TR-IIS-09-005

Protocols for Secure Multi-party Computation: Design, Implementation and Performance Evaluation

I-Cheng Wang, Kung Chen, Tsan-sheng Hsu,
Churn-Jung Liau, Chih-Hao Shen, and Da-Wei Wang

December 1, 2009 || Technical Report No. TR-IIS-09-005
http://www.iis.sinica.edu.tw/page/library/LIB/TechReport/tr2009/tr09.html

Protocols for Secure Multi-party Computation: Design, Implementation and Performance Evaluation

I-Cheng Wang*, Kung Chen‡, Tsan-sheng Hsu*,
Churn-Jung Liau*, Chih-Hao Shen†, and Da-Wei Wang*

Institute of Information Science,
Academia Sinica, Taiwan
Email: {icw, tshsu, liaucj, wdw}@iis.sinica.edu.tw*

Department of Computer Science,
National Chengchi University, Taiwan
Email: chenk@cs.nccu.edu.tw‡

Department of Computer Science,
University of Virginia, USA
Email: shench@email.virginia.edu†

Abstract

Protocols for secure multi-party computation allow participants to share a computation while each party learns only what can be inferred from their own inputs and the output of the computation. However, the execution time of a secure protocol may be too high so that it is not practical unless some tradeoffs being made between data access and confidentiality. In this technical report, we propose a set of information theoretically secure protocols based on scalar product protocol and aim to provide some empirical basis for making such tradeoffs in computing exponentiation. A detailed performance evaluation was carried out by taking advantage of the compositional nature of our protocols. We come up with a time function which provides good prediction of the execution time of the proposed exponentiation protocols based on the execution time of scalar products. Using the time function, we obtain several interesting tradeoffs between execution time and privacy. In particular, compromising some private information enables a reduction in the execution time from years, if not centuries, to days or even minutes. Based on our results, we argue that there are indeed reasonable tradeoffs between privacy and execution time. Furthermore, our study indicates that a system intelligently offering users possible tradeoff options will make secure multi-party computation a more attractive approach to enhance privacy in practice.

Contents

List of Figures

List of Tables

1 Introduction

Secure multiparty computation(SMC) is a research topic aiming at the double-edged privacy problem: How can several potentially distrustful parties take advantage of their private data without revealing their privacy? After Yao's [1] general solution to two-party secure computation was proposed, Goldreich et al. [2] soon gave another general solution to multiparty computation. Both proposals are so elegant that they provide secure two-party/multi-party protocols for binary AND and XOR gates, which can be further generalized to all computable functions. However, despite their academic significance, both solutions have computation costs too prohibitive to be feasible in real applications.

A function f is *complete* if a secure protocol for f implies the existence of secure protocols for all computable functions. Yao and Goldreich et al. propose the idea to solve secure two-party/multi-party computation by giving secure protocols for complete functions. Extended from the idea, the scalar product has gathered more and more attention because of its completeness and integer-based computing power. However, there is no systematic approach to construct all computable functions from scalar products.

In reality, privacy cannot be absolute. Other values, including security and social welfare, compete with it. Therefore, compromise has to be made. In the report [3], the data access and confidentiality tradeoff is well articulated. The value of data-intensive research is highly variable, and it is impossible to specify a universally applicable optimal tradeoff between privacy and data access. The report calls for the development of tools that, on a case-by-case basis, would increase data access without compromising data protection or conversely, increase confidentiality without compromising data access. Our goal is to develop such tools via protocols for SMC.

Loosely speaking, SMC involves computing functions with inputs from two or more parties in a distributed network while ensuring that no additional information, other than what can be inferred from each participant's input and output, is revealed to parties not privy to that information. In this paper, we present a protocol that computes the exponentiation–one of the most important functions in mathematics–in a two-party setting. In the development of this protocol, we used a compositional approach with the scalar product protocols serving as building blocks.

Based on the performance data of our protocol, we argue that the tradeoff between confidentiality and execution time is a real issue by demonstrating various tradeoff points of the exponentiation. The performance data shows that it might either take a very long time or not even be possible to keep both the base and the exponent values absolutely secret and complete the computation. Revealing some information about the base and/or the exponent allows the computation to be completed in radically different time bounds. Our protocol was carefully implemented, and furthermore, a thorough performance evaluation was carried out to ensure the accuracy of these results.

The paper is organized as follows. We give a short review of related works in Section 2. In Section 3, notations and the scalar-product based specifications are described in detail. Next, Section **??** lists several scenarios of SMC in which the exponentiation had concrete tradeoffs between privacy and efficiency. Besides that, the experimental results and how to estimate the time cost for each scenario or each of our protocol are shown here as well. Finally, we conclude this paper and lay out the future work in Section 8.

2 Related Works

Since Yao's [1] and Goldreich et al.'s [2] proposals, researchers have been looking for new complete functions and have been looking at the foundations of completeness. Kilian shows that the oblivious transfer is complete [4], and so are the functions with imbedded OR [5]. Furthermore, it is claimed that the integer-based scalar products are more practical to real applications than binary-based oblivious transfer [6].

Over the past decade, more and more proposals for the secure scalar products seem to be released each year. Du and Zhan [7] proposed the invertible-matrix and the commodity-based approaches. The former approach enabled the tradeoff between efficiency and privacy, and the latter was based on Beaver's commodity model [8]. Goethals et al. [9] proposed the computationally secure scalar-product protocol, the security of which depended on the intractability of the composite residuosity problem. Our previous work [10] demonstrates the potential of scalar-product-based protocols, among which the commodity-based approach has extraordinary performance, though a neutral third party is needed when it comes to two-party computation.

Moreover, much effort has gone into building various applications using secure scalar products. Atallah and Du reduced geometry problems to scalar products [11]. Du et al. constructed secure protocols for statistical analysis [12] and scientific computations [13]. In addition, Bunn and Ostrovsky [14] offered a secure k-means clustering protocol based on scalar products using the composite-residuosity approach. Zhan et al. [15] have recently constructed an efficient privacy-preserving collaborative recommender system based on the scalar product protocol using Beaver's commodity model.

There are also plenty of theoretical studies on scalar products. Chiang et al. [16] proposed a privacy measurement based on information theory, with which they analyzed various scalar-product approaches. They proved that the invertible-matrix approach discloses at least half the information whereas the commodity-based approach is perfectly secure. Wang et al. [17] proved that no information-theoretically secure two-party protocol exist for scalar products. Moreover, the closure property of the commodity-based approach is preliminarily verified according to the security definition based on information theory [18].

Regarding the secure computation of exponentiation and the discussion of tradeoffs, Algesheimer et al. [19] presented a protocol for exponentiation with a shared exponent modulo a shared secret. Damgård et al. [20] gave more efficient constant round protocols for securely computing the exponentiation with respect to public/shared exponents and moduli. Recently, Nielsen and Schwartzbach [21] have given tradeoff examples of SMC problems and shown the timing results.

3 Preliminaries

In this section, we introduce the notations used hereafter and specifications of the building blocks.

For a secure two-party problem, the two parties hold private inputs X_1, X_2 respectively. After the execution of some protocol, they hold private outputs Y_1, Y_2. The subscript of a variable denotes the party who owns the variable. There might be a list of public variables, $plist$, and we use $(X_1, X_2)\{plist\} \mapsto (Y_1, Y_2)$ to denote it. The domain $\mathbb{Z}_n$ denotes a ring consisting of elements $\{0, \ldots, n-1\}$, and the results of addition and multiplication in $\mathbb{Z}_n$ are the modular summation and the modular product. If not stated otherwise, the computations of our proposals are over $\mathbb{Z}_n$, where n is two's power, namely, $n = 2^{k+1}, k \in \mathbb{N}$. Moreover, to extend the domain from natural number to integer, elements $\{1, \ldots, \lfloor \frac{n-1}{2} \rfloor\}$ remain positive numbers, while elements $\{n-1, \ldots, n - \lfloor \frac{n-1}{2} \rfloor\}$ are interpreted as negative integers analogous to the binary system in modern computers. As a result, the subtraction to p is equivalent to the addition to $(n-p)$.

$$x - p \pmod n = x + n - p \pmod n.$$

In this paper, there are two different concepts of the "scalar product." When it comes to lower case letters, it means all secure scalar product approaches; when it comes to capitalized words (Scalar-Product), it means one of these secure scalar product approaches. The formulation for the Scalar-Product protocol follows Goldreich's principle [22], namely that the intermediate results during protocol execution are always shared among participants. In a protocol π composed of Scalar-Products, current outputs can be inputs to the next Scalar-Product, which are actually the intermediate results of π and should be shared. Moreover, the intermediate results are shared by addition rather than multiplication. In ring $\mathbb{Z}_n$, the multiplicative sharing reveals information when either of the values of the shares is zero, while the additive sharing has been proven to be perfect [18]. The secure Scalar-Product protocol is specified as

Specification 3.1 (Scalar-Product) *Party 1 and Party 2 want to collaboratively execute the secure protocol*

$$((x[1]_1, \ldots, x[d]_1), (x[1]_2, \ldots, x[d]_2)) \mapsto (y_1, y_2)$$

such that $y_1 + y_2 = x[1]_1 \cdot x[1]_2 + \cdots + x[d]_1 \cdot x[d]_2$ *and* $x[i]_1, x[i]_2, y_1, y_2 \in \mathbb{Z}_n$*, for* $i = 1, \ldots, d$.

Here we merely specify Scalar-Product instead of providing a concrete approach because we focus on building more protocols on top of Scalar-Product. Similar to the software specification, as long as a new subroutine matches the interface, it can replace the old one and work perfectly. In our scalar-product based protocols, as long as a new solution matches the specification 4.16, it can be used as the building block of our proposed protocols. A scalar-product based composition theory is proved for semi-honest adversary models [23], so our protocols preserve the entropy of the secret inputs as strong as the underlying scalar product protocol preserves the entropy of its inputs.

In passing, it should be noted that we do not deal with the problem of combining the shared output variables to produce the final results, for we are designing building blocks which can be used to build even larger protocols. The step of combining shared variables to produce the final results does not come until after computation is completed. It is unnecessary to combine intermediate variables.

As far as we know, there is no systematic approach to construct all computable functions directly from scalar products. Hence we specified some more building blocks to facilitate solving more SMC problems. In the remainder of this section, we summarize the specifications of those building blocks that we will employ to develop protocols for exponentiation. The readers are referred to [24] for the details of those building blocks.

4 Building Blocks

We separate this section into four subsections proposing primitive, useful building blocks, building blocks for fixed point numbers, and general solutions.

4.1 Primitive building blocks

Four primitive building blocks are specified, which are $\mathbb{Z}_n$-to-$\mathbb{Z}_2$, $\mathbb{Z}_2$-to-$\mathbb{Z}_n$, Product, and Square.

4.1.1 $\mathbb{Z}_n$-to-$\mathbb{Z}_2$ and $\mathbb{Z}_2$-to-$\mathbb{Z}_n$

$\mathbb{Z}_n$-to-$\mathbb{Z}_2$ and $\mathbb{Z}_2$-to-$\mathbb{Z}_n$ convert to and fro between $\mathbb{Z}_n$ sharing and bitwise $\mathbb{Z}_2$ sharing. In addition to be primitive building blocks, these two protocols establish the possibility of secure computation for all functions. Albeit the inefficiency, we can always apply Yao's circuit evaluation idea after the $\mathbb{Z}_n$-to-$\mathbb{Z}_2$ protocol and followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_n$ protocol. These two protocols make our proposal as general as the classic circuit approaches.

Specification 4.1 **($\mathbb{Z}_n$-to-$\mathbb{Z}_2$ $\{k'\}$)** *Party 1 and Party 2 share a number in $\mathbb{Z}_n$, and they want to securely convert the $\mathbb{Z}_n$ sharing into bitwise $\mathbb{Z}_2$ sharing. More specifically, Party 1 and Party 2 want to collaboratively execute the secure protocol $(x_1, x_2)\{k'\} \mapsto ((y_1^0, \ldots, y_1^{k'}), (y_2^0, \ldots, y_2^{k'}))$ such that $(y^{k'} y^{k'-1} \cdots y^1 y^0)_2 = x_1 + x_2$, where $n = 2^{k+1}$, $k' \leq k$, $x_1, x_2 \in \mathbb{Z}_n$, $y_1^i, y_2^i \in \mathbb{Z}_2$, and $y^i = y_1^i + y_2^i \pmod 2)$, for $i = 0, 1, \cdots, k'$.*

To convert from $\mathbb{Z}_n$ sharing to bitwise $\mathbb{Z}_2$ sharing, we emulate the carry ripple adder with binary Scalar-Product protocol, whose $n = 2$. Let $x_1 = (x_1^{k'} \cdots x_1^0)_2$, $x_2 = (x_2^{k'} \cdots x_2^0)_2$, and the adder operates as long addition:

$$\begin{array}{cccccc} & c^{k'+1} & c^{k'} & \cdots & c^1 & c^0 \\ & & x_1^{k'} & \cdots & x_1^1 & x_1^0 \\ +) & & x_2^{k'} & \cdots & x_2^1 & x_2^0 \\ \hline & & y^{k'} & \cdots & y^1 & y^0 \end{array}$$

where $c^0 = 0$ and $c^{i+1} = c^i x_1^i + c^i x_2^i + x_1^i x_2^i \pmod 2$ are the carry bits; $y^i = c^i + x_1^i + x_2^i \pmod 2$ is the i-th summation bit. Next, the $\mathbb{Z}_n$-to-$\mathbb{Z}_2$ $\{k'\}$ protocol is as follows:

PROTOCOL $\mathbb{Z}_n$-to-$\mathbb{Z}_2$ $\{k'\}$ $(n = 2^{k+1}, k' \leq k)$

1. Party j sets $c_j^0 = 0$, and $y_j^0 = x_j^0$, for $j = 1, 2$.
2. For $i = 0, \ldots, k' - 1$, repeat Step 2a to Step 2b.
   (a) The two parties jointly execute the binary Scalar-Product protocol $((c_1^i, x_1^i, x_1^i), (x_2^i, c_2^i, x_2^i)) \mapsto (t_1^i, t_2^i)$, where $t_1^i + t_2^i \pmod 2 = c_1^i x_2^i + x_1^i c_2^i + x_1^i x_2^i \pmod 2$.
   (b) For $j = 1, 2$, Party j computes
   $$c_j^{i+1} = c_j^i x_j^i + t_j^i \pmod 2,$$
   $$y_j^{i+1} = x^{j\,i+1} + c_j^{i+1} \pmod 2.$$

Specification 4.2 **($\mathbb{Z}_2$-to-$\mathbb{Z}_n$ $\{k'\}$)** *Party 1 and Party 2 bitwise, additively share a number in $\mathbb{Z}_2$, and they want to securely convert the bitwise $\mathbb{Z}_2$ sharing into the $\mathbb{Z}_n$ sharing. More specifically, the two parties want to execute the secure protocol $((x_1^0, \ldots, x_1^{k'}), (x_2^0, \ldots, x_2^{k'}))\{k'\} \mapsto (y_1, y_2)$ such that $y_1 + y_2 = (x^{k'} x^{k'-1} \cdots x^1 x^0)_2$, where $n = 2^{k+1}$, $k' \leq k$, $y_1, y_2 \in \mathbb{Z}_n$, $x_1^i, x_2^i \in \mathbb{Z}_2$, and $x^i = x_1^i + x_2^i \pmod 2)$, for $i = 0, 1, \cdots, k'$.*

According to the above requirement, the outputs can be rewritten as the following function:

$$\begin{aligned} y_1 + y_2 = &\textstyle\sum_{i=0}^{k'} x^i \cdot 2^i = \sum_{i=0}^{k'} (x_1^i + x_2^i \bmod 2) \cdot 2^i \\ = &\textstyle\sum_{i=0}^{k'} (x_1^i + x_2^i - 2x_1^i x_2^i) \cdot 2^i \\ = &\textstyle\sum_{i=0}^{k'} x_1^i \cdot 2^i + \sum_{i=0}^{k'} x_2^i \cdot 2^i - \sum_{i=0}^{k'} x_1^i x_2^i \cdot 2^{i+1} \end{aligned}$$

In the above function, we divide the computation into two parts. One is locally computable ($\sum x_1^i \cdot 2^i$ and $\sum x_2^i \cdot 2^i$) while the other needs the Scalar-Product protocol ($\sum x_1^i x_2^i \cdot 2^{i+1}$).

PROTOCOL $\mathbb{Z}_2$-to-$\mathbb{Z}_n$ $\{k'\}$ $(n = 2^{k+1}, k' \leq k)$

1. Party 1 and Party 2 jointly run the Scalar-Product protocol $((2x_1^0, \ldots, 2^{k'+1} x_1^{k'}), (x_2^0, \ldots, x_2^{k'})) \mapsto (t_1, t_2)$ such that $t_1 + t_2 = 2x_1^0 \cdot x_2^0 + \cdots + 2^{k'+1} x_1^{k'} \cdot x_2^{k'}$.
2. Party j computes $y_j = \sum_{i=0}^{k'} x_j^i \cdot 2^{k'} - t_j$, for $j = 1, 2$.

4.1.2 Product

To compute a polynomial function collaboratively, we need to be able to perform two-party addition and multiplication. Since we adopt the principle of the additive sharing, the two-party addition is trivial. However, to execute a secure two-party multiplication, it is necessary to use the Scalar-Product protocol.

Specification 4.3 (Product) *Party 1 and Party 2 additively share the multiplicand and the multiplicator. They want to securely execute the protocol $((x_1, y_1), (x_2, y_2)) \mapsto (z_1, z_2)$ such that $z_1 + z_2 = (x_1 + x_2)(y_1 + y_2)$.*

With a little modification, the outputs can be rewritten as

$$z_1 + z_2 = x_1 y_1 + x_2 y_2 + (x_1 y_2 + y_1 x_2).$$

After the above factoring, it is obvious that $x_1 y_1$ and $x_2 y_2$ are individually computable for Party 1 and Party 2 respectively. However, the other terms should be computed via the Scalar-Product protocol. The following protocol describes the details.

PROTOCOL Product

1. Party 1 and Party 2 jointly execute the Scalar-Product protocol $((x_1, y_1), (y_2, x_2)) \mapsto (t_1, t_2)$ such that $t_1 + t_2 = x_1 y_2 + y_1 x_2$.
2. Party j individually computes $z_j = t_j + x_j y_j$, for $j = 1, 2$.

A polynomial is a function constructed from variables and constants using the operations of addition, subtraction, and multiplication. With the additive sharing, we can easily add and subtract; with the Product protocol, we can multiply as well. Therefore, secure two-party computation on any polynomial evaluation can be composed of the Product protocol and the help of additive sharing.

4.1.3 Square

The Square protocol, which is very similar to the Product protocol and employs the same strategy, namely dividing the computation into the part which can be done individually and the rest which must be performed collaboratively. However, the Square protocol reduces the dimension of the scalar product from two to one.

Specification 4.4 (Square) *Party 1 and Party 2 share a number in $\mathbb{Z}_n$, and they want to collaboratively execute the secure protocol $(x_1, x_2) \mapsto (y_1, y_2)$ such that $y_1 + y_2 = (x_1 + x_2)^2$.*

The protocol details are as follows:

PROTOCOL Square

1. Party 1 and Party 2 jointly execute the Scalar-Product protocol $(x_1, x_2) \mapsto (t_1, t_2)$ such that $t_1 + t_2 \pmod n = x_1 \cdot x_2$.
2. Party j individually computes $y_j = x_j^2 + 2t_j$, for $j = 1, 2$.

4.2 Useful building blocks

Based on primitive building blocks introduced in Section 4.1, several useful protocols are proposed here, including Comparison, Zero, If-Then-Else, Shift, Rotation, Logarithm, Division/Remainder, and Exponentiation.

4.2.1 Comparison

There are many variations of binary comparison: less than ($<$), greater than ($>$), less than or equal to ($\leq$), grater than or equal to ($\geq$), and equal to ($=$). However, all of them are reducible to the less than operator ($<$). In order to compare x and y, it is intuitive to compare $(x-y)$ and 0 since we share the intermediate results additively; at the same time, it is effortless to subtract under additive sharing. Our proposal to compare x and y is to compute the most significant bit of $(x - y)$. According to the binary system on modern computers, if the most significant bit of $(x-y)$ is 1, $(x-y)$ is a negative number inferring that x is less than y.

Specification 4.5 (Comparison) *Party 1 and Party 2 share a number in $\mathbb{Z}_n$, and they want to know the sign of the number. In other words, they want to collaboratively execute the secure protocol $(x_1, x_2) \mapsto (y_1, y_2)$ such that*

$$y_1 + y_2 = \begin{cases} 1 & \textit{if } x_1 + x_2 < 0, \\ 0 & \textit{otherwise.} \end{cases}$$

Recall that we compute the comparison by checking whether the shared number is negative, i.e., whether the most significant bit of the shared number is 1. The protocol details are as follows:

PROTOCOL Comparison

1. Two parties collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol $(x_1, x_2) \mapsto ((b_1^0, \ldots, b_1^k), (b_2^0, \ldots, b_2^k))$, such that $b^i = b_1^i + b_2^i \pmod 2$, and $(b^k \cdots b^0)_2 = x_1 + x_2$.
2. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^k, b_2^k) \mapsto (y_1, y_2)$, such that $y_1 + y_2 = (b^k)_2$ and $b^k = b_1^k + b_2^k \pmod 2$.

4.2.2 Zero

To test if two additively shared numbers are equal to each other is equivalent to test if the difference of these two numbers is zero, and the Zero protocol does the job. More specifically, the Zero protocol examines whether a shared number is zero or not. The straightforward idea is to test if it is neither less nor greater than zero. As mentioned, this is one of the variations of binary comparison.

Specification 4.6 (Zero) *Party 1 and Party 2 share a number in $\mathbb{Z}_n$, and they want to know if the number is equal to zero. In other words, they want to collaboratively execute the secure protocol $(x_1, x_2) \mapsto (y_1, y_2)$ such that*

$$y_1 + y_2 = \begin{cases} 1 & \textit{if } x_1 + x_2 = 0, \\ 0 & \textit{otherwise.} \end{cases}$$

The protocol details are as follows:

PROTOCOL Zero

1. Two parties collaboratively execute the Comparison protocol $(x_1, x_2) \mapsto (t_1, t_2)$, such that
$$t_1 + t_2 = \begin{cases} 1 & \text{if } x_1 + x_2 < 0, \\ 0 & \text{otherwise.} \end{cases}$$
2. Party 1 and Party 2 collaboratively execute the Comparison protocol $(-x_1, -x_2) \mapsto (s_1, s_2)$, such that
$$s_1 + s_2 = \begin{cases} 1 & \text{if } -(x_1 + x_2) < 0, \\ 0 & \text{otherwise.} \end{cases}$$
3. Party 1 and Party 2 collaboratively execute the Product protocol $((1 - t_1, 1 - s_1), (-t_2, -s_2)) \mapsto (y_1, y_2)$ such that $y_1 + y_2 = (1 - t_1 - t_2)(1 - s_1 - s_2)$.

4.2.3 If-Then-Else

The If-Then-Else protocol is useful for functions with alternatives.

Specification 4.7 (If-Then-Else) *Party 1 and Party 2 additively share the predicate, IF-clause value, and the ELSE-clause value. They want to securely execute the protocol $((b_1, x_1, y_1), (b_2, x_2, y_2)) \mapsto (z_1, z_2)$ such that*

$$z_1 + z_2 = \begin{cases} x_1 + x_2 & \textit{if} \quad b_1 + b_2 = 1, \\ y_1 + y_2 & \textit{if} \quad b_1 + b_2 = 0. \end{cases}$$

According to the above requirement, the outputs can be rewritten as the following function:

$$\begin{aligned} z_1 + z_2 =& (b_1 + b_2)(x_1 + x_2) + (1 - b_1 - b_2)(y_1 + y_2) \\ =& (y_1 + y_2) + (b_1 + b_2)(x_1 - y_1 + x_2 - y_2) \end{aligned}$$

Again, with the strategy dividing the components into individually computable ones and those need the Scalar-Product protocols, we propose the following If-Then-Else protocol.

PROTOCOL If-Then-Else

1. Party j individually computes $s_j = x_j - y_j$, for $j = 1, 2$.
2. Party 1 and Party 2 collaboratively execute a Product protocol $((b_1, s_1), (b_2, s_2)) \mapsto (t_1, t_2)$ such that $t_1 + t_2 = (b_1 + b_2)(s_1 + s_2)$.
3. Party j individually computes $z_j = t_j + y_j$, for $j = 1, 2$.

4.2.4 Shift

Shift-Left, Shift-Right, and Shift$\{s\}$ protocols are designed. Note that the input and the output of the Shift-Left and Shift-Right protocols are in $\mathbb{Z}_2$, whereas the input and the output of the Shift$\{s\}$ protocol are in $\mathbb{Z}_n$ shares.

Specification 4.8 (Shift-Left) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x_1^0, x_1^1, \ldots, x_1^k, s_1), (x_2^0, x_2^1, \ldots, x_2^k, s_2)) \mapsto ((y_1^0, y_1^1, \ldots, y_1^k), (y_2^0, y_2^1, \ldots, y_2^k))$ *such that*

$$y^i = \begin{cases} x^{i-s} & \textit{if} \quad i - s \geq 0, \\ 0 & \textit{otherwise,} \end{cases}$$

where $s = s_1 + s_2 \ (\textit{mod}\ n), x^i = x_1^i + x_2^i \ (\textit{mod}\ 2)$, *and* $y^i = y_1^i, y_2^i \ (\textit{mod}\ 2)$, *for* $i = 0, 1, \ldots, k$.

Let $b^i \in \mathbb{Z}_2$ for $i = 0, 1, \cdots, k$, and the design idea of the Shift-Left protocol is as Figure 1 shown.

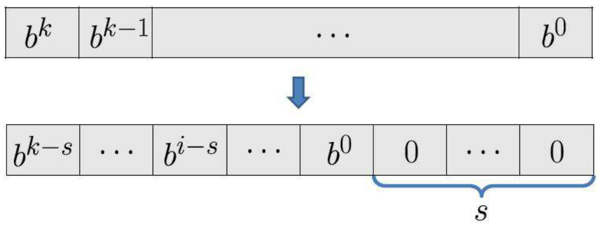


Figure 1. Shift-Left

PROTOCOL Shift-Left

1. Party 1 and Party 2 jointly run the Exp(pb, se, n_y) $\in \mathbb{Z}_n$ protocol $(s_1, s_2)\{2, k+1\} \mapsto (u_1, u_2)$ such that $u_1 + u_2 = 2^{s_1 + s_2}$.
2. Party 1 and Party 2 collaboratively run the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{k\}$ protocol $((x_1^0, \ldots, x_1^k), (x_2^0, \ldots, x_2^k))\{k\} \mapsto (t_1, t_2)$ such that $t_1 + t_2 = (x^k x^{k-1} \cdots x^1 x^0)_2$.
3. These two parties jointly run the Product protocol $((t_1, u_1), (t_2, u_2)) \mapsto (v_1, v_2)$ such that $v_1 + v_2 = (t_1 + t_2)(u_1 + u_2)$.
4. Party 1 and Party 2 jointly execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol $(v_1, v_2)\{k\} \mapsto ((y_1^0, \ldots, y_1^k), (y_2^0, \ldots, y_2^k))$, such that $(y^k \cdots y^0)_2 = v_1 + v_2$, and $y^i = y_1^i + y_2^i \ (\text{mod}\ 2)$, for $i = 0, 1, \ldots, k$.

Specification 4.9 (Shift-Right) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x_1^0, x_1^1, \ldots, x_1^k, s_1), (x_2^0, x_2^1, \ldots, x_2^k, s_2)) \mapsto ((y_1^0, y_1^1, \ldots, y_1^k), (y_2^0, y_2^1, \ldots, y_2^k))$ *such that*

$$y^i = \begin{cases} x^{i+s} & \textit{if} \quad i + s \leq k, \\ 0 & \textit{otherwise,} \end{cases}$$

where $s = s_1 + s_2 \ (\textit{mod}\ n), x^i = x_1^i + x_2^i \ (\textit{mod}\ 2)$, *and* $y^i = y_1^i, y_2^i \ (\textit{mod}\ 2)$, *for* $i = 0, 1, \ldots, k$.

Let $b^i \in \mathbb{Z}_2$ for $i = 0, 1, \cdots, k$, and the design idea of the Shift-Right protocol is as Figure 2 shown.

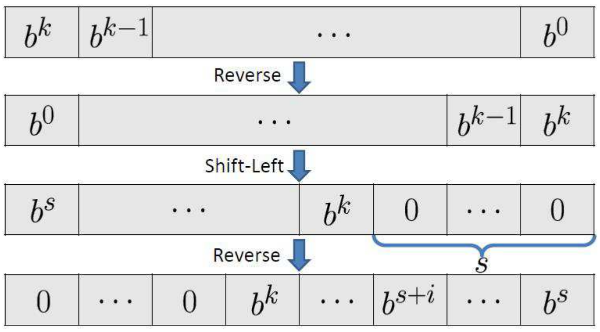


Figure 2. Shift-Right

PROTOCOL Shift-Right

1. Party j individually sets $[d_j^0, d_j^1, \ldots, d_j^{k-1}, d_j^k] = [x_j^k, x_j^{k-1}, \ldots, x_j^1, x_j^0]$, for $j = 1, 2$.
2. Party 1 and Party 2 jointly run the Shift-Left protocol $((d_1^0, d_1^1, \ldots, d_1^k, s_1), (d_2^0, d_2^1, \ldots, d_2^k, s_2)) \mapsto ((f_1^0, f_1^1, \ldots, f_1^k), (f_2^0, f_2^1, \ldots, f_2^k))$ such that

$$f^i = \begin{cases} d^{i-s} & \text{if} \quad i - s \geq 0, \\ 0 & \text{otherwise,} \end{cases}$$

where $s = s_1 + s_2 \ (\text{mod}\ n), d^i = d_1^i + d_2^i \ (\text{mod}\ 2)$, and $f^i = f_1^i, f_2^i \ (\text{mod}\ 2)$, for $i = 0, 1, \ldots, k$.

3. Party j individually sets $[y_j^0, y_j^1, \ldots, y_j^{k-1}, y_j^k] = [f_j^k, f_j^{k-1}, \ldots, f_j^1, f_j^0]$, for $j = 1, 2$.

Specification 4.10 (Shift$\{s\}$) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $(x_1, x_2)\{s\} \mapsto (y_1, y_2)$ *such that* $y_1 + y_2 = \lfloor \frac{x_1+x_2}{2^s} \rfloor$.

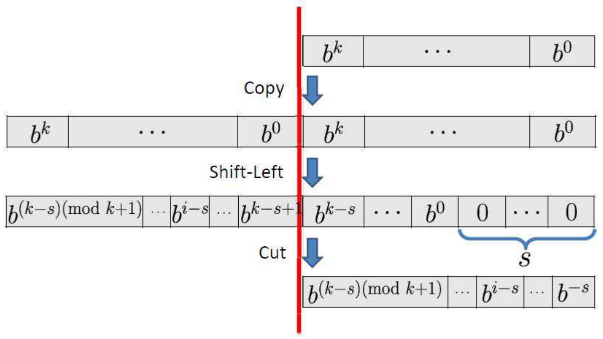
The protocol details are as follows:

PROTOCOL Shift$\{s\}$

1. Party 1 and Party 2 jointly execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol $(x_1, x_2)\{k\} \mapsto ((b_1^0, \dots, b_1^k), (b_2^0, \dots, b_2^k)$, such that $(b^k \cdots b^0)_2 = x_1 + x_2$, and $b^i = b_1^i + b_2^i \pmod 2$, for $i = 0, 1, \dots, k$.
2. Party j individually sets $[d_j^0, d_j^1, \dots, d_j^{k-s-1}, d_j^{k-s}] = [b_j^k, b_j^{k-1}, \dots, b_j^{s+1}, b_j^s]$, for $j = 1, 2$.
3. Party 1 and Party 2 collaboratively run the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{k-s\}$ protocol
$$((d_1^0, \dots, d_1^{k-s}), (d_2^0, \dots, d_2^{k-s}))\{k-s\} \mapsto (y_1, y_2)$$
such that $y_1 + y_2 = (d^{k-s} d^{k-s-1} \cdots d^1 d^0)_2$.

4.2.5 Rotation

Specification 4.11 (Rotate-Left) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x_1^0, x_1^1, \dots, x_1^k, s_1), (x_2^0, x_2^1, \dots, x_2^k, s_2)) \mapsto ((y_1^0, y_1^1, \dots, y_1^k), (y_2^0, y_2^1, \dots, y_2^k))$ *such that* $y^i = x^{(i-r) \pmod{k+1}}$ *where* $r = [(s_1 + s_2) \pmod n] \pmod{k+1}$, *for* $i = 0, 1, \dots, k$.



b^k ⋯ b^0
Copy
b^k ⋯ b^0 b^k ⋯ b^0
Shift-Left
$b^{(k-s)(\mathrm{mod}\ k+1)}$ ⋯ b^{i-s} ⋯ b^{k-s+1} b^{k-s} ⋯ b^0 0 ⋯ 0
s
Cut
$b^{(k-s)(\mathrm{mod}\ k+1)}$ ⋯ b^{i-s} ⋯ b^{-s}

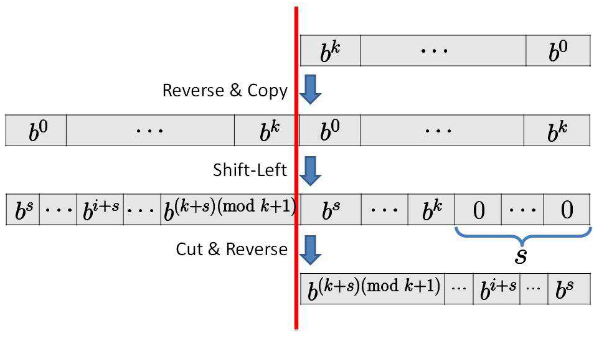
Figure 3. Rotate-Left

Let $b^i \in \mathbb{Z}_2$ for $i = 0, 1, \cdots, k$, and the design idea of the Rotate-Left protocol is as Figure 3 shown.

PROTOCOL Rotate-Left

1. Party 1 and Party 2 collaboratively execute the the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n^2}\{k\}$ protocol
$$((x_1^0, \dots, x_1^k), (x_2^0, \dots, x_2^k)) \mapsto (X_1, X_2)$$
such that $X_1 + X_2 \pmod{n^2} = (x^k x^{k-1} \cdots x^1 x^0)_2$, where $x^i = x_1^i + x_2^i \pmod 2$, for $i = 0, 1, \cdots, k$.
2. Party j individually computes $T_j = X_j \cdot (2^{k+1} + 1) \pmod{n^2}$, for $j = 1, 2$.
3. Party 1 and Party 2 jointly run the Exp(pb, se, n_y) $\in \mathbb{Z}_n$ protocol $(s_1, s_2)\{2, k+1\} \mapsto (u_1, u_2)$ such that $u_1 + u_2 = 2^{s_1+s_2}$.
4. These two parties collaboratively run the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n^2}\{k\}$ protocol
$$(u_1, u_2)\{k\} \mapsto ((d_1^0, \dots, d_1^k), (d_2^0, \dots, d_2^k)) \mapsto (U_1, U_2)$$
such that $U_1 + U_2 \pmod{n^2} = u_1 + u_2 \pmod n$.
5. These two parties jointly run the Product protocol $((T_1, U_1), (T_2, U_2)) \mapsto (V_1, V_2)$ such that $V_1 + V_2 \pmod{n^2} = (T_1 + T_2)(U_1 + U_2) \pmod{n^2}$.
6. Party 1 and Party 2 jointly execute the $\mathbb{Z}_{n^2}$-to-$\mathbb{Z}_2\{2k+1\}$ protocol
$$(V_1, V_2)\{2k+1\} \mapsto ((f_1^0, \dots, f_1^{2k+1}), (f_2^0, \dots, f_2^{2k+1}))$$
such that $(f^{2k+1} f^{2k} \cdots f^1 f^0)_2 = V_1 + V_2 \pmod{n^2}$, where $f^i = f_1^i + f_2^i \pmod 2$, for $i = 0, 1, \cdots, 2k+1$.
7. Party j individually sets $[y_j^0, y_j^1, \dots, y_j^{k-1}, y_j^k] = [f_j^{k+1}, \dots, f_j^{2k+1}]$.

Specification 4.12 (Rotate-Right) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x_1^0, x_1^1, \dots, x_1^k, s_1), (x_2^0, x_2^1, \dots, x_2^k, s_2)) \mapsto ((y_1^0, y_1^1, \dots, y_1^k), (y_2^0, y_2^1, \dots, y_2^k))$ *such that* $y^i = x^{(i+r) \pmod{k+1}}$ *where* $r = [(s_1 + s_2) \pmod n] \pmod{k+1}$, *for* $i = 0, 1, \dots, k$.



b^k ⋯ b^0
Reverse & Copy
b^0 ⋯ b^k b^0 ⋯ b^k
Shift-Left
b^s ⋯ b^{i+s} ⋯ $b^{(k+s)(\mathrm{mod}\ k+1)}$ b^s ⋯ b^k 0 ⋯ 0
s
Cut & Reverse
$b^{(k+s)(\mathrm{mod}\ k+1)}$ ⋯ b^{i+s} ⋯ b^s

Figure 4. Rotate-Right

Let $b^i \in \mathbb{Z}_2$ for $i = 0, 1, \cdots, k$, and the design idea of the Rotate-Right protocol is as Figure 4 shown.

PROTOCOL Rotate-Right

1. Party j individually sets $[d_j^0, d_j^1, \dots, d_j^{k-1}, d_j^k] = [x_j^k, x_j^{k-1}, \dots, x_j^1, x_j^0]$, for $j = 1, 2$.
2. Party 1 and Party 2 collaboratively execute the Rotate-Left protocol $((d_1^0, d_1^1, \dots, d_1^k, s_1), (d_2^0, d_2^1, \dots, d_2^k, s_2)) \mapsto ((f_1^0, f_1^1, \dots, f_1^k), (f_2^0, f_2^1, \dots, f_2^k))$ such that $f^i = d^{(i-r) \pmod{k+1}}$ where $r = [(s_1 + s_2) \pmod n] \pmod{k+1}$, for $i = 0, 1, \dots, k$.

3. Party j individually sets $[y_j^0, y_j^1, \ldots, y_j^{k-1}, y_j^k] = [f_j^k, f_j^{k-1}, \ldots, f_j^1, f_j^0]$, for $j = 1, 2$.

4.2.6 Division/Remainder

We construct our protocols over the finite group $\mathbb{Z}_n$, in which the division is normally defined as the multiplication to divisor's multiplicative inverse. However, such definition is not practical, and neither is it feasible since element $t \in \mathbb{Z}_n$ may not even have multiplicative inverse if t and n are not coprime. As a result, we need to give division a practical and feasible definition, and we choose to follow the integer division in modern computers. Given two integers $x, y \in \mathbb{N}$, the integer division is defined as the follows:

$$\left\lfloor \frac{x}{y} \right\rfloor = q, \text{where } x = y \cdot q + r, \text{and } 0 \leq r < y.$$

Our solution to the division protocol is actually an emulation for a $(k+1)$-bit divider. During the computation of $\left\lfloor \frac{x}{y} \right\rfloor$, we iteratively check whether $x \geq y \cdot 2^i$, for $i = k-1, \ldots, 0$. If it is true, the i-th bit of q is 1, and x is subtracted by $y \cdot 2^i$; otherwise, the i-th bit of q is 0, and x remains untouched. This iterative method is actually the algorithm for long division.

However, we need to mention that it is unfeasible to compute $x \cdot 2^i$ in $\mathbb{Z}_n$ since we need double precision to correctly represent $y \cdot 2^i$, for $i = k-1, \ldots, 0$; otherwise, $y \cdot 2^i \pmod n$ will give us unpredictable results. Therefore, in our solution we first convert both the dividend and the divisor from $\mathbb{Z}_n$ sharing to Z_{n^2} sharing, by which $y \cdot 2^i$ can always be correctly represented. After the conversion, we emulate the divider with the Scalar-Product protocol.

Based on the Comparison, If-Then-Else, $\mathbb{Z}_2$-to-$\mathbb{Z}_n$, and $\mathbb{Z}_n$-to-$\mathbb{Z}_2$ protocols, we specify the following Div/Rem protocol, which represents "Division/Remainder".

Specification 4.13 (Div/Rem) *Two parties share the dividend and the divisor in $\mathbb{Z}_n$, and they want to jointly execute the secure protocol $((x_1, y_1), (x_2, y_2)) \mapsto ((q_1, r_1), (q_2, r_2))$, where $x_1 + x_2 = (y_1 + y_2)(q_1 + q_2) + (r_1 + r_2)$ and $0 \leq (r_1 + r_2) < (y_1 + y_2)$.*

PROTOCOL Div/Rem

1. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n^2}\{k\}$ protocol
$$(x_1, x_2) \mapsto ((b_1^0, \ldots, b_1^k), (b_2^0, \ldots, b_2^k)) \mapsto (X_1, X_2)$$
such that $X_1 + X_2 \pmod{n^2} = x_1 + x_2 \pmod n$.
2. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n^2}\{k\}$ protocol
$$(y_1, y_2) \mapsto ((c_1^0, \ldots, c_1^k), (c_2^0, \ldots, c_2^k)) \mapsto (Y_1, Y_2)$$
such that $Y_1 + Y_2 \pmod{n^2} = y_1 + y_2 \pmod n$.

Algorithm 1 Calculate $q = \lfloor \frac{x}{y} \rfloor, r = x - y \cdot q$

```
$q \leftarrow 0$
for $i \leftarrow k-1, k-2, \cdots, 0$ do
  if $x \geq y \cdot 2^i$ then
    $x \leftarrow x - y \cdot 2^i$
    $bit \leftarrow 1$
  else
    $bit \leftarrow 0$
  end if
  $q \leftarrow q + bit \cdot 2^i$
end for
$r \leftarrow x$
```

3. Party j sets $X_j^k = X_j$, for $j = 1, 2$.
4. For $i = k-1, k-2, \ldots, 0$, repeat Step 4a to Step 4d.
   (a) Party j computes $t_j^i = X_j^{i+1} - Y_j \cdot 2^i \pmod{n^2}$, for $j = 1, 2$.
   (b) Party 1 and Party 2 jointly run the Comparison protocol $(t_1^i, t_2^i) \mapsto (s_1^i, s_2^i)$ such that
   $$s_1^i + s_2^i \pmod{n^2} = \begin{cases} 1 & \text{if } t_1^i + t_2^i < 0, \\ 0 & \text{otherwise.} \end{cases}$$
   (c) Party j individually computes $q_j^i = 1 - s_j^i \pmod{n^2}$, for $j = 1, 2$, such that
   $$q_1^i + q_2^i \pmod{n^2} = \begin{cases} 0 & \text{if } s_1^i + s_2^i = 1, \\ 1 & \text{if } s_1^i + s_2^i = 0. \end{cases}$$
   (d) Party 1 and Party 2 run the If-Then-Else protocol $((s_1^i, X_1^{i+1}, t_1^i), (s_2^i, X_2^{i+1}, t_2^i)) \mapsto (X_1^i, X_2^i)$ such that
   $$X_1^i + X_2^i \pmod{n^2} = \begin{cases} X_1^{i+1} + X_2^{i+1} & \text{if } s_1^i + s_2^i = 1, \\ t_1^i + t_2^i & \text{if } s_1^i + s_2^i = 0. \end{cases}$$
5. For $j = 1, 2$, Party j computes $r_j = X_j^0 \pmod n$, and $q_j = \sum_{i=0}^{k-1} q_j^i \cdot 2^i \pmod n$.

4.2.7 Square Root

Specification 4.14 (Sqrt) *Party 1 and Party 2 want to collaboratively execute the secure protocol $(x_1, x_2) \mapsto (y_1, y_2)$ such that $y_1 + y_2 = \lfloor \sqrt{x_1 + x_2} \rfloor$.*

The protocol details are as follows:

PROTOCOL Sqrt

1. Two parties collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol $(x_1, x_2) \mapsto ((b_1^0, \ldots, b_1^k), (b_2^0, \ldots, b_2^k))$, such that $b^i = b_1^i + b_2^i \pmod 2$, and $(b^k \cdots b^0)_2 = x_1 + x_2$.
2. If $k + 1 \pmod 2 = 1$, Party j sets $b_j^{k+1} = 0$, for $j = 1, 2$.
3. For $j = 1, 2$, Party j individually sets $d_j^e, g_j^e, y_j^e = 0$, where $e = \lceil \frac{k+1}{2} \rceil + 1$.

Algorithm 2 Calculate $y = \lfloor \sqrt{x} \rfloor$

```
$(b^k b^{k-1} \cdots b^1 b^0)_2 \leftarrow x$
if $k + 1 \pmod 2 = 1$ then
  $b^{k+1} \leftarrow 0$
end if
$d, g, y \leftarrow 0$
for $i \leftarrow \lceil \frac{k+1}{2} \rceil$ to 1 do
  $d = 4d + 2b^{2i-1} + b^{2i-2}$
  if $d \geq 2g + 1$ then
    $f \leftarrow d - (2g + 1)$
    $g \leftarrow 2g + 2$
    $y \leftarrow 2y + 1$
  else
    $g \leftarrow 2g$
    $y \leftarrow 2y$
  end if
end for
```

4. For $i = \lceil \frac{k+1}{2} \rceil, \cdots, 1$, repeat from Step 4a to Step 4f.

   (a) Party 1 and Party 2 jointly run the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^{2i-1}, b_2^{2i-1}) \mapsto (t_1^{2i-1}, t_2^{2i-1})$, such that $t_1^{2i-1} + t_2^{2i-1} \pmod n = (b^{2i-1})_2$.

   (b) Party 1 and Party 2 jointly run the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^{2i-2}, b_2^{2i-2}) \mapsto (t_1^{2i-2}, t_2^{2i-2})$, such that $t_1^{2i-2} + t_2^{2i-2} \pmod n = (b^{2i-2})_2$.

   (c) For $j = 1, 2$, Party j individually sets $d_j^i = 4d_j^{i+1} + 2t_j^{2i-1} + t_j^{2i-2}$.

   (d) Two parties jointly execute the Comparison protocol $(d_1^i - 2g_1^i - 1, d_2^i - 2g_2^i) \mapsto (c_1, c_2)$, where

$$c_1 + c_2 = \begin{cases} 1 & \text{if } (d_1^i + d_2^i) - 2(g_1^i + g_2^i) - 1 < 0, \\ 0 & \text{otherwise.} \end{cases}$$

   (e) Party 1 and Party 2 collaboratively run the If-Then-Else protocol $((c_1, d_1^i, d_1^i - 2g_1^i - 1), (c_2, d_2^i, d_2^i - 2g_2^i)) \mapsto (d_1^{i-1}, d_2^{i-1})$ such that

$$d_1^{i-1} + d_2^{i-1} = \begin{cases} d^i & \text{if } c_1 + c_2 = 1, \\ d^i - 2g^i - 1 & \text{if } c_1 + c_2 = 0, \end{cases}$$

   where $d^i = d_1^i + d_2^i, g^i = g_1^i + g_2^i$.

   (f) Party 1 sets $g_1^{i-1} = 2g_1^i + 2 - 2c_1, y_1^{i-1} = 2y_1^i + 1 - c_1$ while Party 2 sets $g_2^{i-1} = 2g_2^i - 2c_2, y_2^{i-1} = 2y_2^i - c_2$ such that

$$g_1^{i-1} + g_2^{i-1} = \begin{cases} 2g^i & \text{if } c_1 + c_2 = 1, \\ 2g^i + 2 & \text{if } c_1 + c_2 = 0, \end{cases}$$

$$y_1^{i-1} + y_2^{i-1} = \begin{cases} 2y^i & \text{if } c_1 + c_2 = 1, \\ 2y^i + 1 & \text{if } c_1 + c_2 = 0, \end{cases}$$

   where $y^i = y_1^i + y_2^i$.

5. Party j individually sets $y_j = y_j^0$, for $j = 1, 2$.

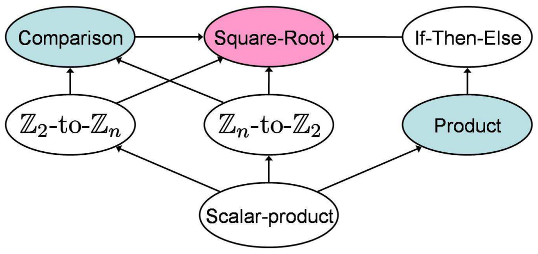

Comparison
Square-Root
If-Then-Else
$\mathbb{Z}_2$-to-$\mathbb{Z}_n$
$\mathbb{Z}_n$-to-$\mathbb{Z}_2$
Product
Scalar-product

Figure 5. The hierarchy of the Square-root protocol.

4.2.8 Logarithm

Specification 4.15 (Logarithm) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $(x_1, x_2) \mapsto (y_1, y_2)$ *such that* $y_1 + y_2 = \lfloor \log_2 (x_1 + x_2) \rfloor$.

Algorithm 3 Calculate $t = \lfloor \log_2 x \rfloor$

```
$(b^k b^{k-1} \cdots b^1 b^0)_2 \leftarrow x$
$t \leftarrow b^1$
for $i \leftarrow 2$ to $k$ do
  if $b^i = 1$ then
    $t \leftarrow i$
  end if
end for
```

PROTOCOL Logarithm

1. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol $(x_1, x_2)\{k\} \mapsto ((b_1^0, \ldots, b_1^k), (b_2^0, \ldots, b_2^k))$ such that $(b^k b^{k-1} \cdots b^1 b^0)_2 = x_1 + x_2$ where $x_1, x_2 \in \mathbb{Z}_n, b_1^i, b_2^i \in \mathbb{Z}_2$, and $b^i = b_1^i + b_2^i \pmod 2$, for $i = 0, 1, \cdots, k$.

2. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^1, b_2^1)\{0\} \mapsto (t_1^1, t_2^1)$ such that $t_1^1 + t_2^1 = (b^0)_2$ where $t_1^1, t_2^1 \in \mathbb{Z}_n$.

3. For $i = 2$ to k, repeat Step 3a and Step 3b.

   (a) Party 1 and Party 2 collaboratively run the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^i, b_2^i)\{0\} \mapsto (d_1^i, d_2^i)$ such that $d_1^i + d_2^i = (b^i)_2$ where $d_1^i, d_2^i \in \mathbb{Z}_n$.

   (b) These two parties jointly run the If-Then-Else protocol $((d_1^i, i, t_1^{i-1}), (d_2^i, 0, t_2^{i-1})) \mapsto (t_1^i, t_2^i)$ such that

$$t_1^i + t_2^i = \begin{cases} i & \text{if } \ d_1^i + d_2^i = 1, \\ t_1^{i-1} + t_2^{i-1} & \text{if } \ d_1^i + d_2^i = 0. \end{cases}$$

4. Party j locally sets $y_j = t_j^k$, for $j = 1, 2$.

4.3 Building Blocks for Fixed-Point Numbers

Since the Scalar-Product is integer-based, we can transform the fixed-point numbers (such as x) to integers (such as $\hat{x}, \tilde{x}$) and use integers to simulate fixed-point number addition, subtraction, multiplication, and other arithmetic. In Section 4.3, let x, y be fixed-point numbers with s bits after the decimal point, and

$$\hat{x} = 2^s x, \quad \tilde{x} = 2^s \hat{x} = 2^{2s} x,$$
$$\hat{y} = 2^s y, \quad \tilde{y} = 2^s \hat{y} = 2^{2s} y.$$

It is apparent that $\hat{x}, \hat{y}, \tilde{x}, \tilde{y}$ are integers. We propose (flo) Scalar-Product$\{s\}$, (flo) Product$\{s\}$, and (flo) Square$\{s\}$ simulating fixed-point number arithmetic. Note that both the inputs and outputs of these protocols are variables with symbol $\hat{}$, which means they represent fixed-point numbers with s bits after the fixed point, where s is a $public$ type integer.

4.3.1 (flo) Scalar-Product$\{s\}$

Specification 4.16 ((flo) Scalar-Product$\{s\}$) *Party 1 and Party 2 want to collaboratively execute the secure protocol*

$$((\hat{x}[1]_1, \ldots, \hat{x}[d]_1), (\hat{x}[1]_2, \ldots, \hat{x}[d]_2))\{s\} \mapsto (\hat{y}_1, \hat{y}_2)$$

such that $\hat{y}_1 + \hat{y}_2 = \lfloor \frac{\hat{x}[1]_1 \cdot \hat{x}[1]_2 + \cdots + \hat{x}[d]_1 \cdot \hat{x}[d]_2}{2^s} \rfloor$ *and* $\hat{x}[i]_1, \hat{x}[i]_2, \hat{y}_1, \hat{y}_2 \in \mathbb{Z}_n$, *for* $i = 1, \ldots, d$.

PROTOCOL (flo) Scalar-Product$\{s\}$

1. Party 1 and Party 2 jointly execute the Scalar-Product protocol
$$((\hat{x}[1]_1, \ldots, \hat{x}[d]_1), (\hat{x}[1]_2, \ldots, \hat{x}[d]_2)) \mapsto (\tilde{t}_1, \tilde{t}_2)$$
such that $\tilde{t}_1 + \tilde{t}_2 = \hat{x}[1]_1 \cdot \hat{x}[1]_2 + \cdots + \hat{x}[d]_1 \cdot \hat{x}[d]_2$.
2. Party 1 and Party 2 jointly run the Shift$\{s\}$ protocol $(\tilde{t}_1, \tilde{t}_2)\{s\} \mapsto (\hat{y}_1, \hat{y}_2)$, where $\hat{y}_1 + \hat{y}_2 = \lfloor \frac{\tilde{t}_1 + \tilde{t}_2}{2^s} \rfloor$.

4.3.2 (flo) Product$\{s\}$

Specification 4.17 ((flo) Product$\{s\}$) *Party 1 and Party 2 share the multiplicand and the multiplicator. They want to securely execute the protocol* $((\hat{x}_1, \hat{y}_1), (\hat{x}_2, \hat{y}_2))\{s\} \mapsto (\hat{z}_1, \hat{z}_2)$ *such that* $\hat{z}_1 + \hat{z}_2 = \lfloor \frac{(\hat{x}_1 + \hat{x}_2)(\hat{y}_1 + \hat{y}_2)}{2^s} \rfloor$.

PROTOCOL (flo) Product$\{s\}$

1. Party 1 and Party 2 jointly execute the Scalar-Product protocol $((\hat{x}_1, \hat{y}_1), (\hat{y}_2, \hat{x}_2)) \mapsto (\tilde{t}_1, \tilde{t}_2)$ such that $\tilde{t}_1 + \tilde{t}_2 = \hat{x}_1 \hat{y}_2 + \hat{y}_1 \hat{x}_2$.
2. Party j individually computes $\tilde{u}_j = \tilde{t}_j + \hat{x}_j \hat{y}_j$, for $j = 1, 2$.
3. Party 1 and Party 2 jointly run the Shift$\{s\}$ protocol $(\tilde{u}_1, \tilde{u}_2)\{s\} \mapsto (\hat{z}_1, \hat{z}_2)$, where $\hat{z}_1 + \hat{z}_2 = \lfloor \frac{\tilde{u}_1 + \tilde{u}_2}{2^s} \rfloor$.

4.3.3 (flo) Square$\{s\}$

Specification 4.18 ((flo) Square$\{s\}$) *Party 1 and Party 2 share a number in* $\mathbb{Z}_n$, *and they want to securely execute the protocol* $(\hat{x}_1, \hat{x}_2)\{s\} \mapsto (\hat{y}_1, \hat{y}_2)$ *such that* $\hat{y}_1 + \hat{y}_2 = \lfloor \frac{(\hat{x}_1 + \hat{x}_2)^2}{2^s} \rfloor$.

PROTOCOL (flo) Square$\{s\}$

1. Party 1 and Party 2 jointly execute the Scalar-Product protocol $(\hat{x}_1, \hat{x}_2) \mapsto (\tilde{t}_1, \tilde{t}_2)$ such that $\tilde{t}_1 + \tilde{t}_2 = \hat{x}_1 \cdot \hat{x}_2$.
2. Party j individually computes $\tilde{u}_j = 2\tilde{t}_j + \hat{x}_j^2$, for $j = 1, 2$.
3. Party 1 and Party 2 jointly run the Shift$\{s\}$ protocol $(\tilde{u}_1, \tilde{u}_2)\{s\} \mapsto (\hat{y}_1, \hat{y}_2)$, where $\hat{y}_1 + \hat{y}_2 = \lfloor \frac{\tilde{u}_1 + \tilde{u}_2}{2^s} \rfloor$.

4.4 General Solutions

Here we design two protocols, the Function(x)$\{s, f(\cdot)\}$ and Function(x, y)$\{s, f(\cdot)\}$ protocols, which provide general solutions to functions that have one or two additively shared parameters between two parties. For generality, we assume that the output is a fixed-point number with s bits after the decimal point. Using the same strategy, these two parties can always securely compute functions with more parameters rather than one or two.

4.4.1 Function(x)$\{s, f(\cdot)\}$

Specification 4.19 (Function(x)) *Party 1 and Party 2 share some number* x. *They want to securely execute the protocol* $(x_1, x_2)\{s, f(\cdot)\} \mapsto (\hat{y}_1, \hat{y}_2)$ *such that* $\hat{y}_1 + \hat{y}_2 = [2^s \cdot f(x_1 + x_2)]$ *where* s *is the number of bits after the decimal point and* $f(\cdot)$ *is some function. (For example,* $f(x) = \log_7 x$ *or* $f(x) = \sqrt[3]{x}$ *where* $x = x_1 + x_2$.*)*

PROTOCOL Function(x)

1. For $i = 0, 1, \cdots, n-1$, repeat Step 1a.
   (a) Party 1 individually sets $\hat{t}[i] = [2^s f(x_1 + i)]$ while Party 2 individually sets
$$u[i] = \begin{cases} 1 & \text{if} \quad i = x_2, \\ 0 & \text{if} \quad i \neq x_2. \end{cases}$$
2. Party 1 and Party 2 collaboratively execute the Scalar-Product protocol
$$((\hat{t}[0], \ldots, \hat{t}[n-1]), (u[0], \ldots, u[n-1])) \mapsto (\hat{y}_1, \hat{y}_2)$$
such that $\hat{y}_1 + \hat{y}_2 = \hat{t}[0] \cdot u[0] + \cdots + \hat{t}[n-1] \cdot u[n-1]$.

4.4.2 Function$(x, y)\{s, f(\cdot)\}$

Specification 4.20 (Function(x, y)) *Party 1 and Party 2 share some x and y. They want to securely execute the protocol $((x_1, y_1), (x_2, y_2))\{s, f(\cdot)\} \mapsto (\hat{z}_1, \hat{z}_2)$ such that $\hat{z}_1 + \hat{z}_2 = [2^s \cdot f(x_1 + x_2, y_1 + y_2)]$ where s is the number of bits after the decimal point and $f(\cdot)$ is some function. (For example, $f(x, y) = \log_y x$ or $f(x, y) = \sqrt[y]{x}$ where $x = x_1 + x_2$ and $y = y_1 + y_2$.)*

PROTOCOL Function(x, y)

1. For $i = 0, \cdots, n-1$, repeat from Step 1a to Step 1c.
   (a) For $j = 0, \cdots, n-1$, repeat Step 1(a)i.
      i. Party 1 individually sets $\hat{t}[j] = [2^s f(x_1 + i, y_1 + j)]$ while Party 2 individually sets
      $$w[j] = \begin{cases} 1 & \text{if} \quad j = y_2, \\ 0 & \text{if} \quad j \neq y_2. \end{cases}$$
   (b) Party 1 and Party 2 collaboratively execute the Scalar-Product protocol
   $$((\hat{t}[0], \ldots, \hat{t}[n-1]), (w[0], \ldots, w[n-1])) \mapsto (\hat{u}[i]_1, \hat{u}[i]_2)$$
   such that $\hat{u}[i]_1 + \hat{u}[i]_2 = \hat{t}[0] \cdot w[0] + \cdots + \hat{t}[n-1] \cdot w[n-1]$.
   (More specifically, $\hat{u}[i]_1 + \hat{u}[i]_2 = \hat{t}[y_2] = [2^s f(x_1 + i, y_1 + y_2)]$.)
   (c) Party 1 individually sets $\hat{v}[i] = \hat{u}[i]_1$ while Party 2 individually sets
   $$q[i] = \begin{cases} 1 & \text{if} \quad i = x_2, \\ 0 & \text{if} \quad i \neq x_2. \end{cases}$$
2. Party 1 and Party 2 collaboratively execute the Scalar-Product protocol $((\hat{v}[0], \ldots, \hat{v}[n-1]), (q[0], \ldots, q[n-1])) \mapsto (\hat{r}_1, \hat{r}_2)$ such that $\hat{r}_1 + \hat{r}_2 = \hat{v}[0] \cdot q[0] + \cdots + \hat{v}[n-1] \cdot q[n-1]$.
   (More specifically, $\hat{r}_1 + \hat{r}_2 = \hat{v}[x_2] = \hat{u}[x_2]_1$.)
3. Party 1 individually sets $\hat{z}_1 = \hat{r}_1$, and Party 2 computes $\hat{z}_2 = \hat{u}[x_2]_2 + \hat{r}_2$.

Since every building block is composed of other ones and based on the Scalar-Product, Table 1 lists the constituents of these building blocks.

We further break those constitutive building blocks in Table 1 into Scalar-Product protocols of which Table 2 shows the domain, dimension, and times.

5 Examples in Statistics

In this section, we give examples for SMC problems in statistics and design protocols solving them. Two scenarios of a shared database are considered. The $secret$ type of database indicates the data of which are shared between Party 1 and Party 2 ; the $private$ type one means each party privately owns their database, and a view with a union is created over both of them to provide a complete view. Figure 6 and Figure 7 show the split and the shard database architecture(aka horizontal partitioning) relatively. Based on these two different scenarios, $(secret)$ and $(private)$ types of protocols are proposed, including Range, Mean, and Var. With these protocols, we can securely compute the range, the mean, and the variance of the linked database.

5.1 Split Database

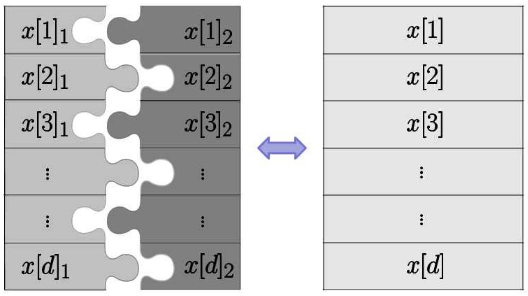


Figure 6. Split database architecture(secret shares)

5.1.1 (secret) Range

Before introducing the (secret) Range protocol, we first design the (secret) Max and the (secret) Min protocols.

Specification 5.1 ((secret) Max) *Party 1 and Party 2 want to collaboratively execute the secure protocol $((x[1]_1, x[2]_1, \cdots, x[d]_1), (x[1]_2, x[2]_2, \cdots, x[d]_2))\{d\} \mapsto (y_1, y_2)$ such that $y_1 + y_2 =$ the maximum of $(x[1], x[2], \cdots, x[d])$ where $x[i]_1, x[i]_2, y_1, y_2 \in \mathbb{Z}_n$, for $i = 1, 2, \cdots, d$.*

Algorithm 4 Calculate max = the maximum of $(x[1], x[2], \cdots, x[d])$

```
$max \leftarrow x[1]$
for $i \leftarrow 2, 3, \cdots, d$ do
    if $max < x[i]$ then
        $max \leftarrow x[i]$
    end if
end for
```

Based on Algorithm 4, the (secret) Max protocol details are as follows:

PROTOCOL (secret) Max

1. Party j individually sets $m_j^1 = x[1]_j$, for $j = 1, 2$.
2. For $i = 2, 3, \cdots, d$, repeat Step 2a and Step 2b.
   (a) Party 1 and Party 2 jointly execute the Comparison protocol $(m_1^{i-1} - x[i]_1, m_2^{i-1} - x[i]_2) \mapsto (c_1, c_2)$ such that
   $$c_1+c_2 = \begin{cases} 1 & \text{if } m_1^{i-1} + m_2^{i-1} - x[i]_1 - x[i]_2 < 0, \\ 0 & \text{otherwise.} \end{cases}$$
   (b) These two parties collaboratively run the If-Then-Else protocol $((c_1, x[i]_1, m_1^{i-1}), (c_2, x[i]_2, m_2^{i-1})) \mapsto (m_1^i, m_2^i)$ such that
   $$m_1^i + m_2^i = \begin{cases} x[i]_1 + x[i]_2 & \text{if } c_1 + c_2 = 1, \\ m_1^{i-1} + m_2^{i-1} & \text{if } c_1 + c_2 = 0. \end{cases}$$
3. Party j individually sets $y_j = m_j^d$, for $j = 1, 2$.

Specification 5.2 ((secret) Min) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1]_1, x[2]_1, \cdots, x[d]_1), (x[1]_2, x[2]_2, \cdots, x[d]_2))\{d\} \mapsto (y_1, y_2)$ *such that* $y_1 + y_2$ = *the minimum of* $(x[1], x[2], \cdots, x[d])$ *where* $x[i]_1, x[i]_2, y_1, y_2 \in \mathbb{Z}_n$, *for* $i = 1, 2, \cdots, d$.

Algorithm 5 Calculate min = the minimum of $(x[1], x[2], \cdots, x[d])$

```
$min \leftarrow x[1]$
for $i \leftarrow 2, 3, \cdots, d$ do
    if $min > x[i]$ then
        $min \leftarrow x[i]$
    end if
end for
```

Based on Algorithm 5, the (secret) Min protocol details are as follows:

PROTOCOL (secret) Min

1. Party j locally sets $m_j^1 = x[1]_j$, for $j = 1, 2$.
2. For $i = 2, 3, \cdots, d$, repeat Step 2a and Step 2b.
   (a) Party 1 and Party 2 jointly execute the Comparison protocol $(x[i]_1 - m_1^{i-1}, x[i]_2 - m_2^{i-1}) \mapsto (c_1, c_2)$ such that
   $$c_1+c_2 = \begin{cases} 1 & \text{if } x[i]_1 + x[i]_2 - m_1^{i-1} - m_2^{i-1} < 0, \\ 0 & \text{otherwise.} \end{cases}$$
   (b) These two parties collaboratively run the If-Then-Else protocol $((c_1, x[i]_1, m_1^{i-1}), (c_2, x[i]_2, m_2^{i-1})) \mapsto (m_1^i, m_2^i)$ such that
   $$m_1^i + m_2^i = \begin{cases} x[i]_1 + x[i]_2 & \text{if } c_1 + c_2 = 1, \\ m_1^{i-1} + m_2^{i-1} & \text{if } c_1 + c_2 = 0. \end{cases}$$
3. Party j locally sets $y_j = m_j^d$, for $j = 1, 2$.

Specification 5.3 ((secret) Range) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1]_1, x[2]_1, \cdots, x[d]_1), (x[1]_2, x[2]_2, \cdots, x[d]_2))\{d\} \mapsto (y_1, y_2)$ *such that* $y_1 + y_2$ = *the range of* $(x[1], x[2], \cdots, x[d])$ *where* $x[i] = x[i]_1 + x[i]_2$ *and* $x[i], x[i]_1, x[i]_2, y_1, y_2 \in \mathbb{Z}_n$, *for* $i = 1, 2, \cdots, d$.

Algorithm 6 Calculate r = the range of $(x[1], \ldots, x[d])$

```
$u \leftarrow \text{Max}(x[1], \ldots, x[d])$
$v \leftarrow \text{Min}(x[1], \ldots, x[d])$
$r \leftarrow u - v$
```

PROTOCOL (secret) Range

1. Party 1 and Party 2 collaboratively execute the (secret) Max protocol $((x[1]_1, x[2]_1, \cdots, x[d]_1), (x[1]_2, x[2]_2, \cdots, x[d]_2))\{d\} \mapsto (y_1, y_2)$ such that $y_1 + y_2$ = the maximum of $(x[1], x[2], \cdots, x[d])$.
2. Party 1 and Party 2 collaboratively execute the (secret) Min protocol $((x[1]_1, x[2]_1, \cdots, x[d]_1), (x[1]_2, x[2]_2, \cdots, x[d]_2))\{d\} \mapsto (v_1, v_2)$ such that $v_1 + v_2$ = the minimum of $(x[1], x[2], \cdots, x[d])$.
3. Party j individually computes $y_j = u_j - v_j$, for $j = 1, 2$.

5.1.2 (secret) Mean

Specification 5.4 ((secret) Mean) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1]_1, \ldots, x[d]_1), (x[1]_2, \ldots, x[d]_2))\{d\} \mapsto (q_1, q_2)$ *such that* $q_1 + q_2 = \lfloor \frac{\sum_{i=1}^d (x[i]_1 + x[i]_2)}{d} \rfloor$.

PROTOCOL (secret) Mean

1. Party j individually computes $t_j = \sum_{i=1}^d x[i]_1$, for $j = 1, 2$.
2. Party 1 and Party 2 collaboratively execute the Div{divisor} protocol $(t_1, t_2)\{d\} \mapsto ((q_1, r_1), (q_2, r_2))$ such that $q_1 + q_2 = \lfloor \frac{t_1 + t_2}{d} \rfloor$.

5.1.3 (secret) Variance

Specification 5.5 ((secret) Var) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1]_1, \ldots, x[d]_1), (x[1]_2, \ldots, x[d]_2))\{d\} \mapsto (q_1, q_2)$ *such that* $q_1 + q_2 = \lfloor \frac{d(s_1+s_2) - (t_1+t_2)^2}{d^2} \rfloor$ *where* $s_1 + s_2 = \sum_{i=1}^d (x[i]_1 + x[i]_2)^2$, *and* $t_j = \sum_{i=1}^d x[i]_j$, *for* $j = 1, 2$.

PROTOCOL (secret) Var

Algorithm 7 Calculate var = the variance of $(x[1], x[2], \cdots, x[d])$

```
$s \leftarrow$ SquareSum$(x[1], x[2], \cdots, x[d])$
$t \leftarrow$ Sum$(x[1], x[2], \cdots, x[d])$
$var = (d \cdot s - t^2)/d^2$
```

1. Party 1 and Party 2 collaboratively run the Scalar-Product protocol $((2x[1]_1, 2x[2]_1, \cdots, 2x[d]_1), (x[1]_2, x[2]_2, \cdots, x[d]_2)) \mapsto (u_1, u_2)$ such that $u_1 + u_2 = 2x[1]_1 \cdot x[1]_2 + \cdots + 2x[d]_1 \cdot x[d]_2$.
2. Party j individually computes $s_j = \sum_{i=1}^{d} x[i]_j^2 + u_j$, for $j = 1, 2$.
3. Party 1 and Party 2 collaboratively run the Square protocol $(t_1, t_2) \mapsto (v_1, v_2)$ such that $v_1 + v_2 = (t_1 + t_2)^2$.
4. Party j computes $z_j = p \cdot s_j - v_j$, for $j = 1, 2$.
5. Party 1 and Party 2 jointly execute the Div/Rem{divisor} protocol $(z_1, z_2)\{d^2\} \mapsto ((q_1, r_1), (q_2, r_2))$ such that $q_1 + q_2 = \left\lfloor \frac{z_1+z_2}{d^2} \right\rfloor$.

5.2 Shard Database

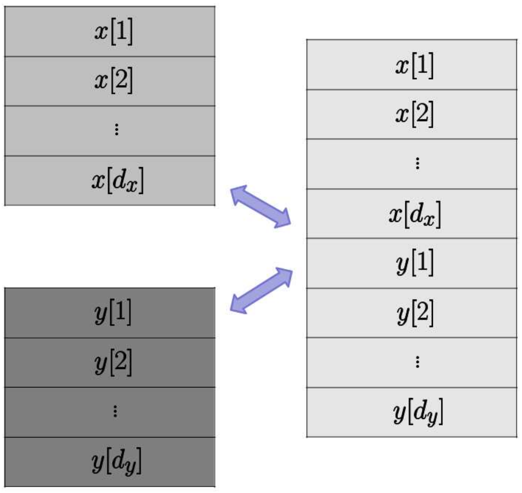


Figure 7. Shard database architecture(Horizontal partitioning)

5.2.1 (private) Range

Before introducing the (private) Range protocol, we first design the (private) Max and the (private) Min protocols.

Specification 5.6 ((private) Max) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $(x, y) \mapsto (z_1, z_2)$ *such that*

$$z_1 + z_2 = \begin{cases} y & \textit{if} \quad x - y < 0, \\ x & \textit{otherwise}, \end{cases}$$

where $x, y, z_1, z_2 \in \mathbb{Z}_n$.

Algorithm 8 Calculate max = the maximum of (x, y)

```
**if** $x > y$ **then**
    $max \leftarrow x$
**else**
    $max \leftarrow y$
**end if**
```

Based on Algorithm 8, the (private) Max protocol details are as follows:

PROTOCOL (private) Max

1. Party 1 and Party 2 jointly run the Comparison protocol $(-x, y) \mapsto (t_1, t_2)$ such that
$$t_1 + t_2 = \begin{cases} 1 & \text{if } y - x < 0, \\ 0 & \text{otherwise.} \end{cases}$$
2. Party 1 and Party 2 jointly run the If-Then-Else protocol $((t_1, x, 0), (t_2, 0, y)) \mapsto (z_1, z_2)$ such that
$$z_1 + z_2 = \begin{cases} x & \text{if} \quad t_1 + t_2 = 1, \\ y & \text{if} \quad t_1 + t_2 = 0. \end{cases}$$

Specification 5.7 ((private) Min) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $(x, y) \mapsto (z_1, z_2)$ *such that*

$$z_1 + z_2 = \begin{cases} x & \textit{if} \quad x - y < 0, \\ y & \textit{otherwise}, \end{cases}$$

where $x, y, z_1, z_2 \in \mathbb{Z}_n$.

Algorithm 9 Calculate min = the minimum of (x, y)

```
**if** $x < y$ **then**
    $min \leftarrow x$
**else**
    $min \leftarrow y$
**end if**
```

Based on Algorithm 9, the (private) Min protocol details are as follows:

PROTOCOL (private) Min

1. Party 1 and Party 2 jointly run the Comparison protocol $(x, -y) \mapsto (t_1, t_2)$ such that
$$t_1 + t_2 = \begin{cases} 1 & \text{if } x - y < 0, \\ 0 & \text{otherwise.} \end{cases}$$

2. Party 1 and Party 2 jointly run the If-Then-Else protocol $((t_1, x, 0), (t_2, 0, y)) \mapsto (z_1, z_2)$ such that

$$z_1 + z_2 = \begin{cases} x & \text{if} \quad t_1 + t_2 = 1, \\ y & \text{if} \quad t_1 + t_2 = 0. \end{cases}$$

Specification 5.8 ((private) Range) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1]_1, \ldots, x[d_x]), (y[1], \ldots, y[d_y])) \mapsto (z_1, z_2)$ *such that* $z_1 + z_2 =$ *the maximum - the minimum of the union database* $(x[1], \ldots, x[d_x], y[1], \ldots, y[d_y])$.

Algorithm 10 Calculate r = the range of $(x[1], \ldots, x[d_x], y[1], \ldots, y[d_y])$

```
$s_x \leftarrow \text{Max}(x[1], \ldots, x[d_x])$
$s_y \leftarrow \text{Max}(y[1], \ldots, y[d_y])$
$t_x \leftarrow \text{Min}(x[1], \ldots, x[d_x])$
$t_y \leftarrow \text{Min}(y[1], \ldots, y[d_y])$
$u \leftarrow \text{Max}(s_x, s_y)$
$v \leftarrow \text{Min}(t_x, t_y)$
$r \leftarrow u - v$
```

Based on the primitives proposed in Section 4, the Range protocol can be constructed as follows:

PROTOCOL (private) Range

1. Party 1 sets s_x = the maximum and t_x = the minimum of $(x[1], \ldots, x[d_x])$, while Party 2 sets s_y = the maximum and t_y = the minimum of $(y[1], \ldots, y[d_y])$.
2. Party 1 and Party 2 collaboratively execute the (private) Max protocol $(s_x, s_y) \mapsto (u_1, u_2)$ such that
$$u_1 + u_2 = \begin{cases} s_y & \text{if} \quad s_x - s_y < 0, \\ s_x & \text{otherwise.} \end{cases}$$
3. Party 1 and Party 2 collaboratively run the (private) Min protocol $(t_x, t_y) \mapsto (v_1, v_2)$ such that
$$v_1 + v_2 = \begin{cases} t_x & \text{if} \quad t_x - t_y < 0, \\ t_y & \text{otherwise.} \end{cases}$$
4. Party j individually computes $z_j = u_j - v_j$, for $j = 1, 2$.

5.2.2 (private) Mean

Specification 5.9 ((private) Mean) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1], \ldots, x[d_x]), (y[1], \ldots, y[d_y])) \mapsto (q_1, q_2)$ *such that* $q_1 + q_2 = \lfloor \frac{\sum_{i=1}^{d_x} x[i] + \sum_{i=1}^{d_y} y[i]}{d_x + d_y} \rfloor$.

PROTOCOL (private) Mean

1. Party 1 computes $t_1 = \sum_{i=1}^{d_x} x[i]$, while Party 2 computes $t_2 = \sum_{i=1}^{d_y} y[i]$.
2. Party 1 and Party 2 collaboratively execute the Div protocol $((t_1, d_x), (t_2, d_y)) \mapsto ((q_1, r_1), (q_2, r_2))$ such that $q_1 + q_2 = \lfloor \frac{t_1 + t_2}{d_x + d_y} \rfloor$.

5.2.3 (private) Variance

Specification 5.10 ((private) Var) *Party 1 and Party 2 want to collaboratively execute the secure protocol* $((x[1], \ldots, x[d_x]), (y[1], \ldots, y[d_y])) \mapsto (q_1, q_2)$ *such that* $q_1 + q_2 = \lfloor \frac{(d_x+d_y)(s_1+s_2) - (t_1+t_2)^2}{(d_x+d_y)^2} \rfloor$ *where* $s_1 = \sum_{i=1}^{d_x} x[i]^2$, $t_1 = \sum_{i=1}^{d_x} x[i]$, $s_2 = \sum_{i=1}^{d_y} y[i]^2$, *and* $t_2 = \sum_{i=1}^{d_y} y[i]$.

Algorithm 11 Calculate var = the variance of $(x[1], x[2], \cdots, x[d_x], y[1], y[2], \cdots, y[d_y])$

```
$d \leftarrow d_x + d_y$
$s \leftarrow \text{SquareSum}(x[1], x[2], \cdots, x[d_x], y[1], y[2], \cdots, y[d_y])$
$t \leftarrow \text{Sum}(x[1], x[2], \cdots, x[d_x], y[1], y[2], \cdots, y[d_y])$
$var = (d \cdot s - t^2)/d^2$
```

PROTOCOL (private) Var

1. Party 1 computes $s_1 = \sum_{i=1}^{d_x} x[i]^2$ and $t_1 = \sum_{i=1}^{d_x} x[i]$ while Party 2 computes $s_2 = \sum_{i=1}^{d_y} y[i]^2$ and $t_2 = \sum_{i=1}^{d_y} y[i]$.
2. Party 1 and Party 2 collaboratively execute the Product protocol $((d_x, s_1), (d_y, s_2)) \mapsto (u_1, u_2)$ such that $u_1 + u_2 = (d_x + d_y)(s_1 + s_2)$.
3. Party 1 and Party 2 collaboratively run the Square protocol $(t_1, t_2) \mapsto (v_1, v_2)$ such that $v_1 + v_2 = (t_1 + t_2)^2$.
4. Party 1 and Party 2 collaboratively execute the Square protocol $(d_x, d_y) \mapsto (w_1, w_2)$ such that $w_1 + w_2 = (d_x + d_y)^2$.
5. Party j computes $z_j = u_j - v_j$, for $j = 1, 2$.
6. Party 1 and Party 2 jointly execute the Div protocol $((z_1, w_1), (z_2, w_2)) \mapsto ((q_1, r_1), (q_2, r_2))$ such that $q_1 + q_2 = \left\lfloor \frac{z_1 + z_2}{w_1 + w_2} \right\rfloor$.

Table 3 lists the constituents of building blocks introduced in this section.

We further break those constitutive building blocks in Table 3 into Scalar-Product protocols of which Table 4 shows the domain, dimension, and times.

6 Tradeoffs

In order to show tradeoffs between privacy and efficiency, in this section we take exponentiation (x^y) and division ($\lfloor \frac{x}{y} \rfloor$) as concrete examples discussing possible tradeoffs. For exponentiation, we also analyze the information leaking out according to *public* variables, present the experimental results, and estimate the time cost for each tradeoff helping make a reasonable tradeoff decision.

6.1 Tradeoffs: Exponentiation

Party 1 and Party 2 share the base x and the exponent y. Let the positive integers n_x and n_y be the range of x and the range of y respectively, where $x < n_x \leq n$, $y < n_y \leq n$, $n_x = 2^{k_x+1}$, $n_y = 2^{k_y+1}$, and $k_x, k_y \in \mathbb{N}$. As we can see in the discussion below, if tighter bounds are/is known for x and/or y, i.e., n_x and/or n_y are/is much smaller than n, the execution time needed would be much smaller. We argue that in cases where the users know or can derive tighter upper bounds for x and/or y due to the nature of the problem, and this knowledge can greatly improve computational efficiency. In some other cases, we envision that the computational cost is prohibitively high, so that the parties would agree to reveal some information about x and/or y in order to get the results in reasonable time. Nielsen and Schwartzbach define three variable types which are $secret$, $public$, and $private$ [25]. Here we only define two variable types: $secret$ and $public$. The $secret$ type of variable indicates the value of which is shared between Party 1 and Party 2 , while the $public$ type of variable means the value of which resides in plain view on these two parties. For simplicity, hereafter we set *sb* = $secret$ base, *se* = $secret$ exponent, *pb* = $public$ base, and *pe* = $public$ exponent.

Since the performance of the protocol is dependent on the number of bits necessary to represent the final results, a very effective way to improve the performance is to reveal the upper bounds of the base and/or the exponent. Below is the description of the exponentiation protocol with n_x and n_y. Note that $n_x = n_y = n$ is the case where no information about x, y is revealed.

Algorithm 12 is a simple repeated squaring algorithm to compute the exponentiation [26]. Note that in this algorithm the exponent y has to be treated as a sequence of bits. It is apparent that a secure two-party protocol based on the algorithm must first invokes the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$ protocol to transform an additive shared y to a shared bit string which is the binary representation of y. Because of $x < n_x, y < n_y$, the answer of x^y is in $\mathbb{Z}_{{n_x}^{n_y}}$. Therefore, to ensure the correctness all the steps of the protocol must be done in $\mathbb{Z}_{{n_x}^{n_y}}$. More specifically, we need $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$ and $\{k_x\}\mathbb{Z}_2$-to-$\mathbb{Z}_{n^n}$ protocols to transform the base x from $\mathbb{Z}_n$ shares to $\mathbb{Z}_{{n_x}^{n_y}}$ shares. Besides that, $k_y + 1$ times of the $\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^{n_y}}\{0\}$ protocol are necessary to transform the shared bits of the exponent y from $\mathbb{Z}_2$ to $\mathbb{Z}_{{n_x}^{n_y}}$ shares. The If-Then-Else, Square, and Product protocols are required as well to complete this algorithm. Figure 8 is the construction of the Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_{{n_x}^{n_y}}$ protocol.

Specification 6.1 (Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_{{n_x}^{n_y}}$)
Party 1 and Party 2 share the base and the exponent. They want to securely execute the protocol $((x_1, y_1), (x_2, y_2))\{n_x, n_y\} \mapsto (z_1, z_2)$ *such that* $z_1 + z_2 = (x_1 + x_2)^{y_1+y_2}$*, where* $z_1, z_2 \in \mathbb{Z}_{{n_x}^{n_y}}$*,* n_x, n_y *are public ranges of x and y respectively. More specifically,* $x < n_x \leq n$*, and* $y < n_y \leq n$*.*

Algorithm 12 Calculate $u = x^y$
```
$(b^k b^{k-1} \cdots b^1 b^0)_2 \leftarrow y$
$u \leftarrow 1$
if $b^0 = 1$ then
    $u \leftarrow x$
end if
$v \leftarrow x$
for $i = 1, 2, \cdots, k$ do
    $v = v^2$
    if $b^i = 1$ then
        $u = u \cdot v$
    end if
end for
```

PROTOCOL Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_{{n_x}^{n_y}}$

1. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^{n_y}}\{k_x\}$ protocol,
$$(x_1, x_2)\{k_x\} \mapsto ((e_1^0, \ldots, e_1^{k_x}), (e_2^0, \ldots, e_2^{k_x})) \mapsto (X_1, X_2),$$
where $X_1 + X_2 \pmod{{n_x}^{n_y}} = x_1 + x_2 \pmod{n}$.
2. Party 1 and Party 2 jointly execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$ protocol $(y_1, y_2)\{k_y\} \mapsto ((b_1^0, \ldots, b_1^{k_y}), (b_2^0, \ldots, b_2^{k_y}))$ such that $(b^{k_y} b^{k_y-1} \cdots b^1 b^0)_2 = y_1 + y_2$, where $y_1, y_2 \in \mathbb{Z}_{n_y}$, $b_1^i, b_2^i \in \mathbb{Z}_2$, and $b^i = b_1^i + b_2^i \pmod 2$, for $i = 0, 1, \cdots, k_y$.
3. Party 1 and Party 2 jointly execute the $\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^{n_y}}\{0\}$ protocol $(b_1^0, b_2^0)\{0\} \mapsto (d_1^0, d_2^0)$ such that $(b^0)_2 = d_1^0 + d_2^0 \pmod{{n_x}^{n_y}}$, where $d_1^0, d_2^0 \in \mathbb{Z}_{{n_x}^{n_y}}$, $b_1^0, b_2^0 \in \mathbb{Z}_2$, and $b^0 = b_1^0 + b_2^0 \pmod 2$.
4. Party 1 and Party 2 jointly execute the If-Then-Else protocol $((d_1^0, X_1, 1), (d_2^0, X_2, 0)) \mapsto (u_1^1, u_2^1)$ such that $u_1^1 + u_2^1 \pmod{{n_x}^{n_y}} =$
$$\begin{cases} X_1 + X_2 & \text{if} \quad d_1^0 + d_2^0 = 1, \\ 1 & \text{if} \quad d_1^0 + d_2^0 = 0. \end{cases}$$
It is clear from the context that + represents addition in $\mathbb{Z}_{n_x^{n_y}}$.
5. Party j individually sets $v_j^1 = X_j$, for $j = 1, 2$.
6. For $i = 1, 2, \cdots, k_y$, repeat from Step 6a to Step 6d.
    (a) The two parties jointly execute the Square protocol $(v_1^i, v_2^i) \mapsto (v_1^{i+1}, v_2^{i+1})$ such that $v_1^{i+1} + v_2^{i+1} \pmod{{n_x}^{n_y}} = (v_1^i + v_2^i)^2$.
    (b) The two parties jointly execute the Product protocol $((u_1^i, v_1^i), (u_2^i, v_2^i)) \mapsto (w_1^i, w_2^i)$ such that $w_1^i + w_2^i \pmod{{n_x}^{n_y}} = (u_1^i + u_2^i)(v_1^i + v_2^i)$.
    (c) Party 1 and Party 2 jointly execute the $\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^{n_y}}\{0\}$ protocol $(b_1^i, b_2^i)\{0\} \mapsto (d_1^i, d_2^i)$ such that $(b^i)_2 = d_1^i + d_2^i \pmod{{n_x}^{n_y}}$, where $d_1^i, d_2^i \in \mathbb{Z}_{{n_x}^{n_y}}$, $b_1^i, b_2^i \in \mathbb{Z}_2$, and $b^i = b_1^i + b_2^i \pmod 2$.

(d) The two parties jointly execute the If-Then-Else protocol $((d_1^i, w_1^i, u_1^i), (d_2^i, w_2^i, u_2^i)) \mapsto (u_1^{i+1}, u_2^{i+1})$ such that

$$u_1^{i+1}+u_2^{i+1} \pmod{{n_x}^{n_y}} = \begin{cases} w_1^i + w_2^i & \text{if} \quad d_1^i + d_2^i = 1, \\ u_1^i + u_2^i & \text{if} \quad d_1^i + d_2^i = 0. \end{cases}$$

It is clear from the context that + represents addition in $\mathbb{Z}_{{n_x}^{n_y}}$.

7. Party j individually sets $z_j = u_j^{k_y+1}$, for $j = 1, 2$.

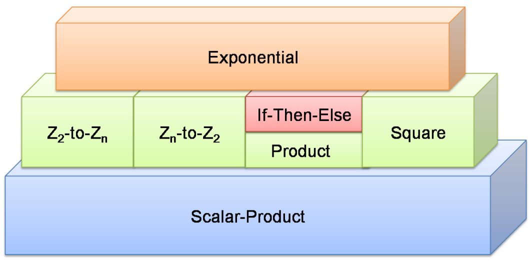


Figure 8. The Composition of the Exponentiation protocol

There are situations where the final results of the exponentiation are bounded above by numbers much smaller than n^n; here we consider the case where the result is guaranteed to be in Z_n. The corresponding protocol is described below:

Specification 6.2 (Exp(sb, se) $\in \mathbb{Z}_n$) *Party 1 and Party 2 share the base and the exponent. They want to securely execute the protocol* $((x_1, y_1), (x_2, y_2)) \mapsto (z_1, z_2)$ *such that* $z_1 + z_2 = (x_1 + x_2)^{y_1+y_2}$.

PROTOCOL Exp(sb, se) $\in \mathbb{Z}_n$

1. Party j individually sets $X_j = x_j$, for $j = 1, 2$.
2. Party 1 and Party 2 jointly execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol $(y_1, y_2) \mapsto ((b_1^0, \dots, b_1^k), (b_2^0, \dots, b_2^k))$ such that $(b^k b^{k-1} \cdots b^1 b^0)_2 = y_1 + y_2$, where $y_1, y_2 \in \mathbb{Z}_n$, $b_1^i, b_2^i \in \mathbb{Z}_2$, and $b^i = b_1^i + b_2^i \pmod 2$, for $i = 0, 1, \cdots, k$.
3. Party 1 and Party 2 jointly execute the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^0, b_2^0) \mapsto (d_1^0, d_2^0)$ such that $(b^0)_2 = d_1^0 + d_2^0$, where $d_1^0, d_2^0 \in \mathbb{Z}_n$, $b_1^0, b_2^0 \in \mathbb{Z}_2$, and $b^0 = b_1^0 + b_2^0 \pmod 2$.
4. Party 1 and Party 2 jointly execute the If-Then-Else protocol $((d_1^0, X_1, 1), (d_2^0, X_2, 0)) \mapsto (u_1^1, u_2^1)$ such that

$$u_1^1 + u_2^1 \pmod n = \begin{cases} X_1 + X_2 & \text{if} \quad d_1^0 + d_2^0 = 1, \\ 1 & \text{if} \quad d_1^0 + d_2^0 = 0. \end{cases}$$

5. Party j individually sets $v_j^1 = X_j$, for $j = 1, 2$.
6. For $i = 1, 2, \cdots, k$, repeat from Step 6a to Step 6d.

(a) The two parties jointly execute the Square protocol $(v_1^i, v_2^i) \mapsto (v_1^{i+1}, v_2^{i+1})$ such that $v_1^{i+1} + v_2^{i+1} \pmod n = (v_1^i + v_2^i)^2$.

(b) The two parties jointly execute the Product protocol $((u_1^i, v_1^i), (u_2^i, v_2^i)) \mapsto (w_1^i, w_2^i)$ such that $w_1^i + w_2^i \pmod n = (u_1^i + u_2^i)(v_1^i + v_2^i)$.

(c) Party 1 and Party 2 jointly execute the $\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$ protocol $(b_1^i, b_2^i) \mapsto (d_1^i, d_2^i)$ such that $(b^i)_2 = d_1^i + d_2^i$, where $d_1^i, d_2^i \in \mathbb{Z}_n$, $b_1^i, b_2^i \in \mathbb{Z}_2$, and $b^i = b_1^i + b_2^i \pmod 2$.

(d) The two parties jointly execute the If-Then-Else protocol $((d_1^i, w_1^i, u_1^i), (d_2^i, w_2^i, u_2^i)) \mapsto (u_1^{i+1}, u_2^{i+1})$ such that

$$u_1^{i+1}+u_2^{i+1} \pmod n = \begin{cases} w_1^i + w_2^i & \text{if} \quad d_1^i + d_2^i = 1, \\ u_1^i + u_2^i & \text{if} \quad d_1^i + d_2^i = 0. \end{cases}$$

7. Party j individually sets $z_j = u_j^{k+1}$, for $j = 1, 2$.

For some other situations where either the value of x or y can be revealed completely, we can then take full advantage of it and have much faster protocols.

1. Exp(pb x, se) $\in \mathbb{Z}_{x^n}$: Since the base resides in plain view on these two parties as x and the exponent is still securely shared in $\mathbb{Z}_n$, the answer of x^y is in $\mathbb{Z}_{x^n}$. We can omit the step transforming the base from $\mathbb{Z}_n$ shares to $\mathbb{Z}_{x^n}$ ones. At the same time, securely executing the Square protocol for x^2 is not necessary anymore.

2. Exp(sb, pe y) $\in \mathbb{Z}_{n^y}$: Since the exponent resides in plain view on these two parties as y and the base is still securely shared in $\mathbb{Z}_n$, the answer of x^y is in $\mathbb{Z}_{n^y}$. Party 1 and Party 2 can individually transform y to a bit string. Details of this protocol are provided as followings.

Specification 6.3 (Exp(sb, pe y) $\in \mathbb{Z}_{n^y}$) *Party 1 and Party 2 share the base while the exponent is public. They want to securely execute the protocol* $(x_1, x_2)\{y\} \mapsto (z_1, z_2)$ *such that* $z_1 + z_2 = (x_1 + x_2)^y$, *where* $z_1, z_2 \in \mathbb{Z}_{n^y}$.

PROTOCOL Exp(sb, pe y) $\in \mathbb{Z}_{n^y}$

(a) Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n^y}\{k\}$ protocol,

$$(x_1, x_2) \mapsto ((e_1^0, \dots, e_1^k), (e_2^0, \dots, e_2^k)) \mapsto (X_1, X_2),$$

where $X_1 + X_2 \pmod{n^y} = x_1 + x_2 \pmod n$.

(b) Party 1 and Party 2 individually turn the public exponent y from $\mathbb{Z}_n$ into $\mathbb{Z}_2$ such that $(b^{k_y} b^{k_y-1} \cdots b^1 b^0)_2 = y$, where $k_y = \lfloor \log_2 y \rfloor$, $y \in \mathbb{Z}_n$, $b^i \in \mathbb{Z}_2$, for $i = 0, 1, \cdots, k_y$.

(c) Party 1 sets $u_1^1 = b^0 X_1 + (1 - b^0)$, while Party 2 sets $u_2^1 = b^0 X_2$, such that

$$u_1^1 + u_2^1 \pmod{n^y} = \begin{cases} X_1 + X_2 & \text{if } b^0 = 1, \\ 1 & \text{if } b^0 = 0. \end{cases}$$

It is clear from the context that + represents addition in $\mathbb{Z}_{n^y}$.

(d) Party j individually sets $v_j^1 = X_j$, for $j = 1, 2$.

(e) For $i = 1, \cdots, k_y$, repeat Step 2(e)i and Step 2(e)ii.

i. The two parties jointly execute the Square protocol $(v_1^i, v_2^i) \mapsto (v_1^{i+1}, v_2^{i+1})$ such that $v_1^{i+1} + v_2^{i+1} \pmod{n^y} = (v_1^i + v_2^i)^2$. It is clear from the context that + represents addition in $\mathbb{Z}_{n^y}$.

ii. If $b^i = 1$, the two parties jointly execute the Product protocol $((u_1^i, v_1^i), (u_2^i, v_2^i)) \mapsto (u_1^{i+1}, u_2^{i+1})$ such that $u_1^{i+1} + u_2^{i+1} \pmod{n^y} = (u_1^i + u_2^i)(v_1^i + v_2^i)$. If $b^i = 0$, Party j individually sets $u_j^{i+1} = u_j^i$, for $j = 1, 2$. It is clear from the context that + represents addition in $\mathbb{Z}_{n^y}$.

(f) Party j individually sets $z_j = u_j^{k_y+1}$, for $j = 1, 2$.

There are other tradeoffs. For example, keep the exponent in shares; at the same time reveal the base and the upper bounds of the exponent. Or on the one hand, keep the base in shares; on the other hand, disclose both the exponent and the upper bounds of the base. The former and the latter are represented as Exp(pb x, se, n_y) $\in \mathbb{Z}_{x^{n_y}}$ and Exp(sb, pe y, n_x) $\in \mathbb{Z}_{{n_x}^y}$ respectively.

1. Exp(pb x, se, n_y) $\in \mathbb{Z}_{x^{n_y}}$: This is quite the same as Exp(pb x, se) $\in \mathbb{Z}_{x^n}$ except the domain of the computation reduced from $\mathbb{Z}_{x^n}$ to $\mathbb{Z}_{x^{n_y}}$.

2. Exp(sb, pe y, n_x) $\in \mathbb{Z}_{{n_x}^y}$: This one is supposed have better performance than Exp(sb, pe y) $\in \mathbb{Z}_{n^y}$ since the domain of the computation reduced from $\mathbb{Z}_{n^y}$ to $\mathbb{Z}_{{n_x}^y}$.

Then we enumerate two situations which may indirectly disclose some information.

1. Exp(pb x, se) $\in \mathbb{Z}_n$: Once the base x is not a secret and the answer of x^y is guaranteed in $\mathbb{Z}_n$, the range of y is no longer in n but limited to $\log_x n$, if $x \neq 1$.

$$x^y < n \Rightarrow y < \log_x n.$$

2. Exp(sb, pe y) $\in \mathbb{Z}_n$: Much the same, when it comes to a public exponent y and a guaranteed answer in $\mathbb{Z}_n$, the range of x is constrained to $\lfloor \sqrt[y]{n} \rfloor$ rather than supposed n.

$$x^y < n \Rightarrow x < \lfloor \sqrt[y]{n} \rfloor.$$

Note that hereafter in this paper, we present the worst cases of those protocols with *pe*. More specifically, we view every bit of the *public* exponent y as 1. That is $y = 2^{k_y+1} - 1$, where $k_y = \lfloor \log_2 y \rfloor$. We break the aforementioned exponentiation protocols into their constitutive building blocks of which Table 6 lists the domain and times.

Based on Table 1 and Table 2, we further break the constitutive building blocks of those exponentiation protocols in Table 6 into Scalar-Product protocols of which Table 8 lists the domain, dimension, and times.

6.2 Tradeoffs: Division

Specification 6.4 (Div/Rem $\{k_x, k_y\}$) *Two parties share the dividend and the divisor in $\mathbb{Z}_n$, and they want to jointly execute the secure protocol $((x_1, y_1), (x_2, y_2))\{k_x, k_y\} \mapsto ((q_1, r_1), (q_2, r_2))$, where $n_x = 2^{k_x+1} > (x_1 + x_2) = (y_1 + y_2)(q_1 + q_2) + (r_1 + r_2)$ and $0 \leq (r_1 + r_2) < (y_1 + y_2) < 2^{k_y+1} = n_y$.*

PROTOCOL Div/Rem $\{k_x, k_y\}$

1. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n_x \cdot n_y}\{k_x\}$ protocol

$$(x_1, x_2) \mapsto ((b_1^0, \dots, b_1^{k_x}), (b_2^0, \dots, b_2^{k_x})) \mapsto (X_1, X_2)$$

such that $X_1 + X_2 \pmod{n_x \cdot n_y} = x_1 + x_2 \pmod{n}$.

2. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n_x \cdot n_y}\{k_y\}$ protocol

$$(y_1, y_2) \mapsto ((c_1^0, \dots, c_1^{k_y}), (c_2^0, \dots, c_2^{k_y})) \mapsto (Y_1, Y_2)$$

such that $Y_1 + Y_2 \pmod{n_x \cdot n_y} = y_1 + y_2 \pmod{n}$.

3. Party j sets $X_j^{k_x} = X_j$, for $j = 1, 2$.

4. For $i = k_x - 1, \dots, 1, 0$, repeat Step 4a to Step 4d.

(a) Party j computes $t_j^i = X_j^{i+1} - Y_j \cdot 2^i \pmod{n_x \cdot n_y}$, for $j = 1, 2$.

(b) Party 1 and Party 2 jointly run the Comparison protocol $(t_1^i, t_2^i) \mapsto (s_1^i, s_2^i)$ such that

$$s_1^i + s_2^i \pmod{n_x \cdot n_y} = \begin{cases} 1 & \text{if } t_1^i + t_2^i < 0, \\ 0 & \text{otherwise.} \end{cases}$$

(c) Party j individually computes $q_j^i = 1 - s_j^i \pmod{n_x \cdot n_y}$, for $j = 1, 2$, such that

$$q_1^i + q_2^i \pmod{n_x \cdot n_y} = \begin{cases} 0 & \text{if } s_1^i + s_2^i = 1, \\ 1 & \text{if } s_1^i + s_2^i = 0. \end{cases}$$

(d) Party 1 and Party 2 run the If-Then-Else protocol $((s_1^i, X_1^{i+1}, t_1^i), (s_2^i, X_2^{i+1}, t_2^i)) \mapsto (X_1^i, X_2^i)$ such that $X_1^i + X_2^i \pmod{n_x \cdot n_y} =$

$$\begin{cases} X_1^{i+1} + X_2^{i+1} & \text{if } s_1^i + s_2^i = 1, \\ t_1^i + t_2^i & \text{if } s_1^i + s_2^i = 0. \end{cases}$$

5. For $j = 1, 2$, Party j computes $r_j = X_j^0 \pmod n$, and $q_j = \sum_{i=0}^{k-1} q_j^i \cdot 2^i \pmod n$.

The two parties can agree on revealing the range of the dividend or the range of the divisor to get better efficiency. We list the specification and details for these two protocols in the following.

Specification 6.5 (Div/Rem$\{$divisor $k_y\}$) *Two parties share the dividend in $\mathbb{Z}_n$, and they want to jointly execute the secure protocol $((x_1, y_1), (x_2, y_2))\{k_y\} \mapsto ((q_1, r_1), (q_2, r_2))$, where $x_1 + x_2 = (y_1 + y_2)(q_1 + q_2) + (r_1 + r_2)$ and $0 \le (r_1 + r_2) < (y_1 + y_2) \pmod n < 2^{k_y+1}$.*

PROTOCOL Div/Rem$\{$divisor $k_y\}$

1. Party j individually sets $Y_j = y_j, X_j^{k-k_y} = x_j$ where $k_y = \lfloor \log_2 y \rfloor$, for $j = 1, 2$.
2. For $i = k - k_y - 1, k - k_y - 2, \dots, 0$, repeat Step 2a to Step 2d.
   (a) Party 1 individually computes $t_1^i = X_1^{i+1} - Y_j \cdot 2^i$ while Party 2 individually sets $t_2^i = X_2^{i+1}$.
   (b) Party 1 and Party 2 jointly run the Comparison protocol $(t_1^i, t_2^i) \mapsto (s_1^i, s_2^i)$ such that
   $$s_1^i + s_2^i = \begin{cases} 1 & \text{if } t_1^i + t_2^i < 0, \\ 0 & \text{otherwise.} \end{cases}$$
   (c) For $j = 1, 2$, Party j individually computes $q_j^i = 1 - s_j^i$ such that
   $$q_1^i + q_2^i = \begin{cases} 0 & \text{if } s_1^i + s_2^i = 1, \\ 1 & \text{if } s_1^i + s_2^i = 0. \end{cases}$$
   (d) Party 1 and Party 2 run the If-Then-Else protocol $((s_1^i, X_1^{i+1}, t_1^i), (s_2^i, X_2^{i+1}, t_2^i)) \mapsto (X_1^i, X_2^i)$ such that
   $$X_1^i + X_2^i = \begin{cases} X_1^{i+1} + X_2^{i+1} & \text{if } s_1^i + s_2^i = 1, \\ t_1^i + t_2^i & \text{if } s_1^i + s_2^i = 0. \end{cases}$$
3. For $j = 1, 2$, Party j computes $r_j = X_j^0 \pmod n$, and $q_j = \sum_{i=0}^{k-1} q_j^i \cdot 2^i \pmod n$.

Specification 6.6 (Div/Rem$\{$dividend $k_x\}$) *Two parties share the divisor in $\mathbb{Z}_n$, and they want to jointly execute the secure protocol $((x_1, y_1), (x_2, y_2))\{k_x\} \mapsto ((q_1, r_1), (q_2, r_2))$, where $n_x = 2^{k_x+1} > (x_1 + x_2) = (y_1 + y_2)(q_1 + q_2) + (r_1 + r_2)$ and $0 \le (r_1 + r_2) < (y_1 + y_2)$.*

PROTOCOL Div/Rem$\{$dividend $k_x\}$

1. Party 1 and Party 2 collaboratively execute the $\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$ protocol followed by the $\mathbb{Z}_2$-to-$\mathbb{Z}_{n_x \cdot n}\{k\}$ protocol
   $$(y_1, y_2) \mapsto ((c_1^0, \dots, c_1^k), (c_2^0, \dots, c_2^k)) \mapsto (Y_1, Y_2)$$
   such that $Y_1 + Y_2 \pmod{n_x \cdot n} = y_1 + y_2 \pmod n$.
2. Party 1 individually sets $X_1^{k_x} = x$ while Party 2 individually sets $X_2^{k_x} = 0$.
3. For $i = k_x - 1, \dots, 1, 0$, repeat Step 3a to Step 3d.
   (a) Party j computes $t_j^i = X_j^{i+1} - Y_j \cdot 2^i \pmod{n_x \cdot n}$, for $j = 1, 2$.
   (b) Party 1 and Party 2 jointly run the Comparison protocol $(t_1^i, t_2^i) \mapsto (s_1^i, s_2^i)$ such that
   $$s_1^i + s_2^i \pmod{n_x \cdot n} = \begin{cases} 1 & \text{if } t_1^i + t_2^i < 0, \\ 0 & \text{otherwise.} \end{cases}$$
   (c) For $j = 1, 2$, Party j individually computes $q_j^i = 1 - s_j^i \pmod{n_x \cdot n}$ such that
   $$q_1^i + q_2^i \pmod{n_x \cdot n} = \begin{cases} 0 & \text{if } s_1^i + s_2^i = 1, \\ 1 & \text{if } s_1^i + s_2^i = 0. \end{cases}$$
   (d) Party 1 and Party 2 run the If-Then-Else protocol $((s_1^i, X_1^{i+1}, t_1^i), (s_2^i, X_2^{i+1}, t_2^i)) \mapsto (X_1^i, X_2^i)$ such that $X_1^i + X_2^i \pmod{n_x \cdot n} =$
   $$\begin{cases} X_1^{i+1} + X_2^{i+1} & \text{if } s_1^i + s_2^i = 1, \\ t_1^i + t_2^i & \text{if } s_1^i + s_2^i = 0. \end{cases}$$
4. For $j = 1, 2$, Party j computes $r_j = X_j^0 \pmod n$, and $q_j = \sum_{i=0}^{k-1} q_j^i \cdot 2^i \pmod n$.

We break the aforementioned division protocols into their constitutive building blocks of which Table 9 lists the domain and times.

Based on Table 1 and Table 2, we further break the constitutive building blocks of those division protocols in Table 9 into Scalar-Product protocols of which Table 10 lists the domain, dimension, and times.

7 Performance Evaluation

No doubt, there are tradeoffs between privacy and efficiency. Nevertheless, making an adequate decision is never an easy task without more evidence. Below we provide methods estimating the time cost for each protocol based on the Scalar-Product.

We adopt the secure scalar product protocol that Du and Zhan proposed, namely, the commodity-based approach [7], which is based on Beaver's commodity model [8]. This approach has extraordinary performance among several secure scalar product ones, though a neutral third party, the commodity server, is needed [10]. We implemented this scalar product protocol and all of our building blocks in Ruby(1.8.6 patchlevel 111). Table 11 shows the environment of our experiments. To minimize the probabilistic variation, our experimental results are the average of 100 effective executions. There are two parts of the experiments:

1. We focus on the execution time of the Scalar-Product protocol with different dimensions and a different number of bits of domain. Table 12 shows the experimental results. Recall that $2^{k+1} = n$, and with given k in this table, timings are not apparently different when it comes to different dimension $d \leq 32$. Therefore, we simply set $d = 1$ and find the time estimation function $EF(\cdot)$ which is a polynomial curve fitting our experimental results (see Figure 9).

   $$EF(k) = 2 \cdot 10^{-10}k^2 - 5 \cdot 10^{-8}k + 0.0566,$$
   $$R^2 = 0.9993.$$

   Then, given two necessary parameters for the Scalar-Product, dimension d and the number of bits of n (that is $k+1$), we have two ways to estimate the time cost. First, with extrapolation we can look up the prepared experimental data, like Table 12. Second, when $d \leq 32$, we can easily use $EF(k)$ to estimate the time.

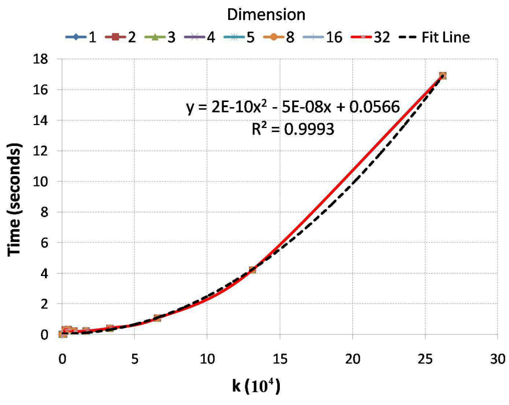


Figure 9. The timings of the commodity-based Scalar-Product protocol with different k and different dimensions.

2. We concentrate on the execution time of the Exp(sb, se) $\in \mathbb{Z}_{n^n}$ protocol with a different number of bits of n. At the same time, we estimate the time cost by using Table 8 and $EF(\cdot)$. Both the experimental results and the estimated time are listed in Table 13. As this table shown, it is convincing that $EF(\cdot)$ is a useful function which helps estimate the execution time of protocols composed of the Scalar-Product.

   Again, we use Table 8 and $EF(\cdot)$ to construct Table 14, showing the estimated time of almost all possible tradeoffs of the exponentiation. Recall that $n_x = 2^{k_x+1}, n_y = 2^{k_y+1}, n = 2^{k+1}$; for simplicity, we set $x = 2^{k_x+1}, y = 2^{k_y+1}$, where $k_x, k_y = \lfloor \frac{k}{2} \rfloor$. We can see that the time to compute the exponentiation may range from seconds to centuries, depending on the degree of information one is willing to reveal.

8 Conclusions and Future Works

A set of information theoretically secure two-party protocols have been developed based on scalar product. The ultimate goal to design such protocols is to build a compiler for secure multiparty computation environments. The protocols presented in this paper is part of protocol along this line. More protocols need to be designed to achieve the ultimate goal.

Although the pursuit of efficient general solutions to SMC problems is admirable, a more modest but likely more successful pursuit is to carefully consider tradeoff options during the protocol design phase. In this paper, protocols for secure two-party computation of exponentiation were designed based on the repeated squaring algorithm using scalar products as building blocks. A careful performance evaluation was carried out in which an analysis of the execution time for our protocols was conducted by breaking them into their constitutive building blocks and estimating the execution times using the performance evaluation results of our implemented Scalar-Product protocol. We noted that the estimated execution times were comparable to those we measured in the experimental runs of the protocols. This suggests that using scalar products as building blocks may make execution time estimation easier.

The results of this evaluation shed some light on the tradeoffs between necessary execution time and the amount of private information revealed as well as suggest future research directions. Within the commodity-based secure scalar product protocol, the results demonstrated that certain kinds of calculations, exponentiation using the repeated squaring algorithm in this case, become infeasible if no information of the shared base and the exponent, except they are in $\mathbb{Z}_n$, is provided. Future research directions include designing more efficient protocols for cases where such secrecy is mandatory. Other kinds of calculations should also be considered. Exponentiation was discussed in this paper since it is widely used for various applications in many disciplines. Lastly, the proposed protocols can handle only integers. Hence we are also eager to extend them to handle floating point numbers so that we can apply our work to more real world applications.

References

[1] A. C. Yao, "Protocols for secure computation," in *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, November 1982, pp. 160–164.

[2] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *STOC '87: Proceedings of the 19th Annual ACM Symposium on Theory of Com-*

puting. New York, NY, USA: ACM Press, 1987, pp. 218–229.

[3] Committee on National Statistics, "Improving access to and confidentiality of research data: Report of a workshop," in *NRC '00: National Research Council*, C. Mackie and N. Bradburn, Eds., Commission on Behavioral and Social Sciences and Education. Washington, D.C.: National Academy Press, 2000.

[4] J. Kilian, "Founding cryptography on oblivious transfer," in *STOC '88: Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1988, pp. 20–31.

[5] ——, "A general completeness theorem for two party games," in *STOC '91: Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 1991, pp. 553–560.

[6] W. Du, "A study of several specific secure two-party computation problems," Ph.D. dissertation, Purdue University, August 2001.

[7] W. Du and Z. Zhan, "A practical approach to solve secure multi-party computation problems," in *NSPW '02: Proceedings of the 2002 Workshop on New Security Paradigms*. New York, NY, USA: ACM Press, 2002, pp. 127–135.

[8] D. Beaver, "Commodity-based cryptography (extended abstract)," in *STOC '97: Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM Press, 1997, pp. 446–455.

[9] B. Goethals, S. Laur, H. Lipmaa, and T. Mielikainen, "On private scalar product computation for privacy-preserving data mining," *Information Security and Cryptology V ICISC 2004*, pp. 104–120, 2004.

[10] I.-C. Wang, C.-H. Shen, T.-S. Hsu, C.-C. Liau, D.-W. Wang, and J. Zhan, "Towards empirical aspects of secure scalar product," in *ISA '08: IEEE International Conference on Information Security and Assurance*, April 2008, pp. 573–578.

[11] M. J. Atallah and W. Du, "Secure multi-party computational geometry," *Algorithms and Data Structures, Lecture Notes in Computer Science*, vol. 2125, pp. 165–179, 2000.

[12] W. Du and M. J. Atallah, "Privacy-preserving cooperative statistical analysis," in *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 102–110.

[13] ——, "Privacy-preserving cooperative scientific computations," in *CSFW '01: Proceedings of the 14th IEEE Workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 273–282.

[14] P. Bunn and R. Ostrovsky, "Secure two-party k-means clustering," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2007, pp. 486–497.

[15] J. Zhan, I.-C. Wang, C.-L. Hsieh, T.-S. Hsu, C.-J. Liau, and D.-W. Wang, "Towards efficient privacy-preserving collaborative recommender systems," in *GrC '08: IEEE International Conference on Granular Computing*, Aug. 2008, pp. 778–783.

[16] Y.-T. Chiang, D.-W. Wang, C.-J. Liau, and T.-S. Hsu, "Secrecy of two-party secure computation," *Data and Applications Security XIX, Lecture Notes in Computer Science*, vol. 3654, pp. 114–123, 2005.

[17] D.-W. Wang, C.-J. Liau, Y.-T. Chiang, and T.-S. Hsu, "Information theoretical analysis of two-party secret computation," *Data and Applications Security XX, Lecture Notes in Computer Science*, vol. 4127, pp. 310–317, 2006.

[18] C.-H. Shen, J. Zhan, D.-W. Wang, T.-S. Hsu, and C.-J. Liau, "Information-theoretically secure number-product protocol," in *ICMLC '07: International Conference on Machine Learning and Cybernetics*, vol. 5, 19-22 Aug. 2007, pp. 3006–3011.

[19] J. Algesheimer, J. Camenisch, and V. Shoup, "Efficient computation modulo a shared secret with application to the generation of shared safe-prime products," *Advances in Cryptology X CRYPTO 2002*, vol. 2442/2002, 2002.

[20] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *TCC '06: Proceedings of the 3rd Theory of Cryptography Conference*, 2006, pp. 285–304.

[21] J. D. Nielsen, "Languages for secure multiparty computation and towards strongly typed macros," Ph.D. dissertation, Department of Computer Science, University of Aarhus Denmark, February 2009.

[22] O. Goldreich, *Foundations of Cryptography, Volume II Basic Applications*, 1st ed. Cambridge University Press, 2004.

[23] R. Canetti, “Security and composition of multiparty cryptographic protocols,” *Journal of Cryptology*, vol. 13, pp. 143–202, 2000.

[24] C.-H. Shen, J. Zhan, T.-S. Hsu, C.-J. Liau, and D.-W. Wang, “Scalar-product based secure two-party computation,” in *GrC ’08: IEEE International Conference on Granular Computing*, Aug. 2008, pp. 556–561.

[25] J. D. Nielsen and M. I. Schwartzbach, “A domain-specific programming language for secure multiparty computation,” in *PLAS ’07: Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM, 2007, pp. 21–30.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, Massachusetts: The MIT Press, 2001.

Protocol	Domain	Constituents	Times
$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k'\}$	$\mathbb{Z}_2$	Scalar-Product	k'
$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{k'\}$	$\mathbb{Z}_n$	Scalar-Product	1
Product	$\mathbb{Z}_n$	Scalar-Product	1
Square	$\mathbb{Z}_n$	Scalar-Product	1
Comparison	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	1
Zero	$\mathbb{Z}_n$	Product	1
	$\mathbb{Z}_n$	Comparison	2
If-Then-Else	$\mathbb{Z}_n$	Product	1
Shift-Left	$\mathbb{Z}_n$	Exp(pb, se, $n_y) \in \mathbb{Z}_n$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{k\}$	1
	$\mathbb{Z}_n$	Product	1
	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
Shift-Right	$\mathbb{Z}_n$	Shift-Left	1
Shift$\{s\}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{k-s\}$	1
Rotate-Left	$\mathbb{Z}_n$	Exp(pb, se, $n_y) \in \mathbb{Z}_n$	1
	$\mathbb{Z}_{n^2}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^2}\{k\}$	2
	$\mathbb{Z}_{n^2}$	Product	1
	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_2$	$\mathbb{Z}_{n^2}$-to-$\mathbb{Z}_2\{2k+1\}$	1
Rotate-Right	$\mathbb{Z}_n$	Rotate-Left	1
Log	$\mathbb{Z}_n$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	k
	$\mathbb{Z}_n$	If-Then-Else	$k-1$
Div/Rem	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	2
	$\mathbb{Z}_{n^2}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^2}\{k\}$	2
	$\mathbb{Z}_{n^2}$	Comparison	k
	$\mathbb{Z}_{n^2}$	If-Then-Else	k
Function(x)	$\mathbb{Z}_n$	Scalar-Product	1
Function(x, y)	$\mathbb{Z}_n$	Scalar-Product	$2^{k+1}+1$
(flo) Scalar-Product	$\mathbb{Z}_n$	Scalar-Product	1
	$\mathbb{Z}_n$	Shift	1
(flo) Square	$\mathbb{Z}_n$	Scalar-Product	1
	$\mathbb{Z}_n$	Shift	1
(flo) Product	$\mathbb{Z}_n$	Scalar-Product	1
	$\mathbb{Z}_n$	Shift	1

Table 1. Constituents of our protocols

Protocol	Domain	Constituents	Times
(secret) Max$\{d\}$	$\mathbb{Z}_n$	Comparison	$d-1$
	$\mathbb{Z}_n$	If-Then-Else	$d-1$
(secret) Min$\{d\}$	$\mathbb{Z}_n$	Comparison	$d-1$
	$\mathbb{Z}_n$	If-Then-Else	$d-1$
(secret) Range$\{d\}$	$\mathbb{Z}_n$	(secret) Max	1
	$\mathbb{Z}_n$	(secret) Min	1
(secret) Var$\{d\}$	$\mathbb{Z}_n$	Scalar-Product	1
	$\mathbb{Z}_n$	Square	1
	$\mathbb{Z}_n$	Div/Rem	1
(private) Max	$\mathbb{Z}_n$	Comparison	1
	$\mathbb{Z}_n$	If-Then-Else	1
(private) Min	$\mathbb{Z}_n$	Comparison	1
	$\mathbb{Z}_n$	If-Then-Else	1
(private) Range	$\mathbb{Z}_n$	(private) Max	1
	$\mathbb{Z}_n$	(private) Min	1
(private) Var	$\mathbb{Z}_n$	Product	1
	$\mathbb{Z}_n$	Square	2
	$\mathbb{Z}_n$	Div/Rem	1

Table 3. Constituents of the statistic protocols

Protocol	Domain	Dimension	Times
(secret) Max$\{d\}$	$\mathbb{Z}_2$	3	$k(d-1)$
	$\mathbb{Z}_n$	1	$(d-1)$
	$\mathbb{Z}_n$	2	$(d-1)$
(secret) Min$\{d\}$	$\mathbb{Z}_2$	3	$k(d-1)$
	$\mathbb{Z}_n$	1	$(d-1)$
	$\mathbb{Z}_n$	2	$(d-1)$
(secret) Range$\{d\}$	$\mathbb{Z}_2$	3	$2k(d-1)$
	$\mathbb{Z}_n$	1	$2(d-1)$
	$\mathbb{Z}_n$	2	$2(d-1)$
(secret) Var$\{d\}$	$\mathbb{Z}_2$	3	$k(k-k_y)$
	$\mathbb{Z}_n$	1	$k-k_y+1$
	$\mathbb{Z}_n$	2	$k-k_y$
	$\mathbb{Z}_n$	d	1
(private) Max	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_n$	1	1
	$\mathbb{Z}_n$	2	1
(private) Min	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_n$	1	1
	$\mathbb{Z}_n$	2	1
(private) Range	$\mathbb{Z}_2$	3	$2k$
	$\mathbb{Z}_n$	1	2
	$\mathbb{Z}_n$	2	2
(private) Var	$\mathbb{Z}_2$	3	$2k^2+3k$
	$\mathbb{Z}_n$	1	1
	$\mathbb{Z}_n$	2	1
	$\mathbb{Z}_{n^2}$	1	k
	$\mathbb{Z}_{n^2}$	2	k
	$\mathbb{Z}_{n^2}$	$k+1$	2

Table 4. The Complexity of the statistic protocols

Protocol	Domain	Constituents	Times
Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_{{n_x}^{n_y}}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$	1
	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_{{n_x}^{n_y}}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^{n_y}}\{k_x\}$	1
	$\mathbb{Z}_{{n_x}^{n_y}}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^{n_y}}\{0\}$	k_y+1
	$\mathbb{Z}_{{n_x}^{n_y}}$	If-Then-Else	k_y+1
	$\mathbb{Z}_{{n_x}^{n_y}}$	Square	k_y
	$\mathbb{Z}_{{n_x}^{n_y}}$	Product	k_y
Exp(sb, se) $\in \mathbb{Z}_{n^n}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	2
	$\mathbb{Z}_{n^n}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^n}\{k\}$	1
	$\mathbb{Z}_{n^n}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^n}\{0\}$	$k+1$
	$\mathbb{Z}_{n^n}$	If-Then-Else	$k+1$
	$\mathbb{Z}_{n^n}$	Square	k
	$\mathbb{Z}_{n^n}$	Product	k
Exp(sb, se, n_x) $\in \mathbb{Z}_{{n_x}^n}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$	1
	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_{{n_x}^n}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^n}\{k_x\}$	1
	$\mathbb{Z}_{{n_x}^n}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^n}\{0\}$	$k+1$
	$\mathbb{Z}_{{n_x}^n}$	If-Then-Else	$k+1$
	$\mathbb{Z}_{{n_x}^n}$	Square	k
	$\mathbb{Z}_{{n_x}^n}$	Product	k
Exp(sb, se, n_y) $\in \mathbb{Z}_{n^{n_y}}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_{n^{n_y}}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^{n_y}}\{k\}$	1
	$\mathbb{Z}_{n^{n_y}}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^{n_y}}\{0\}$	k_y+1
	$\mathbb{Z}_{n^{n_y}}$	If-Then-Else	k_y+1
	$\mathbb{Z}_{n^{n_y}}$	Square	k_y
	$\mathbb{Z}_{n^{n_y}}$	Product	k_y
Exp(pb x, se) $\in \mathbb{Z}_{x^n}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_{x^n}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{x^n}\{0\}$	$k+1$
	$\mathbb{Z}_{x^n}$	If-Then-Else	k
Exp(sb, pe y) $\in \mathbb{Z}_{n^y}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_{n^y}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n^y}\{k\}$	1
	$\mathbb{Z}_{n^y}$	Square	k_y
	$\mathbb{Z}_{n^y}$	Product	k_y
Exp(pb x, se, n_y) $\in \mathbb{Z}_{x^{n_y}}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_{x^{n_y}}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{x^{n_y}}\{0\}$	k_y+1
	$\mathbb{Z}_{x^{n_y}}$	If-Then-Else	k_y
Exp(sb, pe y, n_x) $\in \mathbb{Z}_{{n_x}^y}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$	1
	$\mathbb{Z}_{{n_x}^y}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{{n_x}^y}\{k_x\}$	1
	$\mathbb{Z}_{{n_x}^y}$	Square	k_y
	$\mathbb{Z}_{{n_x}^y}$	Product	k_y

Table 5. Constituents of the Exponentiation tradeoff protocols

Protocol	Domain	Constituents	Times
Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	k_y+1
	$\mathbb{Z}_n$	If-Then-Else	k_y+1
	$\mathbb{Z}_n$	Square	k_y
	$\mathbb{Z}_n$	Product	k_y
Exp(sb, se) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	$k+1$
	$\mathbb{Z}_n$	If-Then-Else	$k+1$
	$\mathbb{Z}_n$	Square	k
	$\mathbb{Z}_n$	Product	k
Exp(sb, se, n_x) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	k_y+1
	$\mathbb{Z}_n$	If-Then-Else	k_y+1
	$\mathbb{Z}_n$	Square	k_y
	$\mathbb{Z}_n$	Product	k_y
Exp(sb, se, n_y) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	k_y+1
	$\mathbb{Z}_n$	If-Then-Else	k_y+1
	$\mathbb{Z}_n$	Square	k_y
	$\mathbb{Z}_n$	Product	k_y
Exp(pb x, se) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	k_y+1
	$\mathbb{Z}_n$	If-Then-Else	k_y
Exp(sb, pe y) $\in \mathbb{Z}_n$	$\mathbb{Z}_n$	Square	k_y
	$\mathbb{Z}_n$	Product	k_y
Exp(pb x, se, n_y) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_n$	$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{0\}$	k_y+1
	$\mathbb{Z}_n$	If-Then-Else	k_y
Exp(sb, pe y, n_x) $\in \mathbb{Z}_n$	$\mathbb{Z}_n$	Square	$\lfloor \log_2 y \rfloor$
	$\mathbb{Z}_n$	Product	$\lfloor \log_2 y \rfloor$

Table 6. Constituents of the Exponentiation tradeoff protocols

Protocol	Domain	Dimension	Times
Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_{{n_x}^{n_y}}$	$\mathbb{Z}_2$	3	$k_x + k_y$
	$\mathbb{Z}_{{n_x}^{n_y}}$	1	$2k_y + 1$
	$\mathbb{Z}_{{n_x}^{n_y}}$	2	$2k_y + 1$
	$\mathbb{Z}_{{n_x}^{n_y}}$	$k_x + 1$	1
Exp(sb, se) $\in \mathbb{Z}_{n^n}$	$\mathbb{Z}_2$	3	$2k$
	$\mathbb{Z}_{n^n}$	1	$2k + 1$
	$\mathbb{Z}_{n^n}$	2	$2k + 1$
	$\mathbb{Z}_{n^n}$	$k + 1$	1
Exp(sb, se, n_x) $\in \mathbb{Z}_{{n_x}^n}$	$\mathbb{Z}_2$	3	$k_x + k$
	$\mathbb{Z}_{{n_x}^n}$	1	$2k + 1$
	$\mathbb{Z}_{{n_x}^n}$	2	$2k + 1$
	$\mathbb{Z}_{{n_x}^n}$	$k_x + 1$	1
Exp(sb, se, n_y) $\in \mathbb{Z}_{n^{n_y}}$	$\mathbb{Z}_2$	3	$k + k_y$
	$\mathbb{Z}_{n^{n_y}}$	1	$2k_y + 1$
	$\mathbb{Z}_{n^{n_y}}$	2	$2k_y + 1$
	$\mathbb{Z}_{n^{n_y}}$	$k + 1$	1
Exp(pb x, se) $\in \mathbb{Z}_{x^n}$	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_{x^n}$	1	$k + 1$
	$\mathbb{Z}_{x^n}$	2	k
Exp(sb, pe y) $\in \mathbb{Z}_{n^y}$	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_{n^y}$	1	k_y
	$\mathbb{Z}_{n^y}$	2	k_y
	$\mathbb{Z}_{n^y}$	$k + 1$	1
Exp(pb x, se, n_y) $\in \mathbb{Z}_{x^{n_y}}$	$\mathbb{Z}_2$	3	k_y
	$\mathbb{Z}_{x^{n_y}}$	1	$k_y + 1$
	$\mathbb{Z}_{x^{n_y}}$	2	k_y
Exp(sb, pe y, n_x) $\in \mathbb{Z}_{{n_x}^y}$	$\mathbb{Z}_2$	3	k_x
	$\mathbb{Z}_{{n_x}^y}$	1	k_y
	$\mathbb{Z}_{{n_x}^y}$	2	k_y
	$\mathbb{Z}_{{n_x}^y}$	$k_x + 1$	1

Table 7. The Complexity of the Exponentiation tradeoff protocols

Protocol	Domain	Dimension	Times
Exp(sb, se, n_x) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	3	k_y
	$\mathbb{Z}_n$	1	$2k_y + 1$
	$\mathbb{Z}_n$	2	$2k_y + 1$
Exp(sb, se, n_y) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	3	k_y
	$\mathbb{Z}_n$	1	$2k_y + 1$
	$\mathbb{Z}_n$	2	$2k_y + 1$
Exp(sb, se, n_x, n_y) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	3	k_y
	$\mathbb{Z}_n$	1	$2k_y + 1$
	$\mathbb{Z}_n$	2	$2k_y + 1$
Exp(pb x, se) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	3	k_y
	$\mathbb{Z}_n$	1	$k_y + 1$
	$\mathbb{Z}_n$	2	k_y
Exp(pb x, se, n_y) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	3	k_y
	$\mathbb{Z}_n$	1	$k_y + 1$
	$\mathbb{Z}_n$	2	k_y
Exp(sb, pe y) $\in \mathbb{Z}_n$	$\mathbb{Z}_n$	1	k_y
	$\mathbb{Z}_n$	2	k_y
Exp(sb, pe y, n_x) $\in \mathbb{Z}_n$	$\mathbb{Z}_n$	1	k_y
	$\mathbb{Z}_n$	2	k_y
Exp(sb, se) $\in \mathbb{Z}_n$	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_n$	1	$2k + 1$
	$\mathbb{Z}_n$	2	$2k + 1$

Table 8. The Complexity of the Exponentiation tradeoff protocols

Protocol	Domain	Constituents	Times
Div/Rem $\{k_x, k_y\}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_x\}$	1
	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k_y\}$	1
	$\mathbb{Z}_{n_x \cdot n_y}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n_x \cdot n_y}\{k_x\}$	1
	$\mathbb{Z}_{n_x \cdot n_y}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n_x \cdot n_y}\{k_y\}$	1
	$\mathbb{Z}_{n_x \cdot n_y}$	Comparison	k_x
	$\mathbb{Z}_{n_x \cdot n_y}$	If-Then-Else	k_x
Div/Rem $\{$divisor $k_y\}$	$\mathbb{Z}_n$	Comparison	$k - k_y$
	$\mathbb{Z}_n$	If-Then-Else	$k - k_y$
Div/Rem $\{$dividend $k_x\}$	$\mathbb{Z}_2$	$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k\}$	1
	$\mathbb{Z}_{n \cdot n_x}$	$\mathbb{Z}_2$-to-$\mathbb{Z}_{n \cdot n_x}\{k\}$	1
	$\mathbb{Z}_{n \cdot n_x}$	Comparison	k_x
	$\mathbb{Z}_{n \cdot n_x}$	If-Then-Else	k_x

Table 9. Constituents of the Div/Rem tradeoff protocols

Protocol	Domain	Dimension	Times
Div/Rem $\{k_x, k_y\}$	$\mathbb{Z}_2$	3	${k_x}^2 + k_x k_y + 2k_x + k_y$
	$\mathbb{Z}_{n_x \cdot n_y}$	1	k_x
	$\mathbb{Z}_{n_x \cdot n_y}$	2	k_x
	$\mathbb{Z}_{n_x \cdot n_y}$	$k_x + 1$	1
	$\mathbb{Z}_{n_x \cdot n_y}$	$k_y + 1$	1
Div/Rem $\{$divisor $k_y\}$	$\mathbb{Z}_2$	3	$k(k - k_y)$
	$\mathbb{Z}_n$	1	$k - k_y$
	$\mathbb{Z}_n$	2	$k - k_y$
Div/Rem $\{$dividend $k_x\}$	$\mathbb{Z}_2$	3	${k_x}^2 + kk_x + k + k_x$
	$\mathbb{Z}_{n \cdot n_x}$	1	k_x
	$\mathbb{Z}_{n \cdot n_x}$	2	k_x
	$\mathbb{Z}_{n \cdot n_x}$	$k + 1$	1

Table 10. The Scalar-Product Complexity of the Div/Rem tradeoff protocols

k	Experimental Time	Estimated Time
9	5.08	4.02
10	7.50	7.88
11	27.65	26.50
12	129.21	119.66

Table 13. Experimental and estimated time (seconds) of the Exp(sb, se) $\in \mathbb{Z}_{n^n}$ protocol

Protocol	Domain	Dimension	Times
$\mathbb{Z}_n$-to-$\mathbb{Z}_2\{k'\}$	$\mathbb{Z}_2$	3	k'
$\mathbb{Z}_2$-to-$\mathbb{Z}_n\{k'\}$	$\mathbb{Z}_n$	$k'+1$	1
Product	$\mathbb{Z}_n$	2	1
Square	$\mathbb{Z}_n$	1	1
Comparison	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_n$	1	1
Zero	$\mathbb{Z}_2$	3	$2k$
	$\mathbb{Z}_n$	1	2
	$\mathbb{Z}_n$	2	1
If-Then-Else	$\mathbb{Z}_n$	2	1
Shift-Left	$\mathbb{Z}_2$	3	$\lceil\log(k+1)\rceil+k$
	$\mathbb{Z}_n$	1	$\lceil\log(k+1)\rceil+1$
	$\mathbb{Z}_n$	2	$\lceil\log(k+1)\rceil+1$
	$\mathbb{Z}_n$	$k+1$	1
Shift-Right	$\mathbb{Z}_2$	3	$\lceil\log(k+1)\rceil+k$
	$\mathbb{Z}_n$	1	$\lceil\log(k+1)\rceil+1$
	$\mathbb{Z}_n$	2	$\lceil\log(k+1)\rceil+1$
	$\mathbb{Z}_n$	$k+1$	1
Shift$\{s\}$	$\mathbb{Z}_2$	3	k
	$\mathbb{Z}_n$	$k-s+1$	1
Rotate-Left	$\mathbb{Z}_2$	3	$\lceil\log(k+1)\rceil+3k+1$
	$\mathbb{Z}_n$	1	$\lceil\log(k+1)\rceil+1$
	$\mathbb{Z}_n$	2	$\lceil\log(k+1)\rceil$
	$\mathbb{Z}_{n^2}$	2	1
	$\mathbb{Z}_{n^2}$	$k+1$	2
Rotate-Right	$\mathbb{Z}_2$	3	$\lceil\log(k+1)\rceil+3k+1$
	$\mathbb{Z}_n$	1	$\lceil\log(k+1)\rceil+1$
	$\mathbb{Z}_n$	2	$\lceil\log(k+1)\rceil$
	$\mathbb{Z}_{n^2}$	2	1
	$\mathbb{Z}_{n^2}$	$k+1$	2
Div/Rem	$\mathbb{Z}_2$	3	$2k^2+3k$
	$\mathbb{Z}_{n^2}$	1	k
	$\mathbb{Z}_{n^2}$	2	k
	$\mathbb{Z}_{n^2}$	$k+1$	2
Function(x)	$\mathbb{Z}_n$	2^{k+1}	1
Function(x, y)	$\mathbb{Z}_n$	2^{k+1}	$2^{k+1}+1$
(flo) Scalar-Product	$\mathbb{Z}_2$	3	$2k$
	$\mathbb{Z}_n$	d	1
	$\mathbb{Z}_n$	1	2
(flo) Square	$\mathbb{Z}_2$	3	$2k$
	$\mathbb{Z}_n$	1	3
(flo) Product	$\mathbb{Z}_2$	3	$2k$
	$\mathbb{Z}_n$	2	1
	$\mathbb{Z}_n$	1	2

Table 2. The Complexity of our protocols

Role	CPU	Operating System	RAM
Commodity Server	Two AMD Opteron™ 2220 SE 2.81GHz (Dual-Core)	FreeBSD 7.0-STABLE	DDR2 667 20GB
Party 1	Two Intel® Xeon® X5365 3.00GHz (Quad-Core)	FreeBSD 7.0-STABLE	DDR2 667 48GB
Party 2	Two Intel® Xeon® X5482 3.20GHz (Quad-Core)	Ubuntu 2.6.24-16-server	DDR2 800 64GB

Table 11. The environment of our experiments

	Dimension							
k	1	2	3	4	5	8	16	32
0	0.0027	0.0028	0.0035	0.0029	0.0030	0.0030	0.0033	0.0037
1	0.0027	0.0028	0.0031	0.0029	0.0029	0.0030	0.0033	0.0037
2	0.0027	0.0028	0.0029	0.0028	0.0028	0.0030	0.0033	0.0036
3	0.0027	0.0028	0.0029	0.0030	0.0030	0.0031	0.0032	0.0036
4	0.0027	0.0028	0.0029	0.0030	0.0030	0.0030	0.0032	0.0036
7	0.0027	0.0028	0.0029	0.0030	0.0029	0.0030	0.0034	0.0036
15	0.0028	0.0029	0.0029	0.0029	0.0030	0.0030	0.0033	0.0038
31	0.0027	0.0027	0.0028	0.0030	0.0029	0.0032	0.0031	0.0040
63	0.0027	0.0029	0.0029	0.0029	0.0030	0.0031	0.0034	0.0046
127	0.0027	0.0030	0.0029	0.0030	0.0030	0.0031	0.0034	0.0041
255	0.0028	0.0030	0.0031	0.0031	0.0032	0.0031	0.0035	0.0042
511	0.0029	0.0030	0.0030	0.0032	0.0033	0.0033	0.0036	0.0042
1023	0.0033	0.0034	0.0034	0.0034	0.0034	0.0037	0.0039	0.0046
2047	0.3066	0.3063	0.3064	0.3063	0.3067	0.3067	0.3068	0.3076
4095	0.3099	0.3098	0.3099	0.3100	0.3099	0.3100	0.3106	0.3111
8191	0.2205	0.2205	0.2203	0.2204	0.2204	0.2205	0.2215	0.2215
16383	0.1823	0.2305	0.2647	0.2058	0.2253	0.2205	0.2645	0.2310
32767	0.3956	0.3898	0.3943	0.4198	0.4010	0.4098	0.4395	0.4270
65535	1.0598	1.0599	1.0618	1.0627	1.0726	1.0612	1.0604	1.0612
131071	4.2073	4.2026	4.2080	4.2186	4.2222	4.2114	4.2212	4.2219
262143	16.9002	16.8916	16.8791	16.8730	16.9086	16.8971	16.8941	16.8636

Table 12. The timings(seconds) of the commodity-based Scalar-Product protocol with different k and different dimensions

k	12	15	17	19	20	22	24
Exp(sb,se) $\in \mathbb{Z}_{n^n}$	2 minutes	3 hours	3 days	2 months	1 year	2 decades	4 centuries
Exp(sb,se,n_x) $\in \mathbb{Z}_{{n_x}^n}$	45 seconds	58 minutes	21 hours	20 days	1 season	5 years	1 century
Exp(pb x,se) $\in \mathbb{Z}_{x^n}$	22 seconds	28 minutes	10 hours	9 days	1 month	2 years	5 decades
Exp(pb x,se,n_y) $\in \mathbb{Z}_{x^{n_y}}$	11 seconds	13 minutes	5 hours	4 days	25 days	1 year	3 decades
Exp(sb,se,n_y) $\in \mathbb{Z}_{n^{n_y}}$	7 seconds	8 seconds	10 seconds	14 seconds	28 seconds	1 minute	7 minutes
Exp(sb,pe y) $\in \mathbb{Z}_{n^y}$	4 seconds	5 seconds	5 seconds	7 seconds	14 seconds	48 seconds	3 minutes
Exp(sb,pe y, n_x) $\in \mathbb{Z}_{{n_x}^y}$	3 seconds	3 seconds	4 seconds	6 seconds	13 seconds	46 seconds	3 minutes
Exp(sb,se,n_x, n_y) $\in \mathbb{Z}_{{n_x}^{n_y}}$	6 seconds	7 seconds	8 seconds	10 seconds	14 seconds	34 seconds	2 minutes
Exp(sb,se) $\in \mathbb{Z}_n$	10 seconds	12 seconds	14 seconds	15 seconds	16 seconds	18 seconds	20 seconds
Exp(sb,se,n_y) $\in \mathbb{Z}_n$	5 seconds	6 seconds	6 seconds	7 seconds	8 seconds	9 seconds	10 seconds
Exp(sb,pe y) $\in \mathbb{Z}_n$	1 second	2 seconds	2 seconds	2 seconds	3 seconds	3 seconds	3 seconds
Exp(sb,pe y, n_x) $\in \mathbb{Z}_n$	1 second	2 seconds	2 seconds	2 seconds	3 seconds	3 seconds	3 seconds
Exp(sb,se,n_x) $\in \mathbb{Z}_n$	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second
Exp(sb,se,n_x, n_y) $\in \mathbb{Z}_n$	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second
Exp(pb x,se) $\in \mathbb{Z}_n$	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second
Exp(pb x,se,n_y) $\in \mathbb{Z}_n$	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second	$<$ 1 second

Table 14. The time cost of the tradeoff exponentiation protocols with given k