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.

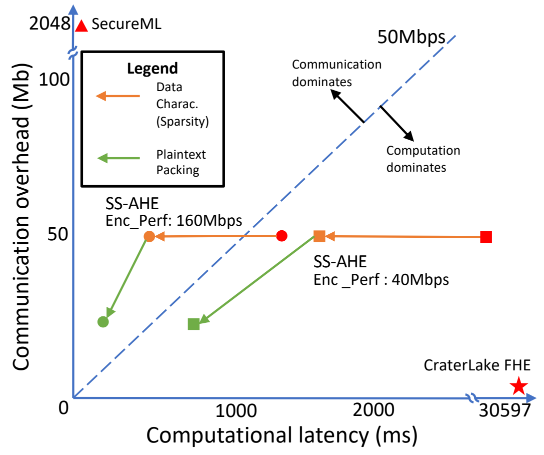

Fig. 3. Exploring optimization space on data characteristics and algorithms.

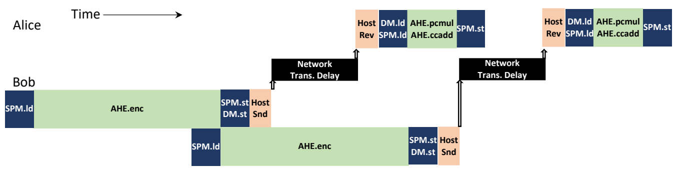

Fig. 4. The pipeline execution process of SHAPER, corresponding to line 1 to 3 in Fig. 1.

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:     $c = \mathbf{QR}(c + b_i a, m)$          ▷ Phase_c, e.g. Multiply-Accumulate Phase
4:     $a = \mathbf{QR}(a \ll k, m)$          ▷ 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 $\Delta$, $\hat{m} = m \gg (l - \Delta - 2)$, $m' = \lfloor \frac{2^{2\Delta+2}}{\hat{m}+1} \rfloor$
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: $b = a - \gamma m$
4: if $b > 2m$ then
5:     $b = b - 2m$          ▷ Barrett Correction
6: else if $b \geq m$ then
7:     $b = b - m$          ▷ 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 **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

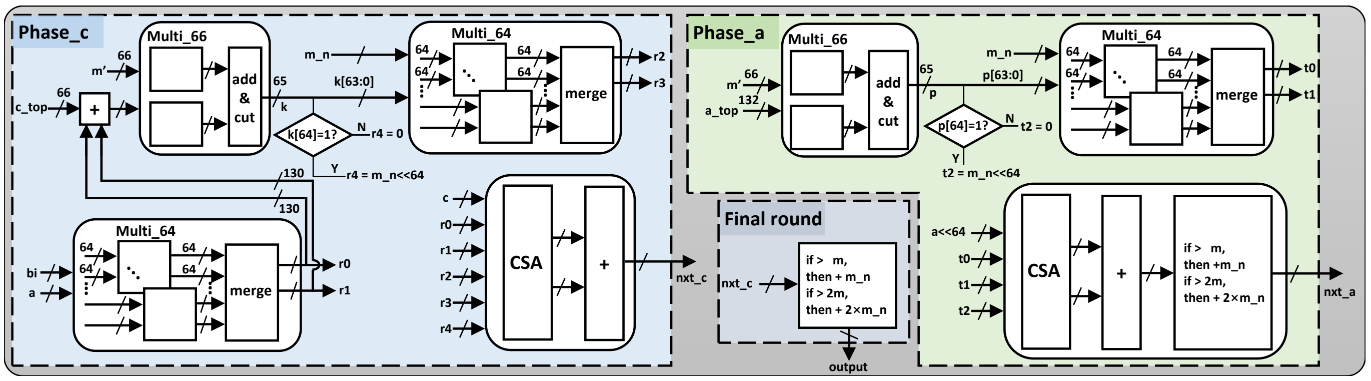

Fig. 5. The architecture of single MM engine based on the MM Algorithm.

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