Semi-Honest 2-Party Faithful Truncation from Two-Bit Extraction

Huan Zou[1,2], Yuting Xiao[1], and Rui Zhang[1,2](✉)

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
{zouhuan, xiaoyuting, r-zhang}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract. As a fundamental operation in fixed-point arithmetic, truncation can bring the product of two fixed-point integers back to the fixed-point representation. In large-scale applications like privacy-preserving machine learning, it is essential to have faithful truncation that accurately eliminates both big and small errors. In this work, we improve and extend the results of the oblivious transfer based faithful truncation protocols initialized by Cryptflow2 (Rathee et al., CCS 2020). Specifically, we propose a new notion of two-bit extraction that is tailored for faithful truncation and demonstrate how it can be used to construct an efficient faithful truncation protocol. Benefiting from our efficient construction for two-bit extraction, our faithful truncation protocol reduces the communication complexity of Cryptflow2 from growing linearly with the fixed-point precision to logarithmic complexity.
This efficiency improvement is due to the fact that we reuse the intermediate results of eliminating the big error to further eliminate the small error. Our reuse strategy is effective, as it shows that while eliminating the big error, it is possible to further eliminate the small error at a minimal cost, e.g., as low as communicating only an additional 160 bits in one round.

Keywords: Secure two-party computation · Secure truncation · Bit extraction

1 Introduction

Secure 2-Party Computation (2PC) allows two parties to compute an arbitrary function of their inputs without revealing anything about them, except for what can be deduced from the function output. When applying 2PC protocols to enhance the privacy of applications analyzing numerical data, such as privacy-preserving machine learning (PPML), an immediate challenge is to overcome the data representation mismatch between the application and the 2PC cryptographic protocols. Typically, the application represents data as float type, while the cryptographic protocols encode the data as big integers.

To address this challenge, two approaches have been adopted: (1) *Floating-point arithmetic* [20], first encodes a floating-point number as a tuple of four integers, then emulates the floating-point addition and multiplication with the four integers; (2) *Fixed-point arithmetic*, first discretizes a floating-point number to a fixed precision 2^{-s}, scales the discretized number by 2^s to be an l-bit signed integer, and encodes this signed integer into the ring $\mathbb{Z}_{2^l}$. For better efficiency, most prior works [8,19,16,21,17] based on the fixed-point arithmetic. In this work, we also focus on fixed-point arithmetic.

Truncation is required after fixed-point integer multiplication. When multiplying two fixed-point integers a and b which have been both scaled by 2^s, their product $c = a \cdot b$ will be scaled by 2^{2s}. To bring c back to the fixed-point representation of scaling by 2^s, we need to execute "truncation" (i.e., a divide-by-2^s protocol). Informally, the truncation functionality takes as input the additive shares c_0, c_1 of c (i.e., $c_0 + c_1 = c$), and returns the additive shares of c', where c' is supposed to be $\frac{c}{2^s}$.

Faithful truncation and truncation errors. Faithful truncation means that the output shares of c' make the equation $c' = \frac{c}{2^s}$ hold with probability 1. Otherwise, it is "probabilistic". Probabilistic truncation introduces the *small error* and *big error*. When the small error occurs, $\frac{c}{2^s} - c' = 2^{-s}$. The occurrence probability of this error is roughly $\frac{1}{2}$. When the big error occurs, there is a sign bit flipping issue. That is, if c is a positive number, then c' is negative, and vice versa. The big error's occurrence probability depends on the magnitude of c. The larger magnitude c has, the more likely that the big error will occur [17].

Faithful truncation is necessary for large-scale applications. Intuitively, one approach to reduce the occurrence probability of the big error as well as minimize the effect of the small error is to increase the computation modulus (i.e., increase the big length l of the encoded data) [16,19]. However, this also leads to increased computation and communication costs in the 2PC protocols [7].

In particular, it is unclear whether increasing computation modulus works for large-scale applications like PPML, where billions of truncation operations are performed. That implies the small error will be accumulated billions of times and the big error is almost certain to occur. Indeed, multiple recent works have shown that additional steps are required to eliminate the big error for training large models [7,12,21], and correct 2PC implementation of the cleartext fixed-point execution is necessary [21]. Therefore, there is a need for efficient faithful truncation protocols that can eliminate both big and small errors.

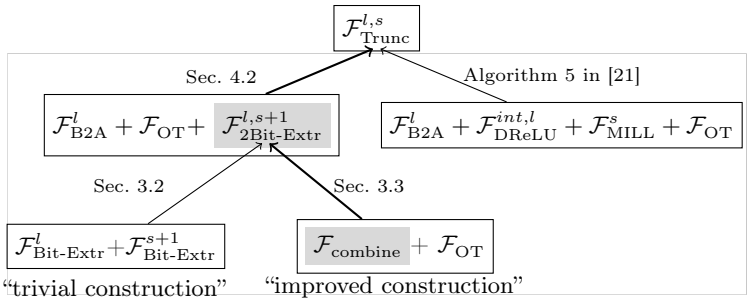
Prior works on eliminating the truncation errors. Cryptflow2 [21], Cheetah [12], [BCG+21] [4] and LLAMA [10] have developed their truncation protocols based on the unsigned integer comparison problem (also known as the millionaire problem). To achieve faithful truncation, these protocols invoke two comparison instances—one for eliminating the big error and the other for eliminating the small error.

[BCG+21] [4] and LLAMA [10] construct their comparison protocols from function secret sharing [5] and enjoy attractive online communication complexity (i.e., each party sends 1 element) as well as round complexity (i.e., 1 round). However, they require a prohibitively expensive offline phase, which can be made efficient with a trusted dealer.

Cryptflow2 [21] and Cheetah [12] construct logarithm rounds comparison protocols from the oblivious transfer (OT) and can be implemented efficiently without a trusted dealer by utilizing fast OT extensions [13,14,23]. Cheetah makes use of the advent of silent OT extension built on vector oblivious linear evaluation [23], while Cryptflow2 instantiates OT using the classical IKNP-style OT extension [13,14]. Furthermore, depending on their applications, Cheetah only eliminates the big error while Cryptflow2 achieves faithful truncation eliminating both the big error and small error.

1.1 Our contributions

We continue the study of efficient faithful truncation construction. We improve and extend previous results from Cryptflow2 [21] in several directions.



† An arrow $A \to B$ means that there exists a protocol realizes the functionality B by using the functionality A as subroutine.
‡ Thick arrows indicate our constructions. Thin arrows indicate the known or trivial constructions. The texts in gray shadowed highlight our new functionality.

Fig. 1. Comparing our faithful truncation construction with Cryptflow2's [21]

New observation. Given an l-bit secret x, we found that acquiring the boolean shares of its l-th bit $x[l]$ and $(s+1)$-th bit $x[s+1]$, confers a significant degree of simplification upon the task of faithfully truncating x by s bits. With the boolean shares of $x[l]$, the parties can recognize the sign bit flipping issue and thus, address the resultant big error. The boolean shares of $x[s+1]$ enable the parties to determine whether a carry-out is generated by the least significant s bits being chopped off, which can mitigate the resultant small error.

New building functionality 2Bit-Extr **for faithful truncation.** Based on our new observation, we propose a new functionality $\mathcal{F}^{l,s}_{\text{2Bit-Extr}}$ for two-bit extraction. This functionality is a special form of bit decomposition, and extends the functionality $\mathcal{F}^{l}_{\text{Bit-Extr}}$ for bit extraction [16]. Instead of a single bit, $\mathcal{F}^{l,s}_{\text{2Bit-Extr}}$ simultaneously decomposes two bits—the MSB (i.e., the most significant bit) and the s-th bit of $x \in \mathbb{Z}_{2^l}$ into their respective binary form.

We also propose a protocol that securely realizes faithful truncation $\mathcal{F}^{l,s}_{\text{Trunc}}$ in the ($\mathcal{F}^{l}_{\text{B2A}}$, $\mathcal{F}_{\text{OT}}$, $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$)-hybrid model.

New construction to realize 2Bit-Extr**.** We first propose a trivial protocol that realizes $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$ by making two calls to $\mathcal{F}^{l}_{\text{Bit-Extr}}$ and $\mathcal{F}^{s+1}_{\text{Bit-Extr}}$ for extracting the MSB and $(s+1)$-th bit of x, respectively. However, we observe that the inputs of the two calls are dependent and some intermediate results are shared, such that it is not necessary to compute them twice. We then propose a more efficient protocol by packing up the two calls. In particular, the protocol first extracts the MSB, then further extracts the $(s+1)$-th bit by reusing those intermediate results that have been computed in the MSB extraction.

We summarize the faithful truncation constructions mentioned above in Fig. 1, and compare their concrete efficiency in Table 1. Compared with the state-of-the-art Cryptflow2 [21] whose communication complexity grows linearly with the fixed-point precision s, the communication complexity of our protocol is logarithm in s. When $l = 32, s = 16$, our protocol only communicates 72% of the bits of Cryptflow2. Given that truncation is as fundamental as multiplication in fixed-point arithmetic, our improvement in communication efficiency can have a significant impact on large-scale applications like PPML, where billions of truncation operations may be involved.

We note that we achieve efficiency improvement by adopting an intermediate result reuse strategy for efficient two-bit extraction construction, which eliminates redundancies present in existing constructions. In Table 1, we also compare our protocol with Cheetah [12], which

Table 1. Comparing the concrete efficiency of OT-based truncation protocols

Application	Protocol	S-Err	B-Err	Comm. (bits)	Round
Truncate an l-bit string by s bits	Cheetah‡ [12]	✗	✓	$\approx \lambda l$	$\lceil \log l \rceil$
	Cryptflow2 [21]	✓	✓	$\approx \lambda l + \lambda s + \lambda + l$	$\lceil \log l \rceil + 1$
	Trivial†	✓	✓	$\approx 3\lambda l + 3\lambda s$	$\lceil \log l \rceil + 3$
	This work	✓	✓	$\approx \lambda l + \lambda + l + k$, $k \in [0, 2\lambda \lceil \log \lceil \frac{s-1}{4} \rceil \rceil)$	$\lceil \log l \rceil + 1$
Truncation example $l = 32, s = 16$	Cheetah‡ [12]	✗	✓	4224	5
	Cryptflow2 [21]	✓	✓	6093	6
	Trivial†	✓	✓	19164	8
	This work	✓	✓	4384	6

⁎ Symbol ✓/ ✗ means that the protocol eliminates / admits the small error (S-Err) (resp., big error (B-Err)). Results regarding this work assume the two optimizations in Section 3.3 are used. The protocol parameter m is set to be 4 for this work and for [21]. We use the security parameter $\lambda = 128$ to calculate the communication bits.
‡ For fair comparison, we assume [12] used the same IKNP-style OT extension [14] to realize $\mathcal{F}_{\text{OT}}$ as Cryptflow2.
† The trivial protocol refers to constructing a faithful truncation protocol from $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$ (Sec. 4.2), while $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$ is realized by trivially invoking two instances of bit extraction [16].

only eliminates the big error. Our comparison suggests that while focusing on eliminating the big error, a small additional cost (as small as communicating $\lambda + l$ bits in one round) can be paid to further remove the small error, where λ is the security parameter. For example, when the fixed-point precision s is set to be 16 (which is typically used by privately training large machine learning models [7]), our protocol further removes the small error by communicating only an additional 160 bits than Cheetah.

1.2 Organization

Section 2 introduces notations, definitions, and primitives used in this work. Section 3 describes the functionality and construction of our proposed notion of two-bit extraction. Section 4 elaborates on faithful truncation and truncation errors, and presents our faithful truncation protocol built from the two-bit extraction. In Section 5, we conduct experiments to empirically compare the practical performance of our faithful truncation protocol with that proposed by Cryptflow2 [21]. Finally, we conclude and discuss future work in Section 6.

2 Preliminaries

Notations. We use λ to denote the computational security parameter. We use $[x, y]$ for $x, y \in \mathbb{Z}$ to denote the set $\{x, x+1, x+2, \ldots, y\}$. We use "||" to denote bit concatenation. We consider two rings $\mathbb{Z}_2$ and $\mathbb{Z}_{2^l}$. For $x \in \mathbb{Z}_2$, we use $\overline{x}$ to denote the bitwise NOT of x. For $x \in \mathbb{Z}_{2^l}$, we use $x_l || \ldots || x_1$ to denote the binary form of x, $x_{i|i\in[1,l]} \in \{0,1\}$. We refer to the l-th bit of x as the most significant bit (MSB), the i-th bit x_i as $x[i]$, and the $j - i + 1$ bits $x_j || \ldots || x_i$ as $x[j:i]$. We use $x \gg s$ to denote arithmetic right shift x by s bits.

Fixed-point representation. Real numbers are encoded into $\mathbb{Z}_{2^l}$ using the fixed-point notation. A real number is first discretized to a limited precision 2^{-s} (denoted as x^{fxd}). Then x^{fxd} is scaled by 2^s to be an integer x^{int} with bit length l, i.e., $x^{\text{int}} = x^{\text{fxd}} \cdot 2^s$. Then this signed integer $x^{\text{int}} \in [-2^{l-1}, 2^{l-1} - 1]$ is further encoded into the ring $\mathbb{Z}_{2^l}$ using the two's complement encoding. This encoding interprets the binary form $x_l || \ldots || x_1$ of x^{int} as $x^{\text{int}} = \sum_{i=1}^{l-1} 2^{i-1} \cdot x_i$ if $x_l = 0$. Otherwise, $x^{\text{int}} = \sum_{i=1}^{l-1} 2^{i-1} \cdot x_i - 2^l$ when $x_l = 1$. We say a signed integer x^{int} and a ring element x correspond to each other if they share the same binary form $x_l || \ldots || x_1$. Since $x^{\text{int}} = x^{\text{fxd}} \cdot 2^s$, given $x \in \mathbb{Z}_{2^l}$, $x[l]$ corresponds to the sign bit

of x^{fxd}, $x[l-1:s+1]$ corresponds to the integer part of x^{fxd}, and $x[s:1]$ corresponds to the fraction part of x^{fxd}.

Secret sharing schemes. We use 2-out-of-2 additive secret sharing schemes over $\mathbb{Z}_{2^l}$ and $\mathbb{Z}_2$. A secret sharing scheme consists of two algorithms, Share and $\mathsf{Reconst}$. On input a value, the probabilistic algorithm Share outputs two shares of it. On input two shares, the deterministic algorithm $\mathsf{Reconst}$ reconstructs a value from them.

Consistent with prior work [8], we refer to shares over $\mathbb{Z}_{2^l}$ and $\mathbb{Z}_2$ as Arithmetic $\langle x\rangle^{\text{A}}$ and Boolean $\langle x\rangle^{\text{B}}$ shares. For $x \in \mathbb{Z}_{2^l}$, its two shares are denoted as $\langle x\rangle_0^{\text{A}}$ and $\langle x\rangle_1^{\text{A}}$ such that $\langle x\rangle_0^{\text{A}} + \langle x\rangle_1^{\text{A}} = x$ where operation $+$ denotes the addition over $\mathbb{Z}_{2^l}$. For $y \in \mathbb{Z}_2$, its two shares are denoted as $\langle y\rangle_0^{\text{B}}$ and $\langle y\rangle_1^{\text{B}}$ such that $\langle y\rangle_0^{\text{B}} \oplus \langle y\rangle_1^{\text{B}} = y$ where operation $\oplus$ denotes the addition over $\mathbb{Z}_2$ (i.e., XOR).

Additive secret sharing schemes are perfectly hiding. Given an arithmetic share $\langle x\rangle^{\text{A}}$ (resp., boolean share $\langle x\rangle^{\text{B}} \in \mathbb{Z}_2$), the value x is completely hidden.

2.1 System model and security

Consistent with our baseline work Cryptflow2 [21], we consider a static, semi-honest adversary $\mathcal{A}$. We use the standard security definition for two-party computation [11] in this work. Let $\mathcal{F} = (\mathcal{F}_0, \mathcal{F}_1)$ be a functionality. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ with inputs x_0 and x_1 run protocol Π to learn $\mathcal{F}$. We say that Π securely realizes $\mathcal{F}$ in the presence of $\mathcal{A}$ if there exists probabilistic polynomial-time algorithms $\mathcal{S}_0$ and $\mathcal{S}_1$ such that:

$$\{(\mathcal{S}_0(1^\lambda, x_0, f_0(x_0,x_1)), f(x_0,x_1))\}_{x_0,x_1,\lambda} \cong \{(\mathsf{View}_0^{\Pi}(x_0,x_1,\lambda), \mathsf{output}^{\Pi}(x_0,x_1,\lambda))\}_{x_0,x_1,\lambda};$$
$$\{(\mathcal{S}_1(1^\lambda, x_1, f_1(x_0,x_1)), f(x_0,x_1))\}_{x_0,x_1,\lambda} \cong \{(\mathsf{View}_1^{\Pi}(x_0,x_1,\lambda), \mathsf{output}^{\Pi}(x_0,x_1,\lambda))\}_{x_0,x_1,\lambda}.$$

In order to conceptually modularize the design of the protocols, the notion of "hybrid model" is introduced. A protocol Π is said to be realized in the $\mathcal{F}$-hybrid model if Π invokes the ideal functionality $\mathcal{F}$ as a subroutine. This allows the simulator $\mathcal{S}$ to simulate $\mathcal{F}$ in the ideal world as long as it "looks" indistinguishable from $\mathcal{F}$-hybrid world.

2.2 Basic Operations

Oblivious transfer. We use $\binom{k}{1}\text{-}\mathsf{OT}_l$ to denote the 1-out-of-k oblivious transfer (OT) functionality. The sender uses k messages $\mathsf{msg}_1, \ldots, \mathsf{msg}_k$ as input (each message is a l-bit string), and the receiver uses $i \in [1,k]$ as inputs. The receiver receives only msg_i as output and the sender receives no output. We use the OT extension protocols from [14] to improve the efficiency of our implementations. The protocols for $\binom{k}{1}\text{-}\mathsf{OT}_l$ [14] communicate $2\lambda + kl$ bits. The simpler $\binom{2}{1}\text{-}\mathsf{OT}_l$ communicates only $\lambda + 2l$ bits [2].

The AND functionality $\mathcal{F}_{\mathbf{AND}}$ takes as input boolean shares of x and y, and returns the boolean shares of $z = x \wedge y$. $\mathcal{F}_{\text{AND}}$ can be realized using the well-known Beaver bit triple [3] of the form $(\langle\delta_x\rangle^{\text{B}}, \langle\delta_y\rangle^{\text{B}}, \langle\delta_z\rangle^{\text{B}})$ such that $\delta_z = \delta_x \wedge \delta_y$. Cryptflow2 (appendix A.1 in [21]) generates two such bit triples using an instance of $\binom{16}{1}\text{-}\mathsf{OT}_2$. The communication complexity per bit triple is $\lambda + 16$ bits. Given one bit triple, the parties need to exchange additional 4 bits to compute a $\mathcal{F}_{\text{AND}}$ call. Hence, the communication complexity of invoking a $\mathcal{F}_{\text{AND}}$ instance is $\lambda + 20$ bits, and the security is in the $\binom{16}{1}\text{-}\mathsf{OT}_2$-hybrid model.

The correlated AND functionality $\mathcal{F}_{\mathbf{cAND}}$ takes as input boolean shares of x, y and z, and returns the boolean shares of $d = x \wedge y$ and $e = x \wedge z$. To generate the corresponding correlated bit triple $(\langle\delta_x\rangle^{\mathrm{B}}, \langle\delta_y\rangle^{\mathrm{B}}, \langle\delta_d\rangle^{\mathrm{B}})$ and $(\langle\delta_x\rangle^{\mathrm{B}}, \langle\delta_z\rangle^{\mathrm{B}}, \langle\delta_e\rangle^{\mathrm{B}})$, Cryptflow2 (appendix A.2 in [21]) uses an instance of $\binom{8}{1}$-OT_2. The communication complexity of invoking a $\mathcal{F}_{\mathrm{cAND}}$ instance is $2\lambda + 22$ bits, and the security is in the $\binom{8}{1}$-OT_2-hybrid model.

Boolean to arithmetic share conversion $\mathcal{F}^{l}_{\mathbf{B2A}}$ converts the same secret x's boolean shares over $\mathbb{Z}_2$ to arithmetic shares over $\mathbb{Z}_{2^l}$. For example, $\mathcal{F}^{2}_{\mathrm{B2A}}$ may convert the boolean shares $\langle x\rangle^{\mathrm{B}}_0 = 0, \langle x\rangle^{\mathrm{B}}_1 = 0$ of secret $x = 0$ to arithmetic shares $\langle x\rangle^{\mathrm{A}}_0 = 1, \langle x\rangle^{\mathrm{A}}_1 = 3$ over Z_4. $\mathcal{F}^{l}_{\mathrm{B2A}}$ can be realized with one call to 1-out-of-2 correlated OT [2] (denoted as $\binom{1}{2}$-COT_l), with communicating $\lambda + l$ bits and is in the $\binom{1}{2}$-COT_l-hybrid model (appendix A.4 in [21]).

2.3 Parallel Prefix Adder (PPA)

Adder is a fundamental concept in the field of digital electronics. In the context of addition, an adder circuit takes two l-bit numbers, a and b, and produces a sum c. The circuit calculates c bit-by-bit from the least significant to the most significant bit. In particular, the i-th bit of c is calculated as $c[i] = a[i] \oplus b[i] \oplus \mathsf{carry}$, where carry is the carry bit from previous calculation of $c[i-1]$ (i.e., the carry-out bit in the $(i-1)$-th bit of $a+b$). The carry calculation problem arises when adding two binary numbers with multiple bits. In particular, calculating $c[i]$ requires the carry bit of calculating $c[i-1]$, and so on. All carry bits have to be computed sequentially, which results in potentially large delays in computing the final sum.

As the most common choice for faster adders, parallel prefix adders (PPA) use a pre-computation technique that allows them to calculate the carry bits in parallel [1]. This is done by dividing the bits into groups and using a series of logical operations to compute the carry bits for each group. The carry bits are then combined in a final step to produce the final sum. For a group from the j-th bit to the i-th bit with $j \geq i$, define the group propagate signal as $P_{j:i}$ and group generate signal as $G_{j:i}$. We refer to $j - i + 1$ as the group length. When the group length equals 1 (i.e., $j = i$), we use the simpler notations P_i and G_i, which are defined as:

$$G_i \stackrel{def}{=} a[i] \wedge b[i], \quad P_i \stackrel{def}{=} a[i] \oplus b[i] \tag{1}$$

When $j > i$, the group signals $(P_{j:i}, G_{j:i})$ are defined as:

$$P_{j:i} \stackrel{def}{=} P_j \wedge P_{j-1} \wedge \ldots \wedge P_i \tag{2}$$
$$G_{j:i} \stackrel{def}{=} G_j \oplus (P_j \wedge G_{j-1}) \oplus (P_j \wedge P_{j-1} \wedge G_{j-2}) \oplus \cdots \oplus (P_j \wedge P_{j-1} \wedge \cdots \wedge P_{i+1} \wedge G_i)$$

We can combine two adjacent groups $(P_{z:y+1}, G_{z:y+1})$ and $(P_{y:x}, G_{y:x})$ into a longer group $(P_{z:x}, G_{z:x})$ of length $z - x + 1$ $(z > y \geq x)$, by defining the dot $\circ$ operator:

$$\begin{aligned}(P_{z:x},\ G_{z:x}) &= (P_{z:y+1},\ G_{z:y+1}) \circ (P_{y:x},\ G_{y:x}) \\ &\stackrel{def}{=} (P_{z:y+1} \wedge P_{y:x},\ \ G_{z:y+1} \oplus P_{z:y+1} \wedge G_{y:x})\end{aligned} \tag{3}$$

The calculation of group signals $(P_{j:i}, G_{j:i})$ is done once i reaches the least significant bit (i.e., $i = 1$). At this point, $G_{j:1}$ is exactly the carry-out bit of calculating $c[j]$.

Take $l = 4$ as an example. Suppose we want to use PPA to learn the carry-out bit of calculating $c[4]$ (i.e., the group generate signal $G_{4:1}$). In the first step, PPA calculates

in parallel the group signals $(P_1, G_1), (P_2, G_2), (P_3, G_3), (P_4, G_4)$ for groups of length 1. In the second step, PPA combines the groups in parallel by equation 3 to obtain the signals $(P_{2:1}, G_{2:1}), (P_{4:3}, G_{4:3})$ for groups of length 2. In the third step, PPA further combines the groups to obtain the desired signals $(P_{4:1}, G_{4:1})$ for the group of length 4.

3 Two-Bit Extraction

In this section, we first define our new notion of two-bit extraction 2Bit-Extr that is customized for faithful truncation. We then present two protocols to realize it.

3.1 Defining Two-Bit Extraction

Before defining our new notion, we recall the related bit extraction notion. Bit extraction is a special case of bit decomposition [16], where a single m-th ($m \leq l$) bit of the arithmetic share $\langle x \rangle^{\mathrm{A}} \in \mathbb{Z}_{2^l}$ should be decomposed into a boolean sharing, i.e., $\langle x[m] \rangle^{\mathrm{B}}$. We can constrain the bit extraction to the MSB extraction, because extracting the m-th bit from x is equivalent to extracting the MSB of $x[m:1]$ with shorter bit length of m. Let $\mathcal{F}^{l}_{\text{Bit-Extr}}$ denote the bit extraction functionality which takes as input the arithmetic shares of $x \in \mathbb{Z}_{2^l}$ and returns the boolean shares of $x[l]$ as outputs.

Two-bit extraction extends the notion of bit extraction, in which two bits—the m-th and s-th ($m > s$) bit of the arithmetic share $\langle x \rangle^{\mathrm{A}}$ should be decomposed into their respective boolean sharing, i.e., $\langle x[m] \rangle^{\mathrm{B}}$ and $\langle x[s] \rangle^{\mathrm{B}}$. Similarly, we constrain two-bit extraction to extracting the MSB and a lower s-th bit ($1 \leq s < l$).

The functionality $\mathcal{F}^{l,s}_{\textbf{2Bit-Extr}}$ for two-bit extraction takes arithmetic shares of $x \in \mathbb{Z}_{2^l}$ as input and returns boolean shares of $x[l]$ and $x[s]$ as outputs.

3.2 Trivial construction for $\mathcal{F}^{l,s}_{\textbf{2Bit-Extr}}$

A trivial two-bit extraction construction can be achieved by invoking two instances of bit extraction: the first invocation $\mathcal{F}^{l}_{\text{Bit-Extr}}$ extracts the MSB of x while the second invocation $\mathcal{F}^{s}_{\text{Bit-Extr}}$ extracts the MSB of $x[s:1]$. The parties can provide $\langle x \rangle^{\mathrm{A}}$ and $\langle x \rangle^{\mathrm{A}}[s:1]$ as inputs to the first and second invocations, respectively, to obtain the desired two-bit extraction.

The correctness of this trivial construction directly follows from the correctness of $\mathcal{F}^{l}_{\text{Bit-Extr}}$ and $\mathcal{F}^{s}_{\text{Bit-Extr}}$. This trivial construction securely realizes $\mathcal{F}^{l,s}_{\text{2Bit-Extr}}$ in the $(\mathcal{F}^{l}_{\text{Bit-Extr}}, \mathcal{F}^{s}_{\text{Bit-Extr}})$-hybrid model. For $b \in \{0, 1\}$, the simulator $\mathcal{S}_b$ of the view of the corrupted party $\mathcal{P}_b$ gets input $(\langle x \rangle^{\mathrm{A}}_b, (\langle x[l] \rangle^{\mathrm{B}}_b, \langle x[s] \rangle^{\mathrm{B}}_b))$ (i.e., the input and output of $\mathcal{P}_b$), which is the identical to the view of $\mathcal{P}_b$ in the corresponding execution (where here $\langle x[l] \rangle^{\mathrm{B}}_b$ and $\langle x[s] \rangle^{\mathrm{B}}_b$ serve as the responses of $\mathcal{F}^{l}_{\text{Bit-Extr}}$ and $\mathcal{F}^{s}_{\text{Bit-Extr}}$, respectively). The simulation is trivial, i.e., $\mathcal{S}_b$ can simply forward $\langle x[l] \rangle^{\mathrm{B}}_b$ and $\langle x[s] \rangle^{\mathrm{B}}_b$ to $\mathcal{P}_b$. Thus, the view of party $\mathcal{P}_b$ can be perfectly simulated.

The invocations to $\mathcal{F}^{l}_{\text{Bit-Extr}}$ and $\mathcal{F}^{s}_{\text{Bit-Extr}}$ are highly interconnected. The first invocation takes input x, and the second invocation takes input $x[s:1]$. This mutual input dependence suggests that a more efficient construction that combines the two invocations and leverages the shared input, may exist.

Input: For $b \in \{0,1\}$, party $\mathcal{P}_b$ uses four boolean shares $\langle P_2\rangle_b^{\mathrm{B}}, \langle P_1\rangle_b^{\mathrm{B}}, \langle G_2\rangle_b^{\mathrm{B}}, \langle G_1\rangle_b^{\mathrm{B}} \in \mathbb{Z}_2$ as input.
Output: $\mathcal{P}_b$ receives two boolean shares $\langle P\rangle_b^{\mathrm{B}}$ and $\langle G\rangle_b^{\mathrm{B}}$ s.t. $P = P_2 \wedge P_1$, $G = G_2 \oplus (P_2 \wedge G_1)$.

1. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\mathcal{F}_{\mathrm{cAND}}$, where $\mathcal{P}_b$ uses $\langle P_2\rangle_b^{\mathrm{B}}$, $\langle P_1\rangle_b^{\mathrm{B}}$, $\langle G_1\rangle_b^{\mathrm{B}}$ as input and receives two boolean shares $\langle P_2 \wedge P_1\rangle_b^{\mathrm{B}}$ and $\langle P_2 \wedge G_1\rangle_b^{\mathrm{B}}$.
2. Party $\mathcal{P}_b$ outputs $\langle P\rangle_b^{\mathrm{B}} = \langle P_2 \wedge P_1\rangle_b^{\mathrm{B}}$, $\langle G\rangle_b^{\mathrm{B}} = \langle G_2\rangle_b^{\mathrm{B}} \oplus \langle P_2 \wedge G_1\rangle_b^{\mathrm{B}}$.

Fig. 2. Protocol Π_{combine} (Combine)

3.3 Improved construction $\Pi^{l,s}_{\mathbf{2Bit\text{-}Extr}}$

This section begins with the definition of the combine functionality $\mathcal{F}_{\mathrm{combine}}$, which is a subroutine used in our improved two-bit extraction construction. Next, we introduce our improved construction for $\mathcal{F}^{l,s}_{\mathrm{2Bit\text{-}Extr}}$. At last, we apply two optimizations to this improved construction.

The combine subroutine. Let $\mathcal{F}_{\mathrm{combine}}$ denote the combine functionality that takes as input the boolean shares of P_2, P_1, G_2 and G_1, and output the boolean shares of $P = P_2 \wedge P_1$ and $G = G_2 \oplus (P_2 \wedge G_1)$.

A combine protocol Π_{combine} appears in Fig. 2. Its correctness directly follows the correctness of $\mathcal{F}_{\mathrm{cAND}}$. Its security is in the $\mathcal{F}_{\mathrm{cAND}}$-hybrid model. For $b \in \{0,1\}$, the simulator $\mathcal{S}_b$ of the view of the corrupted party $\mathcal{P}_b$ gets input $((\langle P_2\rangle_b^{\mathrm{B}}, \langle P_1\rangle_b^{\mathrm{B}}, \langle G_2\rangle_b^{\mathrm{B}}, \langle G_1\rangle_b^{\mathrm{B}}), (\langle P\rangle_b^{\mathrm{B}}, \langle G\rangle_b^{\mathrm{B}}))$ (i.e., the input and output of party $\mathcal{P}_b$). To simulate the responses of $\mathcal{F}_{\mathrm{cAND}}$ received by $\mathcal{P}_b$, $\mathcal{S}_b$ simply forwards $\langle P\rangle_b^{\mathrm{B}}$ and $\langle G\rangle_b^{\mathrm{B}} \oplus \langle G_2\rangle_b^{\mathrm{B}}$ to $\mathcal{P}_b$, which is identical to the view of $\mathcal{P}_b$ in the corresponding real execution. Thus, the view of $\mathcal{P}_b$ can be perfectly simulated. The protocol Π_{combine} only involves one call to $\mathcal{F}_{\mathrm{cAND}}$ which requires communicating $2\lambda + 22$ bits.

The improved construction $\Pi^{l,s}_{\mathbf{2Bit\text{-}Extr}}$. By extracting the MSB and the s-th bit in a batch, we present a more efficient construction for $\mathcal{F}^{l,s}_{\mathrm{2Bit\text{-}Extr}}$. Our key observation is that extracting the MSB and s-th bit of the same value x are highly interconnected: the intermediate results of extracting the MSB can be reused to extract the lower s-th bit.

- Construction overview. Our construction first reduces the bit extraction problem to a carry calculation problem, because $\langle x[i]\rangle_0^{\mathrm{B}} \bigoplus \langle x[i]\rangle_1^{\mathrm{B}} = \langle x\rangle_0^{\mathrm{A}}[i] \bigoplus \langle x\rangle_1^{\mathrm{A}}[i] \bigoplus \mathsf{carry}$, where carry represents the carry-out bit in the $(i-1)$-th bit of $\langle x\rangle_0^{\mathrm{A}} + \langle x\rangle_1^{\mathrm{A}}$. The parties $\mathcal{P}_b$ only need to learn the boolean shares of the corresponding carry-out bit $\langle \mathsf{carry}\rangle_b^{\mathrm{B}}$, as $\langle x[i]\rangle_b^{\mathrm{B}} = \langle x\rangle_b^{\mathrm{A}}[i] \oplus \langle \mathsf{carry}\rangle_b^{\mathrm{B}}$. To solve the carry calculation problem, we rely on PPA (Section 2.3).

Our construction makes use of two subroutines: *(1)* to calculate the group propagate signal and the generate signal from the two input additive shares $\langle x\rangle_0^{\mathrm{A}}$ and $\langle x\rangle_1^{\mathrm{A}}$; and *(2)* to combine the signals of two adjacent groups into the ones for a longer group. In particular, we use $\binom{2}{1}$-OT_1 employing the lookup-table based approach of [9], and the combine functionality $\mathcal{F}_{\mathrm{combine}}$ to instantiate the two subroutines, respectively.

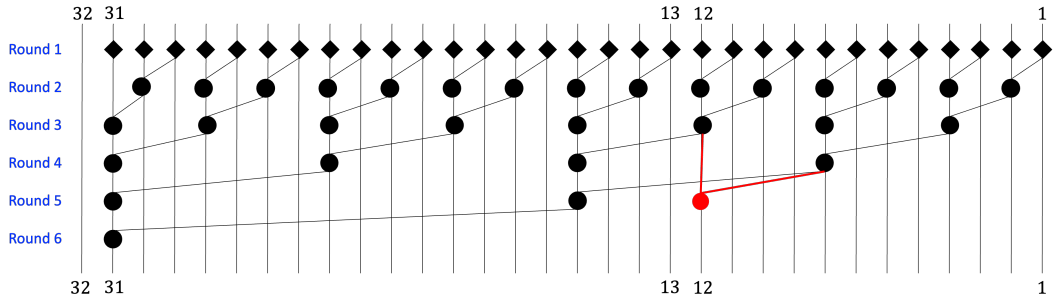
Fig. 3 illustrates the circuit used in our construction to compute the carry-out bits necessary for extracting the MSB and 13-th bit of $x \in \mathbb{Z}_{2^{32}}$. To extract the MSB, we utilize PPA to calculate the carry-out bit in the $(l-1)$-th bit (i.e., denoted as the group generate signal $G_{31:1}$). To calculate $G_{31:1}$, PPA first calculates the group signal P_i and G_i from the two input additive shares, and then iteratively combines the signals of two adjacent

groups into the ones for a longer group. The involved operations are marked as black in Fig. 3. To further extract the s-th bit, i.e., to calculate the group generate signal $G_{12:1}$, we observe that some intermediate group signals of the previous $G_{31:1}$ calculation can be reused. Specifically, in Round 5, we can combine the group signal $(P_{12:9}, G_{12:9})$ generated in Round 3 and $(P_{8:1}, G_{8:1})$ generated in Round 4 to calculate the desired $G_{12:1}$. Our two-bit extraction protocol is formally described in Fig. 4.

- Correctness analysis. We first demonstrate that the adopted 2PC subroutines can accurately implement the circuit of our protocol, and then we show that the implemented circuit can correctly extract the MSB and the s-th bit. Our protocol's circuit only requires two subroutines. For the $\binom{2}{1}$-OT_1 subroutine, we have verified that $\mathsf{Reconst}(\langle G_i\rangle_0^{\mathrm{B}}, \langle G_i\rangle_1^{\mathrm{B}}) = \langle x\rangle_0^{\mathrm{A}}[i] \wedge \langle x\rangle_1^{\mathrm{A}}[i]$ by enumerating all possible values of $\langle x\rangle_0^{\mathrm{A}}[i], \langle x\rangle_1^{\mathrm{A}}[i] \in Z_2$. Additionally, $\mathsf{Reconst}(\langle P_i\rangle_0^{\mathrm{B}}, \langle P_i\rangle_1^{\mathrm{B}}) = \langle x\rangle_0^{\mathrm{A}}[i] \oplus \langle x\rangle_1^{\mathrm{A}}[i]$. As a result, we can confirm the accuracy of calculating the group signals P_i and G_i from the two input additive shares. Moreover, by the correctness of $\mathcal{F}_{\text{combine}}$, combing two groups into a longer group is also correct. Therefore, we conclude that the adopted 2PC subroutines can faithfully realize the implemented circuit.

To show $\mathsf{Reconst}(\langle x[l]\rangle_0^{\mathrm{B}}, \langle x[l]\rangle_1^{\mathrm{B}}) = \langle GL\rangle_0^{\mathrm{B}} \oplus \langle GL\rangle_1^{\mathrm{B}} \oplus \langle x\rangle_0^{\mathrm{A}}[l] \oplus \langle x\rangle_1^{\mathrm{A}}[l] = x[l]$, we in fact have to show $GL = \langle GL\rangle_0^{\mathrm{B}} \oplus \langle GL\rangle_1^{\mathrm{B}}$ is the carry-out bit of calculating the $(i-1)$-th bit of $\langle x\rangle_0^{\mathrm{A}} + \langle x\rangle_1^{\mathrm{A}}$. Since we exploit PPA to solve the carry calculation problem, we essentially have to show the updated boolean shares of GL in step 23 are reconstructed to be the group generate signal $G_{l-1:1}$. Note that step 13 and 17 iteratively combines two adjacent groups into a longer group and step 14 updates GL accordingly. At the last iteration when $i = \log l$ and $j = 1$, GL is updated to be $G_{l-1:1}$. Thus, the circuit can correctly extract the l-th bit.

When $s = 1$, we have $\mathsf{Reconst}(\langle x[1]\rangle_0^{\mathrm{B}}, \langle x[1]\rangle_1^{\mathrm{B}}) = \langle x\rangle_0^{\mathrm{A}}[1] \oplus \langle x\rangle_1^{\mathrm{A}}[1] = x[1]$. When $s > 1$, we need to show the updated GS in step 23 equals the group generate signal $G_{s-1:1}$. Let $\{j_k, \ldots, j_1\}$ denote the positions (in descending order) of the $k \in [1, \lceil \log s \rceil]$ non-zero bits in the binary form $s_{\log l} || \ldots || s_1$ of $s-1$. When $i = j_1$ (i.e., the position of the least significant non-zero bit), we have $2^{j_1-1} \cdot \lfloor \frac{s-1}{2^{j_1}-1} \rfloor = s-1$. So we have GS is initially set to be $G_{s-1:s-2^{j_1-1}}$ in step 22. When $i = j_2$ and $y = 2^{j_2-1} \cdot \lfloor \frac{s-1}{2^{j_2-1}} \rfloor = s - 1 - 2^{j_1-1}$ in step 18, the following step 20 essentially combines the signals $G_{s-1:s-2^{j_1-1}}$ and $G_{s-2^{j_1-1}-1:s-2^{j_1-1}-2^{j_2-1}}$ of two adjacent groups. Then GS is updated to be the combined signal $G_{s-1:s-2^{j_1-1}-2^{j_2-1}}$. When $i = j_{r|r>2}$, we have GS is updated to be $G_{s-1:s-\sum_{t=1}^{r} 2^{j_t-1}}$. When i reaches j_k, GS is updated



† The diamond and the circle symbol refer to the $\binom{2}{1}$-OT_1 and $\mathcal{F}_{\text{combine}}$ subroutine, respectively.

‡ The operations related to the MSB extraction are marked in black while the additional operations required by further extracting the s-th bit are marked in red.

Fig. 3. The circuit of calculating the carry-out bits for extracting the MSB and 13-th bit of $x \in \mathbb{Z}_{2^{32}}$

Input: For $b \in \{0,1\}$, party $\mathcal{P}_b$ uses $\langle x\rangle_b^{\mathrm{A}} \in \mathbb{Z}_{2^l}$ as input.
Output: $\mathcal{P}_b$ receives two boolean shares $\langle x[l]\rangle_b^{\mathrm{B}}$ and $\langle x[s]\rangle_b^{\mathrm{B}}$.

1. **for** $(i = 1; i < l; i = i + 1)$ **do:**
2. Party $\mathcal{P}_0$ first samples $\langle G_i\rangle_0^{\mathrm{B}} \xleftarrow{\$} \mathbb{Z}_2$, then for $j \in \{0,1\}$ sets $\mathsf{msg}_j = (\langle x\rangle_0^{\mathrm{A}}[i] \wedge j) \oplus \langle G_i\rangle_0^{\mathrm{B}}$.
3. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\binom{2}{1}$-OT_1, where $\mathcal{P}_0$ plays the role of sender and $\mathcal{P}_1$ plays the role of receiver. Party $\mathcal{P}_0$ uses $\{\mathsf{msg}_j\}_{j\in\{0,1\}}$ as input, and receives noting. Party $\mathcal{P}_1$ uses $\langle x\rangle_1^{\mathrm{A}}[i]$ as input, and receives a value which is recorded as $\langle G_i\rangle_1^{\mathrm{B}}$.
4. Party $\mathcal{P}_b$ sets $\langle P_i\rangle_b^{\mathrm{B}} = \langle x\rangle_b^{\mathrm{A}}[i]$.
5. Party $\mathcal{P}_b$ initializes $\langle PL\rangle_b^{\mathrm{B}}, \langle GL\rangle_b^{\mathrm{B}}, \langle PS\rangle_b^{\mathrm{B}}$ and $\langle GS\rangle_b^{\mathrm{B}}$ as *null* values.
6. **if** $(s == 1)$ **then:** Party $\mathcal{P}_b$ sets $\langle PS\rangle_b^{\mathrm{B}} = \langle GS\rangle_b^{\mathrm{B}} = 0$.
7. Party $\mathcal{P}_b$ parses $s-1$ as a $(\log l)$-bit binary string $s_{\log l}||\ldots||s_1$, where $s_{i|i\in[1,\log l]} \in \{0,1\}$.
8. **for** $(i = 1; i \le \log l; i = i + 1)$ **do:**
9. **for** $(\ j = 1;\ j + 2^i - 1 \le l;\ j = j + 2^i\ )$ **do:**
10. Party $\mathcal{P}_b$ sets $z = j + 2^i - 1$, $y = j + 2^{i-1} - 1$, $x = j$.
11. **if** $(z == l)$ **then:**
12. **if** ($\langle PL\rangle_b^{\mathrm{B}}$ and $\langle GL\rangle_b^{\mathrm{B}}$ is not *null*) **then:**
13. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\mathcal{F}_{\text{combine}}$, where $\mathcal{P}_b$ uses $\langle P_{l-1:y+1}\rangle_b^{\mathrm{B}}$, $\langle P_{y:x}\rangle_b^{\mathrm{B}}$,$\langle G_{l-1:y+1}\rangle_b^{\mathrm{B}}$, $\langle G_{y:x}\rangle_b^{\mathrm{B}}$ as inputs, and receives $\langle P_{l-1:x}\rangle_b^{\mathrm{B}}$, $\langle G_{l-1:x}\rangle_b^{\mathrm{B}}$.
14. Party $\mathcal{P}_b$ updates $\langle PL\rangle_b^{\mathrm{B}}$ as $\langle P_{l-1:x}\rangle_b^{\mathrm{B}}$, $\langle GL\rangle_b^{\mathrm{B}}$ as $\langle G_{l-1:x}\rangle_b^{\mathrm{B}}$.
15. **else:** Party $\mathcal{P}_b$ sets $\langle PL\rangle_b^{\mathrm{B}} = \langle P_{l-1}\rangle_b^{\mathrm{B}}$, $\langle GL\rangle_b^{\mathrm{B}} = \langle G_{l-1}\rangle_b^{\mathrm{B}}$.
16. **else:**
17. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\mathcal{F}_{\text{combine}}$, where $\mathcal{P}_b$ uses $\langle P_{z:y+1}\rangle_b^{\mathrm{B}}$, $\langle P_{y:x}\rangle_b^{\mathrm{B}}$,$\langle G_{z:y+1}\rangle_b^{\mathrm{B}}$, $\langle G_{y:x}\rangle_b^{\mathrm{B}}$ as inputs, and receives $\langle P_{z:x}\rangle_b^{\mathrm{B}}$, $\langle G_{z:x}\rangle_b^{\mathrm{B}}$.
18. **if** $(s_i == 1$ and $y == 2^{i-1} \cdot \lfloor \frac{s-1}{2^{i-1}} \rfloor)$ **then:**
19. **if** ($\langle PS\rangle_b^{\mathrm{B}}$ and $\langle GS\rangle_b^{\mathrm{B}}$ is not *null*) **then:**
20. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\mathcal{F}_{\text{combine}}$ where $\mathcal{P}_b$ uses $\langle P_{s-1:y+1}\rangle_b^{\mathrm{B}}$, $\langle P_{y:x}\rangle_b^{\mathrm{B}}$, $\langle G_{s-1:y+1}\rangle_b^{\mathrm{B}}$, $\langle G_{y:x}\rangle_b^{\mathrm{B}}$ as input, and receives $\langle P_{s-1:x}\rangle_b^{\mathrm{B}}$, $\langle G_{s-1:x}\rangle_b^{\mathrm{B}}$.
21. Party $\mathcal{P}_b$ updates $\langle PS\rangle_b^{\mathrm{B}}$ as $\langle P_{s-1:x}\rangle_b^{\mathrm{B}}$, $\langle GS\rangle_b^{\mathrm{B}}$ as $\langle G_{s-1:x}\rangle_b^{\mathrm{B}}$.
22. **else:** Party $\mathcal{P}_b$ sets $\langle PS\rangle_b^{\mathrm{B}} = \langle P_{y:x}\rangle_b^{\mathrm{B}}$, $\langle GS\rangle_b^{\mathrm{B}} = \langle G_{y:x}\rangle_b^{\mathrm{B}}$.
23. Party $\mathcal{P}_b$ outputs $\langle x[l]\rangle_b^{\mathrm{B}} = \langle GL\rangle_b^{\mathrm{B}} \oplus \langle x\rangle_b^{\mathrm{A}}[l]$, $\langle x[s]\rangle_b^{\mathrm{B}} = \langle GS\rangle_b^{\mathrm{B}} \oplus \langle x\rangle_b^{\mathrm{A}}[s]$.

Fig. 4. Protocol $\Pi_{\text{2Bit-Extr}}^{l,s}$ (Two-bit extraction)

to be $G_{s-1:s-\sum_{t=1}^{k} 2^{j_t-1}}$. Since $\sum_{t=1}^{k} 2^{j_t-1} = s-1$, GS is updated to be $G_{s-1:1}$ at last. Thus, we have $\mathsf{Reconst}(\langle x[s]\rangle_0^{\mathrm{B}}, \langle x[s]\rangle_1^{\mathrm{B}}) = \langle GS\rangle_0^{\mathrm{B}} \oplus \langle GS\rangle_1^{\mathrm{B}} \oplus \langle x\rangle_0^{\mathrm{A}}[s] \oplus \langle x\rangle_1^{\mathrm{A}}[s] = x[s]$.

- Security analysis. The proposed protocol $\Pi_{\text{2Bit-Extr}}^{l,s}$ is in the $(\binom{2}{1}$-$\mathsf{OT}_1, \mathcal{F}_{\text{combine}})$-hybrid model. We construct two simulators for the following two cases:

Case 1: $\mathcal{P}_0$ is corrupted. The simulator $\mathcal{S}_0$ gets input $(\langle x\rangle_0^{\mathrm{A}}, (\langle x[l]\rangle_0^{\mathrm{B}}, \langle x[s]\rangle_0^{\mathrm{B}}))$ (i.e., the input and output of $\mathcal{P}_0$). $\mathcal{S}_0$ needs to simulate those intermediate messages received by $\mathcal{P}_0$ when invoking $\mathcal{F}_{\text{combine}}$. Before the last round, $\mathcal{S}_0$ returns random $r_1, r_2 \in \mathbb{Z}_2$ as the responses of $\mathcal{F}_{\text{combine}}$. Due to the uniform distribution of r_1, r_2, $\mathcal{P}_0$ cannot distinguish r_1, r_2 from the real boolean shares output by $\mathcal{F}_{\text{combine}}$. In the last round, $\mathcal{S}_0$ returns $r_3 = \langle x[l]\rangle_0^{\mathrm{B}} \oplus \langle x\rangle_0^{\mathrm{A}}[l]$, $r_4 = \langle x[s]\rangle_0^{\mathrm{B}} \oplus \langle x\rangle_0^{\mathrm{A}}[s]$ as the responses of $\mathcal{F}_{\text{combine}}$, which conforms to the view of $\mathcal{P}_0$ in the corresponding execution. Furthermore, r_3 and r_4 are uniformly distributed, because

they are masked by random $\langle x\rangle_0^{\mathrm{A}}[l]$ and $\langle x\rangle_0^{\mathrm{A}}[s]$. Thus, $\mathcal{P}_0$ cannot distinguish r_3, r_4 from the real boolean shares output by $\mathcal{F}_{\text{combine}}$.

Case 2: $\mathcal{P}_1$ is corrupted. The simulator $\mathcal{S}_1$ gets input $(\langle x\rangle_1^{\mathrm{A}}, (\langle x[l]\rangle_1^{\mathrm{B}}, \langle x[s]\rangle_1^{\mathrm{B}}))$ (i.e., the input and output of $\mathcal{P}_1$). $\mathcal{S}_1$ needs to simulate those intermediate messages received by $\mathcal{P}_1$ when invoking $\binom{2}{1}$-OT_1 as well as invoking $\mathcal{F}_{\text{combine}}$. When $\binom{2}{1}$-OT_1 receives $\mathcal{P}_1$'s input, $\mathcal{S}_1$ returns random $r_1 \in \mathbb{Z}_2$ as the response of $\binom{2}{1}$-OT_1. Since the real OT message msg output to $\mathcal{P}_1$ is masked by a random value uniformly sampled by $\mathcal{P}_0$, msg is also uniformly distributed. Thus, $\mathcal{P}_1$ cannot distinguish r_1 from msg. To simulate the responses of $\mathcal{F}_{\text{combine}}$, $\mathcal{S}_1$ operates the same as $\mathcal{S}_0$ does for the corrupted party $\mathcal{P}_0$. That is, the simulator-generated view of the corrupted party $\mathcal{P}_1$ is identically distributed to that of a real execution.

Thus, we conclude that the view of each party $\mathcal{P}_b, b \in \{0,1\}$ can be perfectly simulated.

- Communication complexity. To extract the l-th bit, $\Pi^{l,s}_{\text{2Bit-Extr}}$ involves $l-1$ calls to $\binom{2}{1}$-OT_1 and $l-2$ calls to $\mathcal{F}_{\text{combine}}$. To additionally extract the s-th bit based on the intermediate results of MSB extraction, $0 \le k \le \lceil \log s \rceil - 1$ calls to $\mathcal{F}_{\text{combine}}$ are needed, because there are at most $\lceil \log s \rceil$ non-zero bits in the binary form $s_{\log l} || \ldots || s_1$ of $s-1$. Thus, the communication bits are $3\lambda l + 24l - 5\lambda - 46 + (2\lambda + 22)k$ in total.

- Optimizations for $\Pi^{l,s}_{\mathsf{2Bit\text{-}Extr}}$. Similar to the observation made in [21,6] which construct comparison protocols from binary tree traversal, our $\mathsf{2Bit\text{-}Extr}$ construction in Fig. 4 can be optimized in two ways. First, utilizing $\binom{2^m}{1}$-OT_2 ($m \ge 2$), we can calculate signals for groups of length ≥ 2. For example, by invoking an instance of $\binom{4}{1}$-OT_2, we can directly calculate the signals for group of length 2 from the two input additive shares $\langle x\rangle_0^{\mathrm{A}}$ and $\langle x\rangle_1^{\mathrm{A}}$. Namely, according to equation 1 and equation 2, we can calculate $P_{i+1:i}$ as $(\langle x\rangle_0^{\mathrm{A}}[i+1] \oplus \langle x\rangle_1^{\mathrm{A}}[i+1]) \wedge (\langle x\rangle_0^{\mathrm{A}}[i] \oplus \langle x\rangle_1^{\mathrm{A}}[i])$ and $G_{i+1:i}$ as $(\langle x\rangle_0^{\mathrm{A}}[i+1] \wedge \langle x\rangle_1^{\mathrm{A}}[i+1]) \oplus (\langle x\rangle_0^{\mathrm{A}}[i+1] \oplus \langle x\rangle_1^{\mathrm{A}}[i+1]) \wedge (\langle x\rangle_0^{\mathrm{A}}[i] \wedge \langle x\rangle_1^{\mathrm{A}}[i])$. Taking the above $m=2$ example, we employ the lookup-table based approach of [9] to implement this optimization as follows:

- Party $\mathcal{P}_0$ samples $\langle P_{i+1:i}\rangle_0^{\mathrm{B}}, \langle G_{i+1:i}\rangle_0^{\mathrm{B}} \xleftarrow{\$} \mathbb{Z}_2$. For $j = \{00, 01, 10, 11\}$, party $\mathcal{P}_0$ parses j as a 2-bit binary string $j_2||j_1$, then sets $\mathsf{msg}_j^0 = (\langle x\rangle_0^{\mathrm{A}}[i+1] \oplus j_2) \wedge (\langle x\rangle_0^{\mathrm{A}}[i] \oplus j_1) \oplus \langle P_{i+1:i}\rangle_0^{\mathrm{B}}$ and $\mathsf{msg}_j^1 = (\langle x\rangle_0^{\mathrm{A}}[i+1] \wedge j_2) \oplus (\langle x\rangle_0^{\mathrm{A}}[i+1] \oplus j_2) \wedge (\langle x\rangle_0^{\mathrm{A}}[i] \wedge j_1) \oplus \langle G_{i+1:i}\rangle_0^{\mathrm{B}}$, and finally sets $\mathsf{msg}_j = \mathsf{msg}_j^0 || \mathsf{msg}_j^1$.
- Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\binom{4}{1}$-OT_2 where P_0 plays the sender with inputs $\{\mathsf{msg}_j\}_{j \in \{00,01,10,11\}}$ and P_1 plays the receiver with input $\langle x\rangle_1^{\mathrm{A}}[i+1] \,||\, \langle x\rangle_1^{\mathrm{A}}[i]$. $\mathcal{P}_1$ parses its output as a 2-bit string $\mathsf{msg}^0||\mathsf{msg}^1$, and sets $\langle P_{i+1:i}\rangle_1^{\mathrm{B}}$ as msg^0 and $\langle G_{i+1:i}\rangle_1^{\mathrm{B}}$ as msg^1.

Given this optimization used, the round complexity of $\Pi^{l,s}_{\text{2Bit-Extr}}$ can be brought down $\lfloor \log m \rfloor$ rounds. For the communication complexity, let $l' = \lceil \frac{l-1}{m} \rceil$ and $s' = \lceil \frac{s-1}{m} \rceil$. When extracting the MSB, this optimization involves at most l' calls to $\binom{2^m}{1}$-OT_2 and $l'-1$ calls to $\mathcal{F}_{\text{combine}}$. To further extract the s-th bit, at most 1 call to $\binom{2^m}{1}$-OT_2 and $\lceil \log s' \rceil - 1$ calls to $\mathcal{F}_{\text{combine}}$ are needed. With parameters λ, l, s and m, we can obtain an approximate estimation of the number of communication bits. Our analysis reveals that $m=6$ offers the most significant advantage in terms of communication complexity for the typical values of l and s used by PPML (see Table 2). However, the computational cost also increases super-polynomially with m. While benchmarking the faithful truncation protocol which utilized this optimized two-extraction construction, we concluded that $m=4$ offered a competitive

trade-off between communication and computation. This empirical finding is consistent with our baseline work Cryptflow2 [21] whose protocol also involves parameter m.

The second optimization is to eliminate operations that involve unused propagate signals. For groups reaching the least significant bit, their propagate signals are never used. So we can safely remove operations combing such signals (e.g., the combine operations on the rightmost branches in Fig. 3). With this optimization, instead of invoking a $\mathcal{F}_{\text{combine}}$ instance to combine both the propagate and generate signal, we can invoke a call to $\mathcal{F}_{\text{AND}}$ to combine only the generate signal. A call to $\mathcal{F}_{\text{combine}}$ and $\mathcal{F}_{\text{AND}}$ requires communicating $2\lambda + 22$ and $\lambda+20$ bits, respectively. This optimization removes $\log l$ useless propagate signal calculations and thus saves $\lambda \log l + 2 \log l$ communication bits in total.

4 Truncation Errors and Faithful Truncation

We begin by providing an example of local truncation that can result in both small and big errors. Through this example, we show the importance of learning the boolean shares of the MSB and the $(s+1)$-th bit in eliminating the errors. Finally, we describe the faithful truncation construction in detail, which is based on the aforementioned two-bit extraction.

4.1 Why local truncation fails

Truncation is the process of converting the product x^{int} of two fixed-point integers from a representation scaled by 2^{2s} to the fixed-point representation scaled by 2^s. In the two's complement encoding, dividing x^{int} by 2^s is equivalent to performing an arithmetic right shift of x by s bits. A naive solution to this problem is local truncation: party $\mathcal{P}_b$, holding the additive share $\langle x \rangle_b^{\text{A}}$, outputs $\langle y \rangle_b^{\text{A}} = \langle x \rangle_b^{\text{A}} \gg s$ as its additive share for y, in the hope that y^{int} will equal $\frac{x^{\text{int}}}{2^s}$. However, this local truncation may fail with certain probabilities [16].

Consider the example where $l = 4$, $s = 1$, and the product to be truncated $x^{\text{int}} = 6$ whose corresponding ring element $x = 0110$. Suppose x is additively shared as $\langle x \rangle_0^{\text{A}} = 1001$ and $\langle x \rangle_1^{\text{A}} = 1101$. To obtain the shares of $y^\star = 0011$ corresponding to the truncated product $\frac{x^{\text{int}}}{2} = 3$, the parties arithmetic right shift $\langle x \rangle^{\text{A}}$ by s bits, and output shares $\langle y \rangle_0^{\text{A}} = 1100$ and $\langle y \rangle_1^{\text{A}} = 1110$. But $y = \mathsf{Reconst}(\langle y \rangle_0^{\text{A}}, \langle y \rangle_1^{\text{A}})$ is 1010, which corresponds to -6, instead.

The local truncation is flawed since it ignores small and big errors. Specifically, it neglects the carry-out bit in the s-th bit of $\langle x \rangle_0^{\text{A}} + \langle x \rangle_1^{\text{A}}$, which can lead to the small error if the truncated s-bit has a carry-out of 1. Additionally, the arithmetic right shift is sign-extended, which can cause the big error if the MSBs of the two shares are opposite to that of the secret.

To address the above issue, a preliminary step is required to detect the occurrence of small and big errors. This can be done easily by utilizing the boolean shares of $x[l]$ and $x[s+1]$. For example, $\langle x[s+1] \rangle^{\text{B}}$ can be used to determine the small error indicator t (i.e., the carry-out bit in the s-th bit) as $t = \langle x[s+1] \rangle_0^{\text{B}} \oplus \langle x[s+1] \rangle_1^{\text{B}} \oplus \langle x \rangle_0^{\text{A}}[s+1] \oplus \langle x \rangle_1^{\text{A}}[s+1]$. Moreover, the big error indicator k can be learned by checking whether the MSBs of the shares are opposite to that of the secret (i.e., $x[l] = \langle x[l] \rangle_0^{\text{B}} \oplus \langle x[l] \rangle_1^{\text{B}}$).

Once the error indicators k and t have been determined, the parties can proceed to correct the errors to achieve faithful truncation. This can be accomplished by performing additional computations based on k and t.

Input: For $b \in \{0,1\}$, party $\mathcal{P}_b$ uses $\langle x \rangle_b^{\mathrm{A}} \in \mathbb{Z}_{2^l}$ as input.
Output: Party $\mathcal{P}_b$ receives $\langle y \rangle_b^{\mathrm{A}}$ s.t. $y = x \gg s$.

1. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\mathcal{F}_{\text{2Bit-Extr}}^{l,s+1}$, where $\mathcal{P}_b$ uses $\langle x \rangle_b^{\mathrm{A}}$ as input and receives two boolean shares $\langle x[l] \rangle_b^{\mathrm{B}}$ and $\langle x[s+1] \rangle_b^{\mathrm{B}}$.
2. Party $\mathcal{P}_0$ locally execute the following steps:
   - Samples a random element $\langle k \rangle_0^{\mathrm{A}} \xleftarrow{\$} \mathbb{Z}_{2^l}$;
   - Sets $i_2 = \langle x \rangle_0^{\mathrm{A}}[l]$ and $i_1 = \langle x[l] \rangle_0^{\mathrm{B}}$;
   - For $j \in \{00, 01, 10, 11\}$, parses j as a 2-bit string $j_2 || j_1$, computes $\mathsf{msg}_j = 2^{l-s} \cdot (i_2 \wedge j_2 \wedge \overline{i_1 \oplus j_1}) + (2^l - 2^{l-s}) \cdot (\overline{i_2} \wedge \overline{j_2} \wedge (i_1 \oplus j_1)) - \langle k \rangle_0^{\mathrm{A}}$.
3. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invoke an instance of $\binom{4}{1}$-OT_l, where $\mathcal{P}_0$ plays the role of sender and $\mathcal{P}_1$ plays the role of receiver. Party $\mathcal{P}_0$ uses $\{\mathsf{msg}_j\}_{j \in \{00,01,10,11\}}$ as input, and receives nothing. Party $\mathcal{P}_1$ uses $\langle x \rangle_1^{\mathrm{A}}[l] \,||\, \langle x[l] \rangle_1^{\mathrm{B}}$ as input, and receives a value to be set as $\langle k \rangle_1^{\mathrm{A}}$.
4. Party $\mathcal{P}_b$ locally computes $\langle \mathsf{tmp} \rangle_b^{\mathrm{B}} = \langle x \rangle_b^{\mathrm{A}}[s+1] \bigoplus \langle x[s+1] \rangle_b^{\mathrm{B}}$.
5. Parties $\mathcal{P}_0$ and $\mathcal{P}_1$ invokes an instance of $\mathcal{F}_{\text{B2A}}^{l}$, where $\mathcal{P}_b$ uses $\langle \mathsf{tmp} \rangle_b^{\mathrm{B}} \in \mathbb{Z}_2$ as input, and receives $\langle t \rangle_b^{\mathrm{A}} \in \mathbb{Z}_{2^l}$.
6. Party $\mathcal{P}_b$ outputs $\langle y \rangle_b^{\mathrm{A}} = (\langle x \rangle_b^{\mathrm{A}} \gg s) + \langle t \rangle_b^{\mathrm{A}} + \langle k \rangle_b^{\mathrm{A}}$.

Fig. 5. Protocol $\Pi_{\text{Trunc}}^{l,s}$ (Faithful truncation)

4.2 Faithful truncation from two-bit extraction

Let $\mathcal{F}_{\text{Trunc}}^{l,s}$ denote the faithful truncation functionality that takes arithmetic shares of $x \in \mathbb{Z}_{2^l}$ as input, and returns arithmetic shares of $y = x \gg s$ as output. We propose a faithful truncation protocol in Fig. 5, whose correctness relies on the following proposition.

Proposition 1. *Let $\langle x \rangle_0^A$ and $\langle x \rangle_1^A$ denote the arithmetic shares of $x \in \mathbb{Z}_{2^l}$. Let t denote the carry-out in the s-th bit of $\langle x \rangle_0^A + \langle x \rangle_1^A$. Let k be defined as:*

$$k = \begin{cases} 2^l - 2^{l-s} : & \langle x \rangle_0^A[l] = \langle x \rangle_1^A[l] = 0 \,,\ x[l] = 1; \\ 2^{l-s} : & \langle x \rangle_0^A[l] = \langle x \rangle_1^A[l] = 1 \,,\ x[l] = 0; \\ 0 : & \textit{otherwise.} \end{cases} \tag{4}$$

Then we have:

$$(\langle x \rangle_0^A \gg s) + (\langle x \rangle_1^A \gg s) + k + t = (x \gg s) \tag{5}$$

Proof. The proposition follows from Corollary 4.2 in [21]. When the sign bit flipping issue happens, term k corrects the big error. When the carry-out of the s-th bit is 1 (i.e., the least significant s bits wrap around 2^s), term t corrects the small error.

- Correctness analysis. By the correctness of $\mathcal{F}_{\text{2Bit-Extr}}^{l,s+1}$, we have $\mathsf{Reconst}(\langle x[l] \rangle_0^{\mathrm{B}}, \langle x[l] \rangle_1^{\mathrm{B}}) = x[l]$. Furthermore, by the correctness of $\binom{4}{1}$-OT_l, we have $\mathsf{Reconst}(\langle k \rangle_0^{\mathrm{A}}, \langle k \rangle_1^{\mathrm{A}})$ equals the k defined by equation 4. Namely, our protocol correctly calculates the term k in equation 5. By the correctness of $\mathcal{F}_{\text{2Bit-Extr}}^{l,s+1}$, $\mathsf{Reconst}(\langle x[s+1] \rangle_0^{\mathrm{B}}, \langle x[s+1] \rangle_1^{\mathrm{B}}) = x[s+1]$. So $\mathsf{Reconst}(\langle tmp \rangle_0^{\mathrm{B}}, \langle tmp \rangle_1^{\mathrm{B}}) = \langle x \rangle_0^{\mathrm{A}}[s+1] \oplus \langle x \rangle_1^{\mathrm{A}}[s+1] \oplus x[s+1]$, which is exactly the carry-out bit in the s-th bit of $\langle x \rangle_0^{\mathrm{A}} + \langle x \rangle_1^{\mathrm{A}}$. Next, by the correctness of $\mathcal{F}_{\text{B2A}}^{l}$ which creates the arithmetic shares of the same secret tmp, we have $\mathsf{Reconst}(\langle t \rangle_0^{\mathrm{A}}, \langle t \rangle_1^{\mathrm{A}})$ equals the term t in equation 5. By equation 5, $\mathsf{Reconst}(\langle y \rangle_0^{\mathrm{A}}, \langle y \rangle_1^{\mathrm{A}}) = (\langle x \rangle_0^{\mathrm{A}} \gg s) + (\langle x \rangle_1^{\mathrm{A}} \gg s) + k + t = (x \gg s)$.

- Security analysis. The protocol $\Pi^{l,s}_{\text{Trunc}}$ securely realizes the functionality $\mathcal{F}^{l,s}_{\text{Trunc}}$ in the $(\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}, \binom{4}{1}\text{-}\mathsf{OT}_l, \mathcal{F}^{l}_{\text{B2A}})$-hybrid model. We construct two simulators for two cases.

Case 1: $\mathcal{P}_0$ is corrupted. The simulator $\mathcal{S}_0$ gets input $(\langle x\rangle^{\text{A}}_0, \langle y\rangle^{\text{A}}_0)$ (i.e., the input and output of the corrupted party $\mathcal{P}_0$). $\mathcal{S}_0$ needs to simulate $\mathcal{P}_0$'s received intermediate messages including: $(\langle x[l]\rangle^{\text{B}}_0, \langle x[s+1]\rangle^{\text{B}}_0)$ received from $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$ and $\langle t\rangle^{\text{A}}_0$ received from $\mathcal{F}^{l}_{\text{B2A}}$.

- When $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$ receiving $\langle x\rangle^{\text{A}}_0$ from $\mathcal{P}_0$, $\mathcal{S}_0$ returns random $r_1, r_2 \in \mathbb{Z}_2$ to $\mathcal{P}_0$. Since r_1 and r_2 are uniformly distributed, $\mathcal{P}_0$ cannot distinguish r_1 and r_2 from the real boolean shares $\langle x[l]\rangle^{\text{B}}_0$ and $\langle x[s+1]\rangle^{\text{B}}_0$ output by $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$.
- When $\binom{4}{1}\text{-}\mathsf{OT}_l$ receiving $\{\mathsf{msg}_j\}$ from $\mathcal{P}_0$, $\mathcal{S}_0$ extracts $\langle k\rangle^{\text{A}}_0$ (which is uniformly sampled by $\mathcal{P}_0$) from $\{\mathsf{msg}_j\}$. Concretely, $\mathcal{S}_0$ extracts $\langle k\rangle^{\text{A}}_0 = 2^{l-s} \cdot (\langle x\rangle^{\text{A}}_0[l] \wedge \langle x[l]\rangle^{\text{B}}_0) - \mathsf{msg}_{11}$.
- When $\mathcal{F}^{l}_{\text{B2A}}$ receiving $\langle \mathsf{tmp}\rangle^{\text{B}}_0$ from $\mathcal{P}_0$, $\mathcal{S}_0$ returns $r_3 = \langle y\rangle^{\text{A}}_0 - (\langle x\rangle^{\text{A}}_0 \gg s) - \langle k\rangle^{\text{A}}_0$ to $\mathcal{P}_0$, which conforms to the view of $\mathcal{P}_0$ in the corresponding real execution. Furthermore, since $\langle k\rangle^{\text{A}}_0$ is uniformly distributed, r_3 is also uniformly distributed. Hence, party $\mathcal{P}_0$ cannot distinguish r_3 from the real arithmetic share output by $\mathcal{F}^{l}_{\text{B2A}}$.

Case 2: $\mathcal{P}_1$ is corrupted. The simulator $\mathcal{S}_1$ gets input $(\langle x\rangle^{\text{A}}_1, \langle y\rangle^{\text{A}}_1)$. $\mathcal{S}_1$ needs to simulate $\mathcal{P}_1$'s received intermediate messages including: $(\langle x[l]\rangle^{\text{B}}_1, \langle x[s+1]\rangle^{\text{B}}_1)$ received from $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$, $\langle k\rangle^{\text{A}}_1$ received from $\binom{4}{1}\text{-}\mathsf{OT}_l$, and $\langle t\rangle^{\text{A}}_1$ received from $\mathcal{F}^{l}_{\text{B2A}}$.

- When $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$ receiving $\langle x\rangle^{\text{A}}_1$ from $\mathcal{P}_1$, $\mathcal{S}_1$ operates the same as $\mathcal{S}_0$ does for $\mathcal{P}_0$.
- When $\binom{4}{1}\text{-}\mathsf{OT}_l$ receiving $\langle x\rangle^{\text{A}}_1[l] \,||\, \langle x[l]\rangle^{\text{B}}_1$ from $\mathcal{P}_1$, $\mathcal{S}_1$ returns a random value $r_1 \in \mathbb{Z}_{2^l}$ to $\mathcal{P}_1$. Note that the real message $\langle k\rangle^{\text{A}}_1$ output by $\binom{4}{1}\text{-}\mathsf{OT}_l$ is also uniformly distributed, because $\langle k\rangle^{\text{A}}_1$ is masked by $\langle k\rangle^{\text{A}}_0$ which is uniformly sampled by $\mathcal{P}_0$. Hence, $\mathcal{P}_1$ cannot distinguish r_1 from the real message $\langle k\rangle^{\text{A}}_1$ output by $\binom{4}{1}\text{-}\mathsf{OT}_l$.
- When $\mathcal{F}^{l}_{\text{B2A}}$ receiving $\langle \mathsf{tmp}\rangle^{\text{B}}_1$ from $\mathcal{P}_1$, $\mathcal{S}_1$ returns $r_2 = \langle y\rangle^{\text{A}}_1 - (\langle x\rangle^{\text{A}}_1 \gg s) - r_1$, which conforms to $\mathcal{P}_1$'s view in a real execution. As r_1 is uniformly distributed, r_2 is uniformly distributed. $\mathcal{P}_1$ cannot distinguish r_2 from the real arithmetic share output by $\mathcal{F}^{l}_{\text{B2A}}$.

Thus, we conclude that the view of each party $\mathcal{P}_b, b \in \{0,1\}$ can be perfectly simulated.

- Communication complexity. $\Pi^{l,s}_{\text{Trunc}}$ involves a single call each to $\mathcal{F}^{l,s+1}_{\text{2Bit-Extr}}$, $\binom{4}{1}\text{-}\mathsf{OT}_l$ and $\mathcal{F}^{l}_{\text{B2A}}$. Using the optimized two-bit extraction construction in section 3.3 and setting parameter $m = 4$, the communication bits of our construction are approximately $(2\lambda + 32)(l' + t) + (2\lambda + 22)(l' + k - 1) - (\lambda + 2)\lceil \log l' \rceil + 5l + 3\lambda$, where $l' = \lceil \frac{l-1}{4} \rceil$, $s' = \lceil \frac{s-1}{4} \rceil$, $t \in [0, 1]$ and $k \in [0, \lceil \log s' \rceil - 1]$ are parameters depending on s. For $l = 32$ and $s = 16$, using the parameter $m = 4$ recommended by the authors of Cryptflow2 [21], the concrete communication of our construction is 4384 bits as opposed to 6093 bits for Cryptflow2.

5 Experiments

Table 1 presents a comparison of the theoretical communication complexity for truncation protocols, while this section provides an empirical evaluation of their practical performance.

Benchmarks. We compared our protocol with Cryptflow2 [21], which is currently considered the state-of-the-art OT-based faithful truncation protocol under the 2PC setting. To ensure a fair comparison, we used the recommended protocol parameter $m = 4$, as suggested by the authors of Cryptflow2 (Section 6.1 in [21]), for both our optimized protocol (Section 3.3) and Cryptflow2. We evaluated two bit length $l = 32$ and $l = 64$. For each bit length, we varied the fixed-point precision s in the range of $\{8, 10, 12, 14, 16\}$. These values

of l and s are representative in the field of privacy-preserving machine learning (PPML) [7]. For each combination of l and s, a batch of 2^{20} truncations was evaluated. This number of truncations is typically required in PPML, e.g., one epoch of training the textbook LeNet model on the MNIST dataset [15] roughly requires one million truncations.

Hardwares and softwares. We simulated the two parties with two virtual machines having 2.90 GHz Intel Core i5-9400 processors with 6 CPUs and 8 GBs of RAM. The simulated bandwidth between the two machines was 100 Mbps and the echo latency was 40 ms. Our implementation was built upon the SCI library [18] which implements Cryptflow2 [21]. SCI [18] is written in C++ and makes use of the EMP toolkit [22] to generate the application-level OT types like $\binom{2^m}{1}$-OT. The code was compiled by gcc 9.4.0 on Ubuntu 20.04.

Result analysis. The experiment results are presented in Table 2. It is noteworthy that our protocol consistently communicated fewer bits and ran faster than Cryptflow2 for all values of l and s being evaluated. Our improvement is due to eliminating redundancies in the faithful truncation construction of Cryptflow2: while Cryptflow2 uses two related comparison instances to detect the small error and big error respectively, we use only one two-bit extraction instance.

Our improvement is mainly dominated by the fixed-point precision s. The larger value of s, the more communication bits and running time our protocol can save compared to Cryptflow2. This is because Cryptflow2's communication complexity grows linearly with s, while ours is logarithmic. When $s = 16$, we observe the most significant improvement, as we save roughly 1.5 Gbit communication bits compared to Cryptflow2. This result is as expected, because our two-bit extraction protocol can output the intermediate group generate signal $G_{16:1}$ of the MSB extraction directly as the corresponding carry-out bit for the s-th bit extraction without requiring $\mathsf{combine}$ operations. Namely, in this case, to additionally eliminate the small error, our faithful truncation protocol involves only a single call to the boolean to arithmetic share conversion $\mathcal{F}^{l}_{\text{B2A}}$ which communicates $\lambda + l$ bits.

It is worth noting that when l is large, the efficiency bottleneck of both our protocol and Cryptflow2 is eliminating the big error. For example, when s is fixed, increasing l from 32 to 64 doubles the running time of both protocols. Additionally, when $l = 32$ with $s = 16$ fixed, our protocol saves 28% communication bits compared to Cryptflow2. However, when $l = 64$, our protocol only saves 16% communication bits. These findings suggest that the elimination of the big error incurs significant costs when l is large. To enhance the overall efficiency of faithful truncation, it would be beneficial to investigate techniques that can reduce the costs of eliminating the big error in future research.

Table 2. Empirically comparing our faithful truncation protocol with Crypflow2 [21]

Bit Length	**Precision**	**Crypflow2** [21]		**This work**	
l	s	Time (s)	Comm. (Gbit)	Time (s)	Comm. (Gbit)
32	8	77.50	4.21	68.67	3.63
	10	82.00	4.60	75.03	4.01
	12	82.78	4.63	71.88	3.80
	14	87.64	5.06	77.52	4.23
	16	88.37	5.09	69.00	3.63
64	8	160.11	8.75	145.21	8.11
	10	164.08	9.13	153.25	8.49
	12	165.01	9.16	146.10	8.26
	14	178.32	9.67	160.20	8.76
	16	180.04	9.70	145.37	8.11

† Results were reported for a batch of 2^{20} truncations. The network had 100 Mbps and its echo latency was 40ms.

6 Conclusions

In this work, we investigate efficient constructions for faithful truncation, a crucial operation in fixed-point arithmetic. We extend previous studies of oblivious transfer based constructions [21,12] by proposing a building functionality two-bit extraction customized for faithful truncation. Our faithful truncation protocol capitalizes on the efficient constructions for two-bit extraction, resulting in a reduction of the communication complexity of [21] from linear in s to logarithmic in s, where s is the fixed-point precision. This work highlights the possibility of removing the small error at a negligible cost by reusing the intermediate results from eliminating the big error. In the future work, we would like to investigate techniques that can further reduce the costs of eliminating the big error.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by the National Natural Science Foundation of China under Grant Nos, 62172411, 62172404, 61972094, 62202458.

References

1. Abbas, K.: Handbook of Digital CMOS Technology, Circuits, and Systems (2020)
2. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: CCS'13. https://doi.org/10.1145/2508859.2516738
3. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO '91. https://doi.org/10.1007/3-540-46766-1_34
4. Boyle, E., Chandran, N., Gilboa, N., Gupta, D., Ishai, Y., Kumar, N., Rathee, M.: Function secret sharing for mixed-mode and fixed-point secure computation. In: EUROCRYPT'21. https://doi.org/10.1007/978-3-030-77886-6_30
5. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: EUROCRYPT'15. https://doi.org/10.1007/978-3-662-46803-6_12
6. Couteau, G.: New protocols for secure equality test and comparison. In: ACNS'18. https://doi.org/10.1007/978-3-319-93387-0_16
7. Dalskov, A.P.K., Escudero, D., Keller, M.: Fantastic four: Honest-majority four-party secure computation with malicious security. In: USENIX Security'21. https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov
8. Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: NDSS'15. https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation
9. Dessouky, G., Koushanfar, F., Sadeghi, A., Schneider, T., Zeitouni, S., Zohner, M.: Pushing the communication barrier in secure computation using lookup tables. In: NDSS'17. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/pushing-communication-barrier-secure-computation-using-lookup-tables/
10. Gupta, K., Kumaraswamy, D., Chandran, N., Gupta, D.: LLAMA: A low latency math library for secure inference. In: PoPETs'22 https://doi.org/10.56553/popets-2022-0109
11. Hazay, C., Lindell, Y.: Efficient secure two-party protocols: Techniques and constructions (2010)
12. Huang, Z., Lu, W., Hong, C., Ding, J.: Cheetah: Lean and fast secure two-party deep neural network inference. In: USENIX Security'22. https://www.usenix.org/conference/usenixsecurity22/presentation/huang-zhicong
13. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: CRYPTO'03. https://doi.org/10.1007/978-3-540-45146-4_9

14. Kolesnikov, V., Kumaresan, R.: Improved OT extension for transferring short secrets. In: CRYPTO'13. https://doi.org/10.1007/978-3-642-40084-1_4
15. LeCun, Y., Cortes, C.: The mnist database of handwritten digits (2005)
16. Mohassel, P., Rindal, P.: Aby^3: A mixed protocol framework for machine learning. In: CCS'2018. https://doi.org/10.1145/3243734.3243760
17. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: SP'17. https://doi.org/10.1109/SP.2017.12
18. mpc–msri/EzPC: Secure and Correct Inference (SCI) Library. https://github.com/mpc-msri/EzPC/tree/master/SCI (2016)
19. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: improved mixed-protocol secure two-party computation. In: USENIX Security'21. https://www.usenix.org/conference/usenixsecurity21/presentation/patra
20. Rathee, D., Bhattacharya, A., Sharma, R., Gupta, D., Chandran, N., Rastogi, A.: Secfloat: Accurate floating-point meets secure 2-party computation. In: SP'22. https://doi.org/10.1109/SP46214.2022.9833697
21. Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: Cryptflow2: Practical 2-party secure inference. In: CCS '20. https://doi.org/10.1145/3372297.3417274
22. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit (2016)
23. Yang, K., Weng, C., Lan, X., Zhang, J., Wang, X.: Ferret: Fast extension for correlated OT with small communication. In: CCS '20. https://doi.org/10.1145/3372297.3417276