Scalar-product-based Secure Two-party Computation

Chih-Hao Shen*, Justin Zhan†, Tsan-Sheng Hsu*, Churn-Jung Liau*, and Da-Wei Wang*

†CyLab Japan, Carnegie Mellon University, USA
Email: justinzh@andrew.cmu.edu

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

Abstract—Secure multiparty computation is a very important research topic in cryptography. A secure multi-party computation involves N untrustful parties. It takes input $x_i$ from the $i$th party, and returns to that party. The sub-problem of secure multiparty computation that has received special attention by researchers because of its close relation to many cryptographic tasks is secure two-party computation. This area of research is concerned with the question: 'Can two party computation be achieved more efficiently and under weaker security assumptions than general multiparty computation?' Yao’s protocol [17] provided security against passive adversaries. Lindell and Pinkas [14] proposed secure two protocols that are secure against active adversaries. In this paper, we would like to propose a set of information theoretically secure protocols based on scalar product protocol. The detailed complexity analysis has been provided.

I. INTRODUCTION

Secure multiparty computation is a research topic aiming at the double-edged privacy problem. The core issue of the topic is how several potentially distrustful parties can cooperate to conduct certain computations over their private data without compromising their privacy. In cryptography, secure multiparty computation is a problem that was initially suggested by Andrew C. Yao [17] in 1982. In this publication, the millionaire problem was introduced: Alice and Bob are two millionaires who want to find out which is richer without revealing the precise amount of their wealth. Yao proposed a solution allowing Alice and Bob to satisfy their curiosity while respecting the constraints. This problem and result gave way to a generalization called multi-party computation protocols. In a secure multiparty computation, we have a given number of participants $p_1, p_2, ..., p_N$, each having a private data, respectively $d_1, d_2, ..., d_N$. The participants want to compute the value of a public function $F$ on N variables at the point $(d_1, d_2, ..., d_N)$. A secure multiparty computation protocol is secure if no participant can learn more from the description of the public function and the result of the global calculation than what he/she can learn from his/her own entry - under particular conditions depending on the model used. After Yao’s [17] general solution to two-party secure computation was proposed, Goldreich et al. [11] soon gave another general solution to multiparty computation. Both proposals are so elegant that they provide secure two-party/multiparty protocols for binary AND and XOR gates, which can be further generalized to all functions. However, despite their academic significance, the computation costs of both solutions are too prohibitive to be practical for large datasets. A function $f$ is complete if a secure protocol for $f$ implies the existence of secure protocols for all functions. Yao and Goldreich et al. propose the idea that solving the secure multiparty 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 computation power. Kilian shows that the oblivious transfer is complete [12], and so are the functions with imbedded OR [13]. Furthermore, it is believed that the integer-based scalar product is more practical to real applications than the binary-based oblivious transfer is [5].

In the past decade, proposals for the secure scalar product are flourishing. Du et al. [8] propose the invertible-matrix and the commodity-based approaches. The former approach enables the tradeoff between efficiency and privacy, and the latter is based on Beaver’s commodity model [2]. Goethals et al. [9] propose the computationally secure scalar-product protocol, security of which depends on the intractability of the composite residuosity class problem.

Moreover, there are many efforts on building various applications on secure scalar products. Atallah et al. reduce geometry problems to scalar products [1]. Du et al. construct secure protocols for statistical analysis [7] and scientific computations [6]. More recently, Bunn et al. [3] give a secure protocol for k-means clustering, which is based on scalar products under the composite-residuosity approach.

There is also plenty of theoretical studies on scalar products. Based on information theory, Chiang et al. [4] propose a privacy measurement, by which they analyze various scalar-product approaches. They prove that the invertible-matrix approach discloses at least half the information and the commodity-based approach is perfectly secure. Wang et al. [16] prove that there is no information-theoretically secure two-party protocol for the scalar product. Moreover,
the closure property of the commodity-based approach is preliminarily verified according to the security definition based on information theory [15].

In this paper, we would like to propose a set of practical protocols based on secure scalar-product protocol. They are polynomial evaluation, comparison, if-then-else evaluation, and integer division/remainder protocols. We provide detailed analysis for each protocol.

This paper is organized as follows. Section II introduces the notations and the scalar-product protocol as the building block to all the other protocols. Next, the bottom-up construction of the three primitives which are Product, Comparison, and Division protocols, is thoroughly presented in Section III. We then analyzes the round and computation complexity of the three primitives in Section IV. After that, we specifically build up secure protocols computing the Point-inclusion and the Variance functions in Section V. Finally, Section VI concludes the paper and lays out the future work.

II. Preliminaries

In this section, we introduce the notation used hereafter and the formulation for the building block—Scalar-product.

The notation \((X_1, X_2) \mapsto (Y_1, Y_2)\) for a secure two-party protocol represents that Party 1 and Party 2 have private inputs \(X_1\) and \(X_2\) respectively. After protocol execution, Party 1 gets output \(Y_1\), while Party 2 gets \(Y_2\). For clarity purpose, footnotes always indicate the data owner. The domain \(\mathbb{Z}_n\) denotes the finite group consisted 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. Our proposal mainly focuses on the computations over \(\mathbb{Z}_n\) and \(n\) is two’s power, i.e., \(n = 2^k + 1, k \in \mathbb{N}\). As a result, without explicit mention, the arithmetic is always defined in \(\mathbb{Z}_n\). Moreover, to extend the domain from natural number to integer, while elements \(\{1, \ldots, \lfloor \frac{n+1}{2} \rfloor\}\) remain positive numbers, elements \(\{n-1, \ldots, n-\lfloor \frac{n+1}{2}\}\) are interpreted as negative integers analogous to the binary system in modern computers. Hence, the substraction to \(p\) is equivalent to the addition to \((n-p)\).

In this paper, the formulation for the Scalar-product protocol follows Goldreich’s principle [10] that the intermediate results during a protocol execution are always shared among participants. In a protocol \(\pi\) composed of scalar products, current outputs can be inputs to the next scalar product, which is actually the intermediate results of \(\pi\) and should be shared. Moreover, the intermediate results are shared by addition rather than multiplication. In finite group \(\mathbb{Z}_n\), the multiplicative sharing reveals information when either the shares is zero, on the other hand, the additive sharing is proven to be perfect [15]. Specifically, we define the secure Scalar-product protocol as

**Definition 1** (Scalar Product). Party 1 and Party 2 want to collaboratively compute the scalar product of their private input vectors. That is, they want to execute the secure protocol \(((x[1], \ldots, x[d_1]), (x[1], \ldots, x[d_2])) \mapsto (y_1, y_2)\), such that

\[
y_1 + y_2 = \left[ \begin{array}{c} x[1]_1 \\ \vdots \\ x[d_1]_1 \\ \vdots \\ x[d_2]_1 \end{array} \right]^T \left[ \begin{array}{c} x[1]_1 \\ \vdots \\ x[d_1]_2 \\ \vdots \\ x[d_2]_2 \end{array} \right] = \sum_{i=1}^d x[i]_1 \cdot x[i]_2
\]

where \(x[i]_1, x[i]_2, y_1, y_2 \in \mathbb{Z}_n\). Additionally, \(+\) and \(\cdot\) are the modular addition and the modular multiplication in \(\mathbb{Z}_n\).

Here we merely define the Scalar-product protocol instead of providing a specific approach because we focus on building more protocols on top of the 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 within a huge program. In our scalar-product based protocols, as long as a new approach matches the Definition 1, it can be used as the building block of our proposed protocols. Moreover, our protocols are information-theoretically composed. They are as strong as the specific Scalar-product approach. For example, based on a computationally secure approach, our proposed protocols are also computationally secure.

At last, we need to mention that we do not deal with the problem combining the final results. The combination of the final result involves the fairness problem that the first party who receives the result might disrupt the collaboration prematurely. One of the possible solutions is to adopt the commitment protocols. However, we do not discuss this issue in this paper for it is beyond our focus.

III. Three Primitives

First of all, we would like to explain our construction principles. During our bottom-up construction, the Scalar-product protocol is used as the initial building block. If a protocol is composed of building blocks and its participants only compute locally, the protocol becomes another building block. In other words, we construct a new protocol by executing building blocks or local computation by the participant himself. However, no communication between participant is allowed during protocol composition. After that, the new protocol becomes another building block which can be used to compose other protocols. Such construction is theoretically secure so that its security strength totally relies on the security of the underlying Scalar-product implementation.

Next, we sequentially introduce our proposed three primitives to secure two-party computation—the Product, the Comparison, and the Division protocols.

A. Polynomial Evaluation

To compute a polynomial function collaboratively, we need to be able to perform two-party addition and multiplications. Because 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 Scalar-product protocols.

**Definition A.1** (Product). Party 1 and Party 2 additively share the multiplicand and the multiplicator. They want to
securely execute the multiplication. In short, they want to 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
\[ z_1 + z_2 = x_1y_1 + x_2y_2 + (x_1y_2 + y_1x_2). \]

After the above factoring, it is obvious that \(x_jy_j\) is locally computable to Party \(j\), \(j = 1, 2\), but we need to execute the Scalar-product protocol to compute the other terms. The following protocol describes the details instructions.

**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
\[ t_1 + t_2 = \left[ \begin{array}{c} x_1 \\ y_1 \end{array} \right]^T \left[ \begin{array}{c} y_2 \\ x_2 \end{array} \right] = x_1y_2 + y_1x_2 \]

2) Party \(j\) locally computes \(z_j = t_j + x_jy_j\), 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, and with the Product protocol we can multiply, too. Therefore, secure two-party computation on any polynomial evaluation can be composed of the Product protocol and the help of additive sharing.

**B. Comparison**

There are many types of binary comparison: less than (<), greater than (>), greater than or equal to (\(\geq\)), less than or equal to (\(\leq\)), and equal to (\(=\)). However, we know that all of them can be reduced to the less than operator.\(^1\) Moreover, in order to compare \(x\) and \(y\), it is intuitive to compare \((x - y)\) and 0 since we share the intermediate results additively. It is effortless to subtract under additive sharing. More specifically, our proposal to compare \(x\) and \(y\) is to compute the most significant bit of \((x - y)\).

Ahead of the primitive Comparison protocol, we propose two protocols, \(Z_n\rightarrow-Z_2\) and \(Z_2\rightarrow-Z_n\), which convert to and fro between \(Z_n\) sharing and bitwise \(Z_2\) sharing. In addition to be the building blocks of the Comparison protocol, the \(Z_n\rightarrow-Z_2\) and \(Z_2\rightarrow-Z_n\) protocols establish the possibility of secure computation for all functions. Albeit the inefficiency, we can always apply Yao’s circuit evaluation idea after the \(Z_n\rightarrow-Z_2\) protocol and followed by the \(Z_2\rightarrow-Z_n\) protocol. These two protocols make our proposal as general as the classic circuit evaluation approaches.

**DEFINITION B.1 (\(Z_n\rightarrow-Z_2\))**. Party 1 and Party 2 additively share a number in \(Z_n\), and they want to securely convert the \(Z_n\) sharing into bitwise \(Z_2\) sharing. More specifically, Party 1 and Party 2 want to collaboratively execute the secure protocol \((x_1, x_2) \mapsto ((y_0^1, \ldots, y_k^0), (y_0^2, \ldots, y_k^1))\), such that
\[ (y_ky_{k-1} \cdots y_1y_0^0) = x_1 + x_2 \]

where \(x_1, x_2 \in \mathbb{Z}_n\), \(y_1^0, y_2^0 \in \mathbb{Z}_2\), and \(y_i = y_i^1 \pmod 2\).

To convert from \(Z_n\) sharing to bitwise \(Z_2\) sharing, we emulate the carry ripple adder with binary Scalar-product protocol, whose \(n = 2\). Let \(x_1 = (x_1^1 \cdot \ldots \cdot x_1^n)\), \(x_2 = (x_2^1 \cdot \ldots \cdot x_2^n)\), and the adder operates as the following long addition:
\[ c^{k+1} \quad c^k \quad \ldots \quad c_1 \quad c_0 \]
\[ \begin{array}{cccc}
\vdots \\
x_1^k & \cdots & x_1^1 & x_1^0 \\
\vdots \\
x_2^k & \cdots & x_2^1 & x_2^0
\end{array} \]
\[ y_k \quad \ldots \quad y_1 \quad y_0 \]

where \(c^0 = 0\) and \(c^{i+1} = c_i^1 + c_i^0 + x_1 x_2 \mod 2\) are the carry bits; \(y_i = c_i^1 + x_1^i + x_2^i \mod 2\) is the \(i\)-th summation bit. Next, we present the \(Z_n\rightarrow-Z_2\) protocol as follows:

**PROTOCOL \(Z_n\rightarrow-Z_2\) \((n = 2^{k+1})\)**

1) Party \(j\) locally sets \(c_i^0 = 0\), and \(c_i^1 = y_i^0\), for \(j = 1, 2\).
2) For \(i = 0, \ldots, k-1\), repeat Step 2a to Step 2b.\(^2\)
   a) Party 1 and Party 2 collaboratively execute the binary Scalar-product protocol
   \[ ((c_1^i, x_1^i, x_1^0), (x_2^i, c_2^i, x_2^0)) \mapsto (t_1^i, t_2^i) \]
   such that
   \[ t_1^i + t_2^i \pmod 2 = \left[ \begin{array}{c} c_1^i \\
   x_1^i \\
   1
   \end{array} \right] \left[ \begin{array}{c}
x_2^i \\
   c_2^i \\
   x_2^0
   \end{array} \right]^T \pmod 2 \]
   b) For \(j = 1, 2\), Party \(j\) computes
   \[ c_j^{i+1} = c_j^i x_j^i + t_j^i \pmod 2 \]
   \[ y_j^{i+1} = x_j^{i+1} + c_j^{i+1} \pmod 2 \]

**DEFINITION B.2 (\(Z_2\rightarrow-Z_n\))**. Party 1 and Party 2 bitwise, additively share a number in \(Z_2\), and they want to securely convert the bitwise \(Z_2\) sharing into the \(Z_n\) sharing. More specifically, Party 1 and Party 2 want to execute the secure protocol \([(x_0^0, \ldots, x_0^n), (x_2^0, \ldots, x_2^n)] \mapsto (y_1, y_2)\), such that
\[ y_1 + y_2 = (x_1^k x_{k-1} \cdots x_1 x_0^0) \]

where \(x_1^1, x_2^1 \in \mathbb{Z}_2\), \(y_1, y_2 \in \mathbb{Z}_n\), and \(x^i = x_1^i + x_2^i \pmod 2\).

According to the above requirement, the outputs can be rewritten as the following function:
\[ y_1 + y_2 = \sum_{i=0}^{k} x_1^i \cdot 2^i = \sum_{i=0}^{k} (x_1^i + x_2^i \pmod 2) \cdot 2^i \]
\[ = \sum_{i=0}^{k} x_1^i \cdot 2^i + \sum_{i=0}^{k} x_2^i \cdot 2^i - \sum_{i=0}^{k} x_1^i x_2^i \cdot 2^{i+1} \]

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

**PROTOCOL \(Z_2\rightarrow-Z_n\) \((n = 2^{k+1})\)**

Since \(n = 2^{k+1}\), the overflow bit \(c^{k+1}\) is discarded.
1) Party 1 and Party 2 execute the Scalar-product protocol \(((x_1^0, \ldots, x_k^0), (2^0 x_1^1, \ldots, 2^{k+1} x_{\frac{k}{2}}^0)) \mapsto (t_1, t_2)\), such that
\[
t_1 + t_2 = \begin{bmatrix}
x_1^0 \\
\vdots \\
x_k^0 \\
2 \cdot x_1^0 \\
\vdots \\
2^{k+1} \cdot x_{\frac{k}{2}}^0 \\
\end{bmatrix}
\]

2) Party \( j \) computes \( y_j = \sum_{i=0}^{k} x_j^i \cdot 2^k - t_j \), for \( i = 1, 2 \).

**Definition B.3 (Comparison).** Party 1 and Party 2 additively share a number in \( \mathbb{Z}_n \), and they want to know whether the number is positive or negative. As a result, Party 1 and Party 2 want to collaboratively execute the secure protocol \((x_1, x_2) \mapsto (y_1, y_2)\), such that
\[
y_1 + y_2 = \begin{cases} 
1 & \text{if } x_1 + x_2 < 0, \\
0 & \text{otherwise}.
\end{cases}
\]

Recalled 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.

**Protocol Comparison**

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

**C. Integer Division**

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:

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

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 \( y \geq x \cdot 2^i \), for \( i = k - 1, \ldots, 0 \). If it is true, the \( i \)-th bit of \( q \) is 1, and \( y \) is subtracted by \( x \cdot 2^i \); otherwise, the \( i \)-th bit of \( q \) is 0, and \( y \) 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 \( x \cdot 2^i \), for \( i = k - 1, \ldots, 0 \); otherwise, \( x \cdot 2^i \pmod{n} \) will give us unpredictable results. Therefore, in our solution we first convert both the dividend and the divisor from \( \mathbb{Z}_n \) sharing to \( \mathbb{Z}_n^2 \) sharing, by which \( x \cdot 2^i \) can always be correctly represented. After the conversion, we emulate the divider with the Scalar-product protocol.

Before the Division protocol, we present the If-Then-Else protocol, which is not only the building block for the Division protocol but also useful for the function with alternatives.

**Definition C.1 (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 if-then-else statement. That is, Party 1 and Party 2 want to jointly execute the protocol \(((b_1, x_1, y_1), (b_1, x_2, y_2)) \mapsto (z_1, z_2)\), such that
\[
z_1 + z_2 = \begin{cases} 
x_1 + x_2 & \text{if } b_1 + b_2 = 1 \\
y_1 + y_2 & \text{if } b_1 + b_2 = 0
\end{cases}
\]

According to the above requirement, the outputs can be rewritten as the following function:
\[
z_1 + z_2 = (b_1 + b_2)(x_1 + x_2) + (1 - b_1 - b_2)(y_1 + y_2)
\]
\[
= (x_1 + x_2) + (b_1 + b_2)(x_1 - y_1 + x_2 - y_2)
\]

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

**Protocol If-Then-Else**

1) Party \( j \) locally 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 \( j \) locally computes \( z_j = t_j + x_j \), for \( j = 1, 2 \).

**Definition C.2 (Integer Division/Remainder).** Party 1 and Party 2 additively share the dividend and the divisor in \( \mathbb{Z}_n \). They want to securely execute the integer division. In short, Party 1 and Party 2 want to collaboratively execute the secure protocol \(((x_1, y_1), (x_2, y_2)) \mapsto ((q_1, r_1), (q_2, r_2))\), such that
\[
y_1 + y_2 = (q_1 + q_2)(x_1 + x_2) + (r_1 + r_2)
\]
\[
0 \leq (r_1 + r_2) < (q_1 + q_2)
\]

**Protocol Division/Remainder**

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

3) Party \( j \) sets \( Y_j^k = Y_j \), for \( j = 1, 2 \).

4) For \( i = k-1, k-2, \ldots, 0 \), repeat Step 4a to Step 4d.

\( \text{a) Party } j \text{ computes } t_j^i = Y_j^{i+1} - X_j \cdot 2^i \mod n^2 \),

for \( j = 1, 2 \).

\( \text{b) Party } 1 \text{ and Party } 2 \text{ collaboratively execute the Comparison protocol} \ (t_1^i, t_2^i) \rightarrow (s_1^i, s_2^i), \) such that

\[
s_1^i + s_2^i \mod n^2 = \begin{cases} 1 & \text{if } t_1^i + t_2^i < 0 \\ 0 & \text{otherwise} \end{cases}
\]

\( \text{c) Party } 1 \text{ and Party } 2 \text{ execute the If-Then-Else protocol} \ ((s_1^i, 0, 0), (s_2^i, 0, 1)) \rightarrow (q_1^i, q_2^i), \) such that

\[
q_1^i + q_2^i \mod n^2 = \begin{cases} 0 & \text{if } s_1^i + s_2^i = 1 \\ 1 & \text{if } s_1^i + s_2^i = 0 \end{cases}
\]

\( \text{d) Party } 1 \text{ and Party } 2 \text{ execute the If-Then-Else protocol} \ ((s_1^i, Y_1^{i+1}, t_1^i), (s_2^i, Y_2^{i+1}, t_2^i)) \rightarrow (Y_1^i, Y_2^i), \) such that

\[
Y_1^i + Y_2^i = \begin{cases} Y_1^{i+1} + Y_2^{i+1} & \text{if } s_1^i + s_2^i = 1 \\ t_1^i + t_2^i & \text{if } s_1^i + s_2^i = 0 \end{cases}
\]

5) For \( j = 1, 2 \), Party \( j \) computes

\[
q_j = \sum_{i=0}^{k-1} q_j^i \cdot 2^i \mod n, \ \ \ r_j = Y_j^0 \mod n
\]

At last, figure 1 gives the hierarchy of our scalar-product-based, bottom-up construction.

---

**IV. ANALYSIS**

In Section III we elaborate on the protocol instructions, and we then list the table of complexity summarization here in Table I. The complexity is measured in proportion to the Scalar-product protocol. The “Domain” and the “Dimension” are the inputs’ domain and the number of vector elements. Moreover, the “Round” is actually measured as the number of the Scalar-product protocol executions. According to the table, the Product, the \( \mathbb{Z}_2^{n-1} \rightarrow \mathbb{Z}_2 \), and the If-Then-Else protocols have constant round complexity \( O(1) \); the \( \mathbb{Z}_2^{n-1} \rightarrow \mathbb{Z}_2 \) and the Comparison protocols have linear round complexity \( O(k) \); and the Division protocol has quadratic round complexity \( O(k^2) \). Recalled that \( n = 2^{k+1} \), therefore, the round complexity here is based on the bit length of \( n \).

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Domain</th>
<th>Dimension</th>
<th>Round</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>( \mathbb{Z}_n )</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( \mathbb{Z}_n \rightarrow \mathbb{Z}_2 )</td>
<td>( \mathbb{Z}_2 )</td>
<td>3 ( k )</td>
<td></td>
</tr>
<tr>
<td>( \mathbb{Z}_2 \rightarrow \mathbb{Z}_n )</td>
<td>( \mathbb{Z}_n )</td>
<td>( k+1 )</td>
<td>1</td>
</tr>
<tr>
<td>If-Then-Else</td>
<td>( \mathbb{Z}_n )</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE I**

The complexity of our protocols

---

**V. SPECIFIC EXAMPLES**

To demonstrate the power of the primitives: the Product, the Comparison, and the Division protocols, we practically give two examples of real world computation and compose them with the primitives.

**A. Example: Point Inclusion**

Party 1 is located at \((x[1], x[2])\), and Party 2 has a circle described by its center \((x[1], x[2])\) and radius \(r_2\). They want to know whether Part 1 is inside Party 2’s circle, but Party 1 will not disclose his location, and neither will Party 2 reveal the circle’s information. Therefore, they need a secure protocol \((x[1], x[2]), (x[1], x[2], r_2)) \rightarrow (p_1, p_2)\), such that

\[
p_1 + p_2 = \begin{cases} 1 & \text{if } \sum_{i=1}^k (x[i] - x[i])^2 - r_2^2 < 0 \\ 0 & \text{otherwise} \end{cases}
\]

Based on the primitives proposed in Section III, the Point inclusion protocol can be constructed as the follows:

1) Party 1 and Party 2 jointly execute the Product protocol \((x[k], x[k]), (n - x[k], n - x[k])) \rightarrow (y[k], y[k])\), where \(y[k] = y[k] + y[k] = (x[k] - x[k])^2, \) for \( k = 1, 2 \).

2) Party 1 computes \( t_1 = y[1] + y[2] \).

3) Party 2 computes \( t_2 = y[1] + y[2] - r_2^2 \).

4) Party 1 and Party 2 collaboratively execute the Comparison protocol \((t_1, t_2) \rightarrow (p_1, p_2)\), such that

\[
p_1 + p_2 = \begin{cases} 1 & \text{if } t_1 + t_2 < 0 \\ 0 & \text{otherwise} \end{cases}
\]

**B. Example: Variance**

Party 1 and Party 2 have private inputs \((x[1], \ldots, x[p_1])\) and \((x[1], \ldots, x[p_2])\) respectively. They want to collaboratively and securely compute the variance of the union of their private inputs. More specifically, they need a protocol \((x[1], \ldots, x[p_1], x[1], \ldots, x[p_2])) \rightarrow (q_1, q_2)\), such that

\[
q_1 + q_2 = \frac{(p_1 + p_2)(s_1 + s_2) - (t_1 + t_2)^2}{(p_1 + p_1)^2},
\]

where \(s_j = \sum_{i=1}^{p_j} x[i]^2\) and \(t_j = \sum_{i=1}^{p_j} x[i], \) for \( j = 1, 2\).
Based on the primitives proposed in Section III, the Variance protocol can be constructed as the follows:

1) For $j = 1, 2$, Party $j$ computes
   \[ s_j = \sum_{i=1}^{p_j} x[i], \quad t_j = \sum_{i=1}^{p_j} x[i]_j. \]

2) Party 1 and Party 2 collaboratively execute the Product protocol \((p_1, s_1), (p_2, s_2) \mapsto (u_1, u_2)\), such that
   \[ u_1 + u_2 = (p_1 + p_2)(s_1 + s_2). \]

3) Party 1 and Party 2 collaboratively execute the Product protocol \((t_1, t_1), (t_2, t_2) \mapsto (v_1, v_2)\), such that
   \[ v_1 + v_2 = (t_1 + t_2)(t_1 + t_2). \]

4) Party 1 and Party 2 collaboratively execute the Product protocol \((p_1, p_1), (p_2, p_2) \mapsto (w_1, w_2)\), such that
   \[ w_1 + w_2 = (p_1 + p_2)(p_1 + p_2). \]

5) Party $j$ computes $z_j = u_j - v_j$, for $j = 1, 2$.

6) Party 1 and Party 2 jointly execute the Division protocol
   \((w_1, z_1), (w_2, z_2) \mapsto (q_1, r_1), (q_2, r_2)\), such that
   \[ q_1 + q_2 = \left\lfloor \frac{z_1 + z_2}{w_1 + w_2} \right\rfloor. \]

VI. CONCLUSIONS AND FUTURE WORKS

In this paper, 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.

REFERENCES


