IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 819–843. DOI:10.46586/tches.v2024.i2.819-843

SHAPER: A General Architecture for Privacy-Preserving Primitives in Secure Machine Learning

Ziyuan Liang[1], Qi'ao Jin[1], Zhiyong Wang[1], Zhaohui Chen[2,3,4], Zhen Gu[3,4,5], Yanhheng Lu[4,6] and Fan Zhang[1]

[1] Zhejiang University, Hangzhou, China,
liangziyuan,{jin_qi_ao,wangzhiyong,fanzhang}@zju.edu.cn
[2] Peking University, Beijing, China
[3] DAMO Academy, Alibaba group, Beijing, China,
chenzhaohui.czh,{guzhen.gz}@alibaba-inc.com
[4] Hupan Lab, Hangzhou, China
[5] Tsinghua University, Beijing, China
[6] Alibaba Group, Shanghai, China, yanheng.lyh@alibaba-inc.com

Abstract. Secure multi-party computation and homomorphic encryption are two primary security primitives in privacy-preserving machine learning, whose wide adoption is, nevertheless, constrained by the computation and network communication overheads. This paper proposes a hybrid Secret-sharing and Homomorphic encryption Architecture for Privacy-pERsevering machine learning (*SHAPER*). SHAPER protects sensitive data in encrypted or randomly shared domains instead of relying on a trusted third party. The proposed algorithm-protocol-hardware co-design methodology explores techniques such as plaintext Single Instruction Multiple Data (SIMD) and fine-grained scheduling, to minimize end-to-end latency in various network settings. SHAPER also supports secure domain computing acceleration and the conversion between mainstream privacy-preserving primitives, making it ready for general and distinctive data characteristics. SHAPER is evaluated by FPGA prototyping with a comprehensive hyper-parameter exploration, demonstrating a 94× speed-up over CPU clusters on large-scale logistic regression training tasks.

Keywords: Privacy-Preserving Machine Learning · Multi-Party Computation · Additive Homomorphic Encryption · Hardware Accelerator

1 Introduction

Cross-agency data collaboration maximizes the accuracy of Machine learning (ML) models. Nonetheless, from the perspective of user privacy and business interests, concerns about data privacy and security arise [ARC19]. In practice, ML cannot be applied directly to health or financial data for competitive and regulatory reasons. These sensitive data sets are isolated by different parties, which is also known as the "isolated data island" problem. To solve this problem, privacy-preserving machine learning (PPML) [XBJ21] allows participants to collaborate on training and inference procedures by applying privacy-preserving computing techniques, *e.g.* multi-party computation (MPC) [Yao82], homomorphic encryption (HE) [FV12], and trusted execution environment (TEE) [CD16]. These security primitives prevent the raw data, model weights, and gradient values from being revealed to any other participants. Since the algorithms and protocols of PPML heavily depend

Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

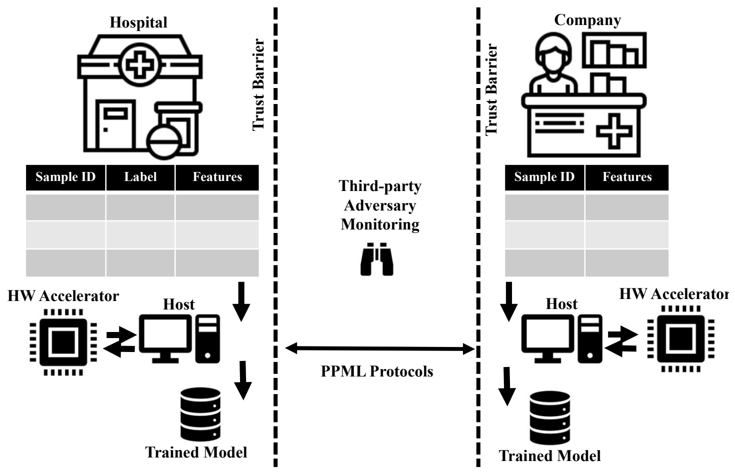

Figure 1: PPML allows two parties to securely train ML models on sensitive data.

on the data characteristics, scale, ownership, and security model, debates on technical roadmap never stop.

Fig. 1 shows an example of PPML in a healthcare scenario. A hospital and a pharmaceutical company collaborate to develop a predictive model for personalized medicine while protecting patient data. The parties have access to different sensitive patient records (labels and features). The parties use privacy-computing techniques to jointly train the model. The computational load is divided between the two parties, with each party performing local calculations and exchanging encrypted updates. The PPML scheme can prevent data security from being compromised beyond the trust barriers. On the one hand, the semi-honest parties act curiously and try to extract data privacy from each other. On the other hand, third-party adversaries can monitor the communication in the insecure network. The goal is to create an accurate model while preserving the privacy of individual patient data.

MPC covers a series of privacy-preserving techniques that support secure computation protocols on mathematically masked data. Garbled circuit (GC) is a secure two-party logical computation protocol, where the evaluation of each gate requires the transmission of a ciphertext look-up table. Secret sharing (SS) guarantees information-theoretic security by randomly sharing the raw data. However, arithmetic on the SS domain relies on intensive in-order data interaction. Even though MPC is versatile to different PPML scenarios, the network overhead always hinders the further development of MPC-based PPML with complex models in real-time applications.

HE-based schemes support multiple operators on encrypted data. Fully HE (FHE) schemes can ideally support any multiplication level by refreshing its noise budget with bootstrapping. Nevertheless, ciphertext evaluation and bootstrapping always require complex modular operations, which introduce tremendous computational overhead. Existing academic FHE accelerators are still expensive and only feasible on small-scale training and inference scenes [SFK+22]. On the other hand, additive HE (AHE) provides partial linear operators except for ciphertext-to-ciphertext multiplication with affordable overhead. However, purely AHE-based two-party schemes require a trusted party to generate and manage the secret key [HHIL+17].

There are gaps between research and practice when it comes to PPML applications in the real world. In the real world, three or more parties usually involve more commercial interests and regulations, so two-party PPML is the most common use case. In addition, the features are usually sparse in practice, such as intelligent risk control or wire fraud detection, because of the feature engineering such as one-hot [CZW+21]. And the performance of

PPML must also be considered. A task that takes a few minutes in non-private ML takes several hours when converted to PPML. Recent works show that hybrid SS-AHE solutions achieve 130× speedup with practical dataset and network bandwidth [CZW+21, FZT+21], compared with fully MPC-based PPML schemes. The key insight is to prevent the characteristics of the training set, *e.g.* sparsity, from being masked in the SS domain or encrypted in the AHE domain. This is achieved by keeping the samples as plaintext within their owner and only transferring small-size intermediate values in the HE domain. Participants can evaluate the layer functions with sparse operations and share the result in the SS domain other than revealing any sensitive values to one participant or a third party.

However, no existing hybrid PPML work tackles the challenges of co-designing implementations and optimizations of PPML protocols. New architectural considerations and methodologies are required when the complex HE algorithm combines SS protocol in the end-to-end PPML solutions. We observe that the computation and communication complexity, which are the bottlenecks of the two primitives, can complement each other. The standalone SS and HE approaches present a highly polarized communication-computation ratio, for which latency hiding between the data transfer and execution units provides little return. We optimize the hybrid approach based on the intuition that a well-balanced and parallel communication-computation flow can ideally reduce latency by 50%. This observation can also make the architecture less sensitive to network bandwidth, which typically dominates MPC performance. At the algorithmic level, it is helpful to tune hyper-parameters, such as an overflow-free pack level, to mitigate ciphertext explosion. Since practical computational settings are also critical for PPML, a ready-to-use architecture should be suitable for Field Programmable Gate Array (FPGA) platforms.

This paper presents a general architecture that can efficiently execute the SS-AHE hybrid PPML protocols on the large industrial-level training dataset. The architecture can generally handle different PPML tasks using hybrid primitives. The proposed design preserves privacy in either the SS domain or the AHE domain without relying on trusted hardware manufacturers. Compared with existing software-only hybrid PPML schemes [CZW+21, FZT+21], the co-designed architecture takes advantage of hardware units, and has more potential for optimization and acceleration. In summary, this paper makes the following contributions:

- A hybrid Secret-sharing and Homomorphic encryption Architecture for Privacy-pERsevering ML (*SHAPER*) with algorithm-protocol-hardware co-optimization between CPU, hardware accelerators, and network collaboration. SHAPER's hardware design improves throughput, and its software design optimizes data flow through system scheduling for latency overlap and parallelization.

- Vectorized high-performance modular multiplication (MM) engines to improve the efficiency of encryption, decryption, and ciphertext domain evaluation. We present new algorithmic and hardware optimizations for these operations of Paillier, including new MM algorithm, new hardware engine, pipelined execution, etc.

- SHAPER shows universal performance improvement on micro-operations and reduces the end-to-end latency by 94× on large-scale logistic regression training tasks, compared with the software-only benchmarks.

2 Background

Descriptions of backgrounds, threat model, and primitives are discussed in this section before introducing our SHAPER architecture.

2.1 Related Work

Various PPML schemes have been proposed to ensure the security of data and models with different cryptographic primitives. Actually, MPC-based PPML schemes [KVH+21, Kel20, ZXWG22] divide ML models into fragments of circuits, and then engage multiple parties to cooperatively perform circuit computations, including arithmetic and binary circuits, without additional privacy leakage. Afterwards, the results of these fragments of circuits are collected by the parties to construct the complete result of complex computational tasks. Historically, MPC was proposed in [Yao82], which solved the “Millionaire Problem” with GC. Not long after, SS-based MPC [GMW87] was proposed for better performance. Compared to GC, SS requires less computation and communication overhead, and outperforms GC in most scenarios. From a computational point of view, addition and multiplication are two important operations in SS-based MPC. The additions can be easily handled by both arithmetic and binary SS with little computation and no communication overhead. However, due to the data exchange required for each MPC multiplication, the communication overhead dominates the overall performance of SS-based MPC and grows significantly for large-scale datasets. Hence performance is still a barrier to practical application, although these common MPC techniques theoretically provide common solutions to PPML. Another potential direction of fully-MPC PPML frameworks is extending to multi-server settings, such as [MR18, WGC19, LX19, PS20, SGA20].

HE-based PPML [GBDL+16, MLS+20] provides the capability to perform operations on encrypted data to protect privacy. Unlike public-key cryptography, AHE supports not only key generation, encryption, and decryption, but also addition/multiplication over ciphertexts without private keys, thus revealing no information about the corresponding plaintexts. Due to the additions and multiplications that one can perform on the ciphertexts, HE-based PPML requires less communication than MPC-based PPML, but requires more computation for expensive HE encryption/decryption.

Fig. 2 describes a typical linear function in PPML, the sparse matrix is kept by its owner, Alice, and the result is shared between the participants Alice and Bob. Since AHE protects the confidentiality of the vector $\mathbf{y}$, Alice cannot recover the plaintext from the ciphertext $[\mathbf{y}]_b$. On the other hand, Alice shares $[\mathbf{z}]_b$ in line 5, which guarantees that Bob can only learn a masked result $\langle \mathbf{z} \rangle_b$. Recent works on the hybrid SS-AHE PPML [CZW+21, FZT+21] framework achieve 130× speedup over MPC-based schemes. AHE supports additions between ciphertexts and multiplications between ciphertexts and plaintexts. However, most of the existing AHE algorithms, such as Paillier [Pai99], DGK [DGK07], OU [OU98], depend on large integer modular operations, especially modular multiplications (MM) and exponentiation (ME), which incur large computational overheads. Therefore, the overall performance of SS-AHE hybrid PPML is strongly dominated by the efficiency of the basic modular multiplications. Montgomery modular multiplication [Mon85] is the most classical method, while new modular algorithms have also been proposed recently [LC21].

2.2 Preliminaries

2.2.1 Paillier Cryptosystem

We choose Paillier as the AHE example in our architecture. The Paillier cryptosystem consists of the following interfaces.

(1)Key Generation:

- Randomly choose two large primes (p, q) of equal length that satisfy $gcd(pq, (p-1)(q-1)) = 1$.
- Calculate $n = pq$ and $\lambda = lcm(p-1, q-1)$.
- Randomly select $g \leftarrow \mathbb{Z}^*_{n^2}$.

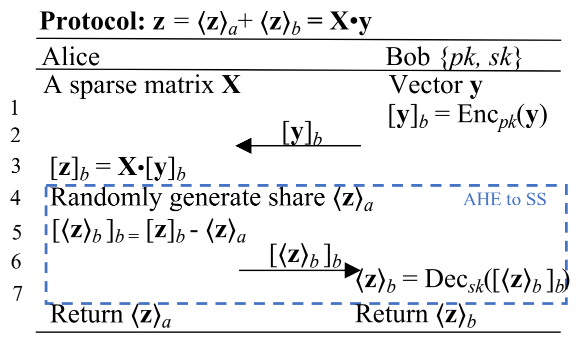

Figure 2: SS-AHE-based secure matrix-vector multiplication.

- Define the function L as $L(x) = \frac{x-1}{n}$. Calculate $\mu = (L(g^{\lambda} \bmod n^2))^{-1} \bmod n$.
- Set (n, g) as the public key and (λ, μ) as the private key.

(2) Encryption:

- Randomly choose a positive integer r that satisfies $0 < r < n$.
- Calculate $c = g^m r^n \bmod n^2$.

(3)Decryption:

- Compute $m = L(c^{\lambda} \bmod n^2)\mu \bmod n$.

Paillier supports ciphertext-ciphertext addition (CCAdd), plaintext-ciphertext addition (PCAdd), and plaintext-ciphertext multiplication (PCMult):

- $c_1 c_2 \bmod n^2 = g^{m_1} r_1^n \times g^{m_2} r_2^n \bmod n^2 = g^{m_1+m_2}(r_1 r_2)^n \bmod n^2 = c_{m_1+m_2}$.
- $c_1 \times g^{m_2} \bmod n^2 = g^{m_1} r^n \times g^{m_2} \bmod n^2 = g^{m_1+m_2}(r)^n \bmod n^2 = c_{m_1+m_2}$
- Similarly, we have $c_1^a \bmod n^2 = c_{ma}$.

As suggested in [DJ01, CGHGN01], we choose primes (p, q) which satisfy $p = q = 3 \bmod 4$ and $gcd(p-1, q-1) = 2$ and set $g = n+1$, so that g^m can be simplified as:

$$\begin{aligned} g^m = (n+1)^m &= \begin{bmatrix} m \\ 0 \end{bmatrix} n^m + \begin{bmatrix} m \\ 1 \end{bmatrix} n^{m-1} + \ldots + \begin{bmatrix} m \\ 2 \end{bmatrix} n^2 + mn + 1 \bmod n^2 \\ &= mn + 1 \bmod n^2 \end{aligned} \tag{1}$$

And we have $\mu = \lambda^{-1}$. The key generation randomly selects $x \leftarrow \mathbb{Z}_n^*$, and adds $h_s = -x^{2n} \bmod n^2$ into the public key. Then the encryption is modified as $c = (mn + 1)h_s^a \bmod n^2$, where a is randomly chosen in $\mathbb{Z}_{2^{\lceil \frac{\lfloor n \rfloor}{2} \rceil}}$.

The optimization has two advantages. First, the exponentiation g^m is simplified as a multiplication mn. Second, since a is much shorter than n, it is easier to compute hs^a than r^n.

2.2.2 Additive Secret Sharing

A value additively shared by two parties refers to $[\![x]\!] = (x_1, x_2)$, where $\sum x_i = x$ over field $\mathbb{F}$, and (x_1, x_2) are random. The addition over additive shares is almost free, as $[\![x+y]\!] = (x_1 + y_1, x_2 + y_2)$. The multiplication over shares is more tricky. A common approach is to use Beaver triples [Bea92]. The Beaver triple is three shared random values

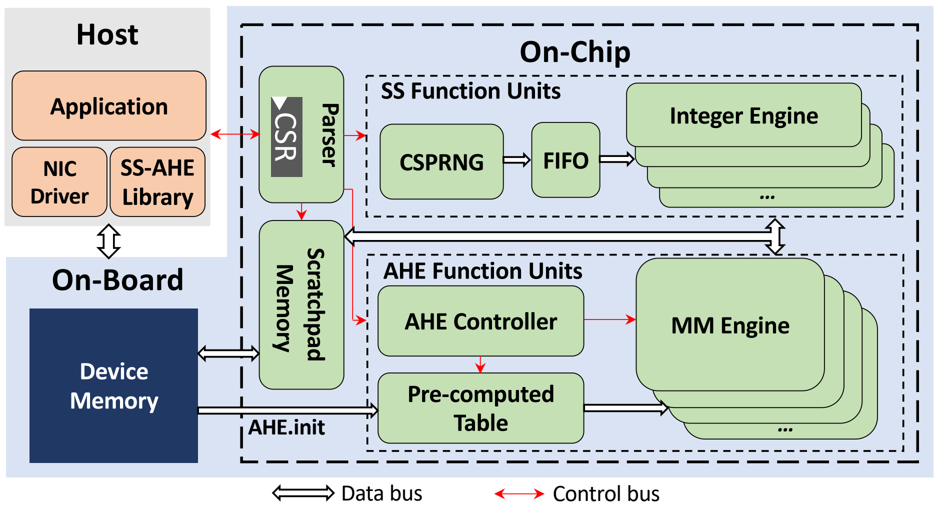

Figure 3: SHAPER Architecture Overview – The grey and blue boxes represent software and hardware components, respectively.

$[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, constrained by $c = ab$. When computing $[\![xy]\!]$, the parties compute and reveal $\alpha = x - a$, $\beta = y - b$, and then the product shares can be constructed with local additions.

$$[\![xy]\!] = [\![c]\!] + \alpha[\![b]\!] + \beta[\![a]\!] + \alpha\beta \tag{2}$$

MPC-based PPML schemes require a large number of beaver triples because each multiplication consumes a triple. Beaver triples can be generated in batch using Paillier [DSZ15, P+13].

2.3 Threat Model

As a co-designed architecture, the threat model of SHAPER takes into account cross-layer assumptions.

At the protocol level, the adversary model follows the semi-honest assumption in a 2-party setting, as SHAPER mainly focuses on implementing and accelerating existing semi-honest schemes [FZT+21, CZW+21]. In the semi-honest model, a probabilistic polynomial-time adversary with semi-honest behaviors controls one of the parties and the adversary can corrupt and control one party, and try to learn more information about the other honest party's input, such as recovering the secret messages sealed in the ciphertexts or shares. Meanwhile, the adversary is required to follow the protocol specification honestly. The semi-honest setting is adopted by most existing PPML models, such as [MZ17, MR18].

At the algorithm level, including SS and AHE, the security is given as a security parameter, which defines the hardness of the algorithm the adversary attempts to break. The parameter is positively related to the key lengths. A 2048/3072-bit Paillier cryptosystem corresponds to a 112/128-bit computational security parameter.

3 Architecture Design

To accelerate the hybrid SS-AHE framework, SHAPER proposes an instruction set and explores efficient design methodologies of AHE, SS, and conversion functions.

3.1 Architecture overview

An overview of our proposed SHAPER architecture is shown in Fig. 3, which includes both software and hardware implementations. The host application controls the start and convergence conditions of the training tasks, and also consults the hyper-parameters between the participants, such as the optimal plaintext packing level, the pre-computation window size, etc. The on-chip hardware modules aim at fast computation on basic primitives, mainly including AHE and SS function units. Since AHE computation is still a performance bottleneck of hybrid PPML schemes [CZW+21], we design new MM algorithms and hardware engines in the AHE units, implement algorithmic optimizations in hardware, and improve the scheduling modules to achieve better acceleration.

SHAPER focuses on 2-party PPML, which is the most common case in industrial applications. The application calls the SS-AHE library, which supports execution flows encapsulated as kernel functions. The kernel functions update algorithm parameters and architecture flags by setting control and status registers (CSRs), and implement the security primitives with customized instructions summarized in Table 1. SHAPER analyzes the control flow dependencies and packs the instructions in VLIW style, ensuring that the packed instructions in a VLIW instruction can be executed in parallel. SHAPER adopts the static scheduling scheme. Each VLIW instruction packs RISC instructions which decode and issue synchronously. Since the instructions are executed sequentially and deterministically, memory allocation is scheduled in a static manner. To communicate with other participants, all network interaction is handled by a network interface card (NIC). The host application always waits for the NIC and SHAPER to interrupt. Since the runtime and driver layers are common components in HW/SW co-design, we omit them in Table 1 for brevity.

On the SHAPER hardware, the parser unpacks the instructions and dispatches them to the appropriate function units. **AHE.init** reloads data from device memory during the offline phase when the host updates its key pair. Other AHE instructions consist of a series of MM operations handled by the AHE controller. **SS.gen** returns a vector of random shares sampled from the cryptographic-secure pseudorandom number generator (CSPRNG). **Int.add** and **Int.mul** perform a series of integer arithmetic operations in a continuous address space. The memory hierarchy consists of the on-board device memory and the on-chip scratchpad managed by **DM.ld/st** and **SPM.ld/st**.

3.2 Algorithm-Protocol Co-Optimization

Fig. 4 describes the methodology for analyzing and exploring the PPML solutions. We map a task to the coordinate point according to the computational and communication overhead. The network bandwidth is represented as a dotted guideline, points on which have the same communication and computation latency. The schemes above the guideline (*e.g. SecureML* [MZ17]) are communication dominated. On the other hand, the communication-less FHE solutions (*e.g. CraterLake* [SFK+22]) cost most of the time for ciphertext evaluation. The position of SS-AHE-based solutions depends on computational power, especially the performance of cryptographic engines. In our work, the following optimizations are applied to explore an optimal solution.

3.2.1 Data Characteristic

In real-world scenes, the training dataset is sparse due to incomplete user information and one-hot encoding [CZW+21]. Since SS-AHE schemes preserve the data sparsity, the number of instructions is significantly reduced.

Table 1: The instruction set supported by SHAPER.

Instruction	Arguments*	Description
AHE.init	*len, dm_ptr*	Initialize the pre-computed table
AHE.enc	*i_pt_ptr, i_pk_ptr, o_ct_ptr*	Encrypt a plaintext or secret share to ciphertext
AHE.dec	*i_ct_ptr, i_sk_ptr, o_pt_ptr*	Decrypt a ciphertext to plaintext or secret share
AHE.ccadd	*i_cta_ptr, i_ctb_ptr, o_ct_ptr*	A ciphertext adds another ciphertext
AHE.pcadd	*i_pt_ptr, i_ct_ptr, o_ct_ptr*	A plaintext adds a ciphertext
AHE.pcmul	*i_pt_ptr, i_ct_ptr, o_ct_ptr*	A plaintext multiplies a ciphertext
SS.gen	*len, o_pt_ptr*	Generate fresh secret shares
Int.add	*len, i_pt_ptr, i_pt_ptr, o_pt_ptr*	Addition in SS or plaintext domain
Int.mul	*len, i_pt_ptr, i_pt_ptr, o_pt_ptr*	Multiplication in SS or plaintext domain
DM.ld/st	*len, dm_ptr, host_ptr*	Device memory load/store a block of data from/to the host
SPM.ld/st	*len, spm_ptr, dm_ptr*	SPM load/store a block of data from/to the device memory

* The argument *len* is the length of input or output data. The arguments *_ptr* is the physical base address of a specific data structure.

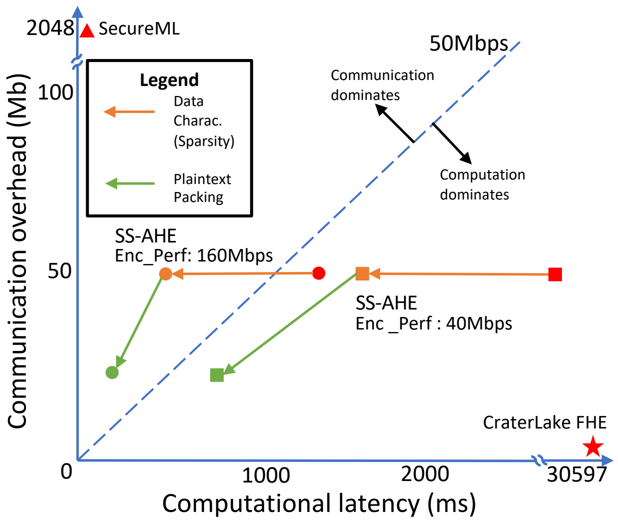

Figure 4: Exploring optimization space on data characteristics and algorithms. The example network bandwidth is 50Mbps. Applying the optimizations reduces the encryption overhead for the 40Mbps and 160Mbps throughput configurations.

3.2.2 Plaintext Packing

Packing multiple ciphertexts of short plaintexts into one ciphertext greatly reduces the number of ciphertexts and allows SIMD-style computation [P+13], as explained in Sec. 3.5. The packing strategy reduces the communication overhead for transmission and the computational overhead for decryption at the expense of additional homomorphic computation over ciphertexts. .

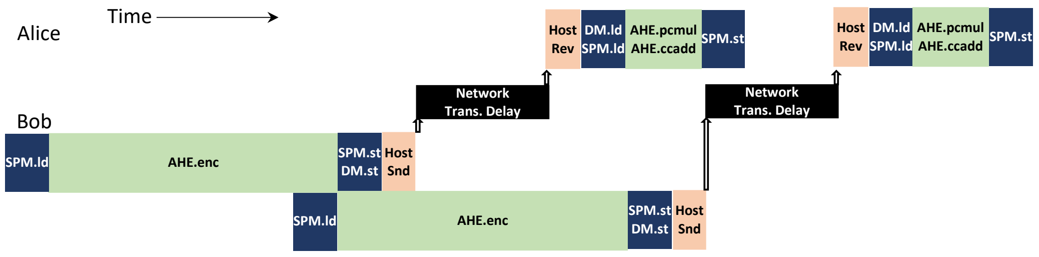

Figure 5: The pipeline execution process of SHAPER, corresponding to line 1 to 3 in Fig. 2. Two successive executions overlap their latency.

3.2.3 Latency Hiding

Since the SS-AHE schemes have balanced overhead, overlapping computation and communication brings more benefits. Fig. 5 shows the pipeline execution process of SHAPER, corresponding to line 1 to 3 in Fig. 2. The AHE encryption is the most time-consuming operation in the example, and can hide other delays. Once the first encryption is complete, the second encryption and the transmission of the first ciphertext are performed in parallel in a pipelined flow. In this case, the computation instructions overlap the communication delay. SHAPER consumes the data as soon as the source data is created with multi-buffer transfer.

3.3 Efficient AHE Function Units

The AHE unit of SHAPER consists of a Paillier controller and several MM engines. The controller manipulates MM engines to compute the functions of the Paillier cryptosystem with key length $|n| = 3072$ in parallel. Each MM engine implements our proposed fast MM algorithm, which supports a 5-stage pipeline. To accelerate the modular exponentiation (ME) in the Paillier encryption, a set of Ultra-RAMs (URAMs) and Block-RAMs (BRAMs) are deployed to store the public/private keys of the device, as well as some pre-computed values.

When executing an AHE instruction, the controller divides it into multiple multiplications and exponentiations based on DJN optimizations of Paillier [CGHGN01]. Several optimizations suggested in [DSZ15] are considered, including Chinese-Remainder-Theorem (CRT) optimization and fixed-base pre-computation (see Appendix B), which scales down both the base size and the exponent size of the ME. The call to a single ME is divided into multiple multiplications in SHAPER, and the controller then schedules the datapath between different MM engines to compute the ME collaboratively.

The performance of MM engines has a large impact on the efficiency of various AHE interfaces and higher-level applications. Therefore, we propose an efficient MM construction with optimizations in both algorithm and hardware implementation.

3.3.1 The MM Algorithm

Our proposed MM algorithm is inspired by the shift-sub algorithm in [LC21] (see Appendix A), which has the advantage of dealing with large integers. The algorithm requires multiple serial full adders, one for each bit of b, which results in long data paths. To avoid multiple serial additions of large integers, we propose a high-radix shift-sub MM algorithm as described in Alg. 1. Our high-radix shift-sub deals with k bits of b in a single iteration, rather than a single bit, where k is the radix width. Single-bit shift sub in [LC21] deals with a single bit of b in each iteration. Therefore, the strategy does not

Algorithm 1 High-radix Shift-sub Modular Multiplication. $\mathbf{MM}(a, b, m)$

```
Require: Radix width $k$
Require: $a = \sum_{i=0}^{\tau-1} a_i 2^{ki}$, $b = \sum_{i=0}^{\tau-1} b_i 2^{ki}$, $\forall i \in [0, \tau-1], a_i, b_i < 2^k$. $a, b < m < 2^{k\tau}$, $m \bmod 2 = 1$
Ensure: $c = ab \mod m$
1: $c = 0$
2: for $i = 0$ to $\tau - 2$ do
3:     $c = c + b_i a$                              ▷ Multiply-Accumulate Phase ($Phase\_c$)
4:     $a = \mathbf{QR}(a \ll k, m, k)$              ▷ Shift-Reduction Phase ($Phase\_a$)
5: end for
6: $c = \mathbf{QR}(c + b_{\tau-1} a, m, k + \lceil \log \tau \rceil)$   ▷ Final Round of Accumulation-Reduction
7: return $c$.
```

Algorithm 2 Quick Barrett Reduction with MSBs Approximation. $\mathbf{QR}(a, m, \Delta)$

```
Require: Length upper bound $\Delta$, $\hat{m} = m \gg (l - \Delta - 2)$, $m' = \lfloor \frac{2^{2\Delta+2}}{\hat{m}+1} \rfloor$
Require: $m \in [2^{l-1}, 2^l)$, $a \in [0, 2^{l+\Delta})$, $l \geq \Delta + 2$
Ensure: $b = a \mod m$
1: $a' = a \gg (l - \Delta - 2)$                ▷ MSB Shift
2: $\gamma = (a'm') \gg (2\Delta + 2)$           ▷ Barrett Reduction
3: $b = a - (\gamma + 1)m$
4: if $b < 0$ then
5:     $b = b + m$                               ▷ Correction
6: else
7:     $b = b - m < 0 ? b : b - m$
8: end if
9: return $b$
```

work well in hardware design as k grows, since it leads to too many cycles when dealing with significantly large a and b. A more efficient MM algorithm is needed to speed up Paillier in hardware.

Our high-radix MM algorithm processes k bits of b in each round, and has τ rounds in total. Each round consists of a Multiply-Accumulate phase (*Phase_c*) and a Shift-Reduce phase (*Phase_a*). In the i-th round, *Phase_c* multiplies the i-th piece of b by the current round's a and adds the product to the accumulation of previous rounds. In the first $\tau - 1$ rounds, *Phase_a* updates the next round's a with the current round's a. a is modulo reduced after shifting k bits to the left. In the final round, *Phase_a* modulo reduces the accumulation of *Phase_c* to get the final result. The correctness of the algorithm is guaranteed:

$$ab \bmod m = a \times \sum_{i=0}^{\tau-1} b_i 2^{ki} \bmod m = \sum_{i=0}^{\tau-1} (a 2^{ki} \times b_i) \bmod m \tag{3}$$

Note that except for the final round, there is no data dependency between *Phase_c* and *Phase_a*. Therefore, *Phase_c* and *Phase_a* can be executed in parallel to reduce latency. After the latencies of multiplication and addition in *Phase_c* are hidden by parallelization, the total execution time of one MM is reduced by more than 30%.

Since the modular reductions of *Phase_a* have additional length constraints, we propose a quick modular reduction algorithm **QR** in Alg. 2. We note that the inputs of the modular reduction have upper bounds. Each round's $a \ll k$ is less than $m \ll k$, and the final round's c is less than $\tau m \ll k$. Therefore, the **QR** algorithm limits the length of the dividend to no more than $(l + \Delta)$, where l is the length of the modulus m. The Δ is set to k in the first $\tau - 1$ rounds and to $k + \lceil \log \tau \rceil$ in the final round. The radix k has a large impact on the total number of rounds, as well as the efficiency and consumption of hardware implementations in SHAPER. Different choices of k are discussed in the next subsection about hardware implementation.

We propose and adopt a new strategy using the Most Significant Bits (MSB) approximation to simplify the reduction. Unlike the remainder (*i.e.*, the output of the modular reduction), the quotient in a division is mainly determined by the MSBs of the dividend and divisor, while the lower bits contribute little to the quotient. Therefore, the algorithm approximates a quotient γ with the MSBs of a and m, and then computes an approximate remainder with $a - \gamma m$, which is then modified to the result with a conditional subtraction. The error between the approximated and the precise quotients is proven to be within 1 if we use the most significant $\Delta + 2$ bits of m for the approximation, as in Eq.(4,5). (Note that $\forall x, y \in \mathbb{R}$, if $|x - y| \leq 1$, then $|\lfloor x \rfloor - \lfloor y \rfloor| \leq 1$.) Furthermore, the existing Barrett approximation [Bar86], which converts the division of a' by $\hat{m} + 1$ into the product of a' and m', is further used to simplify the quotient computation, which involves a deviation of no more than 1, as Eq.(6,7) shows. Therefore, at most two conditional subtractions are needed in the algorithm.

$$\frac{a}{m} \geq \frac{a' \times 2^{l-\Delta-2}}{m} = \frac{a'}{m/2^{l-\Delta-2}} > \frac{a'}{m \gg (l-\Delta-2)+1} = \frac{a'}{\hat{m}+1} \tag{4}$$

$$\begin{aligned} \frac{a}{m} - \frac{a'}{\hat{m}+1} &< \frac{a'+1}{\hat{m}} - \frac{a'}{\hat{m}+1} = \frac{(a'+1)+\hat{m}}{\hat{m}(\hat{m}+1)} \\ &\leq \frac{\hat{m}2^{\Delta+1}+\hat{m}}{\hat{m}(\hat{m}+1)} = \frac{2^{\Delta+1}+1}{\hat{m}+1} < 1 \end{aligned} \tag{5}$$

$$\begin{aligned} \frac{a'}{\hat{m}+1} - \frac{a'm'}{2^{2\Delta+2}} &= \frac{a'[2^{2\Delta+2} - m'(\hat{m}+1)]}{2^{2\Delta+2}(\hat{m}+1)} \\ &= \frac{a'[2^{2\Delta+2} \mod (\hat{m}+1)]}{2^{2\Delta+2}(\hat{m}+1)} \geq 0 \end{aligned} \tag{6}$$

$$\begin{aligned} \frac{a'}{\hat{m}+1} - \frac{a'm'}{2^{2\Delta+2}} &= \frac{a'[2^{2\Delta+2} - m'(\hat{m}+1)]}{2^{2\Delta+2}(\hat{m}+1)} \\ &< \frac{a'(\hat{m}+1)}{(\hat{m}+1)2^{2\Delta+2}} = \frac{a'}{2^{2\Delta+2}} < 1 \end{aligned} \tag{7}$$

In fact, **QR** computes $b = a - (\gamma + 1)m$ instead of $b = a - \gamma m$. A small change is made to accommodate the hardware implementation. Additions over large integers are not cheap in hardware. Direct computation of $b = a - \gamma m$ results in two additions in the worst case. But when switching to $(\gamma + 1)m$, the hardware engine determines whether to compute $b - m$ or $b + m$ based on the sign bit of b. Computing $\gamma + 1$ is cheap because γ is short, and this small cost reduces the additions of large integers here from 2 to 1. The optimization effectively reduces the consumption of on-board resources in the hardware implementation.

3.3.2 The MM Engine

The MM engine (shown in Fig. 6) is the fundamental processing element that supports our MM algorithm. It can be divided into two modules: one for *Phase_c* and one for quick reduction in *Phase_a* respectively. *Phase_c* contains a block multiplication (BM) module and an adder module for accumulation, consisting of a carry-save adder and a ripple-carry adder. *Phase_a* contains a multiplier to compute γ, a BM module for $-(\gamma + 1)m$, an adder for $a - (\gamma + 1)m$, and a Conditional Subtraction (CS) module for correction. A CS

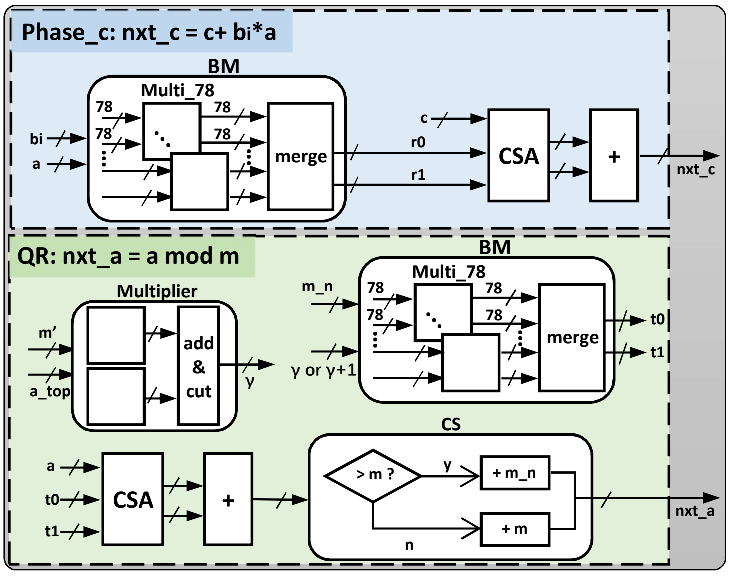

Figure 6: The architecture of a single MM engine based on the MM Algorithm.

module contains an adder that performs either $-m$ or $+m$. Note that the subtraction $-m$ is replaced by $+m\_n$, where m_n is the complement of m. *Phase_a* and *Phase_c* are updated in parallel to compress the total number of clock cycles. Two phases need no data exchange except for *Phase_c* fetching a at the beginning of each round. Also, to run an integrated MM, a controller is needed to schedule the input/output of the MM engine in each round.

We implement the BM modules with multiple multipliers on a smaller scale. Specifically, a $k \times 3072$ multiplication is divided into $k \times k$ parts, whose subproducts are combined into two large integers. The $k \times k$ multipliers are implemented using on-board digital signal processing (DSP) units.

When exploring the design space of on-board resources and hardware clock cycles, we choose radix $k = 72$. A single piece of DSP supports 27×18 multiplication. For example, implementing 64×64 multiplication requires 12 DSP units, but the DSP utilization rate is only about 70%, since 12 DSP units theoretically allow 81×72 multiplication. The utilization rate peaks at 100% when the multiplier width equals 54 or 108. At 54, a multiplier needs only 6 DSPs, but the round number τ rises to 57 with a 3072×3072 multiplication, leading to more clock cycles of the MM engine. In contrast, in the case of 108, there are only 29 rounds, but one multiplier requires 24 DSPs, making the MM engine too large. A larger MM engine consumes more resources (especially DSP). Therefore, fewer MM engines can be placed in the FPGA implementations, which reduces the overall hardware parallelism. In addition, a larger engine has longer data paths, further reducing the frequency of the hardware implementation.

Considering the trade-off between time and space, we choose the case of the 78-bit multiplier, which requires 40 rounds of multiplication and 12-DSP multipliers. Although the DSP utilization rate cannot reach 100%, it is still higher than 90%. For hardware compatibility of the final round, the hardware implementation should support $k + \lceil \log \tau \rceil$ bit quick reduction, as shown in Fig. 6. Therefore, the radix k is finally fixed to 72.

Another bottleneck in the design of an MM engine is the 3072-bit addition, because it introduces large logic delays that limit the hardware frequency. In our design, however, we optimize serial adders with a prediction strategy, along with splitting two addends into multiple 128-bit chunks. Each such chunk (x, y) uses two 128-bit ripple-carry adders to compute two potential sums, $x + y$ and $x + y + 1$. Using the carry bit propagated from the lower chunk, a multiplexer selects one of the summations and propagates the corresponding carry bit to the higher chunk. Since $x+y+1 \leq 2 \times (2^{128} - 1) + 1 = 2^{129} - 1$, the propagation will not lead to the growth of the carry bit, and there is at most one carry bit during the propagation, which guarantees the correctness of the strategy.

We set the chunk size to 128, taking into account resource consumption and maximum frequency. When the chunks are large, the logic delay within each chunk is still large, and the frequency cannot be improved efficiently. However, when the chunks are particularly small, although the resource consumption for each chunk is reduced, the logic delay outside the chunks to merge the subsums into the final output increases. And this also leads to a decrease in frequency. Therefore, we set the chunk size to 128 to get the peak frequency.

The acceleration of the MM engine over the MM implementation on a standard CPU is discussed in section 5.3.

3.3.3 Paillier Controller

A 3072-bit Paillier cryptosystem has a 3072-bit message space ($|n| = 3072$) and a 6144-bit ciphertext space ($|n^2| = 6144$). Making the hardware compatible with the 6144-bit modulo operation is a waste of on-board resources. And our Paillier controller uses Chinese Remainder Theory to convert 6144-bit modulo operations in encryption and decryption to 3072-bit modulo operations. We follow the dataflow of open-sourced Paillier implementation of [DSZ15], and transform it into a micro-instruction control flow that takes more advantage of the MM engines. CRT gives a unique solution to simultaneous linear congruences with coprime moduli. Since $n^2 = p^2q^2$, CRT transforms modulo operations over n^2 into modulo operations over p^2 and q^2.

The prerequisite for CRT optimization is obtaining the private key, since the private key λ can be computed as $\lambda = (p-1)(q-1)/2$. In public key cryptosystems, it is assumed that the encryptor does not have the private key. However, in HE (including AHE) scenarios, the encrypting party usually has the private key. HE scenarios in hybrid PPML schemes are analogous to proxy execution, such as the matrix multiplication in Fig. 2. Both encryption and decryption are handled locally by the client, and the server only handles execution over ciphertexts. Therefore, it makes sense to optimize Paillier encryption on the client side with CRT.

Encryption. Eq.8 shows the DJN-Paillier encryption.

$$c = (mn + 1)hs^a \mod n^2 \tag{8}$$

To optimize hardware computation with CRT, the controller first computes the projections of c over the modulo field p^2 and q^2.

$$\begin{aligned} c_p = c \mod p^2 = [(mn \mod p^2) + 1] \times ((hs \mod p^2)^a \mod p^2) \mod p^2 \\ c_q = c \mod q^2 = [(mn \mod q^2) + 1] \times ((hs \mod q^2)^a \mod q^2) \mod q^2 \end{aligned} \tag{9}$$

$(hs \mod p^2)^a \mod p^2$ and $(hs \mod q^2)^a \mod q^2$ are under fixed bases depending on the public keys, and can be computed with precomputed tables. The details of computing ME under fixed bases with precomputation are explained in Sec. 4.2. After computing c_p and c_q, the ciphertext c can be recovered.

$$c = c_p + [(c_q - c_p)(p^{-2} \mod q^2) \mod q^2] \times p^2 = c_p + t_c \times p^2 \tag{10}$$

$(p^{-2} \mod q^2) \mod q^2$ also depends on the keys, and will be precomputed during key generation. The control flow of Paillier encryption at the micro-instruction level is listed as follows. ME_P refers to modular exponentiation with precomputation. Note that the final step of encryption, $c_p + t_c \times p^2$, necessarily requires 6144-bit computations, so the hardware sends back c_p and t_c to the host to obtain the ciphertext.

```
# hsp_table, hsq_table: precomputed exponentiation table
# p2, q2: precomputed p^2, q^2
# pq2: precomputed p^{-2} mod q^2
ME_P hsp_table, a, tmp1
MM m, n, p2, tmp2
ADDI tmp2, tmp2, 1
MM tmp1, tmp2, p2, cp
ME_P hsq_table, a, tmp1
MM m_ptr, n, q2, tmp2
ADDI tmp2, tmp2, 1
MM tmp1, tmp2, q2, cq
SUB cq, cp
MM cq, pq2, q2, tc
# Sending back cp and tc to the host
```

Decryption. Decryption can also benefit from CRT. Eq.11 shows the decryption of DJN-Paillier.

$$m = L(c^\lambda \mod n^2)\lambda^{-1} \mod n = \frac{(c^\lambda \mod n^2) - 1}{n} \times \lambda^{-1} \mod n \tag{11}$$

The controller follows the same CRT strategy to calculate the intermediate $d = c^\lambda \mod n^2$.

$$\begin{aligned} d_p &= d \mod p^2 = c^\lambda \mod p^2 = (c \mod p^2)^\lambda \mod p^2 \\ d_q &= d \mod q^2 = c^\lambda \mod q^2 = (c \mod q^2)^\lambda \mod q^2 \\ d &= d_p + [(d_q - d_p)(p^{-2} \mod q^2) \mod q^2] \times p^2 = d_p + t_d \times p^2 \end{aligned} \tag{12}$$

To continue the decryption, a naive approach is to send d_p and t_d back to the host, which recovers d, calculates $\frac{d-1}{n}$, and returns it to the hardware. However, this approach obviously lacks efficiency, since it involves a round of communication for each decryption. Therefore, we expect the controller to compute m directly using d_p and t_d.

$$m = \frac{d-1}{n} \times \lambda^{-1} \mod n = (\frac{d_p - 1}{n} + \frac{t_d \times p^2}{n}) \times \lambda^{-1} \mod n (d_p = 1) \tag{13}$$

For any legal plaintext-ciphertext pair (m, c), the correctness of Paillier holds as $c^\lambda - 1 = \lambda mn (\text{mod} n^2)$. Since $n = pq$, we have $d - 1 \mid p$, and of course $d_p = d \bmod p^2 > 0$. Suppose p is the smaller prime between (p, q), then $d_p < p^2 < n$. Therefore eq.14 holds.

$$\lfloor \frac{d_p - 1}{n} \rfloor = 0 \tag{14}$$

Since $d - 1 \mid n$, we get m from Eq.15.

$$m = \lceil \frac{t_d \times p^2}{n} \rceil \times \lambda^{-1} \mod n \tag{15}$$

The control flow of the Paillier decryption at the micro-instruction level is listed as follows. RED modulo reduces a 6144-bit input with a 3072-bit modulus, and MDIV returns the quotient of the three inputs $x \times y/z$. Both RED and MDIV can be supported by a slightly modified MM engine. Specifically, *Phase_a* can independently handle RED in τ rounds, and MDIV can be implemented by collecting the corrected γ in each round of *Phase_a* and merging them into the final quotient.

```
RED c, p2, tmp1
ME tmp1, lambda, p2, dp
RED c, q2, tmp1
ME tmp1, lambda, q2, dq
SUB cq, cp
MM cq, pq2, q2, td
MDIV td, p2, n, tmp2
CMP cp, 1
JE s
ADDI tmp2, tmp2, 1
s: MM tmp2, inv_lambda, n, m
# Return plaintext m
```

3.4 Secret Sharing Function Units

Although computation is not the critical overhead in SS-based schemes compared to communication, we design a dedicated SS unit in SHAPER due to the following latency-related concern: In hybrid PPML schemes such as [CZW+21,FZT+21], the computation of AHE and SS is interleaved, which means that the data must be transmitted frequently between the host and the hardware if the hardware is not capable of computing SS functions locally. Therefore, each transfer requires reading/writing from device memory, and introduces non-negligible redundant latency.

The SS unit consists of multiple integer processing engines and a CSPRNG. The CSPRNG generates the random numbers used in SS schemes. And the integer processing engines support computation over 64-bit integers, which is a common choice in SS-based schemes.

For higher random number generation throughput, we have optimized the CSPRNG in the SS unit with several improvements. Actually, existing CSPRNG constructions in PPML usually use ECB-mode AES encryption, which benefits a lot from AES-NI hardware extensions. However, recent works [XHY+20] pointed out that SHA3-based CSPRNG would outperform AES hardware implementations because SHA-3 takes advantage of its 1600-bit Keccak structure and fast binary executions, resulting in higher throughput during each iteration. In addition to the SHA-3 Keccak engine, we also use a first-in-first-out (FIFO) buffer to cache the generated random numbers. The SHA-3 Keccak engine dynamically generates random numbers and pushes them into the buffer when it is not full. And SS units pop random numbers from the buffer as needed.

3.5 Conversions between Primitives with Packing

SHAPER supports an efficient SIMD-style conversion between SS and AHE. We adopt and improve the conversion protocols in [FZT+21]. Details are shown in Appendix C. The conversions can be optimized with the packing strategy proposed in [P+13].

The strategy packs different ciphertexts into one. For example, if the server normally computes x_1y_1 and x_2y_2 over ciphertexts, two ciphertexts $c_{x_1y_1}$ and $c_{x_2y_2}$ are sent back to the client. And the client decrypts them to get the plaintexts. However, when using the packing strategy, the server computes the ciphertext of $x_1y_1 + x_2y_2 \times 2^i$. Then the client decrypts the ciphertext and truncates the plaintext to get x_1y_1 and x_2y_2. This reduces the number of decryptions from 2 to 1. The packing strategy works if $x_1y_1 < 2^i$, otherwise an overflow occurs and the MSBs of x_1y_1 get mixed with the LSBs of x_2y_2. Also, $x_1y_1 + x_2y_2 \times 2^i$ should stay within the message space.

The Paillier message space (3072 bits) is too large for the values in the models (within 64 bits), and decryption in Paillier costs much more than encryption. Therefore, the space

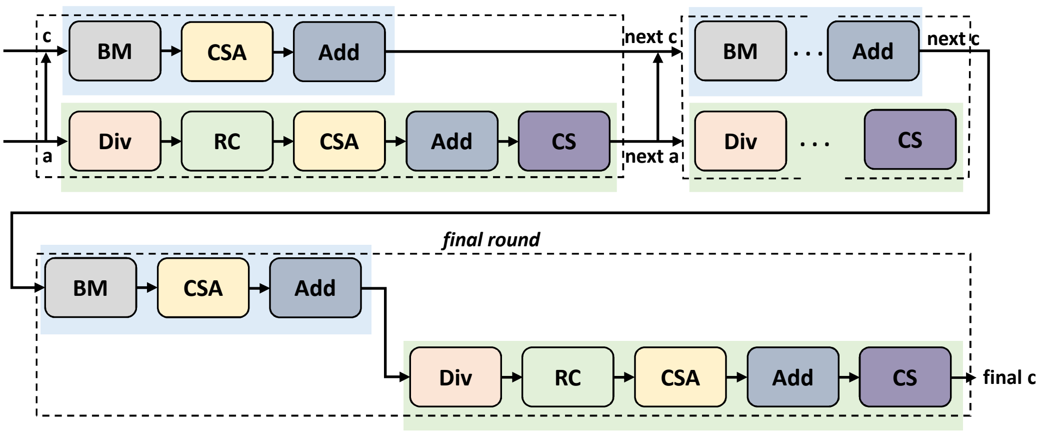

Figure 7: Pipelined modular multiplication engine. Each round of MM in Alg. 1 is divided into five stages. In the first τ-1 rounds, the accumulation phase (c) and the QR phase (a) are executed in parallel, and in the final round the phases run in serial.

can be divided into smaller buckets, with a smaller value in each bucket. The computation results in each bucket will not interfere with others if the bucket size is large enough to ensure that there is no overflow in the subsequent computation. Since there are fewer ciphertexts to send and decrypt, both communication and computation are greatly reduced.

The optimal bucket size depends on the computation under encrypted values in different protocols. In hybrid schemes, the optimal bucket size is roughly $(m+1)l + \log(a+1) + \sigma$ to ensure that there is no overflow in each bucket [P+13], where l is the share length, usually equal to 64. m and a are the numbers of multiplication with plaintext shares and addition with other shares. σ is the statistical security parameter of the scheme, which is 40 by default. For example, AHE handles matrix multiplication in [CZW+21], where one multiplication and multiple additions are processed with each ciphertext. And the bucket size can be set to 180 by default, where each ciphertext contains about 17 buckets. The 180-bit bucket size remains valid unless the number of additions in the matrix multiplication exceeds 2^{12}. Then the decryption overhead can be reduced at the expense of more ciphertext additions and plaintext multiplications for packing.

4 Performance Optimizations and Security Enhancement

Several optimizations are applied to our implementation to improve performance and FPGA resource efficiency.

4.1 Parallel Modular Operations

We observe that there are a large number of matrix computations in real-world PPML scenarios [CZW+21, FZT+21], consisting of multiple AHE operations without data dependency. Meanwhile, a single Paillier encryption can also benefit from parallelization, since fixed-base pre-computation is involved to improve efficiency. Therefore, we provide support for vectorized modular operations in our design.

The pipeline implementation of the MM engine has five stages for each iteration, as shown in Fig. 7. Each stage takes 4 execution cycles. Since *Phase_a* has a longer datapath than *Phase_c*, we divide the datapath into five. The division (Div) stage computes the Barrett division in Alg. 2 to obtain γ. The Reduction Computation (RC) stage multiplies $\gamma + 1$ by $-m$. The CSA stage includes the carry-save adders to merge $c + b_i a - km$ into a single addition, and the Add stage includes an optimized ripple-carry adder. The CS stage performs the conditional subtraction of a. The pipelined datapath of *Phase_c* consists of 3 stages, BM, CSA, and Add. The block multiplication (BM) stage computes $b_i a$ in Alg. 1, and the hardware implementation of the BM stage is identical to the RC stage in *Phase_a*. *Phase_c* and *Phase_a* in the same round can be executed in parallel, so the latency of *Phase_c* can be overlapped by *Phase_a* except for the final round. Therefore, pipelining brings almost 5× performance improvement to the MM engine. Besides pipelining, we also use multiple MM engines on FPGA to improve parallelism.

4.2 Optimal Pre-computation Window

The ME in Paillier encryption is under fixed bases depending on the public keys. An optimization using this insight is to store all ME of short powers (*e.g.* window) in the offline phase. Large integer ME are converted to multiple MM operations in the online phase [BGMW92]. Enlarging the precomputation window can reduce ME latency. However, the maximum window size is limited by the on-chip memory size, since a larger window has a larger enumeration space. The size of the pre-computed table S_{pre} for encryption depends on the window size w and the key length $|n|$. Eq.16 shows the theoretical estimation of S_{pre} based on w and n.

$$S_{pre} = |n|/2w \times (2^w - 1) \times |n| \times 2 = |n|^2(2^w - 1)/w \tag{16}$$

The precomputed table sizes for different window sizes in a 3072-bit Paillier cryptosystem are shown in Table 2. The number of MM operations in a precomputed ME also depends on the window size. When the window size is 4, the required memory size for the precomputed table is about 34 MB. If the window size increases to 8, the size increases to about 287 MB. However, the number of windows is only reduced from 384 to 192, and the cost-effectiveness of the increased storage and reduced multiplication is significantly lower. Considering the URAM size of the hardware implementation, SHAPER provides the interface **AHE.init** to reload the precomputed table with the default window size of 4.

Table 2: Pre-computed table sizes under different window sizes when $|n| = 3072$.

Window Size w (bit)	1	2	3	4	5	6	7	8
Window Num	1536	768	512	384	308	256	220	192
Table Size S_{pre} (MB)	9	14	21	34	56	95	164	287

5 Implementation and Evaluation

We implement the prototype of SHAPER on a Xilinx 16nm VU13P FPGA using the Xilinx Vivado toolchain.

5.1 FPGA Resource Utilization

Table 3 shows the resource usage of SHAPER. The maximum clock frequency is 285 MHz by default. As the most expensive module, 14 MM engines are used considering performance and LUT consumption. 32 integer engines are used to support vectorized SS operations, contributing little to the overall consumption. Only one CSPRNG is used because its throughput is over 2.6 Gbps, which is sufficient for existing hybrid schemes. The resource consumption, excluding the controllers, is about 40% LUT/FF and 75% DSP in terms of the total FPGA resource. 75% of URAM is used for the precomputed table, which is a good balance between performance and area.

Table 3: Resource utilization on the Xilinx FPGA.

Module	LUT	FF	BRAM	URAM	DSP	Num
MM Engine	49750	93421	0	0	364	14
Pre-Computed Table	38912	49152	1024	960	0	1
CSPRNG	3447	3234	29	0	0	1
FIFO	228	1924	45	0	0	1
Integer Engine	311	3234	0	0	12	32
ScratchPad	2595	3637	512	0	0	1
Misc.	1613	3880	290	0	0	1
Total*	42%	43%	71%	75%	43%	-

* Measured by percentage in terms of Xilinx VU13P FPGA.

5.2 MM Throughput Comparisons

Table 4: Comparison between the Hardware Performance of MM implementation.

Design	Length	DSP	Freq. (MHz)	Cycle	Latency (μs)	Throughput Per DSP	
						op/s	Mbit/s
[YHC20]	1024	9	500	4405	8.81	12612	12.31
[BJ20]	2048	16	122	2005	16.42	3806	7.43
[XYCL22]	1024	131	285	345	1.21	6306	6.15
SHAPER	3072	364	285	172	0.60	4552	13.34

MM engine is important in the hardware implementation of SHAPER, and its throughput significantly influences the performance of high-level functions and applications. We evaluate the throughput of the MM engine in Table 4, compared with state-of-art MM designs in [BJ20, YHC20, XYCL22]. Since it is unfair to discuss throughput without considering resources, these designs are evaluated based on the average throughput generated by each piece of DSP. In specific, the throughput is measured as the operations or output bits executed by the MM engine. Existing MM implementations usually focus on 1024 or 2048-bit MM operations, while the MM in SHAPER needs to fit in the 3072-bit Paillier cryptosystem. It makes SHAPER handle fewer MM instructions than [XYCL22, YHC20], as their benchmarks are tested with 1024-bit MM. However, considering the output length, our MM engine shows advantages compared with other proposals.

It is notable that our MM engine requires more DSP units than other designs. Because of the pipelining optimization, the DSP units cannot be reused across different stages. So our design assigns different DSPs for each stage, leading to more DSP utilization.

5.3 Function-Level Comparisons

We evaluate the latency of general functions, including modular operations, Paillier functions, and several MPC-level functions. The latencies of these micro-benchmarks are shown in Table 5. The inputs of the MM and ME benchmarks are as long as the key length, and the length of plaintexts in Paillier is set to 64 bits. We test the performance of SHAPER with different settings. To fairly compare the latency of our MM engine in SHAPER with the cryptography processor (CP) in [BJ20], we implement a single MM

Table 5: Comparison between the Performance of Function-level Micro-benchmarks.

Design	Key Length	FPGA	Freq.(MHz)	Function Latency *(μs)									
				MM	ME	Enc	Dec	CCAdd	PCAdd	PCMult	S2H	H2S	TriGen
[BJ20]	2048	Xilinx Artix-7	122	1.37	4208	15038	15060	4.54	-	-	-	-	-
SHAPER(1xMM)	2048	Xilinx Artix-7	125	0.94	2890	487	2895	3.76	1.88	365	-	-	-
SHAPER(14xMM)	2048	Xilinx UltraScale+	285	0.03	90.5	15.3	90.7	0.12	0.06	11.4	15.3	16.8	58.9
SHAPER(14xMM)	3072	Xilinx UltraScale+	285	0.04	200	33.7	201	0.17	0.09	16.9	33.8	28.8	113
Speedup•	-	-	-	46.4×	46.5×	985×	166×	38.5×	-	-	-	-	-
[DSZ15]	2048	CPU	-	-	-	18000	18000	8.0	8.0	769	-	-	1473
[FZT$^+$21]	2048	CPU	-	-	-	-	-	-	-	-	250	600	-
Speedup•	-	-	-	-	-	1179×	198×	68.2×	136×	67.3×	16.3×	35.7×	25.0×

* **CCAdd** - Ciphertext-ciphertext addition;
PCAdd - Plaintext-ciphertext addition;
PCMult - Plaintext(64bit)-ciphertext multiplication;
S2H / **H2S** - Transformation from SS / HE to HE / SS;
TriGen - Beaver triple generation.
• The speedup is compared between benchmarks with 2048-bit key.

engine on Xilinx Artix-7 FPGA (*i.e.* the hardware environment in [BJ20]). Our MM engine has a similar frequency with CP in [BJ20]. However, the results show that our MM engine has a big advantage in cycles with the 2048-bit key. The MM engine of SHAPER significantly reduces the cycles, resulting in an order-of-magnitude improvement in efficiency.

Table 5 shows the speedup of 14xMM SHAPER on Xilinx UltraScale+ with 2048-bit key compared with multi-core FPGA of [BJ20] (Line 5), and the published CPU benchmarks in [DSZ15, FZT$^+$21] (Line 8).

The results show that the MM and ME efficiency of SHAPER is improved 46 times compared to [BJ20]. The MM and ME in SHAPER have similar hardware performance improvements because SHAPER follows a conventional approach to constructing ME using MM. Thus, the ME improvement is mainly due to the MM improvements. Paillier encryption in SHAPER has a significant advantage according to the results. This is reasonable because our encryption benefits a lot from exclusive optimizations, including DJN, CRT, and pre-computation, which makes the speedup increase about 6 times higher than the decryption speedup. Decryption in SHAPER can only benefit from CRT, and the applied optimizations still contribute a lot to the speedup.

Hardware solutions for higher-level functions are not provided in [BJ20]. So we only compare them to software solutions. In this case, SHAPER performs 16.3-1179 times better than the well-optimized software solutions in [DSZ15]. The software speedup of encryption and decryption is higher than the hardware speedup. We also compare the performance of the conversion interfaces (S2H/H2S) with the software benchmarks in [FZT$^+$21]. In the software benchmarks, H2S takes longer than S2H, because it requires AHE decryption while S2H requires less expensive encryption. However, they have similar latencies in the SHAPER implementation, because the packing strategy starts to work, and the decryption overhead can be shared by multiple H2S calls. The triple-generation interface also benefits from packing.

The MM engines in SHAPER are implemented to support the Paillier cryptosystem with the 3072-bit key. And the latency of different functions is also shown in Table 5. Note that the latency of PCMult is lower than the latency of ME. PCMult includes an ME operation, but the exponent (*i.e.* the plaintext message) is much shorter than the exponents in the ME benchmarks (the key length).

5.4 End-to-End PPML Comparisons

In fact, we can hardly find a hardware competitor since most existing accelerators only consider small datasets, which is far less than real-world business-to-business applications. We test the latencies of SHAPER when computing the CAESAR scheme [CZW$^+$21],

Table 6: Comparison between the Latencies of End-to-end logistic regression training with different sample sizes. The time unit is in minutes.

Design	**Platform**	**Security** (bit)	**Sample Size**				
			200K	400K	600K	800K	1M
CAESAR [CZW+21](Paillier)	CPU	112	615	1239	1862	2477	3101
CAESAR [CZW+21](OU)	CPU	112	66	133	200	266	333
SHAPER	FPGA	112	11	17	23	28	33

Table 7: Comparison between the Performance of End-to-end logistic regression training.

Design	**Platform**	**Primitive**	**Security** (bit)	**Latency** (min)	**Speedup***
SHAPER	FPGA	Hybrid	128	46	**67.4**×
SHAPER	FPGA	Hybrid	112	33	**94.0**×
CAESAR [CZW+21](Paillier)	CPU	Hybrid	112	3101	1×
CAESAR [CZW+21](OU)	CPU	Hybrid	112	333	9.31×
CraterLake [SFK+22]	ASIC	HE	128	3100	1.00×
SecureML [MZ17]	CPU	SS	128	14729	0.211×

* Take CAESAR (Paillier) as the baseline.

depending on the RTL simulation on Vivado for FPGA IP system with PCIe 3.0 bandwidth.

We evaluate real-world PPML applications using practical network settings, *i.e.* 40 Mbps network bandwidth and 40ms latency. In our experiments, we choose logistic regression (LR) as the benchmark application. On the one hand, although some complex ML approaches have more application scenarios than LR, and recent work [RCK+21] has tried to explore HE-based neural networks (NN), LR still has a wider range of real-world industrial applications that benefit from its simpler structure and interoperability, such as disease detection in hospitals, and fraud detection in financial companies. On the other hand, existing SS-AHE hybrid PPML proposals [CZW+21,FZT+21] do not provide support for complex ML such as NN. Therefore, it is better to compare the performance of LR tasks for a fair comparison. Note that SHAPER has potential in other ML applications besides LR, and we will explore SHAPER applications in other scenarios in the future, such as XGBoost, NN, etc.

We compare the results with the results on the CPU. Table 6 shows the performance on sparse logistic regression with different samples with 0.02% sparsity. Note that communication latency is not the largest contributor to total latency, so most of the communication latency is overlapped by computation latency. It is demonstrated that SHAPER has a significant performance advantage compared with the software benchmarks in [CZW+21]. Note that SHAPER performs a higher speedup when executing larger sample sizes since the overhead overlapping is more effective when the size grows.

Besides, we compare with other solutions using pure HE or SS which are also presented for comparison in Table 7. The result of CraterLake [SFK+22] is estimated based on scaling their results over small sets, as their scheme has a computational overhead that is linear in the size of the dataset. CraterLake [SFK+22], which requires expensive bootstrapping, is designed for non-interactive outsourcing computing scenarios, where communication latency is ignored. Note that schemes using garbled circuits are out of our scope due to the huge communication traffic.

Data in Table 7 is tested on sparse logistic regression with 1M samples. One participant has 30k features and label, and the other has 70k features. Paillier-based CAESAR is adopted as the performance baseline, whose latencies are taken from [CZW+21]. SHAPER performs 94× faster than CAESAR executed on the CPU when both using 2048-bit Paillier (112-bit security). The acceleration is mainly due to the fast Paillier encryption and pipeline execution. Even when the key is lengthened for 128-bit security (3072-bit key), SHAPER still performs 7.2× better than OU-based CAESAR with 112-bit security. In addition, most solutions of hybrid schemes show significant efficiency gains compared with SS or HE-based solutions, confirming the performance advantage of hybrid PPML schemes. SecureML performs the worst among the solutions since SS-based solutions mask sparse features to dense data shares.

6 Conclusion

In this paper, we propose SHAPER to accelerate hybrid SS-AHE PPML protocols. The algorithm-protocol-hardware co-design methodology explores the full-stack techniques to minimize the end-to-end latency in various network settings. SHAPER further supports secure domain computing acceleration and the conversion between mainstream privacy-preserving primitives, making it ready for general and distinctive data characteristics. We provide a prototype of SHAPER on an off-the-shelf FPGA with several hardware optimizations. Our evaluation shows that SHAPER provides significant speedup over CPU clusters on a large-scale logistic regression training task.

Acknowledgements

Ziyuan Liang, Qi'ao Jin, Zhiyong Wang, and Fan Zhang were supported in part by National Natural Science Foundation of China (62227805, 62072398), by the Natural Science Foundation of Jiangsu Province (BK20220075), by the Fok Ying-Tung Education Foundation for Young Teachers in the Higher Education Institutions of China (No.20193218210004) and by Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

References

[ARC19] Mohammad Al-Rubaie and J Morris Chang. Privacy-preserving machine learning: Threats and solutions. *IEEE S& P*, 17(2):49–58, 2019.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 311–323. Springer, 1986.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—CRYPTO'91: Proceedings 11*, pages 420–432. Springer, 1992.

[BGMW92] Ernest F Brickell, Daniel M Gordon, Kevin S McCurley, and David B Wilson. Fast exponentiation with precomputation. In *Advances in Cryptology-Eurocrypt' 92*, pages 200–207. Springer, 1992.

[BJ20] Milad Bahadori and Kimmo Järvinen. A programmable soc-based accelerator for privacy-enhancing technologies and functional encryption. *IEEE VLSI*, 2020.

[CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[CGHGN01] Dario Catalano, Rosario Gennaro, Nick Howgrave-Graham, and Phong Q Nguyen. Paillier's cryptosystem revisited. In *ACM CCS*, pages 206–214, 2001.

[CZW$^+$21] 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 *ACM SIGKDD*, pages 2652–2662, 2021.

[DGK07] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430. Springer, 2007.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *International workshop on public key cryptography*, pages 119–136. Springer, 2001.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[FZT$^+$21] Wenjing Fang, Derun Zhao, Jin Tan, Chaochao Chen, Chaofan Yu, Li Wang, Lei Wang, Jun Zhou, and Benyu Zhang. Large-scale secure xgb for vertical federated learning. In *CIKM*, pages 443–452, 2021.

[GBDL$^+$16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*. ACM, 1987.

[HHIL$^+$17] 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. *arXiv preprint arXiv:1711.10677*, 2017.

[Kel20] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *ACM CCS*, pages 1575–1590, 2020.

[KVH$^+$21] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 2021.

[LC21] Yamin Li and Wanming Chu. Shift-sub modular multiplication algorithm and hardware implementation for RSA cryptography. In *HIS*. Springer, 2021.

[LX19] Yi Li and Wei Xu. Privpy: General and scalable privacy-preserving data mining. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1299–1307, 2019.

[MLS$^+$20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX*, pages 2505–2522, 2020.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MR18] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *ACM CCS*, pages 35–52, 2018.

[MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, pages 19–38. IEEE, 2017.

[OU98] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *Eurocrypt*, pages 308–318. Springer, 1998.

[P$^+$13] Pille Pullonen et al. Actively secure two-party computation: Efficient beaver triple generation. *Instructor*, 2013.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, pages 223–238. Springer, 1999.

[PS20] Arpita Patra and Ajith Suresh. Blaze: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042*, 2020.

[RCK$^+$21] Brandon Reagen, Woo-Seok Choi, Yeongil Ko, Vincent T Lee, Hsien-Hsin S Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 26–39. IEEE, 2021.

[SFK$^+$22] 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*. ACM, 2022.

[SGA20] Jinhyun So, Basak Guler, and Salman Avestimehr. A scalable approach for privacy-preserving collaborative machine learning. *Advances in Neural Information Processing Systems*, 33:8054–8066, 2020.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[XBJ21] Runhua Xu, Nathalie Baracaldo, and James Joshi. Privacy-preserving machine learning: Methods, challenges and directions. *arXiv preprint arXiv:2108.04417*, 2021.

[XHY$^+$20] Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, and Xiaoyang Zeng. Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture. *IEEE TCAS-I*, 2020.

[XYCL22] Hao Xiao, Sijia Yu, Biqian Cheng, and Guangzhu Liu. Fpga-based high-throughput montgomery modular multipliers for rsa cryptosystems. *IEICE Electronics Express*, 19(9):20220101–20220101, 2022.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

[YHC20] Zhaoxiong Yang, Shuihai Hu, and Kai Chen. Fpga-based hardware accelerator of homomorphic encryption for efficient federated learning. *arXiv preprint arXiv:2007.10560*, 2020.

[ZXWG22] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. Ppmlac: high performance chipset architecture for secure multi-party computation. In *ISCA*, pages 87–101, 2022.

A Radix-1 Shift-sub MM

The algorithm of original shift-sub MM published in [LC21] is shown in Alg. A.

Algorithm 3 Shift-sub Modular Multiplication.

```
INPUTS: $a = \sum_{i=0}^{n-1} a_i 2^i$, $b = \sum_{i=0}^{n-1} b_i 2^i$, $a, b < m < 2^n$, $m \mod 2 = 1$.
OUTPUTS: $c = ab \mod m$
ALGORITHM:
  $c = 0$
  for $i = 0$ to $n - 1$ do
      $c = c + b_i a$
      if $c \geq m$ then
          $c = c - m$
      end if
      $a \ll 1$
      if $a \geq m$ then
          $a = a - m$
      end if
  end forreturn c
```

B Fixed-base precomputation of Paillier encrytion

The ME operations in Paillier encryption are executed with a fixed base (h_s in the public key) after the key parameters are determined. Hence the AHE unit can precompute and store some specific ME results in advance, and the actual ME operation in Paillier encryption can be converted into multiple MM operations. The core idea is to divide the exponent into small pieces.

The implementation in [DSZ15] divides the exponents into bit series. It precomputes $\{h_s^{1<<i}\}$: $\{h_s^{0\cdots01}, h_s^{0\cdots10}, \cdots, h_s^{1\cdots00}\}$, and stores them in the table. Then for each bit "1" in the actual exponent, the AHE unit multiplies the corresponding precomputed intermediates. The number of multipliers equals the hamming weight of the exponent.

The above fixed-base precomputation strategy can be further extended to wider window sizes, and the original is a special case where the window size equals 1. For the precomputation strategy with window size w, there are total l/w windows, where each window needs $2^w - 1$ precomputed values (except for the all-zero case). The larger the windows are, the less the times of the final multiplications. However, the size of the precomputed table grows significantly. The implementations need to consider the balance between latency and storage.

Note that when cooperated with the CRT strategy, the precomputed table stores the intermediates with bases p and q, instead of h_s.

C Conversion Protocols

SHAPER adopts the conversion protocols (including S2H & H2S) in [FZT+21]. The protocols are shown below. A small modification is applied to the original protocols in

[FZT+21]. Since Paillier cryptosystem supports addition between plaintext and ciphertext (PCAdd), SHAPER replaces the "Enc + CCAdd" process in the original protocols with PCAdd.

Algorithm 4 H2S Protocol

INPUT: A value $[\![x]\!] = (x_1, x_2)$ shared by the client $\mathcal{C}$ and server $\mathcal{S}$. AHE public key pk.
PROTOCOL:

```
1. $\mathcal{C}$ encrypts $x_1$ with $pk$, sends $c_{x_1}$ to $\mathcal{S}$.
2. $\mathcal{S}$ calculates $AHE.PCAdd(x_2, c_{x_1})$, outputs $c_x$.
```

Algorithm 5 S2H Protocol

INPUT: Ciphertext c_x hold by $\mathcal{S}$. AHE private key sk hold by $\mathcal{C}$. AHE public key pk.
PROTOCOL:

```
1. $\mathcal{S}$ generates random share $x_2$.
2. $\mathcal{S}$ calculates $c_{x_1} = AHE.PCAdd(-x_2, c_x)$, sends $c_{x_1}$ to $\mathcal{C}$.
3. $\mathcal{C}$ decrypts $c_{x_1}$, and obtains $x_1$.
4. Output share $[\![x]\!] = (x_1, x_2)$.
```