Optimizations for Secure Multiparty Computation Protocols

Leilani Tam von Burg, Christopher Harth-Kitzerow*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: leilani.tam-von-burg@tum.de, christopher.harth-kitzerow@outlook.de

***Abstract*—The BGW protocol is a protocol for secure multiparty computation based on Shamir's secret sharing scheme. It allows the computation of functions by representing them as arithmetic circuits composed of addition and multiplication gates. Many steps of the protocol are quite efficient as they do not require encryption or communication. However, multiplication gates require communication and impact the efficiency of the protocol negatively. Therefore, optimization techniques improving the multiplication operation have been developed. In this paper, we focus on the optimization technique of Beaver triples.**

***Index Terms*—BGW Protocol, Shamir's secret sharing, Beaver Triples**

1. Introduction

There exist many applications where there is a need for computations that keep the inputs secret. Examples of this are secure auctions, voting, secure machine learning or computations on databases that hold private information. Secure Multiparty Computation Protocols do exactly this. They enable a joint computation on a group of parties without disclosing the private inputs of the participants [1].

The idea of Secure Multiparty Computation Protocols was first introduced by Yao in the 1980s [1], [2]. He illustrated the necessity for this type of computation with the two millionaires problem. Here, two millionaires want to determine who is richer without disclosing their individual wealth. Yao's protocol is based mainly on garbled circuits [2].

Later, different protocols were developed to expand from two party to multiparty computation and improve on efficiency. The implementation of Fairplay in 2004 is considered the first proper implementation of such a protocol [3]. Admittedly, its scalability and performance was very limited. Today, more efficient implementations exist and practical uses are becoming more and more common. This paper introduces the BGW protocol and one possible optimization technique often implemented in combination.

2. BGW Protocol for Secure Multiparty Computation

The BGW protocol was introduced by Ben-Or, Goldwasser and Wigderson [4]. It differs from other well known protocols in that it is not based on garbled circuits. Instead, it is based on Shamir secret sharing [5]. The protocol can compute any function f over a field F by representing the function as an arithmetic circuit composed of addition, multiplication and multiplication-by-constant gates. Multiplication by a constant can be represented by an addition and will therefore be omitted in the further discussion. The evaluation of the circuit is done gate-by-gate.

Since we require the computation to be secure, each input wire is only known by one party that desires to keep it private. Additionally, no intermediate values should be revealed during the evaluation of the circuit. In order to achieve the desired privacy during the evaluation of the function, the value of each wire is kept secret by hiding it in a polynomial of degree t which is shared between the parties.

2.1. Shamir's Secret Sharing

Since secret sharing is a fundamental part of the BGW protocol, it will be shortly introduced. The idea behind secret sharing can be illustrated by the following example. Imagine a treasure chest which requires multiple keys to be opened. These keys are held by different parties. Therefore, the treasure can only be accessed when the parties come together to unlock the chest.

The protocol can be split into two phases: a sharing phase and a reconstruction phase. During the sharing phase, the secret s is split into shares held by the different parties. Firstly, the secret s must be "locked in the chest". This is done by encrypting the secret in a polynomial of the following form.

$$f(x) = a_t x^t + ... + a_1 x + a_0 \quad (1)$$

where $a_0 = s$ and $a_t, ..., a_1$ are random coefficients, such that $f(0) = s$. Each party P_i then receives one point $(\alpha_i, [f(\alpha_i)])$ on the polynomial, which we refer to as its share. This value can be interpreted as the "key" held by that specific party. For clarity, we will refer to shares by using square brackets throughout the paper.

The threshold t is chosen such that it corresponds to the assumed maximum number of faulty parties. Therefore, $t + 1$ points are required to reconstruct the secret by interpolating the polynomial. Less points will not reveal the secret. This corresponds to the reconstruction phase.

doi: 10.2313/NET-2022-01-1_05

2.2. Security scenarios for the BGW protocol

When choosing a threshold value t for the BGW protocol, we differentiate between two different types of faulty parties.

Semi-honest security A semi-honest adversary is considered honest-but-curious. This means it follows the protocol honestly but it may try to learn as much information as possible during the execution. Therefore, we consider it a passive adversary. This includes parties colluding and pooling their information together in order to learn as much as possible [1].

For every n-ary function $f(x_1, ..., x_n)$, there exists a protocol for computing f with perfect security in the presence of a semi-honest adversary controlling $t < n/2$ parties. This means we require an honest majority in this case [4]. Evidently, we cannot allow any adversaries in a two party computation. In this situation, the secret inputs are encrypted with linear functions. Therefore, knowing the gradient allows direct reconstruction of the secret.

Malicious security A malicious adversary is active, which means it can take any action it desires and deviate from the protocol. Therefore, it can provide any input it wants as well, which can affect the honest parties inputs. A malicious adversary can control up to $t < n/3$ parties while ensuring perfect security [1].

It should be noted here, that the BGW protocol is secure from an information-theoretic standpoint when adhering to the above choices for the threshold t [4]. It does not rely on cryptographic assumptions. This means that the protocol is secure for adversaries with unlimited computing power. For example, quantum computers do not pose a threat to the security.

2.3. BGW Protocol

The BGW protocol can be subdivided into three main phases:

1) Input sharing phase
2) Computation of the circuit gate-by-gate (additions, multiplications)
3) Output Reconstruction phase

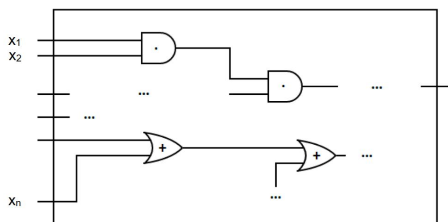

Figure 1: Example of a function represented by an arithmetic circuit with inputs $x_1, ..., x_n$. The input values as well as the intermediate values are secret shared to ensure their privacy over the entirety of the circuit.

2.3.1. Input sharing phase. The input sharing phase of the BGW protocol follows the Shamir secret sharing scheme. Each party P_i encrypts its input x_i in a random polynomial $f_i(x)$ of degree t, where $f_i(0) = x_i$ and then sends each party P_j a share $[f_i(\alpha_j)]$. This way, all parties obtain shares of the other parties inputs.

2.3.2. Computation of the circuit. At each gate, the parties compute the shares of the output wire using the shares of the input wires. The intermediate values stay hidden throughout the circuit.

Addition Gates The computation of addition gates is inexpensive since it does not require communication. Let a and b be the input values and $g_a(x)$ and $g_b(x)$ be the polynomials hiding the input values according to the secret sharing scheme. $g_a(x) + g_b(x) = h_{a+b}(x)$ is the operation at the gate and $\alpha_1, ..., \alpha_n$ are the interpolation points of the individual parties shares. Initially, each party P_i holds the shares $[g_a(\alpha_i)]$ and $[g_b(\alpha_i)]$ of the input which it can add locally. Then, each party holds a share $[h_{a+b}(\alpha_i)]$ of the output $h_{a+b}(x)$. This share of the output can be used directly as the input at the next gate, since there is no need for communication during addition operations. Computations can be done locally on the shares throughout the circuit (as long as they are all additions) until the final output, where the shares are combined to recover the sum. To do so, the constant term $h_{a+b}(0) = a + b$ is evaluated. The output polynomial is still of degree t.

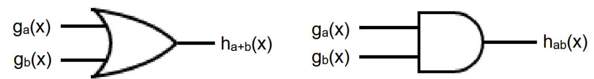

Figure 2: Illustration of an addition gate on the left and a multiplication gate on the right with the secret shared wire inputs $g_a(x)$ and $g_b(x)$ of the input values a and b.

Multiplication Gates The computation of multiplication gates is not as straight forward. Simply multiplying the shares locally as it is done for addition causes two problems. Firstly, the degree becomes $2t$ after a single computation. Multiple multiplication gates cause the degree of the polynomial to become too large. There are not enough interpolation points to recover the result anymore. Additionally, the product of two random polynomials is not fully random anymore. We need a way to compute the multiplication while keeping the polynomial at degree t and ensuring it stays random. This is referred to as degree reduction and randomization [4].

Degree reduction and randomization Assume input values a and b. Degree reduction relies on following property of polynomials:

For any polynomial $h(x)$ with degree $t < n$, there exist constants $\lambda_1, ..., \lambda_n$ and interpolation points $\alpha_1, \alpha_2, ..., \alpha_n$ such that:

$$h(x) = \lambda_1[h(\alpha_1)] + ... + \lambda_n[h(\alpha_n)]$$
$$h(0) = ab$$

That is, we can represent the result of the multiplication by a linear combination $h(x) = \sum_{i=1}^{2t} \lambda_i [h(\alpha_i)]$ of the parties shares.

This can be illustrated more thoroughly by observing the following equations. Let $h(x) = h_{2t}x^{2t} + ... + h_1 x + ab$ be a polynomial of degree $2t$ hiding the secret $h(0) = ab$. In equation (2), we multiply an invertible Vandermonde matrix with the coefficients of the polynomial. This operation corresponds to the evaluations of the polynomial $h(x)$ on the interpolation points $\alpha_1, \alpha_2, ..., \alpha_n$.

$$\begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & ... & \alpha_1^{2t} \\ 1 & \alpha_2 & \alpha_2^2 & ... & \alpha_2^{2t} \\ \vdots & & & & \\ 1 & \alpha_n & \alpha_n^2 & ... & \alpha_n^{2t} \end{bmatrix} \begin{bmatrix} ab \\ h_1 \\ \vdots \\ h_{2t} \end{bmatrix} = \begin{bmatrix} h(\alpha_1) \\ h(\alpha_2) \\ \vdots \\ h(\alpha_n) \end{bmatrix} \quad (2)$$

However, since our goal is to compute our secret ab with each parties shares $[h(\alpha_i)]$, we must invert the matrix. A simple way to invert the Vandermonde matrix is explained in [6]. For example, a Vandermonde matrix

$$V = \begin{bmatrix} 1 & c_1 & c_1^2 & c_1^3 \\ 1 & c_2 & c_2^2 & c_2^3 \\ 1 & c_3 & c_3^2 & c_3^3 \\ 1 & c_4 & c_4^2 & c_4^3 \end{bmatrix} \quad (3)$$

has the inverse V^{-1}=

$$\begin{pmatrix} \frac{c_2c_3c_4}{(c_4-c_1)(c_3-c_1)(c_2-c_1)} & \frac{-c_1c_3c_4}{(c_4-c_2)(c_3-c_2)(c_2-c_1)} & \frac{c_1c_2c_4}{(c_4-c_3)(c_3-c_2)(c_3-c_1)} & \frac{-c_1c_2c_3}{(c_4-c_3)(c_4-c_2)(c_4-c_1)} \\ \frac{-c_2c_3+c_2c_4+c_3c_4}{(c_4-c_1)(c_3-c_1)(c_2-c_1)} & \frac{c_1c_3+c_1c_4+c_3c_4}{(c_4-c_2)(c_3-c_2)(c_2-c_1)} & \frac{-c_1c_2+c_1c_4+c_2c_4}{(c_4-c_3)(c_3-c_2)(c_3-c_1)} & \frac{c_1c_2+c_1c_3+c_2c_3}{(c_4-c_3)(c_4-c_2)(c_4-c_1)} \\ \frac{c_2+c_3+c_4}{(c_4-c_1)(c_3-c_1)(c_2-c_1)} & \frac{-c_1-c_3-c_4}{(c_4-c_2)(c_3-c_2)(c_2-c_1)} & \frac{c_1+c_2+c_4}{(c_4-c_3)(c_3-c_2)(c_3-c_1)} & \frac{-c_1-c_2-c_3}{(c_4-c_3)(c_4-c_2)(c_4-c_1)} \\ \frac{-1}{(c_4-c_1)(c_3-c_1)(c_2-c_1)} & \frac{1}{(c_4-c_2)(c_3-c_2)(c_2-c_1)} & \frac{-1}{(c_4-c_3)(c_3-c_2)(c_3-c_1)} & \frac{1}{(c_4-c_3)(c_4-c_2)(c_4-c_1)} \end{pmatrix}$$

This shows that in our case, the entries of the inverted matrix depend only on α_i, the points where the function is evaluated and which are public. We introduce the values λ_i corresponding to each of the individual entries of the first line of the inverted matrix.

$$\begin{bmatrix} ab \\ h_1 \\ \vdots \\ h_{2t} \end{bmatrix} = \begin{bmatrix} \lambda_1 & \dots & \lambda_n \\ \vdots & & \\ \dots & & \end{bmatrix} \begin{bmatrix} h(\alpha_1) \\ h(\alpha_2) \\ \vdots \\ h(\alpha_n) \end{bmatrix} \quad (4)$$

We are only interested in the first line of (3) since the only coefficient of interest is the secret ab. This corresponds exactly to the linear combination introduced earlier.

The protocol followed to compute the multiplication is the following.

- Each party P_i computes its share
$$[h(\alpha_i)] := [g_a(\alpha_i)][g_b(\alpha_i)] \quad (5)$$
locally.
- $[h(\alpha_i)]$ is then secret shared with the other parties using a degree t polynomial $H_i(x)$
- Now, each party holds a share $[H_1(\alpha_i)], ..., [H_n(\alpha_i)]$ of each other parties polynomial $H_i(x)$. Each party can compute the output
$$[H(\alpha_i)] = \lambda_1[H_1(\alpha_i)] + ... + \lambda_n[H_n(\alpha_i)] \quad (6)$$
locally.
- Each party P_i now holds a share $[H(\alpha_i)]$ of $H(x) := \lambda_1 H_1(x) + ... + \lambda_n H_n(x)$.

It is important to note that, because each $H_1(x), ..., H_n(x)$ is of degree t, $H(x)$ is again of degree t. Additionally, each $H_1(x), ..., H_n(x)$ is random since they are created by following the secret sharing scheme where all coefficients are random. Therefore, $H(x)$ is also random, since the addition of random functions is still random.

We have achieved a dimensionality reduction and ensured the randomness of the polynomial as desired. Unfortunately, the protocol for the computation of multiplications requires communication. This is inefficient. Therefore, optimization techniques such as Beaver triples have been developed to alleviate the communication costs. This will be further explained in the next section.

Output Reconstruction phase The output of a multiplication can be computed by evaluating

$$\begin{aligned} H(0) &= \lambda_1 H_1(0) + ... + \lambda_n H_n(0) \\ &= \lambda_1[h(\alpha_1)] + ... + \lambda_n[h(\alpha_n)] \\ &= ab \end{aligned}$$

3. Beaver Triples as Optimization Technique

The majority of the cost of the BGW protocol is caused by the communication required for multiplication operations. Ideally, a portion of the cost would be moved to the pre-processing phase. Unfortunately, the operations are dependent on the circuit inputs, which only become available during the online phase. Beaver triples allow for a way to move the majority of the communication to the pre-processing phase given even these circumstances. The basic idea is that the parties produce shared data during the offline phase that does not require information on the inputs [1].

A Beaver triple, also known as multiplication triple is a triple of three secret shared values $[a], [b], [c]$ where a, b are uniform random values unknown to all parties and $c = ab$.

3.1. Generation of the triples

There are different ways of generating the multiplication triples. A first option is trusted party generation. This protocol requires an honest third party that samples the triple (a, b, c) and distributes shares to the parties participating in the computation. The third party does not participate in the actual computation at all and does not have to be trusted with the actual inputs. However, it must compute the triples correctly and refrain from sharing their values. A second option is based on oblivious transfer which shall be shortly introduced.

doi: 10.2313/NET-2022-01-1_05

Oblivious Transfer A sender S holds two secrets x_0, x_1 and a receiver R holds a choice bit $b \in \{0,1\}$. The receiver learns x_b while staying oblivious to the other secret x_{1-b} and the sender does not learn anything about if or what information was transferred.

In the following, an example of generating triples based on oblivious transfer is illustrated. Say Alice and Bob want to generate a beaver triple. Firstly, Alice randomly samples values (x_A, y_A) and r_A and Bob randomly samples values (x_B, y_B) and r_B.

Next, Alice acts as the sender in an oblivious transfer with the input pair $(r_A, x_A \oplus r_A)$. Bob acts as the receiver using y_B as the selection bit. If $y_B = 0$, he learns r_A, else, he learns $x_A \oplus r_A$. In total, he learns $x_A y_B \oplus r_A$. The same thing is done in the other direction, so Alice learns $x_B y_A \oplus r_B$.

Now, Alice computes

$$z_A \leftarrow r_A \oplus x_A y_A \oplus x_B y_A \oplus r_B \tag{7}$$

and Bob computes

$$z_B \leftarrow r_B \oplus x_B y_B \oplus x_A y_B \oplus r_A \tag{8}$$

This results in

$$z_A \oplus z_B = (x_A \oplus x_B)(y_A \oplus y_B) \tag{9}$$

. Therefore, $(x_A, x_B), (y_A, y_B), (z_A, z_B)$ correspond to a Beaver triple.

Another option for the generation of the triples is based on homomorphic encryption [7], a way of computing on encrypted data without having to decrypt it.

3.2. Computing multiplications with Beaver triples

During the online step, the beaver triples are used during computation to diminish the necessary communication. Assume we generated a triple (a, b, c) using one of the generation methods. Additionally, we have the two input values α, β that we would like to securely multiply with each other. The parties hold secret shares of the input values. We define these as $[v_\alpha]$ and $[v_\beta]$. The computation with the beaver triples follows the following protocol [1].

1) Each party computes $[v_\alpha - a]$ and $[v_\beta - a]$ locally. Then, all parties publicly announce their shares in the form $d = v_\alpha - a$ and $e = v_\beta - b$ (the secret values v_α and v_β are hidden by a and b).

2) Following equality holds:

$$\begin{aligned} v_\alpha v_\beta =& (v_\alpha - a + a)(v_\beta - b + b) \\ =& (d + a)(e + b) \\ =& de + db + ae + ab \\ =& de + db + ae + c \end{aligned}$$

A share of $[v_\alpha v_\beta] = de + d[b] + e[a] + [c]$ is computed locally by each party.

Therefore, each party must only broadcast two values d and e per multiplication. This is much more cost effective than the communication required in plain BGW to compute multiplications.

4. Efficiency of BGW Protocol

Since, the BGW protocol uses secret sharing to hide its inputs, it does not rely on encryption. Generally, this is considered more efficient than a cryptographic approach. But as we have seen, certain operations involve complications and communication that strongly impact the efficiency.

The BGW protocol is very efficient for arithmetic circuits containing mostly additions [8]. Unfortunately, it is not ideal for functions requiring many multiplications. As a rule of thumb, we assume communication is much more expensive than computation and decryption. This means that tasks like matrix multiplications are difficult to solve with the BGW protocol. This would require many multiplication gates and a very large cost related to communication. For example, neural networks require extensive matrix multiplications and are not ideal for the BGW protocol.

Additionally, certain operations are expensive to represent as arithmetic circuits. Arithmetic circuits operate over a finite field F that must be set in advance and be large enough to prevent overflow [9]. In order to compute operations such as comparisons, bit-shifts and equality tests, a bit-decomposition is required. This conversion is expensive. Therefore, these are also operations that should be avoided with the BGW protocol.

5. Conclusion

This paper provided insight into the functionality of the BGW protocol for secure multiparty computation. Perhaps the most interesting component of this protocol is the degree reduction step, necessary to allow the computation of multiplication gates in a secure way. Unfortunately, this step also has significant negative impact on the efficiency of the protocol. This is why optimization techniques have been implemented to alleviate this impact, such as the Beaver triples introduced in this paper. All in all, the BGW protocol is often more efficient than protocols relying on a cryptographic approach. This depends on the type of function being evaluated. In general, operations based on many multiplications might be more efficiently computed with a different protocol.

References

[1] D. Evans, V. Kolesnikov, and M. Rosulek, *A Pragmatic Introduction to Secure Multi-Party Computation*, 2018, vol. 2, no. 2-3. [Online]. Available: http://dx.doi.org/10.1561/3300000019

[2] A. C. Yao, "Protocols for secure computations," pp. 160–164, 1982.

[3] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay — a secure two-party computation system," 06 2004.

[4] M. Ben-Or, S. Goldwasser, and A. Wigderson, “Completeness theorems for non-cryptographic fault-tolerant distributed computation,” in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC ’88. New York, NY, USA: Association for Computing Machinery, 1988, p. 1–10.

[5] A. Shamir, “How to share a secret,” *Commun. ACM*, vol. 22, no. 11, p. 612–613, Nov. 1979.

[6] E. Rawashdeh, “A simple method for finding the inverse matrix of vandermonde matrix,” 01 2020.

[7] S. University, “Cs 355: Topics in cryptography.” [Online]. Available: https://crypto.stanford.edu/cs355/18sp/lec7.pdf

[8] T. Rabin, “Secure multiparty computation,” 2014. [Online]. Available: https://www.youtube.com/watch?v=NOtsxHoIcWQ&t=618s&ab_channel=Technion

[9] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, “Sok: General purpose compilers for secure multi-party computation,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1220–1237.

doi: 10.2313/NET-2022-01-1_05