institution, company, *etc.*), and is isolated from other entities due to the ever-improving management regulations or laws [50]. Cross-silo FL enables machine learning among these data silos and supports both vertical and horizontal FL.

Horizontal FL: As shown in Figure 1a, participants in Horizontal FL have different sample spaces but the same feature space, and each participant owns the labels of its samples. In most cases, there is an arbiter for parameter aggregation (the arbiter is a third-party server to assist the FL computations). To train a model, each participant trains local model $\boldsymbol{w}$ with its own samples and encrypts its model weights via PHE (*e.g.*, Paillier [74]). Then all participants send their encrypted weights to the arbiter, and the arbiter directly performs an aggregation over the received ciphertexts to obtain the global model. Eventually, the arbiter sends the aggregated global model to all participants for next-round computation.

Vertical FL: As shown in Figure 1b, participants in Vertical FL have the same sample space but different feature spaces. Only one participant holds the label of the FL task. Before training a model, all participants have to align the samples among different data silos based on the common IDs (similar to joining two tables in a database based on the common IDs). One of the most commonly used algorithms is RSA blind signature-based PSI (RSA-PSI) [45]. After sample alignment, all participants can follow a pre-defined protocol for model training, such as Vertical Linear Regression (Please see Table 1 in [89]) and SecureBoost [42]. During the process, participants use PHE to encrypt their intermediate results and exchange them with other participants or the arbiter.

For interested readers, we have provided a detailed explanation of how cross-silo FL works and its security analysis in Appendix A.

Summary: Various cryptographic technics are used in cross-silo FL. These cryptographic technics are composed of various operations, *e.g.*, data encryption/decryption via additively HE, computation over ciphertext, *etc.*, and we call these operations as *cryptographic operations* in this paper.

3 Analysis of Cryptographic Operations

3.1 Cryptographic Operations

In this section, we present nine cryptographic operations that are widely used in cross-silo FL. Our study is based on the implementation of FATE [2], the most widely adopted open-source framework for cross-silo FL. However, our analysis can also be applied to other cross-silo FL frameworks, *e.g.*, FedLearner [4], TF Encrypted [18], *etc.* Specifically, these nine cryptographic operations are as follows:

O1. Paillier Encryption. This operation uses Paillier [28,73], an additively homomorphic cryptographic algorithm, to encrypt cleartexts into ciphertexts. The operation is mainly used for protecting the intermediate data during model training.

O2. Paillier Decryption. This operation decrypts Paillier ciphertexts into cleartexts. It is used when participants need to decrypt the intermediate results for local model updates in the training phase.

O3. Ciphertext Matrix Addition. This operation is used to add two matrices (or vectors/values) of ciphertexts. As Paillier is used, ciphertexts can be summed up.

O4. Ciphertext & Cleartext Matrix Element-wise Multiplication. This operation performs Hadamard product [58] between ciphertext matrix and cleartext matrix.

O5. Ciphertext & Cleartext Matrix Multiplication[4]. This operation performs matrix multiplication between two matrices of ciphertexts and cleartexts, respectively.

O6. Ciphertext Histogram Building. This operation performs addition operations over encrypted gradient statistics to build decision trees [42].

O7. RSA Encryption/Decryption. This operation conducts encryption or decryption with the public or private key of the RSA algorithm correspondingly. This operation is used when multiple participants try to perform PSI for sample alignment [45].

O8. RSA Blind. This operation blinds the cleartexts with encrypted random numbers.

O9. RSA Unblind. This operation unblinds RSA ciphertexts to remove the random numbers from the ciphertexts.

As shown subsequently (§3.2), these cryptographic operations have a large impact on the performance of cross-silo FL applications due to the following two reasons:

- **High time complexity:** These operations are of high computation complexity, *e.g.*, Paillier encryption has a time complexity of $O(2^N)$. Thus these algorithms are expensive to compute.
- **Large number computation:** Cryptographic operations significantly inflate data, yielding large numbers, *e.g.*, 2048-bit integer. The large number will need to be divided into multiple small numbers and executed on the current CPU architecture with limited parallelism.

3.2 Quantifying the Performance Impact

We now quantify the performance impact of these cryptographic operations with realistic cross-silo FL applications through testbed experiments.

Our testbed is equipped with an Intel Xeon Silver 4114 CPU [5] and 192GB memory. We choose three most widely-adopted vertical FL applications and one horizontal FL application for evaluation: RSA blind signature-based PSI (RSA-PSI), Vertical Logistic Regression (VLR) [53], SecureBoost Decision Tree (SBT) [42] and Horizontal Logistic Regression (HLR). The dataset we use is a commercial dataset from a bank with $\sim 100,000$ samples and 80 features. For vertical FL applications, the dataset is vertically partitioned into two parts: one part contains 80 features while the other contains one feature. We first perform RSA-PSI to obtain the

[4]To efficiently process a large matrix, we will use optimization algorithms such as blocking the matrix and performing multiplications of the blocked matrices. Thus this operation is not a simple combination of matrix element-wise multiplication (**O4**) and addition (**O3**).

Applications & Their Sub-tasks		Involved Operations	w/o CO (s)	w CO (s)	Degradation
RSA-PSI	Computing intersection	**O7, O8, O9**	7.91	20.62	2.60× ↓
VLR (One Epoch) Total: 12.05× ↓	Encrypting logits	**O1**	0	32.05	-
	Aggregating logits	**O3**	0.63	2.04	3.23× ↓
	Computing fore gradients[a]	**O3, O4**	0.74	2.92	3.93× ↓
	Computing gradients	**O3, O4, O5**	3.77	135.96	36.08× ↓
	Decrypting gradients	**O2**	0	0.04	-
	Computing loss	**O1, O3, O4**	4.08	8.22	2.02× ↓
SBT (One Epoch) Total: 3.49× ↓[b]	Encrypting gradients	**O1**	0	54.71	-
	Aggregating gradients	**O3, O6**	12.02	223.91	18.62× ↓
	Split information synchronization	**O2**	3.62	13.03	3.60× ↓
HLR (One Epoch) Total: 60.74× ↓[b]	Computing gradients	**O3, O4, O5**	1.73	177.94	102.80× ↓
	Model update	**O3, O4**	0.0002	0.10	526.10× ↓
	Model re-encryption	**O1, O2**	0	0.69	-

[a] According to Federated Logistic Regression [53], the gradient computation takes two steps: fore gradients computation and gradients computation.
[b] The overall performance degradation of SBT/HLR is smaller than the sum of those sub-tasks because we do not include pure cleartext computation or networking communication sub-tasks in the table.

Table 1: Performance penalty caused by cryptographic operations (CO) with different cross-silo FL applications.

data intersection. Then, we run VLR and SBT over the data intersection, respectively. For horizontal FL, the dataset is horizontally partitioned into two parts, each with $\sim 50,000$ samples. The four applications are executed both with cryptographic operations implemented using GMP (w/ CO) and without cryptographic operations (w/o CO). To implement model training w/o CO, we modify the code of FATE to skip these cryptographic operations. To perform a fine-grained analysis, we also break down these four applications into sub-tasks, and for each sub-task, we show the adopted cryptographic operations. All the applications are executed with ten CPU cores in parallel. Table 1 shows the results, and we make the following observations:

- **Cryptographic operations considerably degrade the performance.** In general, cryptographic operations significantly degrade the performance of cross-silo FL applications. In our experiment, the cryptographic operations cause RSA-PSI, VLR, SBT and HLR to suffer 2.60×, 12.05×, 3.49× and 60.74× performance degradation, respectively. Moreover, the combinations of these cryptographic operations can degrade the performance from $\sim 2.02\times$ to $\sim 526.10\times$.
- **Not all the cryptographic operations are used at all times simultaneously.** Different FL applications use different cryptographic operations, and even within a single application, different sub-tasks use different operations.

3.3 Challenges of Offloading Cryptographic Operations

To accelerate these cryptographic operations, we chose GPU as our first attempt, as it has been widely adopted in various offloading scenarios. However, as our exploration proceeds, we find that the cryptographic operations in cross-silo FL require complicated pipeline computation and significantly inflate the data, posing the following challenges for GPU.

The hardware architecture of GPU is tailored for performing data parallelism over tensors, which are mostly short numbers. However, as we will show in §4.2.1, to efficiently execute cryptographic operations, we have to use several steps to optimize the computation, *e.g.*, Montgomery Modular Multiplication [65], in which pipeline parallelism is needed. Furthermore, massive large numbers should be stored in the shared memory during the pipeline execution. However, these large numbers, *e.g.*, 2048-bit integer numbers, can easily overflow the on-chip memory of the GPU. For example, the amount of shared memory per SM is 96 KB for NVIDIA V100 [13]. No more than 384 2048-bit integer numbers can be stored inside one SM. Therefore, after processing a small amount of data, the GPU has to swap data between the shared memory and external memory, interrupting the pipeline execution. While it is possible to solve the aforementioned limitations, it requires complicated memory orchestration, such as a suitable double-buffering [33], which may pose further challenges.

To this end, our paper does not take this direction but moves one step further beyond the existing GPU architecture, by seeking a more efficient, customized hardware acceleration architecture for cross-silo FL. We envision that with a customized hardware architecture, we can implement fine-grained pipelining for those cryptographic operations with large numbers. The hardware can also support variable bit-widths to match the cross-silo FL scenarios where different public key sizes are used (which yield large numbers with different bit-widths). Furthermore, with a customized design, we are able to invest sufficient on-chip memory for caching large numbers used in the pipeline execution. The data swapping between on-chip and external memory can be part of the pipeline to avoid the performance penalty mentioned above.

In this paper, we follow the rule-of-thumb approach to use FPGA as a prototype and evaluate the potential of ASIC via software tools [30, 51, 87]. However, we confront the following two challenges in our design:

1. Inadequate hardware acceleration due to limited resources. As identified in §3.2, all the cryptographic operations cause a performance penalty, so we should offload all of them to hardware. Furthermore, to realize sufficient accelera-

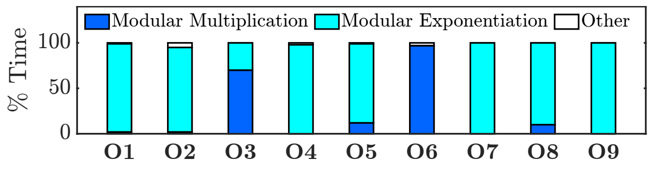


Figure 2: Cryptographic operation computation time analysis.

tion, each operation requires multiple hardware instances of accelerating modules/circuits for high parallelism. However, in practice, the hardware chip has limited resources, and if we naïvely offload all cryptographic operations to the chip, each operation has inadequate resources to be fully accelerated. Taking the DSP resources as an example, our preliminary implementation on VU13P FPGA [23] chip shows that to accelerate Paillier encryption (**O1**) by 2×, we need to use 2630 DSPs. Yet, a high-end FPGA chip, such as VU13P [23], only has 12288 DSPs, leaving < 1365 DSPs for one operation on average[5] (some DSPs are reserved for PCIe, memory controller, *etc*.). Thus, directly offloading *all* operations on VU13P FPGA chip leads to only $\sim 50\%$ acceleration on average. A similar problem also applies to the ASIC design.

2. Insufficient resource utilization due to static offloading. Different from software, hardware function is static after being configured/programmed/taped out, thus it cannot change its function dynamically. Nevertheless, as shown in §3.2, not all the cryptographic operations are used at all times simultaneously. Consequently, if we statically offload all cryptographic operations on the hardware chip, only part of these cryptographic operations is used at a time. Therefore, such static offloading causes low resource utilization and further leads to suboptimal performance.

3.4 Opportunities with Accelerating Basic Operators

To overcome the above challenges, we further take a look at the internal of these nine cryptographic operations. We discover that *all* these operations are composed of two basic operators: modular multiplication and exponentiation. Then we further find that the performance of these operations is mainly decided by the two basic operators.

Paillier Encryption: Given the public key (n,g) and data m $(0 \le m < n)$, the Paillier encryption algorithm takes two steps: (1) selecting a random number r where $0 < r < n$ and $r \in \mathbb{Z}_n^*$; (2) computing ciphertext $c = g^m \cdot r^n \mod n^2$. The formula can also be simplified to $c = (1+mn) \cdot r^n \mod n^2$ by setting $g = (1+n)$. We use $[[\cdot]]$ to denote the Paillier encryption, *e.g.*, $c = [[m]]$.

Homomorphic Addition: Given two plaintext a and b, homomorphic addition guarantees that $[[a]] \bigoplus [[b]] = [[a+b]]$. In Paillier cryptosystem, $[[a]] \bigoplus [[b]]$ is defined as $[[a]] * [[b]]$ mod n^2. The homomorphic addition is used by operations **O3**, **O5** and **O6**.

[5]We will show later that all these nine operations share similar building blocks, thus they require similar resources to implement.

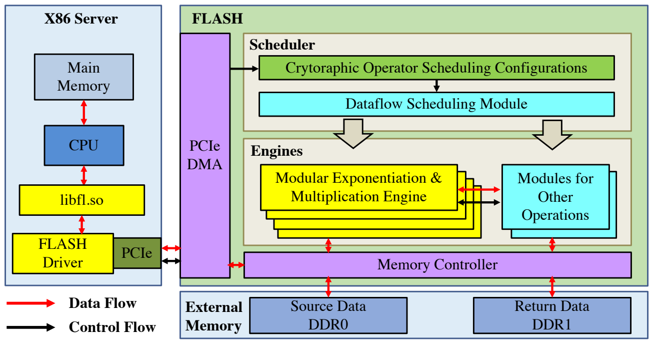


Figure 3: FLASH architecture.

Homomorphic Multiplication: Given a plaintext a and k, the homomorphic multiplication is denoted by $k \cdot [[a]]$. It can be actually regarded as a homomorphic addition: $\Sigma^k [[a]]$. Thus, $k \cdot [[a]] = [[a]]^k \mod n^2$. The homomorphic multiplication is used by operations **O4** and **O5**.

Paillier Decryption: Given the public key (n,g), private key (p,q) and ciphertext c, the Paillier Decryption algorithm can be optimized via Chinese Remainder Theorem (CRT) to reduce the original workload to only about one-quarter of the original decryption algorithm. To use CRT, we define L_p and L_q to be $L_p(x) = \frac{x-1}{p}$ and $L_q(x) = \frac{x-1}{q}$. The decryption algorithm takes the following three steps: (1) computing $m_p = L_p(c^{p-1} \mod p^2) L_p(g^{p-1} \mod p^2)^{-1} \mod p$; (2) computing $m_q = L_q(c^{q-1} \mod q^2) L_q(g^{q-1} \mod q^2)^{-1} \mod q$; and (3) computing plaintext $m = \mathrm{CRT}(m_p, m_q) \mod n$.

RSA-related Operations: The RSA-related operations are used in RSA blind signature-based PSI [45]. It is commonly known that the core of these RSA-related algorithms is either modular multiplication or modular exponentiation.

Through the above mathematical analysis, we find that *all* cryptographic operations used in cross-silo FL are built upon the two basic operators: modular multiplication and exponentiation[6] . Then, we further perform testbed experiments to investigate how these two basic operators impact the performance of the nine original cryptographic operations. The results are shown in Figure 2. Clearly, we find that across *all* nine original operations, the two basic operators occupy $> 95\%$ of the total execution time.

Observation: We can compose all the nine cryptographic operations with these two basic operators, and by accelerating these two basic operators, all the nine original operations used in cross-silo FL applications can be effectively accelerated.

4 The FLASH Design

Inspired by the above observation, we present FLASH, a high-performance hardware acceleration architecture for cross-silo FL. This section describes how we design FLASH in detail. Please note that our design has been fully implemented in our FLASH prototype with FPGAs as well as rigorously evaluated

[6]Appendix B and C provide more details of these operations.

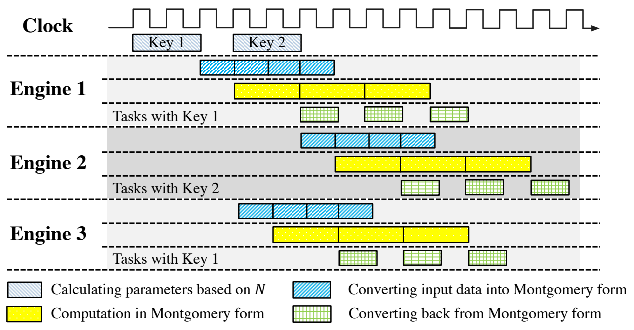

Clock
Key 1
Key 2
Engine 1
Tasks with Key 1
Engine 2
Tasks with Key 2
Engine 3
Tasks with Key 1
Calculating parameters based on N
Converting input data into Montgomery form
Computation in Montgomery form
Converting back from Montgomery form

Figure 4: Pipeline executions of both inter- and intra-engines.

with the Synopsys tools for the ASIC.

4.1 Architecture Overview

Figure 3 shows the overall architecture of FLASH. It contains a hardware acceleration card and an integrated software package. The accelerator card can be plugged into a server via PCIe Gen3×16 interface. The server is installed with cross-silo FL software, *e.g.*, FATE. The software can invoke FLASH's software package to *offload* the cryptographic operations on the card for efficient acceleration.

The idea of our FLASH design is that it does not directly offload all cryptographic operations on the hardware, but leverages the insights of our observation to (1) utilize the limited programmable resource for most performance-critical basic operators: modular exponentiation and multiplication to achieve *adequate acceleration* (§4.2), and (2) design an on-chip dataflow scheduling module to dynamically compose different cryptographic operations on-demand based on these engines, achieving *high resource utilization* (§4.3). In addition, to make FLASH a general solution to support a wide range of cross-silo FL frameworks, our software package provides standard APIs. In this way, different cross-silo FL software can utilize FLASH by harnessing its APIs (§4.4).

4.2 Modular Exponentiation and Multiplication Engines

To implement modular exponentiation and multiplication operators as high-performance engines on hardware, FLASH makes the following design decisions. First, instead of directly offloading the modular exponentiation and multiplication operators, we optimize the algorithms of the two operators to make them suitable for the hardware implementation (§4.2.1). Second, based on the optimized algorithms, FLASH further leverages pipelining technologies to efficiently execute them with high parallelism (§4.2.2). Third, we provide sufficient on-chip memory for pipeline execution and make the data transfer as part of the pipeline to efficiently exchange data between off-chip memory and engines (§4.2.3).

4.2.1 Algorithm Optimization

The mathematical formulas of the two basic operators: modular exponentiation (Equation 1) and multiplication (Equation 2) are as follows:

Algorithm 1 Montgomery Modular Multiplication

```
▷ Given 3 input numbers $X$, $Y$ and $N$, the Montgomery Modular Multiplication outputs $Z = X \cdot Y \cdot R^{-1} \mod N$, where $R$ is a power of 2 and $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.
Input: $X = (X_{d-1}, ..., X_0), Y = (Y_{d-1}, ..., Y_0), N = (N_{d-1}, ..., N_0), N'$, where
    $N' = (-N)^{-1} \mod r$,                          ▷ $N'$ is pre-computed in S1
    $r = 2^w$, $d = \lfloor \log_r N \rfloor + 1$,     ▷ $r$ and $d$ is used to split data
    $\gcd(N, r) = 1$, with $N \times N' \equiv -1 \mod r$
Output: $Z = \text{ModMult}(X, Y, N) = X \times Y \times R^{-1} \mod N$
 1: $Z = (Z_{d-1}, ..., Z_0) = 0$                      ▷ Initialization
 2: for all $i = 0, 1, ..., d-1$ do                     ▷ Loop on $Y$
 3:     $\alpha = [X_0 \times Y_i]_{\text{low}}$
 4:     $\beta = \alpha + Z_0$
 5:     $q = [\beta \times N']_{\text{low}}$
 6:     $\delta_1 = \alpha + [q \times N_0]_{\text{low}}$
 7:     $\delta_2 = \delta_1 + Z_0$
 8:     $Z_0 = [\delta_2]_{\text{low}}$
 9:     $C = [\delta_2]_{\text{high}}$
10:     for all $j = 1, 2, ..., d-1$ do                 ▷ Loop on $X$
11:         $\delta_0 = [X_{j-1} \times Y_i]_{\text{high}} + [q \times N_{j-1}]_{\text{high}}$
12:         $\delta_1 = \delta_0 + [X_j \times Y_i]_{\text{low}} + [q \times N_j]_{\text{low}}$
13:         $\delta_2 = \delta_1 + Z_j + C$
14:         $Z_{j-1} = [\delta_2]_{\text{low}}$
15:         $C = [\delta_2]_{\text{high}}$               ▷ Carry higher bits
16:     end for
17:     $Z_{d-1} = C$
18: end for
19: if $Z \geq N$ then
20:     $Z = Z - N$
21: end if
22: return $Z$
```

$$P = m^e \mod n \qquad m, e, n \in \mathbb{Z}^+ \tag{1}$$

$$P = ab \mod n \qquad a, b, n \in \mathbb{Z}^+ \tag{2}$$

When used in cryptographic operations, all the numbers a, b, m, n are large numbers, leading to high computation complexity. Therefore, before designing FLASH's engines, we first apply some commonly-used optimization strategies in the cryptographic research community to optimize the two basic operators, including Binary Exponentiation [52] and Montgomery Modular Multiplication [65], *etc*. The advantages of using these optimization strategies are: (1) lowering the number of multiplications used in modular exponentiation from $O(2^N)$ to $O(N)$ (N is the bit-width of e), and (2) replacing the modulo operation with the hardware-friendly bit-shifting operation. Appendix D provides details of how they work.

After applying these optimization methods, FLASH's modular exponentiation and multiplication operators require the following four stages to complete the computation:

S1. Preparing common data needed in Montgomery space based on the input data n. Since, in both Paillier and RSA cryptosystems, n is decided by the public key, we can reuse these prepared data for all computations with the same public key. This is common in cross-silo FL applications as they use one key for all cryptographic operations within one application.

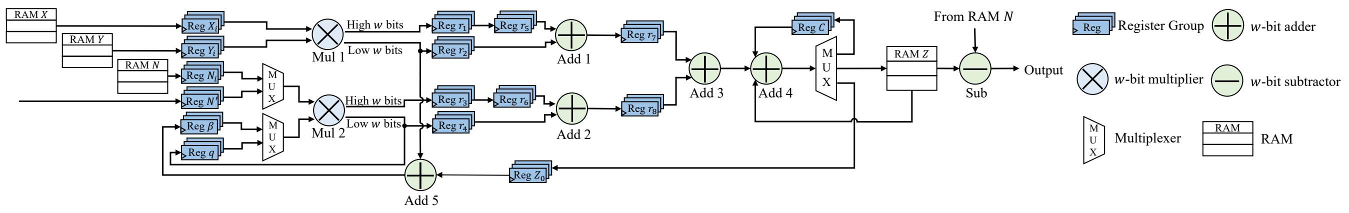


Figure 5: The Montgomery Modular Multiplication engine circuit design. Our circuit uses two multipliers and four on-chip RAMs for efficient pipelining. For more details of how this circuit works, please refer to Appendix E.

S2. Performing input data pre-precessing and converting them into Montgomery space.

S3. Performing computation in Montgomery form. Major operations in this stage include large-number multiplication, addition, and bit-shifting. No modulo operation is needed.

S4. Converting all output data of the operators out of Montgomery space.

4.2.2 Pipelining

We next introduce how FLASH efficiently performs the above four computation stages via inter- and intra-engine pipelining.

Inter-engine pipelining: To enable inter-engine pipelining, FLASH employs an engine pipeline stage manager to control the execution strategies for different stages. Figure 4 gives an overview of how these stages are pipelined. First, FLASH reserves S1 as a standalone stage, which can be executed in advance once it obtains the public key. Second, for all computation tasks with the same public key (Engine 1 and 3 in Figure 4), they can be executed in parallel once their data preparation is completed (S1). The start time of these engines may have a small gap of several clock cycles because FLASH adopts a round-robin strategy to dispatch stage executions. Third, for tasks with different keys (Engine 2 in Figure 4), they can be executed independently.

Intra-engine pipelining: FLASH further performs intra-engine pipelining within the most computation-intensive stage S3 to accelerate the stage's internal execution. The key design goals are: (1) FLASH should support variable bit-widths thus the application can choose the key length based on their security requirements; (2) the hardware resources should be fully utilized to achieve superb performance.

To achieve the first goal, FLASH builds an efficient pipeline that processes data based on radix-2^w arithmetic [60] (we use $w = 64$ in FLASH's implementation). Given any input data, we split it into d w-bit integers. For example, $d = 16$ when bit-width of X is 1024, and $d = 32$ when bit-width of X is 2048. Theoretically, the pipeline can be adapted for input data with any bit-width as long as the bit-width is or can be extended by complementary zeros to the integer multiple of d. After data splitting, the complete algorithm of S3 (Montgomery Modular Multiplication) is shown in Algorithm 1. Note, compared to the original Montgomery Modular Multiplication (shown in Algorithm D.2 in Appendix D.2), we make the following optimizations to make the algorithm more hardware-friendly: (1) the computation of S (*i.e.*, line 6 in Algorithm D.2) is separated into computations of lower w bits and higher w bits so that the bit-width required in operations (*e.g.*, addition) is halved; (2) the first iteration of the inner loop where $j = 0$ is unrolled to remove the conditionals in the original algorithm (*i.e.*, line 7 to 9 in Algorithm D.2) and keep the consistency of computation logic.

In Algorithm 1, the most computation-intensive operations are the multiplications of $X_j \times Y_i$ and $q \times N_j$ respectively (both operations require d^2 w-bit multiplications). Moreover, the data required in these two multiplications are totally independent. Therefore, we use two multipliers (one 64-bit multiplier consists of 32 DSP48E2 slices [22] on our FPGA prototype), Mul 1 and Mul 2, for these two multiplication operations and reuse them to execute the rest of the multiplication operations as well (operations assigned to Mul 1 and Mul 2 are marked with red and green respectively in Algorithm 1). Since the multiplier can continuously process data, to fully utilize the multiplier, we have the following design decisions. First, we design a circuit to fully pipeline the inner loop (line 10 to 16 of Algorithm 1). The circuit is shown in Figure 5 and due to the limited space, we defer a detailed description of how it works in Appendix E. Second, when the multiplications of i-th iteration finish and some other operations are still under execution, *e.g.*, addition operations in the right part of the Figure 5, FLASH allows direct starting the multiplications in $i+1$-th iteration to minimize the delay between different iterations. We also use Figure 6 to visualize how the operations in S3 are efficiently pipelined.

4.2.3 On-chip & Off-chip Memory

To provide sufficient on-chip memory for efficient pipeline execution, FLASH allocates four memory units for each modular multiplication and exponentiation engine (shown in Figure 5). For our FPGA prototype implementation, we use 4× 36Kbit BRAM units. While the on-chip memory is mainly used for pipeline execution, FLASH further exploits external memory (shown in Figure 3) for input, output and intermediate data storage. To achieve high performance, data exchange between on-chip and off-chip memory is part of the pipeline itself, *i.e.*, when the data at the on-chip memory is consumed, FLASH simultaneously fetches new data from the off-chip memory, so that the data fetching time can overlap with the computation time. As the data fetch time is typically shorter than the computation time, it effectively hides the off-chip

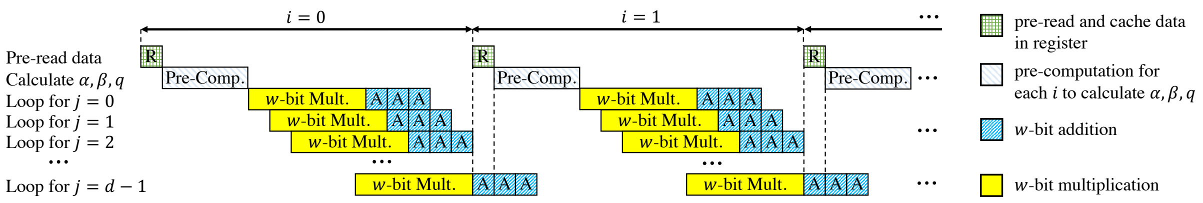


Figure 6: Efficient pipelining of Montgomery Modular Multiplication.

memory access latency, leading to perfect pipelining.

Moreover, we also design a hierarchical data distribution mechanism to solve the design difficulties encountered when manipulating large external memory. Specifically, instead of directly using long paths to send data from the memory controller to all engines, FLASH distributes data at multiple layers. Such a hierarchical mechanism leads to two advantages: (1) low design difficulties: since a small fan-out with short logical paths is required in each layer, the placement of the circuits is much easier; (2) improved performance: because the time constraints of the short path are much easier to meet, such method can allow a high operating frequency. Due to the limited space, we defer a detailed introduction in Appendix F.

4.3 Dataflow Scheduling

We now introduce how FLASH composes various cryptographic operations over basic engines through dataflow scheduling. First, we show how our engines can work at different modes (§4.3.1). Then, we present how different cryptographic operations are constructed by combining particular engines (§4.3.2).

4.3.1 Dynamic Engine Switching

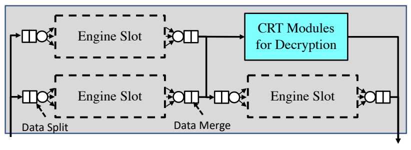
To build various cryptographic operations over basic engines, FLASH needs to enable dynamic engine switching between modular exponentiation and multiplication. Mathematically, modular exponentiation can be realized by performing modular multiplication multiple times. Thus, FLASH leverages a hardware control module to achieve it without reconfiguring hardware (it is almost impossible to reconfigure the ASIC). Specifically, to accelerate modular exponentiation, FLASH constructs a dataflow loop over the multiplication engine multiple times. In contrast, when the engine needs to execute modular multiplication, FLASH directs the dataflow through the modular multiplication engine once. While the design works well for most cryptographic operations that use either modular exponentiation or multiplication, it cannot directly support operations that simultaneously require both modular exponentiation and multiplication, *e.g.*, Paillier encryption (**O1**), matrix multiplication (**O5**), in which FLASH has to decide the ratio of engines in different modes.

How to decide the ratio? We use domain knowledge in cross-silo FL applications to decide the ratio. Taking matrix multiplication operation (**O5**) as an example, it first performs ciphertexts and cleartexts multiplication (requires modular exponentiation) and then ciphertexts addition (requires modular multiplication). Considering the modular exponentiation, the exponent e is a cleartext, which has a common bit-width of 64. As mentioned in §4.2.1, since we use Binary Exponentiation to optimize the modular exponentiation, the number of modular multiplication required may vary from 64 to 127 depending on the specific value of cleartext. On average, 96 modular multiplications are required. Thus, the throughput of the modular exponentiation should be $\sim 1/96$ of modular multiplication. Based on this, we can adjust the ratio of the engines working in different modes to make the throughput of both modular exponentiation and modular multiplication balanced. In this way, the hardware resources can be efficiently utilized and no engines will sit idle.

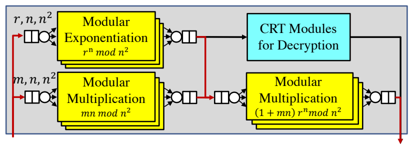
4.3.2 Building Cryptographic Operations

As shown in Figure 7a, the core idea of dataflow scheduling is to use an on-chip controller to dynamically determine: (1) which data paths (they are logical paths that do not reflect the physical wiring) should be active, and (2) what to put in the engine slots, based on which operation is offloaded. Each engine slot contains one data splitting module and one data merging module to distribute data to different engines and aggregate results from these engines, respectively. These data splitting and merging modules have physical wires to all engines, and by configuring which wires are active, we can logically assign engines to these engine slots. We also design a hierarchical data distribution mechanism, as mentioned in §4.2.3, for better performance.

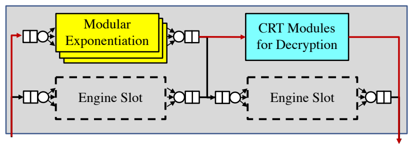
Specifically, we can construct a Paillier encryption operator by following the dataflow scheduling strategy shown in Figure 7b. As mentioned in §3.4, the Paillier encryption follows equation: $c = (1+mn) \cdot r^n \mod n^2$. So we can distribute the data m, n, n^2 to modular multiplication engines (these engines are denoted E_1) to calculate $r_1 = mn \mod n^2$ and distribute the data r, n, n^2 to modular exponentiation engines (these engines are denoted E_2) to calculate $r_2 = r^n \mod n^2$. Then the results can be further sent to modular multiplication engine (these engines are denoted E_3) to calculate $(1+r_1) \times r_2 \mod n^2$. Please note that the $1+r_1$ is completed in the input data pre-precessing stage (S2 in §4.2.1) with a lightweight dedicated hardware module. The ratios of E_1, E_2 and E_3 are determined through the strategies discussed above, thus we can assign particular numbers of engines to these engine slots. Similarly, Figure 7c shows the dataflow used in Paillier decryption. In this case, FLASH uses other modules besides modular exponentiation and multiplication engines to real-



(a) All available dataflow scheduling paths.



(b) Dataflow for encryption.



(c) Dataflow for decryption.

Figure 7: Dataflow scheduling. Black arrow indicates all available paths for dataflow scheduling while red arrow indicates the active paths for a particular cryptographic operation. Each engine slot can have multiple engines.

ize the decryption operation. In particular, we use CRT to optimize the decryption algorithm as discussed in §3.4, thus FLASH implements several CRT modules to accelerate this operation. We put all engines, working in modular exponentiation mode, in the top left corner engine slot to achieve high resource utilization.

4.4 Software Integration

While our current implementation integrates FLASH with FATE, the design of FLASH is generic and works with other cross-silo FL systems/frameworks. They can harness FLASH by using the standard APIs.

For easy integration, FLASH provides Python NumPy-similar APIs as shown in Listing 1. The Python APIs are also wrappers of the C/C++ library: `libfl.so`. Besides providing standard APIs, the `libfl.so` library manipulates the status of all installed FLASH accelerators, such as temperature, workload, *etc*.

By using these APIs, users can easily create encrypted scalar, vector, or matrix via Paillier or RSA encryption method. Users can further perform homomorphic addition and multiplication operations over these data. To reduce the data exchange between the FLASH accelerator and the host, we put the computation results on the off-chip memory unless the `get` API is used. As shown in §4.2.3, the data exchange between on-chip and off-chip memory is efficiently pipelined, leading to better end-to-end performance. Moreover, since `libfl.so` works in a stateless way, it can be easily scaled out to support different tasks from various FL applications.

Listing 1: FLASH's NumPy-like APIs

```python
import flash_np as np
# Generating two Paillier-encrypted arrays accelerated by FLASH
x1 = np.array([1, 2, 3], encryption="paillier")
x2 = np.array([4, 5, 6], encryption="paillier")

x3 = x1 + x2 # Homomorphic addition
x4 = np.array([1, 2, 3], encryption=None)
x5 = x4 * x1 # Ciphertext & cleartext multiplication

x3.decrypt() # Decrypting the ciphertext
x5.decrypt()

x3.get() # Transferring the data from accelerator to host
x5.get()
```

Multi-accelerator Support: The server-side software also enables multi-accelerator support. If there are multiple FLASH accelerators on the server, when applications invoke the APIs, `libfl.so` will break the task into multiple sub-tasks and dispatch them to multiple accelerators. The dispatching strategy is the least workload first and can be configured to use different strategies, such as round-robin.

5 Implementation

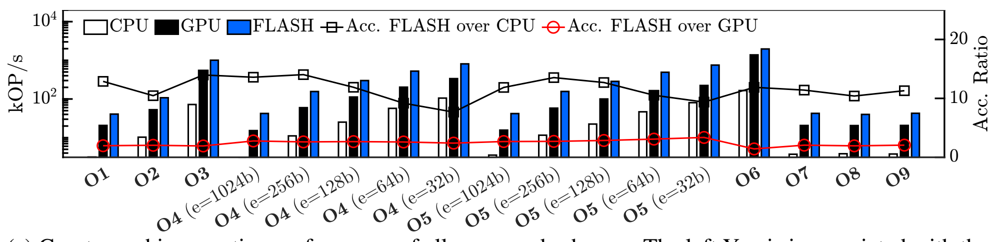
Prototype Implementation with FPGA: We implement FLASH with FPGA using $\sim 30,000$ lines of Verilog [84] code. We use Xilinx Virtex UltraScale+ VU13P chip [23] in our implementation. FLASH implements 300 modular exponentiation and multiplication engines with the chip. As the VU13P chip consists of four dies, we need to distribute components on different dies in a balanced way to achieve high resource utilization. In our implementation, we first place large modules such as PCIe and DDR controllers on separate dies with the consideration that they should be close to the location of their corresponding I/O pins. Then, with the settle-down of large modules, we place different numbers of engines on different dies to make the resource utilization of each die approximately the same to avoid the possibility of local congestion. As a final note, the operation frequency of our FPGA implementation is 300MHz while we achieved $\sim 88\%$ DSP resource utilization, which, to the best of our knowledge, is relatively high in FPGA's industry [92].

Server-side Software Stack Implementation: Our implementation of FLASH's server-side software contains $\sim 10,000$ lines of C/C++ and Python code. This includes modifications of FATE to harness FLASH's acceleration capacity. We mainly modify the `federatedml` module [3] in FATE by replacing normal collection operations with FLASH's NumPy-like APIs. We further use Xilinx DMA (XDMA) IP Reference driver [21] for high-performance direct memory access through the PCIe interface.

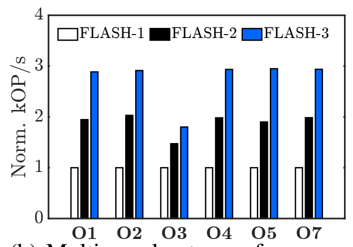
Evaluating FLASH as ASIC: We leverage multiple standard software to assess the FLASH design as an ASIC. Specifically, we first use Synopsys Design Compiler [14] to convert FLASH's design logics into physical implementations, *i.e.*, netlist, over both 12nm and 28nm technology libraries. Then, we use Synopsys VCS [16] to verify that the generated netlist functions correctly and use Synopsys Prime Time [15] for static timing analysis to validate that the netlist satisfies all timing constraints. More evaluation results of the ASIC performance will be discussed in §6.5.

6 Evaluation

In this section, we first present our evaluation methodology (§6.1). Then we show that for the nine cryptographic



(a) Cryptographic operation performance of all compared schemes. The left Y-axis is associated with the bar chart and is in the log scale. The right Y-axis is associated with the line chart.



(b) Multi-accelerator performance with selected operations.

Figure 8: Performance of cryptographic operations.

	FPGA	Logic Cells	DSP	Public Key $N = 1024$bit		Public Key $N = 2048$bit	
				Encryption (kOP/s)	**Decryption** (kOP/s)	**Encryption** (kOP/s)	**Decryption** (kOP/s)
FLASH	VU13P	3,780,000	12,288	40.706	107.707	6.033	19.373
PCP [80]	7VX330T [25]	326,400	1,120	1.40625	1.15625	-	-
HLS [91]	VU9P [23]	2,586,000	6,840	5.238	5.238	-	-
SoC [31]	ZU9EG [26]	600,000	2,520	-	-	0.561	0.563

Table 2: Resource consumption & performance comparison among FLASH and other Paillier accelerators.

	Models	**Datasets**
Vertical FL	RSA-PSI [45] VLR [53] SBT [42]	CreditCard [1]
Horizontal FL	HLR MLP LSTM [57] DenseNet169 [59] ResNet50 [55] VGG16 [82]	CreditCard [1] FMNIST [86] Shakespeare [17] Cifar-10 [66]

Table 3: Models & datasets used in evaluation of FLASH.

operations, FLASH achieves up to 14.0× and 3.4× acceleration over CPU and GPU (§6.2), translating up to 6.8× and 2.0× speedup for realistic FL applications (§6.3), respectively. Finally, we evaluate the performance of FLASH as an ASIC (§6.5).

6.1 Methodology

Environment Setup: We use two X86 servers in our setup. Each server is equipped with a Mellanox CX-4 NIC [6] and connected to a Mellanox SN2100 [7] switch via 40Gbps DAC cables. To reflect realistic networking situations in real-world cross-silo FL deployments, we use `netem` [8] to limit the networking bandwidth to be 50Mbps [7]. As to other hardware configurations, each server is equipped with one Intel Xeon Silver 4114 CPU [5], 192GB memory, and one FLASH acceleration card (In the multi-accelerator experiment, each server will be installed with multiple acceleration cards). We deploy FATE v1.5 as the cross-silo FL framework.

Schemes Compared: We mainly compare the performance achieved by FLASH with that achieved by: (1) Original FATE that uses a highly-optimized GMP library to execute cryptographic operations with CPU (denoted as CPU in the following charts). We choose Intel Xeon Silver 4114 CPU similar to prior works [76]. All CPU experiments are executed with all ten physical cores in parallel. (2) GPU-based accelerator (denoted as GPU). We extend the GPU implementation of HAFLO [43] which only implements logistic regression. Note that only the cryptographic operations are accelerated by GPU in our experiments. We use NVIDIA P4 GPU because it has the same technology of 16nm and achieves the closest INT8 TOPS as FLASH (although $\sim 2\times$ better. P4 reaches ~ 20 INT8 TOPS while FLASH achieves ~ 12.9 INT8 TOPS).

Performance Metrics: We use *the number of operations performed per second (OP/s)* as the metric when evaluating the performance of cryptographic operations, and *acceleration ratio over CPU/GPU* as the metric when evaluating FL applications.

6.2 Cryptographic Operations

To demonstrate that FLASH can efficiently accelerate the nine cryptographic operations, we compare the performance achieved by CPU, GPU, and FLASH, respectively. For operations **O4** and **O5**, we also evaluate different exponent bitwidths (32bit – 1024bit). The experiment results are shown in Figure 8a. In general, FLASH can consistently outperform CPU and GPU for all cryptographic operations. Specifically, FLASH outperforms CPU by 7.7× – 14.0× and GPU by 1.4× – 3.4×, showing that FLASH's hardware architecture fits the computational requirements of these cryptographic operations. Furthermore, we observe that when handling a larger exponent, FLASH tends to achieve a better acceleration ratio. For example, FLASH achieves 13.6× acceleration than CPU when evaluating O4 with $e = 1024$bit, but drops to 7.7× with $e = 32$bit. The results show that when the computation is more intensive, *i.e.*, with a large exponent, FLASH can achieve even better performance.

Multi-accelerator Support: We inspect how FLASH performs when we use multiple FLASH acceleration cards to speed up cryptographic operations. We evaluate one, two and three accelerators, denoted as FLASH-1, FLASH-2 and FLASH-3 respectively. For space limitation, we only pick some operations for demonstration: **O1**, **O2**, **O3**, **O4** with

[7]More details on how network bandwidth affects FLASH are shown in §6.4

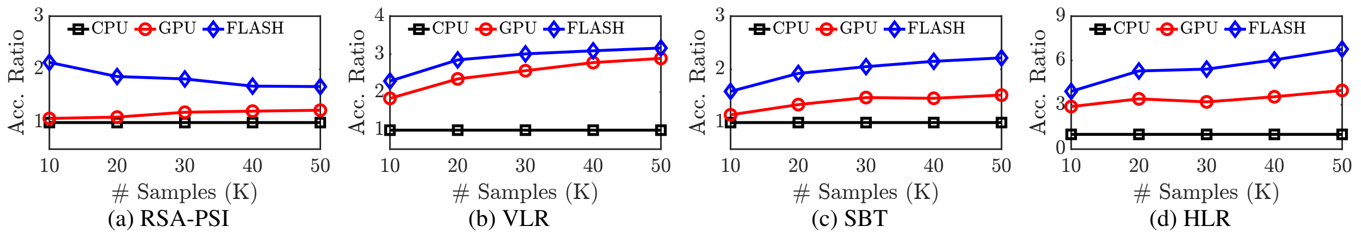


Figure 9: Performance of RSA-PSI, VLR, SBT, and HLR with changing data volumes.

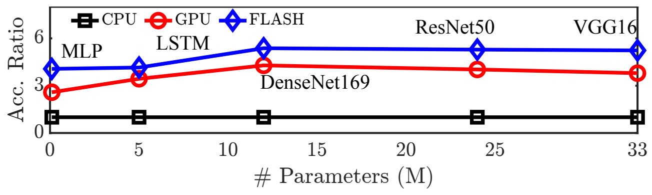


Figure 10: Performance of five deep learning applications.

$e = 1024$bit, **O5** with $e = 1024$bit, and **O7**. The results are shown in Figure 8b. We observe that for most cryptographic operations, *e.g.*, **O1**, **O2**, **O4**, **O5** and **O7**, the overall performance of FLASH is almost linear to the number of accelerators: FLASH-2 achieves $1.90\times$ – $1.98\times$ while FLASH-3 achieves $2.89\times$ – $2.95\times$ speedup for these operations. However, for **O3**, FLASH-2 and FLASH-3 only achieve $1.47\times$ and $1.80\times$ acceleration, respectively. The reason is as follows: the computation workload of **O3** is relatively low, thus the control overhead, *e.g.*, multi-accelerator synchronization, takes a considerable portion, leading to non-linear speedup. However, in the real-world use case, we envision that FLASH with multiple accelerators would still be an efficient solution to accelerate large-scale cross-silo FL applications.

Comparison with Other Paillier Accelerators: To give readers a better understanding of how efficient the FLASH's hardware design is, we further compare FLASH with some state-of-the-art hardware-based solutions, *e.g.*, Paillier Cryptoprocessor (PCP) [80], HLS [91], and SoC [31] based solutions. Moreover, due to the limited hardware resources, some of these works only implement a subset of cryptographic operations supported by FLASH. The comparison results are shown in Table 2. PCP and HLS report their data with public key $N = 1024$bit, while SoC uses $N = 2048$bit, thus we report the performance of FLASH with both $N = 1024$ and 2048bit. The results show that, compared to PCP, HLS and SoC, FLASH consumes $10.97\times$, $1.80\times$, $4.88\times$ DSP resources, but delivers $28.95\times$, $7.77\times$, and $10.75\times$ encryption acceleration and $93.15\times$, $20.56\times$, and $34.38\times$ decryption acceleration, respectively. The results demonstrate that by using inter- & intra-engine pipelining and dataflow scheduling, FLASH can (1) deliver much better performance if utilizing comparable resources, and (2) support more complete functions.

6.3 Cross-silo FL Applications

We then present how FLASH can accelerate real-world cross-silo FL applications, including both vertical and horizontal. The models and datasets used are shown in Table 3. For vertical FL, before performing the model training algorithms, we first run a commonly used sample alignment algorithm: RSA blind signature-based PSI (RSA-PSI). Then, we perform Vertical Logistic Regression (VLR) [53] and Secure Boosting Tree (SBT) [42] algorithms over the data intersection (generated from PSI), respectively. For horizontal FL, we mainly evaluate Horizontal Logistic Regression (HLR) and five deep learning applications with different parameters. Each application runs a fixed number of epochs.

RSA-PSI, VLR, SBT, and HLR: The performance of RSA-PSI, VLR, SBT, and HLR is related to the data volumes. Thus we evaluate FLASH with different data volumes. The results are shown in Figure 9. In general, FLASH consistently outperforms CPU and GPU by achieving $1.6\times$ – $6.8\times$ and $1.1\times$ – $2.0\times$ acceleration ratio respectively. The results have demonstrated that by designing a tailored hardware acceleration architecture for cross-silo FL, we can effectively speed up FL applications and outperform the existing CPU/GPU architectures. Furthermore, we also notice that for RSA-PSI and VLR, GPU tends to reach a similar acceleration ratio as FLASH while processing more data. The reason is that for RSA-PSI and VLR, the cleartext computation, which is purely executed on the CPU, takes a significant portion of the total computation time. For example, in VLR, when handling 50K data samples in one epoch, after sufficient acceleration, the ciphertext computation takes $< 10\%$ of the total computation time. Therefore, the performance is mainly decided by the time of cleartext computation when the cryptographic operations are sufficiently accelerated, which leads to the results that FLASH and GPU achieve similar acceleration ratios over CPU. In contrast, for HLR and SBT, FLASH can achieve a higher acceleration ratio than GPU because the cryptographic operations of these two applications consume a significant portion of the total computation time.

Deep Learning Applications: We have further evaluated five deep learning models of different numbers of parameters with horizontal FL. The results are shown in Figure 10. We find that FLASH can outperform CPU and GPU by achieving $4.1\times$ – $5.4\times$ and $1.2\times$ – $1.6\times$ acceleration ratio respectively due to a similar reason discussed above. Furthermore, we note that for models with more parameters, *e.g.*, DenseNet169, ResNet50, VGG16, FLASH can achieve a higher speedup than models with fewer parameters, *e.g.*, MLP, LSTM. This experiment implies that for more computation-intensive tasks, FLASH can deliver more notable results.

	28nm Technology Library (Actual Op. Frequency: 800MHz)			**12nm Technology Library** (Actual Op. Frequency: 1120MHz)		
	Area/Unit (mm^2)	**# Unit**	**Total Area (mm^2)**	**Area/Unit (mm^2)**	**# Unit**	**Total Area (mm^2)**
PCIe Gen3×16	8.46	1	8.460 (6.56%)	5.25	1	5.250 (4.04%)
DDR4 Controller	7.25	2	14.500 (11.24%)	4.43	2	8.860 (6.81%)
Engine Logic	0.093	800	74.480 (57.72%)	0.046	1900	87.499 (67.26%)
Engine Memory	0.033	800	26.200 (20.30%)	0.014	1900	25.927 (19.93%)
Dataflow Scheduling & Others	5.399	1	5.399 (4.18%)	2.561	1	2.561 (1.97%)
Total	-	-	129.04 (99.26%)	-	-	130.10 (100.08%)

Table 4: ASIC resource evaluation for both 28nm and 12nm technology libraries.

Correctness: In addition to evaluating the performance of the above nine cross-silo FL applications, we also validate the final results of all compared schemes (we avoid the randomness by setting an identical random seed). Results have shown that all schemes yield identical results, showing that FLASH does not affect the correctness of model training.

Summary: Implemented as an FPGA prototype, FLASH has already largely outperformed CPU and achieved moderately better performance than GPU with comparable price. We also understand that high-end GPUs, *e.g.*, A100 [9], H100 [10], may outperform FLASH's FPGA prototype due to more advanced foundry technology, which are also of much higher price. However, they still share the drawbacks as mentioned in §3.3. The goal of our paper is to design a more efficient hardware acceleration architecture for cross-silo FL beyond existing CPU/GPU architectures. As we will demonstrate in §6.5, if implemented as an ASIC, the performance of FLASH can be significantly improved, which should boost the acceleration ratio for these applications to a much higher level.

6.4 FLASH Deep-dive

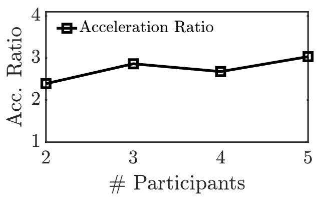
In this part, we mainly investigate (1) how the number of participants and (2) how the varying network bandwidth affects the performance of FLASH respectively.

Number of Participants: We evaluate VLR with two to five participants and measure the acceleration ratio of FLASH over CPU. The experiment result is shown in Figure 11a and we observe that in general, the number of participants does not largely impact the acceleration of FLASH.

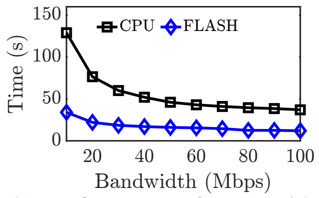
Varying Bandwidth Setting: In this part, we use `netem` [8] to limit the available bandwidth between the two participants from 10Mbps to 100Mbps. We run VLR and measure the execution time of one iteration with both CPU and FLASH. Figure 11b shows the results and we can observe that when the bandwidth is over 50Mbps, the running times of both CPU and FLASH are stable, where FLASH outperforms CPU by $\sim 3\times$. The results show that the varying network bandwidth does not have a noticeable impact on FLASH.

6.5 ASIC Performance Assessment

Given that our FPGA-based prototype implementation of FLASH has performance limitations due to the intrinsic drawback of FPGA (*e.g.*, low operation frequency), in this section we intend to demonstrate some preliminary results of how FLASH performs as an ASIC. As introduced in §5, we use



(a) Performance of VLR with varying participants.



(b) Performance of VLR with varying networking bandwidth.

Figure 11: FLASH deep-dive.

standard software tools to assess the performance of FLASH if implemented as an ASIC. We evaluate FLASH's ASIC implementation with two technology libraries: 28nm and 12nm. Based on the industry experience, we set the operating frequency to be 1000MHz and 1400MHz, respectively, for these two technology libraries. Furthermore, we set the die area to be $\sim 130\text{mm}^2$. We believe this setting could balance the performance and power consumption for FLASH.

The detailed evaluation includes the following steps: First, we perform logic synthesis using Synopsys Design Compiler [14] to convert FLASH's design into netlist under the frequency and die area constraints. Table 4 illustrates the results. With 28nm technology library, we can allocate 800 modular multiplication and exponentiation engines successfully, while with 12nm technology library, we can allocate 1900 such engines. Second, we use Synopsys VCS [16] and Synopsys Prime Time [15] to confirm that both netlists are valid and function correctly. The third step is to estimate the performance gain of FLASH as an ASIC. Since the actual operating frequency after physical design should be lower than logic synthesis, we reduce the actual operation frequency by multiplying 80% by the design target for a conservative purpose.

Then, our final performance estimation is as follows. With 28nm technology library, we can allocate 2.67× engines compared to our FPGA implementation (800 v.s. 300), and the operation frequency of these engines is 2.67× that of the FPGA implementation (800MHz v.s. 300MHz), leading to an overall 7.11× performance gain on modular exponentiation operator (we use modular exponentiation operator as the metrics since it can fulfill the computation capacity of an engine). With 12nm technology library, we can allocate 6.33× engines (1900 v.s. 300) with 3.73× operation frequency (1120MHz v.s. 300MHz), and achieve 23.64× overall performance improvement. To give our readers a better understanding of FLASH's performance as an ASIC, we also evaluate the modular exponentiation operator with a state-of-the-art GPU – NVIDIA A100 [9], our results show that A100 can only achieve 5.78× performance gain than our downscale FPGA prototype. Finally, we estimate the power consumption for a single engine, which is $\sim$ 16.6mWatt with 28nm technology library. Thus, the total power consumption for all engines is $\sim$ 13.28Watt. Although we do not have the power consumption data of other parts, *e.g.*, PCIe controller, we believe the total power consumption of FLASH as an ASIC should be significantly lower than the 120Watt of our FPGA implementation.

7 Discussion

Benefit to Future GPU/TPU Design: Nowadays, GPU [9–13] and TPU [63] have been widely adopted to accelerate deep learning applications. These accelerators mainly target accelerating convolution operations with tensors, where most numbers are short floats. In contrast, FLASH targets accelerating the identified nine cryptographic operations that are widely adopted in cross-silo FL. Moreover, most numbers used in FLASH are large numbers with a bit-width of 2048 bit or even longer. However, in some cross-silo applications, *e.g.*, horizontal deep learning in §6.3, both convolution and cryptographic operations exist. Therefore, we can foresee a co-design of GPU/TPU with FLASH in the future. For example, GPU/TPU can efficiently accelerate the local model training while FLASH is used to accelerate the model encryption and aggregation. We will make FLASH as an IP core in the future, and thus GPU/TPU vendors can use FLASH in their design to accomplish the aforementioned co-design.

FLASH v.s. Other GPU/FPGA Implementations: Some existing works also target accelerating modular exponentiation operations with GPU [43, 54, 61] or FPGA [29, 31, 34, 35, 80, 83, 91], which leverage similar algorithm optimization methods, *e.g.*, Binary Exponentiation [52] and Montgomery Modular Multiplication [65]. Yet, none of them performs a thorough analysis towards *all* cryptographic operations used in cross-silo FL and offloads them efficiently on the hardware-based accelerator as FLASH. Moreover, our idea of composing various cryptographic operations based on the two basic operators via dataflow scheduling is designed for the cross-silo FL scenarios, making FLASH a unique solution compared to prior FPGA-based implementations. As a final note, our design of FLASH is not limited to FPGA but is also applied to ASICs.

Extending to Other Application Domains: While FLASH is introduced for accelerating cross-silo FL, it can speed up applications in other domains as well. First, the Paillier and RSA cryptosystems used in cross-silo FL are also widely adopted in other domains. Thus FLASH can accelerate applications built on them, *e.g.*, electronic voting [47], electronic cash [38], and threshold cryptosystem [46]. Second, since FLASH's core idea is to accelerate modular multiplication and exponentiation operators, cryptographic systems/operations built on them, such as Diffie-Hellman key exchange [56], can also benefit from FLASH.

8 Related Works

Besides the related works discussed in §7, we further cover the following two related directions in this section.

Accelerating FL: Recently, due to the increasing deployment of FL, various research works have emerged to accelerate FL. MAGE proposes to optimize the secure computation from a memory perspective [67]. BatchCrypt tries to optimize the Paillier encryption by encoding a batch of quantized gradients into a long integer and encrypting it in one batch [95]. VF^{2}Boost proposes a novel training protocol to reduce the idle time of each participant [49]. Relative to them, we design FLASH from a different angle: accelerating the cryptographic operations used in FL, and our FLASH could be easily combined with these prior works.

Domain Specific Accelerator (DSA): DSA has recently been an emerging research topic that adopts hardware, *e.g.*, FPGA, ASIC, *etc.*, to accelerate particular applications [37, 62, 64, 75, 75, 76, 79, 93, 94]. For example, Tiara [94] uses FPGA and a programmable switch to accelerate layer-4 load balancing. FlowBlaze [75] offloads complex networking functions to a NetFPGA SmartNIC. hXDP [37] proposes to use FPGA to accelerate eBPF programs for fast XDP execution. MicroRec [62] offloads neural networks to FPGA to implement efficient recommendation systems. Various DSAs have been proposed to accelerate fully homomorphic encryption (FHE) [96], such as HEAX [76], F1 [78], BTS [64] and CraterLake [79]. Similar to them, FLASH follows the principle of DSA to design a hardware-based solution to efficiently accelerate cross-silo FL.

9 Conclusion

This paper presented FLASH, a hardware acceleration architecture for cross-silo FL. We have provided a fully functional FPGA prototype and evaluated our design as an ASIC. Extensive experiments with realistic applications and cryptographic operations have shown that FLASH is a viable solution.

Acknowledgments

We thank the anonymous NSDI reviewers and our shepherd Prof. Andreas Haeberlen for their constructive feedback and suggestions. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621, the NSFC Grant (62062005, 62102046), the Natural Science Foundation of Hunan Province (2022JJ30618), and the Scientific Research Fund of Hunan Provincial Education Department (22B0300). Kai Chen is the corresponding author.

References

[1] Credit Card Cheating Detection. https://www.kaggle.com/arslanali4343/credit-card-cheating-detection-cccd.

[2] Federated AI Technology Enabler. https://fate.fedai.org.

[3] Federated Machine Learning. https://github.com/FederatedAI/FATE/tree/master/python/federatedml.

[4] FedLearner. https://github.com/bytedance/fedlearner.

[5] Intel Xeon Silver 4114 Processor. https://www.intel.com/content/www/us/en/products/sku/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz/specifications.html.

[6] Mellanox ConnectX-4 EN Adapter Card Single/Dual-Port 100 Gigabit Ethernet Adapter. https://www.mellanox.com/products/ethernet-adapters/connectx-4-en.

[7] Mellanox SN2100 Open Ethernet Switch. https://www.mellanox.com/related-docs/prod_eth_switches/PB_SN2100.pdf.

[8] netem. https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[9] NVIDIA A100. https://www.nvidia.com/en-us/data-center/a100/.

[10] NVIDIA H100. https://www.nvidia.com/en-us/data-center/h100/.

[11] NVIDIA P4. https://images.nvidia.com/content/pdf/tesla/184457-Tesla-P4-Datasheet-NV-Final-Letter-Web.pdf.

[12] NVIDIA P40 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/documents/nvidia-p40-datasheet.pdf.

[13] NVIDIA V100 Datasheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.

[14] Synopsys Design Compiler. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[15] Synopsys Prime Time. https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html.

[16] Synopsys VCS. https://www.synopsys.com/verification/simulation/vcs.html.

[17] Text generation with an RNN. https://www.tensorflow.org/text/tutorials/text_generation.

[18] TF Encrypted. https://github.com/tf-encrypted/tf-encrypted.

[19] The GNU Multiple Precision Arithmetic Library. https://gmplib.org.

[20] Timing Closure User Guide. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx14_7/ug612.pdf.

[21] Xilinx DMA. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[22] Xilinx UltraScale Architecture DSP Slice User Guide. https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

[23] Xilinx UltraScale+ FPGA Production Table and Production Selection Guide. https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[24] Xilinx Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics. https://docs.xilinx.com/v/u/en-US/ds923-virtex-ultrascale-plus.

[25] Xilinx Virtex7 FPGAs. https://www.xilinx.com/support/documentation/selection-guides/virtex7-product-table.pdf.

[26] Xilinx Zynq UltraScale+ MPSoCs. https://docs.xilinx.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide.

[27] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 308–318. ACM, 2016.

[28] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4):79:1–79:35, 2018.

[29] Timo Alho, Panu Hämäläinen, Marko Hännikäinen, and Timo D. Hämäläinen. Compact modular exponentiation accelerator for modern FPGA devices. *Comput. Electr. Eng.*, 33(5-6):383–391, 2007.

[30] Arash AziziMazreah and Lizhong Chen. Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 94–105. IEEE, 2019.

[31] Milad Bahadori and Kimmo Järvinen. A programmable soc-based accelerator for privacy-enhancing technologies and functional encryption. *IEEE Trans. Very Large Scale Integr. Syst.*, 28(10):2182–2195, 2020.

[32] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[33] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. Shredder: Gpu-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, page 14. USENIX Association, 2012.

[34] Thomas Blum. Montgomery modular exponentiation on reconfigurable hardware. In *14th IEEE Symposium on Computer Arithmetic (Arith-14 '99), 14-16 April 1999, Adelaide, Australia*, pages 70–77. IEEE Computer Society, 1999.

[35] Thomas Blum and Christof Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE Trans. Computers*, 50(7):759–764, 2001.

[36] Kallista A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.

[37] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA nics. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 973–990. USENIX Association, 2020.

[38] Jan Camenisch, Anna Lysyanskaya, and Mira Meyerovich. Endorsed e-cash. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, pages 101–115. IEEE Computer Society, 2007.

[39] Di Chai, Leye Wang, Kai Chen, and Qiang Yang. Secure federated matrix factorization. *IEEE Intell. Syst.*, 36(5):11–20, 2021.

[40] Di Chai, Leye Wang, Junxue Zhang, Liu Yang, Shuowei Cai, Kai Chen, and Qiang Yang. Practical lossless federated singular vector decomposition over billion-scale data. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, pages 46–55. ACM, 2022.

[41] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X. Liu, Hao Wang, and Cheng Hong. When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 2652–2662. ACM, 2021.

[42] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. SecureBoost: A lossless federated learning framework. *IEEE Intell. Syst.*, 36(6):87–98, 2021.

[43] Xiaodian Cheng, Wanhang Lu, Xinyang Huang, Shuihai Hu, and Kai Chen. HAFLO: gpu-based acceleration for federated logistic regression. *CoRR*, abs/2107.13797, 2021.

[44] Yong Cheng, Yang Liu, Tianjian Chen, and Qiang Yang. Federated learning for privacy-preserving AI. *Commun. ACM*, 63(12):33–36, 2020.

[45] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2010.

[46] Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*, volume 2727 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2003.

[47] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *Int. J. Inf. Sec.*, 9(6):371–385, 2010.

[48] Olga Fink, Torbjørn H. Netland, and Stefan Feuerriegel. Artificial intelligence across company borders. *Commun. ACM*, 65(1):34–36, 2022.

[49] Fangcheng Fu, Yingxia Shao, Lele Yu, Jiawei Jiang, Huanran Xue, Yangyu Tao, and Bin Cui. VF^{2}Boost: Very fast vertical federated gradient boosting for cross-enterprise learning. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 563–576. ACM, 2021.

[50] Michelle Goddard. The eu general data protection regulation (GDPR): European regulation that has a global impact. *International Journal of Market Research*, 59(6):703–705, 2017.

[51] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. HARE: hardware accelerator for regular expressions. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 44:1–44:12. IEEE Computer Society, 2016.

[52] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, 1998.

[53] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *CoRR*, abs/1711.10677, 2017.

[54] Owen Harrison and John Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 350–367. Springer, 2009.

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[56] Martin E Hellman, Bailey W Diffie, and Ralph C Merkle. Cryptographic apparatus and method, April 29 1980. US Patent 4,200,770.

[57] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[58] Roger A Horn. The hadamard product. In *Proc. Symp. Appl. Math*, volume 40, pages 87–169, 1990.

[59] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[60] M.K. Ibrahim. Radix-2^n multiplier structures: a structured design methodology. *IEE Proceedings E (Computers and Digital Techniques)*, 140(4):185–190, 1993.

[61] Keon Jang, Sangjin Han, Seungyeop Han, Sue B. Moon, and KyoungSoo Park. Sslshader: Cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[62] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B. Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, Ce Zhang, and Gustavo Alonso. MicroRec: Efficient recommendation inference by hardware and data structure solutions. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*. mlsys.org, 2021.

[63] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12. ACM, 2017.

[64] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. BTS: an accelerator for bootstrappable fully homomorphic encryption. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 711–725. ACM, 2022.

[65] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[66] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[67] Sam Kumar, David E. Culler, and Raluca Ada Popa. MAGE: nearly zero-cost virtual memory for secure computation. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 367–385. USENIX Association, 2021.

[68] Pankaj Kumbhare and Vamsi Krishna. Designing high-performance video systems in 7 series FPGAs with the AXI interconnect. *Xilinx, Inc., San Jose, CA, USA, Appl. Note*, 7:1–24, 2012.

[69] Gang Liang and Sudarshan S. Chawathe. Privacy-preserving inter-database operations. In *Intelligence and Security Informatics, Second Symposium on Intelligence and Security Informatics, ISI 2004, Tucson, AZ, USA, June 10-11, 2004, Proceedings*, volume 3073 of *Lecture Notes in Computer Science*, pages 66–82. Springer, 2004.

[70] Yang Liu, Yan Kang, Chaoping Xing, Tianjian Chen, and Qiang Yang. A secure federated transfer learning framework. *IEEE Intell. Syst.*, 35(4):70–82, 2020.

[71] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[72] M Rai Nigam and S Bande. AXI interconnect between four master and four slave interfaces. *Int. J. Eng. Res. Gen. Sci*, 2(4):2091–2730, 2014.

[73] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[74] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Trans. Inf. Forensics Secur.*, 13(5):1333–1345, 2018.

[75] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani Brunella, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, and Felipe Huici. FlowBlaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 531–548. USENIX Association, 2019.

[76] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1295–1309. ACM, 2020.

[77] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[78] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*, pages 238–252. ACM, 2021.

[79] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sánchez. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 173–187. ACM, 2022.

[80] Ismail San, Nuray At, Ibrahim Yakut, and Huseyin Polat. Efficient paillier cryptoprocessor for privacy-preserving data mining. *Secur. Commun. Networks*, 9(11):1535–1546, 2016.

[81] Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. POSEIDON: privacy-preserving federated neural network learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[83] Daisuke Suzuki. How to maximize the potential of FPGA resources for modular exponentiation. In *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2007.

[84] Donald E. Thomas and Philip Moorby. *The Verilog hardware description language (2. ed.)*. Kluwer, 1995.

[85] Han Tian, Chaoliang Zeng, Zhenghang Ren, Di Chai, Junxue Zhang, Kai Chen, and Qiang Yang. Sphinx: Enabling privacy-preserving online learning over the cloud. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2487–2501. IEEE, 2022.

[86] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017.

[87] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind. AQUOMAN: an analytic-query offloading machine. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 386–399. IEEE, 2020.

[88] Liu Yang, Ben Tan, Vincent W. Zheng, Kai Chen, and Qiang Yang. Federated recommendation systems. In *Federated Learning - Privacy and Incentive*, volume 12500 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2020.

[89] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.*, 10(2):12:1–12:19, 2019.

[90] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving google keyboard query suggestions. *CoRR*, abs/1812.02903, 2018.

[91] Zhaoxiong Yang, Shuihai Hu, and Kai Chen. Fpga-based hardware accelerator of homomorphic encryption for efficient federated learning. *CoRR*, abs/2007.10560, 2020.

[92] Tian Ye, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor K. Prasanna. Performance modeling and FPGA acceleration of homomorphic encrypted convolution. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*, pages 115–121. IEEE, 2021.

[93] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: an fpga-accelerated embedding-based retrieval system. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 841–856. USENIX Association, 2022.

[94] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 1345–1358. USENIX Association, 2022.

[95] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 493–506. USENIX Association, 2020.

[96] Junxue Zhang, Xiaodian Cheng, Liu Yang, Jinbin Hu, Ximeng Liu, and Kai Chen. Sok: Fully homomorphic encryption accelerators. *CoRR*, abs/2212.01713, 2022.

Appendix

A Cross-silo Federated Learning

Cross-silo federated learning (FL) denotes the scenario where companies or institutions collaboratively train machine learning models without data privacy leakage [39,40,44,48,88,89]. Compared with cross-device FL, where participants are mobile devices, cross-silo FL focuses more on data security and incentive mechanism. From the data partition angle, Yang *et al.* proposed to categorize FL into horizontal FL, vertical FL, and federated transfer learning [70]. Federated transfer learning has rarely been applied in the industry and remains in the research stage. In most cases, cross-device FL contains only horizontal FL, while cross-silo FL usually contains both horizontal and vertical FL. Because of the different data partition situations, horizontal and vertical FL are different in model construction, protocol design as well as the utilized cryptographic systems.

A.1 Cross-silo Horizontal FL

Participants in horizontal FL have different sample ID spaces but the same feature space, as shown in Figure 1a. Each participant owns the labels of their samples. Therefore, horizontal FL enlarges the number of training samples to train a model with better generalization ability. In most cases, there is a third-party central server for parameter aggregation.

The t^{th} iteration of the training process among three participants is shown as follows:

1. All participants negotiate about keys for encryption.
2. Each participant i trains local model $\boldsymbol{w}_i^t$ with its own samples and encrypts its model weights to $[[\boldsymbol{w}_i^t]]$ with either additively homomorphic encryption (*e.g.*, Paillier [74]).
3. Each participant i sends the encrypted weights to the central server.
4. The server receives the encrypted local model weights from all participants and aggregates them to global model weights $\frac{\sum_i [[\boldsymbol{w}_i^t]]}{n}$, where n stands for the number of participants. Because the weights are encrypted via additively homomorphic encryption, we can directly perform an aggregation over the ciphertext.
5. The central server sends the aggregated global weights to all participants.
6. Each participant receives the global weights and decrypts them locally. Then the participant can update its local model $\boldsymbol{w}_i^{t+1}$ with the decrypted global model.

After the federated training process, each participant obtains the same well-trained model. Thus, each participant can perform model inference locally.

A.2 Cross-silo Vertical FL

Participants in vertical FL scenarios have the same sample ID space but different feature spaces, as shown in Figure 1b. Under normal circumstances, only one participant holds the label of the FL task, which is called the active party. The other participants without labeling information are called passive parties. Compared with horizontal FL, vertical FL could enrich the feature information of samples. Unlike horizontal FL, the training process of vertical FL is conducted after the entity alignment stage, which aligns common samples while protecting privacy. Besides, the training schema of vertical FL is also different from horizontal FL. More specifically, each participant only owns part of the model parameters corresponding to the local feature dimensions. Hence, vertical FL cannot simply conduct the secure aggregation as horizontal FL does. In addition, various machine learning algorithms do not have a unified design of the vertical FL implementation.

Taking the federated linear regression [89] between two participants as an example, we illustrate the training process as follows:

1. Participants and the third-party central server negotiate about keys for encryption.
2. Passive party B computes local point estimate $u_{B,j}^t$ and partial loss $L_{B,j}^t$ for the j^{th} aligned sample, then encrypts them to $[[u_{B,j}^t]]$ and $[[L_{B,j}^t]]$ with Paillier [74]. Active party A calculates local point estimate $u_{A,j}^t$.
3. Passive party B sends the encrypted numbers to the active party A.
4. Active party A receives the encrypted numbers and computes the total loss $[[L_j^t]]$ and the intermediate results $[[d_j^t]]$ used to calculate gradients.
5. Active party A sends $[[L_j^t]]$ to server and $[[d_j^t]]$ to passive party B.
6. Party B and party A separately compute encrypted gradients $[[\frac{\partial L_j^t}{\partial \boldsymbol{w}_P^t}]]$ and $[[\frac{\partial L_j^t}{\partial \boldsymbol{w}_A^t}]]$ and add random masks.
7. Both parties send the encrypted and masked gradients to the central server.
8. The third-party central server decrypts the received ciphertext to get the plain-text masked gradients and sends them back.
9. Party B and party A respectively remove the random masks from gradients and update the local partial model.

All participants of vertical FL should be involved in the inference stage since each of them only owns part of the whole model.

A.3 Security Analysis of Cross-silo FL

The adopted FL algorithms in this paper are proved secure under the semi-honest assumption [42, 89]. The semi-honest assumption means that each party does not violate the federated protocols and only tries to infer the sensitive data of other parties from the received messages. For the horizontal FL models, the transmitted model updates are protected by additively HE for aggregation. Therefore, nothing can be learned by the arbiter. Moreover, each party obtains the aggregated model updates and can only calculate the average model updates of the other parties. Hence, given more than two parties, the model updates computed over the local data of one party cannot be leaked to the other parties [89]. For the vertical federated linear models, the transmitted intermediate results are protected by random masks and HE, which reveals no information. Furthermore, from the obtained model updates, one party cannot infer the sensitive data of other parties without prior knowledge of their data structures [89]. For the vertical SecureBoost model, the active party with labels could learn some information agreed in advance, such as the instance spaces and the responsible parties of splits. However, under the protection of HE, the original data records of one party cannot be revealed to other parties, either [42].

B Paillier Cryptosystem

Paillier Cryptosystem [73] is a widely-used additively (*i.e.*, partially) homomorphic encryption scheme. Paillier cryptosystem supports two kinds of operations, including the addition of two values of ciphertext and multiplication between ciphertext and cleartext. We will introduce Paillier key generation, encryption and decryption respectively in the following section.

Key Generation: Key generation of Paillier Cryptosystem follows the following steps.

1. Choose two random prime numbers p and q which satisfy that $\gcd(pq,(p-1)(q-1))=1$, where gcd stands for the greatest common divisor.
2. Compute $n = p \cdot q$.
3. Compute $\lambda = \mathrm{lcm}(p-1, q-1)$, where lcm means the least common multiple.
4. Randomly select an integer g which satisfies that $\gcd(\mathrm{L}_n(g^\lambda \mod n^2), n) = 1$. Function $\mathrm{L}_n(x)$ is defined as $\mathrm{L}_n(x) = (x-1)/n$.
5. Compute $\mu = [\mathrm{L}_n(g^\lambda \mod n^2)]^{-1} \mod n$.

After the above computation, we will obtain the public key: (n, g) and private key: (λ, μ) respectively.

Encryption: The encryption algorithm of Paillier is straightforward and follows the following equation.

$$c = g^m \cdot r^n \mod n^2. \tag{B.1}$$

Optimization of Encryption: The encryption can be accelerated by assigning public key g as $n+1$. Therefore, the encryption algorithm is simplified as follows.

$$\begin{aligned} c &= g^m \cdot r^n \mod n^2 \\ &= (n+1)^m \cdot r^n \mod n^2 \\ &= [(\sum_{i=0}^{m} \binom{m}{i} \cdot n^i) \cdot r^n] \mod n^2 \\ &= [(1 + m \cdot n) \cdot r^n] \mod n^2 \end{aligned} \tag{B.2}$$

One modular exponentiation operation is saved by this optimization. FLASH uses the optimized encryption for better performance.

Decryption: Paillier ciphertext c is decrypted to plaintext m with both public key (n, g) and private key (λ, μ):

$$m = \mathrm{L}_n(c^\lambda \mod n^2) \cdot \mu \mod n \tag{B.3}$$

Optimization of Decryption: The workload of the decryption algorithm of Paillier can be reduced with the Chinese Remainder Theorem (CRT). In this scheme, prime numbers p and q generated with the key pair are considered as the private key. The process of optimized decryption is shown below:

1. Compute $h_p = \mathrm{L}_p(g^{p-1} \mod p^2)^{-1} \mod p$ and $h_q = \mathrm{L}_q(g^{q-1} \mod q^2)^{-1} \mod q$.
2. Compute $m_p = \mathrm{L}_p(c^{p-1} \mod p^2) \cdot h_p \mod p$ and $m_q = \mathrm{L}_q(c^{q-1} \mod q^2) \cdot h_q \mod q$. Function $\mathrm{L}_p(x)$ and $\mathrm{L}_q(x)$ are defined by $\mathrm{L}_p(x) = (x-1)/p$ and $\mathrm{L}_q(x) = (x-1)/q$.
   It can be proved that $m_p = m \mod p$ and $m_q = m \mod q$, where m is the plaintext corresponding to ciphertext c.
3. Apply CRT to recombine the modular residues. $m = \mathrm{CRT}(m_p, m_q) \mod pq$.

With the optimization above, the workload can be reduced to only about one-quarter of the original decryption algorithm, leading to better performance. FLASH also uses optimized decryption in its implementation.

C RSA Intersection

RSA (Rivest–Shamir–Adleman) is an asymmetric public-private key method used to securely transfer data [77]. The whole RSA algorithm mainly contains three operations: key generation, encryption, and decryption.

Key Generation: The generation process is shown below:

1. Randomly choose two distinct prime numbers p and q.
2. Compute $n = p \cdot q$.
3. Compute $\lambda(n) = \mathrm{lcm}(p-1, q-1)$.
4. Randomly choose a number e such that $1 < e < \lambda(n)$ and $\gcd(e, \lambda(n)) = 1$.
5. Compute d by solving $d \cdot e = 1 \mod \lambda(n)$.

Generally speaking, (n, e) is regarded as a public key, while d is regarded as a private key.

Encryption: Using public key (n, e), plain-text message m is encrypted to cipher-text message c:

$$c = m^e \mod n. \tag{C.4}$$

Decryption: Using private key d, cipher-text message c is decrypted to plain-text message m:

$$m = c^d \mod n. \tag{C.5}$$

RSA-based PSI: The RSA-based private set intersection can protect the privacy of sample ID out of the intersection set with the mechanism of blind RSA signature [45,69]. We take the two-party setting as an example. Party A contains three user IDs, *i.e.*, $\mathcal{X}_A = \{x_1, x_2, x_3\}$, while party B contains four user IDs, *i.e.*, $\mathcal{X}_B = \{x_1, x_2, x_4, x_5\}$. They want to find their common users via RSA-based intersection:

1. Party A generates RSA keys n, e, d and sends public key (n, e) to party B.
2. Party B blinds and encrypts its user IDs $\mathcal{X}_B$ to $\mathcal{Y}_B = \{H(x) \mod n \cdot r^e \mod n) \mid x \in \mathcal{X}_B\}$, where r is a unique random number for each x, and sends $\mathcal{Y}_B$ to party A.
3. Party A signs the received $\mathcal{Y}_B$, obtains $\mathcal{Z}_B = \{y^d \mod n = r \cdot H(x)^d \mod n \mid y \in \mathcal{Y}_B\}$ and sends $\mathcal{Z}_B$ to party B.
4. Party A also signs its own user IDs, gets $\mathcal{D}_A = \{H(H(x))^d \mid x \in \mathcal{X}_A\}$ and sends $\mathcal{D}_A$ to party B.
5. Party B unblinds the received $\mathcal{Z}_B$ and obtains $\mathcal{D}_B = \{H(z/r \mod n) = H(H(x))^d \mid z \in \mathcal{Z}_B\}$.

6. Party B computes $\mathcal{D}_A \cap \mathcal{D}_B = \{H(H(x_1))^d, H(H(x_2))^d\}$ and gets common user IDs $\{x_1, x_2\}$.

$H(\cdot)$ denotes the hash function. After party B knows the overlapping users, it could choose whether to inform party A according to different scenarios.

D Modular Exponentiation & Multiplication Algorithm Optimization

D.1 Binary Exponentiation

Modular exponentiation is defined as $P = m^e \mod N$, as shown in Equation 1. In the naïve algorithm, m is multiplied by itself for e times, and the algorithm uses $e-1$ multiplications to obtain the result. Therefore, if e is a large integer, the computation time is dramatic. As a result, to optimize the computation, people usually apply binary exponentiation optimization to reduce the dramatic computation time. Algorithm D.1 shows the process of the binary exponentiation optimization algorithm.

Algorithm D.1 Binary Exponentiation

```
Input: $m$, $e$, $N$, where $N > 0$
Output: $P = m^e \mod N$
1: $P = 1$                                  ▷ Initialization
2: while $e > 1$ do
3:     if e is odd then
4:         $P = P \cdot m \mod N$
5:     end if
6:     $e = e \gg 1$
7:     $m = m^2 \mod N$
8: end while
9: return $P$
```

The idea of binary exponentiation is to reduce the number of multiplications by using the binary representation of the exponent e. As a result, we only need to compute at most $2\lfloor \log_2 e \rfloor$ multiplications, which is much smaller than $e-1$. Since the time complexity of modular exponentiation is determined by the number of multiplications, binary exponentiation can reduce its time complexity from $O(e)$ to $O(\log e)$. Worth mentioning, the modulo computation can be performed after each multiplication because of the distribution law in modular arithmetic: $(a \mod N)(b \mod N) \equiv ab \mod N$.

Summary: By using the binary exponentiation optimization algorithm, we can largely optimize the time complexity of modular exponentiation computation.

D.2 Montgomery Modular Multiplication

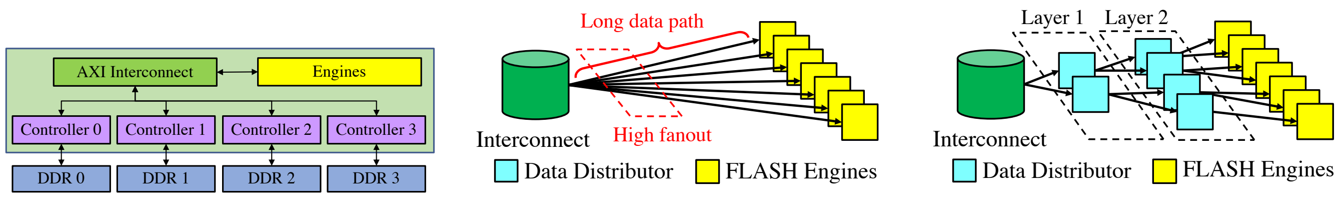
After applying the binary exponentiation optimization algorithm as shown in §D.1, we lower the time complexity of modular exponentiation computation by reducing the number of multiplications. However, after each multiplication, we have to perform one modulo operation. Although we can implement modulo operation on hardware with Cyclic Reduction and Barrett Reduction algorithms [32], the performance of these algorithms is still not satisfying because of the division operations used in these algorithms. Therefore, FLASH utilizes Montgomery Modular Multiplication [65] to replace the modulo operation with a bit-shifting operation, which is more hardware-friendly.

Algorithm D.2 Montgomery Modular Multiplication

```
▷ Given three input numbers $X$, $Y$ and $N$, the Montgomery Modular Multiplication outputs $Z = X \cdot Y \cdot R^{-1} \mod N$, where $R$ is a power of 2 and $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.
Input: $X = (X_{d-1}, ..., X_0)$, $Y = (Y_{d-1}, ..., Y_0)$, $N = (N_{d-1}, ..., N_0)$,
    $N'$, where
    $N' = (-N)^{-1} \mod r$,                ▷ $N'$ is pre-computed in S1
    $r = 2^w$, $d = \lfloor \log_r N \rfloor + 1$,     ▷ $r$ and $d$ is used to split data
    $\gcd(N, r) = 1$, with $N \times N' \equiv -1 \mod r$
Output: $Z = \text{ModMult}(X, Y, N) = X \times Y \times R^{-1} \mod N$
1: $Z = (Z_{d-1}, ..., Z_0) = 0$                    ▷ Initialization
2: for all $i = 0, 1, ..., d-1$ do                    ▷ Loop on $Y$
3:     $q = (Z_0 + X_0 \times Y_i) \times N' \mod r$
4:     $C = 0$
5:     for all $j = 0, 2, ..., d-1$ do                ▷ Loop on $X$
6:         $S = Z_j + X_j \times Y_i + q \times N_j + C$
7:         if $j > 0$ then
8:             $Z_{j-1} = S \mod r$
9:         end if
10:        $C = S \gg w$                             ▷ Carry higher bits
11:    end for
12:    $Z_{d-1} = C$
13: end for
14: if $Z \geq N$ then
15:     $Z = Z - N$
16: end if
17: return $Z$
```

The process of applying Montgomery Modular Multiplication includes three major steps: (1) converting the data into Montgomery space, (2) computing the modular multiplication in the Montgomery space, and (3) converting the data back from Montgomery space. Before going into details, we will first describe Algorithm D.2. This algorithm implements efficient $A * B * R^{-1} \mod N$ in a hardware-friendly way. The key optimization of the algorithm is the introduction of the divider $R = r^d$. Thus the division can be easily implemented by bit-shifting since r is a power-of-2 integer, and after the division, the result is an integer within $[0, 2N)$ and no more modulo operation is needed. Afterward, we will show details of each step in the following sections.

Converting the data into Montgomery space: Before applying Montgomery Modular Multiplication, the input numbers should be converted to Montgomery space. The conversion formula is $A = a \cdot R \mod N$. It can also be written as $A = a \cdot R^2 \cdot R^{-1} \mod N$, so we can leverage Algorithm D.2 to efficiently calculate it. In the formula, a is one of the multiplicands of modular multiplication. A is the Montgomery space of a. N is the modulus. R is a power of 2 and it satisfies the condition that $\lfloor \log_2 R \rfloor = \lfloor \log_2 N \rfloor$.



(a) AXI Interconnect. (b) Naïve data distribution. (c) Hierarchical data distribution.

Figure D.1: FLASH adopts hierarchical data distribution to enable efficient data exchange between on-chip and off-chip memory.

Computing the modular multiplication in the Montgomery space: We can directly use Algorithm D.2 to efficiently calculate the modular multiplication.

Converting the data back from Montgomery space: To convert a number out of Montgomery space, the conversion formula is $p = P \cdot R^{-1} \mod N$. p is the result of modular multiplication. P is the Montgomery format of p. Similarly, we can leverage Algorithm D.2 to complete the computation.

Parameter Computation: In the above steps, we notice that if N, which is usually the public key in cryptosystems, remains unchanged, $R^2 \mod N$ (*e.g.*, used in converting the data into Montgomery space) and $(-N)^{-1} \mod r$ (*e.g.*, line 3 and 8 in Algorithm D.2) remain constant values. We call these constant values parameters in this paper, and we show that we can compute these parameters in advance and avoid duplicate calculations to improve the performance further.

Summary: By applying the Montgomery Modular Multiplication, we mainly replace the modulo operation with a hardware-friendly bit-shifting operation, which improves the performance of modular multiplication/exponentiation on hardware.

E The Montgomery Modular Multiplication Circuit Design

In this section, we will describe how our Montgomery Modular Multiplication Circuit works (shown in Figure 5).

According to Algorithm 1, the outer loop iterates over operand Y. Therefore, the circuit sequentially reads different Y_i from RAM Y and performs execution over them. The workflow of our Montgomery Modular Multiplication circuit contains four steps. Steps 1 and 2 in the following introduction focus on the workflow for a fixed Y_i while steps 3 and 4 show how we bridge the operations between iterations with different i and obtain the final result. We use Reg to represent the register group.

1. X_0 and Y_i are sent to Mul 1 to get a $2w$-bit multiplication result. The higher w bits of the result are cached in Reg r_1. The lower w bits, denoted as α in Algorithm 1, are cached in both Reg α and r_7. The numbers stored in Reg α are used for the calculation of β via Add 5. With β and N' available, their multiplication result q can be obtained from the output of Mul 2. After that, q is sent back to the input of Mul 2 and multiplied with N_0. Similarly, the higher w bits of the multiplication result are cached in Reg r_3 and the lower w bits are sent to Reg r_8. Combining data cached in r_7 and r_8, it is straightforward to get δ_1 and δ_2 with several addition units. The results of addition, Z_0 and C, are cached in Reg Z_0 and Reg C for subsequent operations.
2. Following step 1, the inner loop for j begins. Different iterations of the inner loop are fully pipelined, which implies the jth iteration is executed by the circuit just one cycle after the $(j-1)$th iteration. At the beginning of the pipeline, Mul 1 and Mul 2 simultaneously calculate $X_j \times Y_i$ and $q \times N_j$. The higher w bits and lower w bits of the results are separately cached in different registers. Please note that we use Reg r_5 and r_6 to register the higher w bits in the circuit for one more cycle compared to the lower w bits so that δ_1 can be calculated through the addition between higher w bits from the $(j-1)$th iteration and lower w bits from the jth iteration (*i.e.*, line 12 in Algorithm 1). After the calculation of δ_1, the subsequent calculation of δ_2 can also be simply executed by the adders in the pipeline. The intermediate results, Z_{j-1} and C, are cached in RAM Z and Reg C.
3. We begin the execution of the $(i+1)$th iteration of the outer loop when all the multiplications in the ith iteration accomplish. Although some operations like addition are still in progress, the multipliers are free to start the multiplications in Step 1 for the $(i+1)$th iteration.
4. After accomplishing the outer loop, the result Z should be stored in RAM Z. If $Z < N$, we directly output Z. Otherwise, we calculate $Z - N$ as the final output.

F Hierarchical Data Distribution

While the design in §4.2.3 is efficient, it also introduces a practical problem. As shown in Figure D.1a, FLASH adopts AXI interconnect to manipulate the external memory [68, 72]. However, as the on-chip memory units are placed near each engine for low latency, naïvely distributing data from the AXI interconnect to these memory units, as shown in Figure D.1b, leads to high fan-out near interconnect and long data paths. These two issues will cause (1) large design difficulties for circuits placement because there are too many long paths to be placed near interconnect; (2) degraded performance because long paths cause large delay for the circuits.

To solve the problem, we design a hierarchical data distribution mechanism as shown in Figure D.1c. Instead of directly sending data to all engines, FLASH distributes data

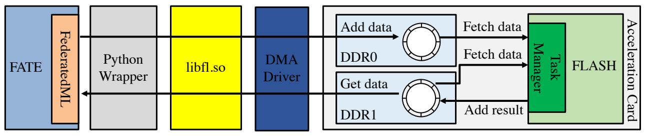


Figure G.2: FLASH integrates with cross-silo FL frameworks by providing an integrated software package.

at multiple layers. At each layer, the data distributors receive data from the previous layer and further distribute data to the data distributors/engines in the next layer. Suppose we have m engines and n layers, the fan-out of each data distributor is $\sim \log_n m$, which is much smaller than m. As a result, FLASH achieves a much smaller fan-out and shortened logical data path. These two advantages first reduce the design complexity because a small fan-out with short logical paths will make the circuits' placement much easier. Furthermore, they also improve performance because they allow a high operating frequency by restricting the delay of all logic paths. In our FPGA implementation, the delay of all logic data paths is within 3.3ns, thus we can achieve a high FPGA operation frequency of 300MHz [20].

G Software Stack Architecture

Figure G.2 illustrates how FLASH integrates with the cross-silo FL software. As introduced in §4.4, FLASH's software stack contains Xilinx DMA Driver [21], `libfl.so` library and its corresponding Python wrapper.