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

***Abstract*—Secure multi-party computation and homomorphic encryption are two primary security primitives in the 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 the 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 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 129× speed-up over CPU clusters on large-scale logistic regression training tasks.**

***Index Terms*—Privacy-Preserving Machine Learning, Multi-Party Computation, Additive Homomorphic Encryption, Hardware Accelerator**

I. 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 [1]. Privacy-preserving machine learning (PPML) [25] allows participants to co-operate on training and inference procedures by applying privacy-preserving computing techniques, *e.g.* multi-party computation (MPC) [26], homomorphic encryption (HE) [9], and trusted execution environment (TEE) [6]. These security primitives prevent the raw data, model weights, and gradient values from revealing to any other participants. Since the algorithms and protocols of PPML heavily depend on the data characteristics, scale, ownership, and security model, debates on technical roadmap never stop.

MPC covers a series of privacy-preserving techniques that support secure computation protocols on mathematically masked data. Garble circuit (GC) is a secure two-party logical computing protocol, where evaluating an *AND* gate requires transferring 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, network overhead always hinders further development of MPC-based PPML with complex models in real-time applications.

HE-based schemes support several 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 computation overhead. Existing academic FHE accelerators are still costly and only feasible on small-scale training and inference scenes [23]. 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 need a trusted party to generate and manage the secret key [13].

Recent works show that hybrid SS-AHE solutions achieve 130× speedup with practical dataset and network bandwidth [5], [16]. The key insight is to keep 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 always holding samples as plaintext within their owner and only transferring small-size HE domain intermediate values. 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.

When the complex HE algorithm combines SS protocol in the end-to-end PPML solutions, new architecture considerations and methodologies are necessary. We observe that the computation and communication complicity, 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 gets little return. We optimize the hybrid approach following the intuition that, in an ideal case, a well-balanced and parallel communication-computation flow can fold latency by 50%. This observation can also make the architecture less sensitive to the network bandwidth, which typically dominates MPC performance. At the algorithm level, it is helpful to tune hyper-parameters, such as an overflow-free pack level, to relieve ciphertext explosion. Considering practical computing power settings are also vital for PPML, a ready-to-use architecture should be suitable for FPGA platforms while flexible to ASIC design as well.

This paper provides a general architecture that can efficiently execute the SS-AHE hybrid PPML protocols on the industrial-level large training dataset. The proposed design preserves privacy in either the SS domain or AHE domain, without any reliance on trusted hardware manufacturers. 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.
- A hardware implementation supporting linear and approximated non-linear tasks with the help of efficient conversion protocols between primary security primitives.
- Vectorized high-performance modular multiplication (MM) engines to improve the efficiency of encryption, decryption, and ciphertext domain evaluation.
- SHAPER shows universal performance improvement on micro-operations and reduces the end-to-end latency by 129× on large-scale logistic regression training tasks.

II. Background and 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 [14], [15], [27] divide ML models into fragments of circuits, and then engage multiple parties to cooperatively execute circuit computations, including arithmetic and binary circuits, without additional privacy leaking. Afterwards, the

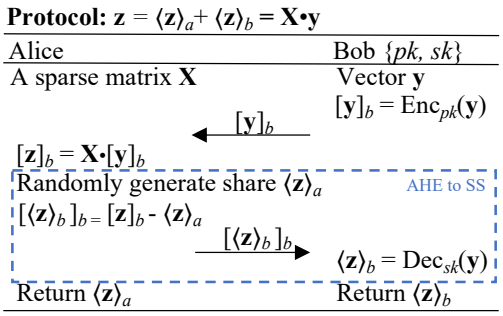

Fig. 1. SS-AHE-based secure matrix-vector multiplication.

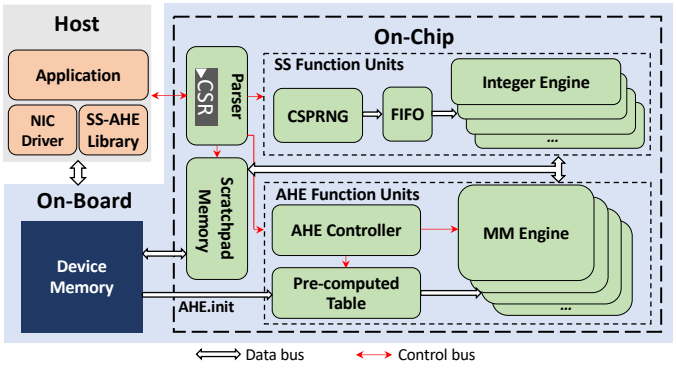

Fig. 2. SHAPER Architecture Overview – The orange and red rounded boxes represent software and hardware components, respectively.

results of these fragments of circuits are gathered from the parties to construct the full result of complex computation tasks. Historically speaking, MPC was proposed in [26], which solved the "Millionaire Problem" with GC. Not long after that, SS-based MPC [12] was proposed for better performance. Compared to GC, SS requires smaller computation and communication overheads, and outperforms GC in most scenarios. In the perspective of computation, addition and multiplication are two major operations in SS-based MPC. Its 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 by each MPC multiplication, the communication overhead dominates the overall performance for SS-based MPC and grows significantly for large-scale datasets.

HE-based PPML [11], [17] provides the capability to execute operations on encrypted data to protect privacy. Compared to public-key cryptography, apart from key generation, encryption, and decryption, HE additionally supports addition/multiplication over ciphertexts without private keys, and therefore no information about the corresponding plaintexts is revealed. Therefore, HE-based PPML requires less communication but more computation than MPC-based PPML for expensive HE encryption/decryption.

Fig. 1 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 vector $\mathbf{y}$, Alice can not recover the plaintext from the ciphertext $[\mathbf{y}]_b$. On the other hand, Alice shares $[\mathbf{z}]_b$ in line 5, which guarantees Bob can only learn a masked result $\langle \mathbf{z} \rangle_b$. Recent works on hybrid SS-AHE PPML [5] framework achieve 130× speedup compared with MPC-based schemes. AHE supports additions between ciphertexts and multiplications between ciphertexts and plaintexts. However, most existing AHE algorithms, such as Paillier [21], DGK [7], OU [20], depend on large integer modular operations, especially modular multiplications and exponentiations, which cost heavy computational overheads. Therefore, the overall performance of SS-AHE hybrid PPML is greatly dominated by the efficiency of the basic modular multiplications. Montgomery modular multiplication [19] is the most classic method while new modular algorithms have also been proposed recently [16].

III. 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.

A. Architecture overview

The overview of our proposed SHAPER architecture is shown in Fig. 2, which includes both software and hardware implementations. The host application controls the start and convergence condition of training tasks, it also consults hyper-parameters between participants. 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 I. SHAPER analyzes the dependency of control flow and packs the instructions as VLIW-style, ensuring the packed ones in a VLIW instruction can execute in parallel. Since the instructions execute in order 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 interruption from the NIC and SHAPER. Since runtime and driver layers are common components in the HW/SW co-design, we omit them in Table I for concise.

On the SHAPER hardware, the parser unpacks the instructions and dispatches them to the corresponding function units. **AHE.init** reloads data from the device memory in 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** execute a batch of integer arithmetic in a continuous address space. The memory hierarchy consists of on-board device memory and on-chip scratchpad, which is managed with **DM.ld/st** and **SPM.ld/st**.

B. Algorithm-Protocol Co-Optimization

Fig. 3 describes the methodology to analyze and explore the PPML solutions. We map a task to the coordinate point according to the computation and communication overhead. The network bandwidth is represented as a dotted guideline, points on which have the same communication and computation latency. The solutions above the guideline (*e.g. SecureML* [18]) are communication dominated. On the other hand, the communication-less FHE solutions (*e.g. CraterLake* [23]) cost most of the time on ciphertext evaluation. The location of SS-AHE-based solutions depends on computation power, especially the performance of cryptographic engines. In our work, the following optimizations are applied to explore an optimal solution.

1) Data Characteristic: In real-world scenes, the training dataset is sparse due to incomplete user information and one-hot encoding [5]. Since SS-AHE schemes keep the data sparsity, the number of instructions significantly reduces.

TABLE I
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.

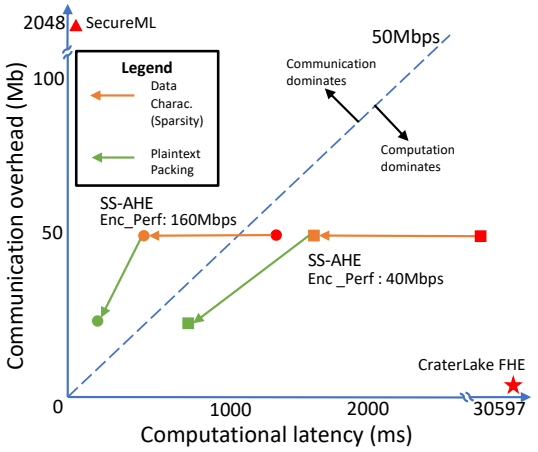

Fig. 3. Exploring optimization space on data characteristics and algorithms.

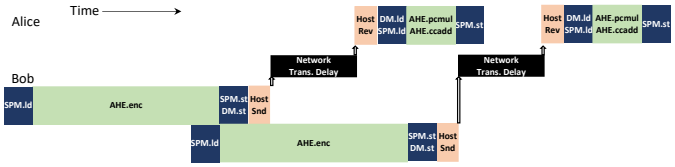

Fig. 4. The pipeline execution process of SHAPER, corresponding to line 1 to 3 in Fig. 1.

2) Plaintext Packing: Packing several plaintexts in a ciphertext relieves size expansion and enables SIMD-style computation as explained in Sec. III-E.

3) Latency Hiding: Since the SS-AHE schemes have balanced overhead, overlapping computation and communication brings more benefits. The example in Fig. 4 shows the most time-consuming operation hides the others in a pipelined flow. SHAPER consumes the data once source data is produced with multi-buffer transferring.

C. Efficient AHE Function Units

The AHE unit of SHAPER consists of a Paillier controller and multiple MM engines. The controller manipulates MM engines to compute the functions of the Paillier cryptosystem with message length $|n| = 3072$ parallelly. Each MM engine implements our proposed fast MM algorithm supporting a 5-stage pipeline. To accelerate the modular exponentiation (ME) in the encryption of Paillier, a piece 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 several multiplications and exponentiations based on DJN optimizations of Paillier [4]. Various optimizations suggested in [8] are taken into consideration, including Chinese-Remainder-Theorem (CRT) optimization and fixed-base pre-computation, which scales down both the base size and the exponent size of the ME. The call to single ME is divided into multiple multiplications in SHAPER, and the controller then schedules the datapath among different MM engines to collaboratively compute ME.

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

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

Require: Radix width k
Require: $a = \sum_{i=0}^{\tau-1} a_i 2^{ki}$, $b = \sum_{i=0}^{\tau-1} b_i 2^{ki}$, $\forall i \in [0, \tau-1], a_i, b_i < 2^k$. $a, b < m < 2^{k\tau}$, $m \mod 2 = 1$
Ensure: $c = ab \mod m$
1: $c = 0$
2: **for** $i = 0$ to $\tau - 1$ **do**
3: $\quad c = \mathbf{QR}(c + b_i a, m)$ ▷ *Phase_c, e.g.* Multiply-Accumulate Phase
4: $\quad a = \mathbf{QR}(a \ll k, m)$ ▷ *Phase_a, e.g.* Shift Phase
5: **end for**
6: **return** c.

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

Require: Length upper bound Δ, $\hat{m} = m \gg (l - \Delta - 2)$, $m' = \lfloor \frac{2^{2\Delta+2}}{\hat{m}+1} \rfloor$
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: $b = a - \gamma m$
4: **if** $b > 2m$ **then**
5: $\quad b = b - 2m$ ▷ Barrett Correction
6: **else if** $b \geq m$ **then**
7: $\quad b = b - m$ ▷ MSB Approximation Correction
8: **end if**
9: **return** b

1) The MM Algorithm: Our proposed MM algorithm is inspired by the shift-sub algorithm in [16], which has the advantage of dealing with large integers. The algorithm requires several serial full adders, one for each bit of b, which leads to long datapaths. 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 [16] is the special case of Alg. 1 with $k = 1$, and the modular reduction of c and a can be simplified to conditional subtraction. However, the strategy does not work as k grows. Thus a more efficient modular reduction is required to speed up the modular reductions.

Since the bit lengths of *Phase_c* and *Phase_a* have an upper bound, we propose a quick modular reduction algorithm $\mathbf{QR}$ in Alg. 2. The algorithm limits the length of the dividend to no more than $(l + \Delta)$, where l is the length of the modulus, and Δ is set to $k + 1$.

Two strategies are adopted to simplify the reduction. The first is the most-significant-bits (MSBs) approximation. The quotient in a

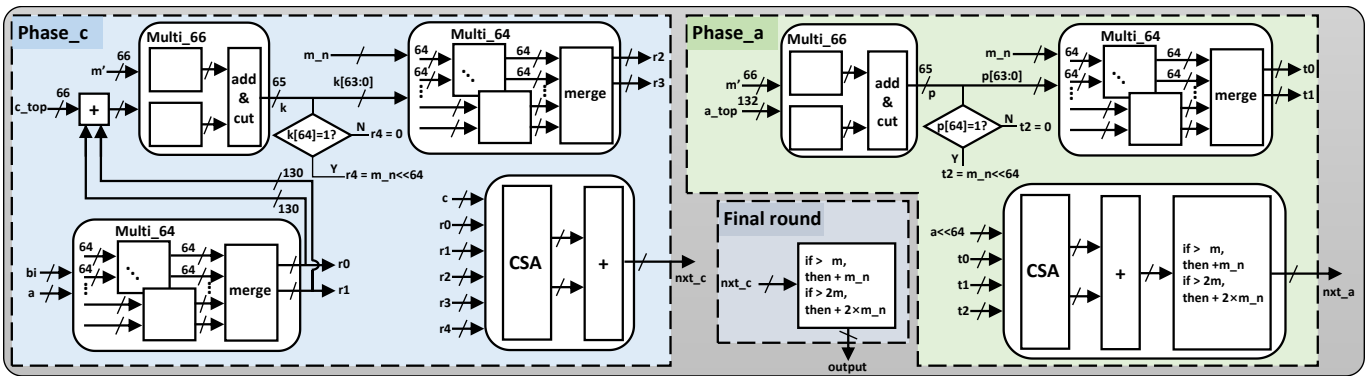

Fig. 5. The architecture of single MM engine based on the MM Algorithm.

division is mainly determined by the most significant bits of the dividend and modulus, while lower bits contribute barely to the quotient. Thus, the algorithm approximates a quotient γ with the MSBs of a and m, and then computes an approximated remainder with $a - \gamma m$, which is afterwards modified to the result with a conditional subtraction. The error between the approximated and precise quotients is proven to be within 1 when we use the most significant $\Delta + 2$ bits of m for approximation, as Eq.(1,2). (Note that $\forall x, y \in \mathbb{R}$, if $|x - y| \leq 1$, then $||\lfloor x \rfloor - \lfloor y \rfloor|| \leq 1$.) The second optimization is Barrett approximation to convert the division of a' by $\hat{m}+1$ to the product of a' and m'. This also involves a deviation of no more than 1 as the proof given in Eq.(3,4). Hence at most two conditional subtractions are required in the algorithm.

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

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

$$\begin{aligned} \frac{a'}{\hat{m}+1} - \frac{a'm'}{2^{2\Delta+2}} &= \frac{a'[2^{2\Delta+2} - m'(\hat{m}+1)]}{2^{2\Delta+2}(\hat{m}+1)} \\ &= \frac{a'[2^{2\Delta+2} \mod (\hat{m}+1)]}{2^{2\Delta+2}(\hat{m}+1)} \geq 0 \end{aligned} \tag{3}$$

$$\begin{aligned} \frac{a'}{\hat{m}+1} - \frac{a'm'}{2^{2\Delta+2}} &= \frac{a'[2^{2\Delta+2} - m'(\hat{m}+1)]}{2^{2\Delta+2}(\hat{m}+1)} \\ &< \frac{a'(\hat{m}+1)}{(\hat{m}+1)2^{2\Delta+2}} = \frac{a'}{2^{2\Delta+2}} < 1 \end{aligned} \tag{4}$$

2) The MM Engine: The MM engine (shown in Fig. 5) is the fundamental processing element to support our MM algorithm. It can be divided into three modules: two modules for Phase_c and Phase_a respectively, and a conditional subtraction module only for the final round. Phase_c contains two block multiplication modules, a 66×132 multiplier for Barrett division, a carry-save adder and an addition module. Phase_a contains an additional conditional subtraction module but only requires one block multiplication compared to Phase_c. A conditional subtraction module contains two addition modules performing $-m$ and $-2m$ in parallel. 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 cycle number. Two phases do not need data exchange except for Phase_c fetching a at the beginning of each round. Besides, to perform an integrated MM, a controller is required to schedule the input / output of the MM engine in each round.

One bottleneck in the design of an MM engine is the large integer addition, because it introduces a large ripple-carry adder, which leads to a long datapath. Whereas, in our design, we optimize serial adders with a prediction strategy, together with dividing two addends into several 256-bit pairs. Each such pair (x, y) uses two 256-bit ripple-carry adders to calculate two potential summations, $x+y$ and $x+y+1$. With the carry bit propagated from the lower pair, a multiplexer selects one of the summations, and propagates the corresponding carry bit to the higher pair.

D. Secret Sharing Function Units

Although compared with communication, computation is not the crucial overhead in SS-based schemes , we design a dedicated SS unit in SHAPER out of the following latency-related concern: in hybrid PPML schemes such as [5], [10], the computation of AHE and SS is interleaved, which means the data has to be transmitted frequently between the host and hardware if the hardware is not capable of locally computing SS functions. Hence each transmission requires reading/writing from the device memory, and it brings in non-negligible redundant latency.

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

For higher throughput of the random number generation, we optimized the CSPRNG in the SS unit with several improvements. Actually, existing CSPRNG constructions in PPML usually adopt ECB-mode AES encryption, which benefits a lot from AES-NI hardware extensions. However, recent works [24] 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, which leads to higher throughput during each iteration. Apart from the SHA-3 Keccak module, we also deploy a first-in-first-out (FIFO) buffer to temporarily store the generated random numbers, into which the SHA-3 Keccak module dynamically generates and pushes random numbers when it is not full.

E. Conversions between Primitives with Packing

SHAPER supports an efficient SIMD-style conversion between SS and AHE. We adopt and improve the conversion protocols in [16] On one hand, Paillier supports the addition between a ciphertext and a plaintext, and the encryption required in the H2S protocol can be

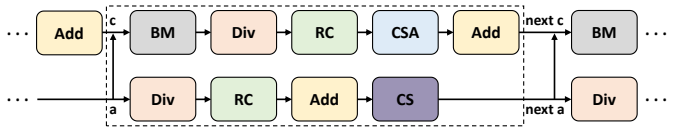

Fig. 6. Pipelined modular multiplication.

removed. On the other hand, the conversions can be optimized with the packing strategy proposed in [22]. The plaintext space (3072 bits) is too large for the values in the models (within 64 bits), and decryption in Paillier costs much more than encryption. Thus the space can be divided into small-scale buckets, with a smaller value in each bucket. Computation results in each bucket will not disturb others if the bucket size is large enough to ensure that there is no overflow in the following computation. So different ciphertexts can be packed into one, and there will be fewer ciphertexts needing to be sent and decrypted, which reduces both communication and computation.

The optimal bucket size depends on the computation under encrypted values in different protocols. In hybrid schemes, the optimal bucket size roughly equals $(m+1)l + log(a+1) + \sigma$ to ensure that there is no overflow in each bucket [22], where l is share length, usually equaling 64. m and a represent the numbers of multiplication with plaintext shares and addition with other shares. σ denotes the statistic security parameter of the scheme, equaling 40 in default. For example, AHE handles matrix multiplication in [5], where one multiplication and multiple additions are processed with each ciphertext. And the bucket size can be set to 180 as default, where each ciphertext contains around 17 buckets. 180-bit bucket size keeps valid unless the number of additions in matrix multiplication exceeds 2^{12}. Then the decryption overhead can be reduced at the expense of several ciphertext additions and plaintext multiplications for packing.

IV. Implementation and Evaluation

We implement SHAPER on a Xilinx 16nm VU13P FPGA with Xilinx Vivado toolchain. Several optimizations are applied to our implementation to improve performance and FPGA resource efficiency.

A. Parallel Modular Operations

1) Parallel Modular Operations: We observe that there are a large amount of matrix computations in real-world PPML scenarios [5], [10], consisting of multiple AHE operations without data dependency. Meanwhile, a single Paillier encryption can also benefit from parallelization, as fixed-base pre-computation is involved to improve efficiency. Thus, 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. 6. Each stage takes 5 execution cycles. Block Multiplication (BM) stage calculates $b_i a$ in Alg. 1. Division (Div) stage computes the Barrett division in Alg. 2 to obtain γ. Reduction Computation (RC) stage multiplies γ with $-m$. CSA stage involves the carry-save adders to merge $c + b_i a - km$ into single addition, and Add stage includes an optimized ripple-carry adder. CS stage handles the conditional subtraction of a. Pipelining brings 5 times performance improvement to the MM engine. Besides pipeline, we also deploy multiple MM engines on FPGA to improve parallelism.

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 stage. Large integer ME is converted to multiple MM Operations in the online stage [3]. Enlarging the pre-computation window can reduce the ME latency. Nevertheless, the maximum window size is restricted by on-chip memory size, as a larger window has a larger enumeration space. If the window size is 4, the required memory size for the pre-computed table is around 34 MB. When the window size grows to 8, the size grows to around 287 MB. In this case, SHAPER provides the interface **AHE.init** to reload the pre-computed table with the default window size of 4.

B. FPGA Resource Utilization

Table II shows the resource usage of SHAPER. The maximum clock frequency is 300 MHz under the default setting. As the most expensive module, 10 MM engines are deployed considering performance and LUT consumption. 32 Integer engines are deployed to support vectorized SS operations, contributing little to the overall consumption. Only one CSPRNG is deployed as its throughput is over 2.6 Gbps, which is enough for existing hybrid schemes. 75% of URAM is occupied for the pre-computed table, which is balanced between performance and area.

TABLE II
Resource utilization on the Xilinx FPGA.

Module	LUT	FF	BRAM	URAM	DSP	Num
MM Engine	99446	136795	0	0	624	10
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*	61%	45%	71%	75%	54%	-

* Measured by percentage in terms of Xilinx VU13P FPGA.

C. Function-Level Comparisons

We evaluate the latency of general functions, including modular operations, Paillier functions, and several MPC-level functions. The latency of these micro-benchmarks is shown in Table III. We test the performance of SHAPER in different settings. The results indicate that SHAPER has a great advantage in cycles compared with [2]. The MM engine of SHAPER greatly reduces the cycles, which leads to an order-of-magnitude efficiency improvement.

Table III shows the HW speedup of 10xMM SHAPER with 2048-bit key compared with 12xCP (cryptography processor) [2], and SW speedup compared with the solutions published in [5], [8], [16]. The results show that MM and ME latency of SHAPER is reduced by 25 times compared with [2]. Paillier encryption in SHAPER has a significant advantage according to the results. It is reasonable as our encryption benefits a lot from exclusive optimizations including DJN and pre-computation, which makes the speedup increase to more than 10 times higher than the decryption speedup. The applied CRT optimization contributes $2\times$ speedup on decryption, MM and ME. Hardware solutions for higher-level functions are not provided in [2]. So we only compare them with software solutions. In this case, SHAPER performs 9.03-657 times better than the well-optimized software solutions [5].

D. End-to-End PPML Comparisons

Actually, 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 build a simulator to get the performance of SHAPER when computing the CAESAR

TABLE III
COMPARISON BETWEEN THE HARDWARE PERFORMANCE AND SPEED OF FUNCTION-LEVEL MICRO-BENCHMARKS.

Design	Key Length	FPGA	Freq.(MHz)	Function Latency (μs)									
				MM	ME	Enc	Dec	CCAdd*	PCAdd	PCMult	S2H	H2S	TriGen
MK20 [2](1xCP)	2048	Xilinx Artix-7	122	16.42	50490	180450	180720	54.42	-	-	-	-	-
SHAPER **(1xMM)**	2048	Xilinx Artix-7	130	1.24	3795	632	7590	6.44	6.69	618	-	-	-
SHAPER **(10xMM)**	2048	Xilinx UltraScale+	300	0.054	164	27.4	329	0.28	0.29	26.8	27.7	30.8	139
SHAPER **(10xMM)**	3072	Xilinx UltraScale+	300	0.080	370	61.6	739	0.42	0.43	40.1	62.0	44.8	249
HW Speedup	-	-	-	25.7×	25.7×	549×	45.8×	16.2×	-	-	-	-	-
CPU Speedup	-	-	-	-	-	657×	54.7×	28.7×	27.6×	28.7×	9.03×	19.5×	10.6×

* **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.

TABLE IV
COMPARISON BETWEEN THE SIMULATED PERFORMANCE OF END-TO-END LOGISTIC REGRESSION TRAINING.

Design	Platform	Primitive	Security (bit)	Latency (min)	Speedup*
SHAPER	FPGA	Hybrid	128	42	**73.8**×
SHAPER	FPGA	Hybrid	112	24	**129**×
CAESAR [5](Paillier)	CPU	Hybrid	112	3101	1×
CAESAR [5](OU)	CPU	Hybrid	112	333	9.31×
CraterLake [23]	ASIC	HE	128	3100	1.00×
SecureML [18]	CPU	SS	128	14729	0.211×

* Take CAESAR (Paillier) as the baseline.

scheme [5], and compare the simulated results with the results on the CPU. Other solutions using pure HE or SS are also introduced for comparison. The result of [23] is estimated based on scaling their results over small sets, as their scheme has computation overhead linear to the size of the dataset. [23], which requires expensive bootstrapping, is designed for non-interactive outsourcing computing scenarios, where communication latency is ignored.

We evaluate real-world PPML applications with practical network settings, *i.e.* 40 Mbps network bandwidth and 40ms latency. Table IV shows the performance on sparse logistic regression with 1M samples, 100K features with 0.02% sparsity. SHAPER performs $129\times$ faster than CAESAR executed on the CPU when both using 2048-bit Paillier (112-bit security). The acceleration is mainly contributed by fast Paillier encryption and pipeline execution. Even when the key is lengthened for 128-bit security (3072-bit key), SHAPER still performs $7.9\times$ better than OU-based CAESAR with 112-bit security. Besides, 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.

V. 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 brings significant speed-up over CPU clusters on a large-scale logistic regression training task.

REFERENCES

[1] M. Al-Rubaie *et al.*, "Privacy-preserving machine learning: Threats and solutions," *IEEE S& P*, vol. 17, no. 2, pp. 49–58, 2019.
[2] M. Bahadori *et al.*, "A programmable soc-based accelerator for privacy-enhancing technologies and functional encryption," *IEEE VLSI*, 2020.
[3] E. F. Brickell *et al.*, "Fast exponentiation with precomputation," in *Advances in Cryptology-Eurocrypt' 92*. Springer, 1992, pp. 200–207.
[4] D. Catalano *et al.*, "Paillier's cryptosystem revisited," in *ACM CCS*, 2001, pp. 206–214.
[5] C. Chen *et al.*, "When homomorphic encryption marries secret sharing: Secure large-scale sparse logistic regression and applications in risk control," in *ACM SIGKDD*, 2021, pp. 2652–2662.
[6] V. Costan *et al.*, "Intel sgx explained," *Cryptology ePrint Archive*, 2016.
[7] I. Damgård *et al.*, "Efficient and secure comparison for on-line auctions," in *ACISP*. Springer, 2007, pp. 416–430.
[8] D. Demmler *et al.*, "Aby-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.
[9] J. Fan *et al.*, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.
[10] W. Fang *et al.*, "Large-scale secure xgb for vertical federated learning," in *CIKM*, 2021, pp. 443–452.
[11] R. Gilad-Bachrach *et al.*, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *ICML*, 2016.
[12] O. Goldreich *et al.*, "How to play any mental game or A completeness theorem for protocols with honest majority," in *STOC*. ACM, 1987.
[13] S. Hardy *et al.*, "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption," *arXiv preprint arXiv:1711.10677*, 2017.
[14] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *ACM CCS*, 2020, pp. 1575–1590.
[15] B. Knott *et al.*, "Crypten: Secure multi-party computation meets machine learning," *Advances in Neural Information Processing Systems*, 2021.
[16] Y. Li *et al.*, "Shift-sub modular multiplication algorithm and hardware implementation for RSA cryptography," in *HIS*. Springer, 2021.
[17] P. Mishra *et al.*, "Delphi: A cryptographic inference service for neural networks," in *USENIX*, 2020, pp. 2505–2522.
[18] P. Mohassel *et al.*, "Secureml: A system for scalable privacy-preserving machine learning," in *IEEE S&P*. IEEE, 2017, pp. 19–38.
[19] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.
[20] T. Okamoto *et al.*, "A new public-key cryptosystem as secure as factoring," in *Eurocrypt*. Springer, 1998, pp. 308–318.
[21] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Eurocrypt*. Springer, 1999, pp. 223–238.
[22] P. Pullonen *et al.*, "Actively secure two-party computation: Efficient beaver triple generation," *Instructor*, 2013.
[23] N. Samardzic *et al.*, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *ISCA'22*. ACM, 2022.
[24] G. Xin *et al.*, "Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture," *IEEE TCAS-I*, 2020.
[25] R. Xu *et al.*, "Privacy-preserving machine learning: Methods, challenges and directions," *arXiv preprint arXiv:2108.04417*, 2021.
[26] A. C. Yao, "Protocols for secure computations (extended abstract)," in *FOCS*. IEEE Computer Society, 1982, pp. 160–164.
[27] X. Zhou *et al.*, "Ppmlac: high performance chipset architecture for secure multi-party computation," in *ISCA*, 2022, pp. 87–101.