Survey of Arithmetic Properties of Multi-Party Computation Schemes

Risto Pärnapuu

December 20, 2019

Abstract

Multi-party computation (MPC) is a method where multiple parties jointly compute some function without revealing their private inputs. There are about three mainly used or discussed primitives that power MPC schemes. These are Garbled Circuits, Secret Sharing and fully homomorphic encryption. In practical terms, the main difference in the MPC schemes is the number of supported operations and and the computational and computational complexities of those operations. Additionally, the security requirements and guarantees of the schemes differ.

In this report, we will give a brief overview of the three primitives that are mainly used for creating MPC schemes and see some examples of MPC schemes based on those primitives. We will survey the MPC schemes in terms of the supported operations, how are those operations performed and what is their computational complexity. Finally we compare the surveyed schemes in those terms.

Contents

1 Introduction

Multi-party computation (MPC, also known as secure MPC) is a process which allows multiple parties $P_1, P_2, \ldots, P_n$ to jointly compute some function f while keeping the inputs $x_1, x_2, \ldots, x_n$ of the parties private. Privacy in this case means that the only new piece of information that is released is the output of f even if some number of the parties are malicious (exact number of supported malicious parties depends on a specific protocol).

In general there are two different primitives that power practical MPC schemes. These are Garbled Circuits and Secret Sharing. The majority of the modern MPC schemes are based on Secret Sharing. There exist ideas for MPC protocols that do not rely on Secret Sharing nor Garbled Circuits. One such example is schemes based of fully homomorphic encryption. However schemes based on it, are generally not considered to be practical due to the computational complexity of circuit evaluation.

The most important practical distinguisher of the protocols is what kind of operations are supported. These operations inherently rely on the primitives underlying the protocol. Most protocols are designed so that they support the basic arithmetic operations like addition and multiplication. Intuitively, addition (and consequently subtraction) is much easier to support than multiplication. Some other operations (e.g. equality checks) are also supported by some of the protocols but in the context of this paper, we will take a look at the two basic operations addition and multiplication. It also happens that, often other protocols can be composed from addition and multiplication.

The number of parties participating in the computation is also often limited due to the restrictive factors of the underlying primitive. It is also worth noting that often these restrictive factors can and have been worked around by modifying the original scheme in a clever way, which intuitively often increases the computational complexity of the scheme. In this paper we will be focusing more on the schemes based on the original primitive and will not be looking that deep into the existing improvements of the schemes.

The aim of this paper is to give an overview of different MPC schemes. We shall analyze different primitive structures powering the MPC schemes in Section 2. For each primitive we will give an one or two examples of the MPC schemes based on it and provide a sufficient analysis so that schemes can be compared based on various metrics in the last Section 3.

2 Primitives powering MPC schemes

In this section we categorize MPC protocols based on the underlying primitive and analyze each protocol individually. First we analyze and give an overview of the primitive itself. Then we will take a look at the general structure of the protocol, what kinds of operations can be performed, how are those operations achieved and what is the complexity of performing those operations.

2.1 Garbled circuits

Garbled circuit is a cryptographic protocol which was first introduced by Andrew Yao [22]. Yao described a scheme that works on boolean circuits to enable secure 2-party computation. The evaluated function between the parties must be represented as a boolean circuit which both parties know. Circuit garbling (or garbling scheme) consists of three distinct parts:

- *Garbling* $Gar(C) = \hat{C}$
  A process which takes the boolean circuit C and converts it into a garbled circuit $\hat{C}$.
- *Encoding* $Enc(x) = \hat{x}$
  Process which takes an input x to the circuit C and converts into an input $\hat{x}$ for the garbled circuit $\hat{C}$.
- *Evaluation* $Eval(\hat{C}, \hat{x}) = C(x)$
  Process which uses the garbled circuit $\hat{C}$ and the obfuscated input $\hat{x}$ to obtain the output of the original circuit C on the original input x.

A garbled circuit $\hat{C}$ in this scheme can be thought as an "encryption" of the original circuit which completely hides everything besides the output of the circuit C. I.e. the input x to the circuit and all the intermediate values will be kept hidden.

Protocol described by Yao works as follows:

1. Two parties agree on a function f and a way to represent it as a boolean circuit C. Both parties receive the circuit.
2. Party one garbles the circuit $Gar(C) = \hat{C}$ and its input $Enc(x) = \hat{x}$ and sends it to party two.
3. Party two needs to garble its input y to $\hat{y}$. Since only party one knows how to garble the inputs, two parties perform an oblivious transfer protocol so that party two learns the obfuscated input $\hat{y}$. The oblivious transfer protocol makes sure that party one doesn't learn anything about party two input y.
4. Now that party two has the garbled circuit $\hat{C}$, obfuscated input $\hat{x}$ of party one and its own obfuscated input $\hat{y}$, it can compute the evaluation function $Eval(\hat{C}, \hat{x}, \hat{y})$ and learn the output of the circuit $C(x, y)$.
5. Party two reveals the output $C(x, y)$ to party one.

Since schemes based on Garbled Circuits work on boolean circuits, any function that can be represented as a boolean circuit can be used in this type of scheme. Main computational complexity of this scheme comes from circuit generation and individual gate executions. In the garbled circuit, all possible bit values of all possible wires are encrypted using an encryption function (let's denote it by E). Initial protocol proposed by Yao took approximately 4 calls to encryption function for every gate in the circuit. The encryption function can be a symmetric key encryption. The use of an encryption function on each gate leads to a much higher computational complexity when comparing garbled circuit evaluation to a conventional calculation.

Over the years, there have been quite a many optimizations on the original Yao's Garbled Circuit that decreased the number of encryption calls needed. Table 1 highlights the most notable improvements of Yao's protocol.

technique	size per gate		calls to E per gate			
			generator		evaluator	
	XOR	AND	XOR	AND	XOR	AND
classical	4	4	4	4	4	4
point and permute	4	4	4	4	**1**	**1**
free XOR	**0**	4	**0**	4	**0**	**1**
GRR3 + free XOR	**0**	**3**	**0**	4	**0**	**1**
GRR2	**2**	**2**	4	4	**1**	**1**
fleXOR	**{0,1,2}**	**2**	**{0,2,4}**	4	**{0,1,2}**	**1**
half gates	**0**	**2**	**0**	4	**0**	**2**
garbled gadgets	**2**	**2**	**3**	**3**	**1**	**1**

Table 1: Optimizations of Garbled Circuits. Size is number of "ciphertexts". Table taken from [21]. Improvements over the classical protocol are marked with bold.

Since we want to compare the complexity of addition and multiplication of n bit integers, we can use the previous table to express the complexly of these operation in number of calls to E. For addition we assume for simplicity that full adder method is used to add two n bit integers, where for each bit there exists 3 XOR gates and 2 AND gates. For addition we get the following table 2.

	calls to E per bit	
	generator	evaluator
classical	$3*4+2*4=20$	$3*4+2*4=20$
point and permute	$3*4+2*4=20$	$3*1+2*1=5$
free XOR	$3*0+2*4=8$	$3*0+2*1=2$
GRR3 + free XOR	$3*0+2*4=8$	$3*0+2*1=2$
GRR2	$3*4+2*4=20$	$3*1+2*1=5$
fleXOR	$\{0,6,12\}+2*4=\{8,14,20\}$	$\{0,3,6\}+2*1=\{2,5,8\}$
half gates	$3*0+2*4=8$	$3*0+2*2=4$
garbled gadgets	$3*3+2*3=15$	$3*1+2*1=5$

Table 2: Computational complexity of addition using Garbled Circuits.

In general though, binary addition still stays linear in terms of number of bits. So for computational complexity we can say that it takes $O(n)$ calls to the encryption function.

For multiplication there is no good iterative combinatorial circuit available. Instead we can take the schoolbook version which says that it uses roughly n^2 AND gates for multiplying two n bit integers. In terms of complexity, this is all we need since we can now estimate the number of calls to H which is $O(n^2)$. Note that in terms of complexity there exist multiplication algorithms that achieve much better computational complexity (e.g. Karatsuba algorithm for large integer multiplication which has complexity of $O(n^{1.585})$. Karatsuba algorithm however surpasses the schoolbook version only if the values are sufficiently large (320–640 bits) [12]. Similar things can be said about other multiplication algorithms that achieve better computational complexity. Thus, for the sake of simplicity, we assume the schoolbook method.

2.2 Secret sharing

Secret sharing is a scheme first introduced by Adi Shamir [19] which enables sharing a secret to a number of other parties by distributing parts of the secret to each party. Parts are constructed in such a way that the secret can only be reconstructed if sufficient number of shares are combined together. The exact number of shares needed for a successful reconstruction depends on the scheme. Schemes that have N parties and need $k < N$ parts of the secret to be reconstructed are sometimes referred to as *threshold Secret Sharing schemes*. Any number of shares below the threshold share should not reveal anything about the actual secret.

A Secret Sharing scheme is considered to be verifiable if parties are able to verify the consistency of individual parts. In verifiable Secret Sharing, parties are not able to lie about their parts of the secret. Verifiability of a scheme is generally added as a separate construction and most schemes are not inherently verifiable. There are multiple ways to add verifiability to a scheme, e.g. using a commitment scheme. We will not be focusing on the verifiability of a scheme and will be looking at the constructions that power the actual computations.

It has been shown that secure multi-party computation can efficiently be based on any linear secret scheme, provided that the access structure of the scheme allows MPC at all [8]. Linearity means that parties can compute arbitrary linear functions of the underlying secrets without interaction.

We shall analyze two different kind of secret sharing schemes. In section 2.2.1 we will take a look Additive secret sharing schemes and in section 2.2.2, we look at Shamir's Secret Sharing schemes.

2.2.1 Additive schemes

Additive Secret Sharing schemes are the simplest ones, based on basic arithmetic properties of a finite field. In an additive scheme, the secret x is assumed to be from some finite field $\mathbb{F}$. The main idea of an additive scheme is to split the secret into n different parts $x_1, x_2, \ldots, x_n$ such that their sum will give back the secret x. This is done by first picking $n-1$ elements uniformly at random from the field $\mathbb{F}$:

$$x_1, x_2, \ldots, x_{n-1} \leftarrow \mathbb{F}$$

and then computing the last element in such way that the $\sum_{i=1}^{n} x_n = x$:

$$x_n = x - x_1 - x_2 - \ldots - x_{n-1}.$$

Intuitively, picking the parts that way gives us additive homomorphic property for MPC schemes where addition function can easily be calculated by adding together shares from all parties. In a sense, we get addition for "free" meaning no additional constructions need to be introduced. Above Secret Sharing also requires all parties to contribute all of their shares to reconstruct the secret, i.e. it's not a threshold scheme. If even one of the shares of some participant is missing, no information about about the secret will be revealed, because individual shares seem random.

2.2.1.1 SPDZ

SPDZ [10] is an additive N-party MPC protocol that computes an arithmetic circuit C. SPDZ uses a preprocessing model which works in two phases. Generally the two phases are called offline and online phases. The offline phase runs before the online phase and generates data that will later be used by the online phase. In the online phase we assume that all parties have access to the oracle that distributes the data generated by the offline phase.

The evaluated function is represented as an arithmetic circuit. Arithmetic circuits consist of either addition or multiplication operations.

First step of any circuit evaluation is to share the inputs of the parties. For the party P_i to share its input x each party must hold some random value r_i

such that

$$r = \sum_{i=1}^{N} r_i$$

and r is uniformly random and not known by any party. Parties then send their share of r to party P_i. This means that P_i learns the value of r. This is called partial opening, since the party P_i learns the value of r. Next P_i, broadcasts the value $\epsilon = x - r$. Then the shares of the parties are set as follow:

1. Some party P_t sets its share as $x_t = r_t + \epsilon = r_t + x - r$
2. Party P_j where $j \neq t$ set its share as $x_j = r_j$

Addition of x and y is performed using the basic properties of an additive scheme where each party $P_i$can locally compute $x_i + y_i$ and hence obtain $\sum_i (x_i + y_i)$.

Multiplication however gets a bit trickier. This is where the offline phase of the protocol is required. SPDZ uses something called *multiplication triplets* [1] which can be generated in the offline phase. Offline phase is based on a somewhat homomorphic encryption (SHE). A multiplication triplet t consists of three values (a, b, c), where a and b are picked uniformly at random and c is computed as $c = ab \pmod{q}$, where q is a prime. SPDZ assumes that these triplets are pre-generated in the offline phase and distributed to the parties so that for the i-th multiplication all parties have access to the triplet t_i. For the two parties to compute the product of two values x and y, they use a and b from the triplet t to mask the secrets to $x' = x - a$ and $y' = y - b$. Then the computation works as follows. First we define following values:

$$\begin{aligned} x'y' &= (x-a)(y-b) = xy - xb - ay + ab \\ ay' &= a(y-b) = ay - ab \\ bx' &= b(x-a) = bx - ab \end{aligned}$$

Next each party broadcasts the values $x_i - a_i$ and $y_i - b_i$ and can then compute the value $a_i(y - b) + b_i(x - a_i) + c_i$. Summation of these values with along with the public values $(x - a)$ and $y - b$ gives us the desired result:

$$\begin{aligned} x'y' + ay' + bx' + c &= xy - xb - ay + ab + ay - ab + bx - ab + c \\ &= xy + (bx - xb) + (ay - ay) + (ab - ab) + (ab + c) = xy \end{aligned}$$

Previous is a generic outline of the online phase of the SPDZ protocol. In original SPDZ protocol offline (preprocessing) phase has computational complexity of $O(\frac{n^2}{s})$, where s is a number that grows with the security parameter of used SHE scheme. Online phase has computaional complexity $O(n \cdot |C| + n^3)$ where $|C|$ is the size of the arithmetic circuit C. The authors implemented the SPDZ protocol and tested the execution time of a single multiplication. They obtained total of 13 ms for the offline phase and 0.05 ms for the online phase.

Over the years, there have been quite many improvements to the original SPDZ protocol, especially in the preprocessing (offline) phase. Original authors themselves have proposed second version of SPDZ, SPDZ2 [9] which also offers quite many performance improvements mainly in the offline phase.

2.2.1.2 Sharemind

Sharemind [3] is platform which provides MPC capabilities for practical data processing applications. It's important to note that the latest Sharemind supports multiple MPC protocols, but we will be analyzing the originally defined Sharemind 3-party MPC protocol that works as follows.

As in SPDZ, addition is also performed using the basic properties of an additive scheme. The summation of shares is obtained by adding together the shares of each party. This can be done locally by each party and does not require additional communication with other parties. Multiplication of two values x and y however needs a bit more work as additive Secret Sharing does not have homomorphic property for multiplication. Sharemind multiplication protocol works as follows. First, the product of two values x and y is expressed as:

$$\begin{aligned} xy =&(x_1 + x_2 + x_3)(y_1 + y_2 + y_3) \\ =&x_1y_1 + x_1y_2 + x_1y_3+ \\ &x_2y_1 + x_2y_2 + x_2y_3+ \\ &x_3y_1 + x_3y_2 + x_3y_3. \end{aligned}$$

Since each party P_i has x_i and y_i, they can locally compute the value x_iy_i. To compute the full sum as seen in the previous equation, parties need to exchange information about the additional shares. This must be done so that the values of the shares are kept confidential (otherwise some party may be able to construct the original values x and y). Sharemind does this by a process of temporary resharing of the original values x and y so that new shares could be exchanged. Resharing protocol takes a shared value x and outputs a shared value x' such that $x = x'$, all shares x'_i are uniformly distributed and x_i and x'_j are independent for $i, j = 1, 2, 3$. It works as follows:

1. P_1 generates a uniformly distributed $r_{12} \leftarrow \mathbb{F}$.
2. P_2 generates a uniformly distributed $r_{23} \leftarrow \mathbb{F}$.
3. P_3 generates a uniformly distributed $r_{31} \leftarrow \mathbb{F}$.
4. P_i sends r_{ij} to P_j.
5. P_1 calculates $x'_1 = x_1 + r_{12} - r_{31}$.
6. P_1 calculates $x'_2 = x_2 + r_{23} - r_{12}$.
7. P_1 calculates $x'_3 = x_3 + r_{31} - r_{23}$.
8. x' is obtained as resharing of $x = x_1 + x_2 + x_3$.

Now that we have the protocol for resharing, we can define the protocol for multiplication which computes the product $p = xy$, where the shares of x and y are distributed between parties P_1, P_2, P_3. Multiplication protocol works as follows:

1. x is reshared and x' is obtained.
2. y is reshared and y' is obtained.
3. Party P_1 sends shares x'_1 and y'_1 to P_2.
4. Party P_2 sends shares x'_2 and y'_2 to P_3.
5. Party P_3 sends shares x'_3 and y'_3 to P_1.
6. Party P_1 computes $p'_1 = x'_1y'_1 + x'_1y'_3 + x'_3y'_1$.
7. Party P_2 computes $p'_2 = x'_2y'_2 + x'_2y'_1 + x'_1y'_2$.
8. Party P_3 computes $p'_3 = x'_3y'_3 + x'_3y'_2 + x'_2y'_3$.
9. p' is reshared and $p = xy$ is obtained.

For reference, Sharemind also supports the following operations which we will not be describing further:

- Casting from Z_2 to $Z_{2^{32}}$;
- Equality check;
- Bit shifting to right;
- Bit extraction;
- Division with public and private divisor.

Computational complexity of addition and multiplication depend on the bit length n of the inputs. Since addition is local, it requires no communication and no rounds. Multiplication however is a single round protocol with communication cost of $15n$, where n is the number of data bits.

2.2.2 Shamir's Secret Sharing

Shamir's Secret Sharing (or polynomial sharing) is a linear Secret Sharing scheme proposed by Adi Shamir [19]. Shamir's Secret Sharing is a threshold scheme, which means that for k out of n shares, a coalition of parties with k or more shares will be able to reconstruct the secret.

Shamir's Secret Sharing scheme builds upon a fact that any k points define a polynomial of degree $k - 1$. The scheme works as follows:

1. Owner of the secret s generates a $k-1$ degree polynomial:

$$f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{k-1}x^{k-1}$$

   such that $a_0 = s$ and $a_1, a_2, \ldots, a_{k-1}$ are sampled randomly.

2. Owner then chooses n distinct non-zero values x_1, x_2, …, x_N and generates the n shares

$$f(x_1), f(x_2), \ldots, f(x_N)$$

3. Generated shares pair tuples $(x_i, f(x_i))$ are shared among participants of the scheme. Note that if the scheme fixes $x_i = i$ then it is sufficient to share only $f(x_i)$ values among the participants.

Any party with k or more shares will be able to reconstruct the secret using polynomial interpolation.

Feldman's scheme [11] is a modification of Shamir's Secret Sharing with the addition of verifiability. This is achieved by using homomorphic encryption which allows multiplication and addition to be performed on ciphertext values, resulting in a ciphertext that is equal to performing the same operations on the plaintexts and then encrypting the result.

2.2.2.1 GRR multiplication protocol

The protocol of Gennaro, Rabin and Rabin (the GRR protocol) [13] enables computing of the product of two secrets α and β that are shared as polynomials $f_\alpha(x)$ and $f_\beta(x)$ with degree $t = k-1$. GRR is based on Shamir's Secret Sharing, with an additional low-cost construction which is a public commitment of the dealer to each one of the shares of the parties. This extra construction adds the verifiability of the shares and works as follow:

1. The dealer chooses two random polynomials $f(x)$ and $g(x)$, both of degree t.

2. Like in the Shamir's Secret Sharing scheme constant term of $f(x)$ will be set as the shared secret s.

3. $g(x)$ will be used to generate t-pairwise independent random strings which will be used to commit to the shares.

4. Each player P_i will receive both $f(i)$ (share of the secret) and $r(i)$, which is the randomness associated with P_i.

5. The dealer publicly commits to the shares of all players by broadcasting $C(f(i), r(i))$, where C is a commitment function.

Note that to enable MPC for multiplication using the previous construction, it is not sufficient for each player to multiply its shares locally as it would generate a non-random polynomial with constant term being $\alpha\beta$, but degree of $2t$.

The problematic parts being non-randomness and degree $2t$. This problem was originally solved [2] by using degree reduction and re-randomization. Authors of the GRR multiplication protocol proposed a way to achieve both of these things in a single step which works as follows.

Let $f_\alpha(i)$ and $f_\beta(i)$ be the shares belonging to the player P_i. Then the product $f_\alpha(x)f_\beta(x)$ has the form

$$f_{\alpha\beta}(x) = f_\alpha(x)f_\beta(x) = \alpha\beta + \gamma_1 x + \ldots + \gamma_{2t}x^{2t}.$$

Additionally, for $1 \leq i \leq 2t+1$ it must hold that $f_{\alpha\beta}(i) = f_\alpha(i)f_\beta(i)$. Which means we can write:

$$A \begin{bmatrix} \alpha\beta \\ \gamma_1 \\ \vdots \\ \gamma_{2t} \end{bmatrix} = \begin{bmatrix} f_{\alpha\beta}(1) \\ f_{\alpha\beta}(2) \\ \vdots \\ f_{\alpha\beta}(2t+1) \end{bmatrix}.$$

where A is a non-singular $(2t+1) \times (2t+1)$ matrix with $a_{ij} = i^{j-1}$. Let's take the inverse of this matrix A^{-1} and define the first row of it as $(\lambda_1, \ldots, \lambda_{2t+1})$ where all elements are known constants. Then we get that:

$$\alpha\beta = \lambda_1 f_{\alpha\beta}(1) + \ldots + \lambda_{2t+1} f_{\alpha\beta}(2t+1).$$

Now, given polynomials $h_1(x), \ldots, h_{2t+1}(x)$ with degree t that satisfy $h_i(0) = f_{\alpha\beta}(i)$ for $1 \leq i \leq 2t+1$ we define

$$H(x) = \sum_{i=1}^{2t+1} \lambda_i h_i(x)$$

Note that $H(0) = \alpha\beta$. Now, if the party P_i uses polynomial $h_i(x)$ to share its share, then polynomial $H(x)$ that is used for sharing the multiplication $\alpha\beta$ is of degree t which solves the first problem with polynomial degree. It also has to be random because all values λ_i are non-zero and there are $n-t$ polynomials chosen at random by the honest parties (because honest parties must follow the protocol).

Such constructions $H(x)$ enables us to define the GRR multiplication protocol which works as follows:

1. Party P_i chooses a polynomial $h_i(x)$ of degree t at random, such that $h_i(0) = f_\alpha(i)f_\beta(i)$.
2. Party P_i shares the value $h_i(x)$.
3. Party P_i gives to party P_j the value $h_i(j)$, for $1 \leq j \leq 2t+1$.
4. Each party P_j computes its share of the multiplication $\alpha\beta$ using the random polynomial H to compute $H(j) = \sum_{i=1}^{2t+1} \lambda_i h_i(j)$.

2.3 Fully homomorphic encryption

Homomorphic encryption is a specific form of encryption such that when computations are performed on encrypted data, the result will be an encryption of the computation as if it were performed initially on the data itself and then encrypted. We can describe the homomorphic property as:

$$f(Enc(p_1), Enc(p_2)) = Enc(g(p_1, p_2)),$$

where f and g represent the computation. Note that f and g don't have to necessarily be the same function. The computations themselves are represented as arithmetic or boolean circuits. Homomorphic encryption can be divided into subgroups by the computations they support. Most common subgroups are:

- Partially homomorphic encryption - Supports evaluating circuits that compose of a single type of gate.
- Somewhat homomorphic encryption - Supports evaluating a subset of circuits that compose of two type of gates.
- Leveled fully homomorphic encryption - Supports evaluating any circuit with a bounded depth.
- Fully homomorphic encryption (FHE) - Supports evaluating any circuit where depth is unbounded.

Some of them are in a sense more “weaker“, e.g. partially homomorphic encryption is only homomorphic for a single type of computation (addition or multiplication). Fully homomorphic encryption is the strongest, as it supports evaluation arbitrary circuits of unbounded depth.

It's clear that homomorphic encryption is generally less secure than non-homomorphic encryption as it is inherently malleable. Thus it's less preferable in most circumstances. However, for powering MPC schemes, fully homomorphic encryption would work great because one can use the homomorphic properties to perform the computations.

Constructing fully homomorphic encryption, however, is not easy. In fact, the first construction of a fully homomorphic encryption (which used lattice-based cryptography) was proposed in 2009 by Craig Gentry [14]. This scheme has been extensively studied and various improvements on the original protocol have been proposed [4, 6, 7, 15, 20]. The main idea however has stayed pretty much the same. They start with a somewhat homomorphic encryption (which means they are able to evaluate addition and multiplication gates, but only for a specific subset of circuits) and convert into a fully homomorphic encryption by reducing the ”noise” which increases each time computations are performed on the ciphertexts and doing it periodically when it grows too large. Such process is referred to as bootstrapping. Over the years, new research [5, 18] has been done to decrease the growth rate of the noise when homomorphic computations are performed. Such schemes are often referred to as second generation FHE schemes.

Traditional FHE schemes are based on assumption that the cyphertexts are encrypted by the same key. However for MPC protocols, it would be much more useful if each party could use their own key. Multi-key fully homomorphic encryption (MKFHE) [18] preserves homomorphic properties even if the inputs are encrypted by parties who each have their own key and the computation results can be jointly decrypted by using the secret keys of all the parties.

On-the-fly MPC protocol based on MKFHE was proposed in the same paper as MKFHE [18]. The protocol involves an untrusted computationally powerful third party who performs computations on the ciphertexts sent by the users. Each user interacts with the third party twice: once when the encrpytions of the data are sent and once again after the computation is performed (decryption phase). The two interactions are independent, meaning that parties to not need to choose the computation before sharing the encrypted data. Proposed MKFHE is based on NTRU FHE encryption which is transformed into a multi-key version. The authors note that such transformation does impact the efficiency of the NTRU somewhat, but they believe that such construction is a leading candidate for a practical FHE scheme. Very general outline of the protocol looks a s follows:

1. Each party generates an independent keypair pk_i, sk_i and an evluation key ek_i.
2. Parties encrypt the inputs x_i using their own public keys: $c_i = Enc(pk_i, x_i)$.
3. Parties send the encrypted inputs to the third party along with the public and evaluation keys, pk_i and ek_i.
4. The third party chooses the function f and an arbitrary subset of inputs of the parties.
5. The third party computes the function f over the inputs of the chosen parties and sends the result to each of the participating parties.
6. Parties together run an interactive MPC protocol to decrypt the output using their secret keys sk_i.

The computational and communication complexity on the parties in this scheme is completely independent of the complexity of the evaluated function and the total number of parties in the system.

Improvements on the previous protocol have been proposed in [17] which proposes a way to make homomorphic evaluations more efficient. The proposed scheme first constructs a single key fully homomorphic encryption where the number of relinearization steps is reduced by separating the homomorphic multiplication and the relinearization. Next, they transform the constructed single-key homomorphic encryption into multi-key homomorphic encryption. Finally, they construct a distributed decryption process which can be implemented independently by each party. Based on these constructions, they construct a two-round MPC protocol which improves on the one proposed in [18].

In terms of computational complexity however, homomorphic calculation in existing FHE schemes is very slow. Compared to conventional calculations, calculations on early FHE schemes were trillions of times slower. Further work [16] has improves on this, but it still remains unusable for practice for most use cases.

3 Comparison of protocols

Firstly, we can compare the surveyed schemes in terms of how many different parties can participate in it. This is summarized in Table 3.

Protocol	**Number of parties**
Garbled Circuits	2
SPDZ	N
Sharemind	3
GRR	N
FHE schemes	N

Table 3: Number of parties supported in analyzed schemes.

In terms of supported operations, both Garbled Circuits and fully homomorphic encryption support all computations that can be represented as boolean circuit, which obviously includes addition and multiplication. The computational complexity of garbled circuit based schemes mainly comes from the fact that circuit scrambling requires quite many calls to an symmetric encryption function which can be costly when compared to the conventional circuit evaluation, especially if the number of gates increases. As seen in Table 1, further work has decreased the number of calls to the encryption function, but the computations would still take much more time compared to schemes based on Secret Sharing. Schemes based on FHE are even slower as calculations take a lot more time when compared to conventional calculations. In fact, calculation done in most of the FHE schemes are millions of times slower than the conventional calculation. Active work is being done on it [16], but so far it remains impractical for most use cases.

Schemes based on secret sharing schemes are however a bit trickier to compare as there is a larger variety of the schemes in general. In terms of supported operations of the analyzed schemes, both SPDZ and Sharemind protocols amongst other computations, support addition and multiplication. GRR multiplication obviously is designed to support only multiplication, but from the basic properties of Shamir's Secret Sharing, it would be very easy to also support for addition. In terms of performance, SPDZ and Sharemind are currently the most actively researched protocols and the latest versions of them are improving in performance constantly.

4 Conclusion

In this report we categorized existing MPC protocols based on the primitives that power them. We found that, in general there are roughly three types of such primitives. These are Garbled Circuits, Secret Sharing and fully homomorphic encryption.

Garbled circuits as described by Yao is a primitive that itself is a 2-party computation protocol which supports evaluation of any computation that can be represented as a boolean circuit.

Secret sharing is by far most generic primitive and offers more protocols that differ from each other unlike Garbled Circuits and fully homomorphic encryption which provide quite concrete MPC schemes. We took a look at additive Secret Sharing schemes and Shamir's Secret Sharing schemes and provided examples of existing MPC protocols based on them.

Fully homomorphic encryption is relatively new research field for which actual possible constructions were first proposed in 2009 [14]. The main issue of the MPC schemes based on FHE is that the computational complexity is very high and actual evaluations of the arithmetic circuits take long time when compared to existing MPC schemes based on for example Secret Sharing.

While being academically very intriguing, MPC schemes based on FHE schemes currently are not used in practice due to the high computational overhead. SPDZ and Sharemind are currently amongst the most researched and actively worked on MPC protocols. Both enable addition and multiplication amongst other computations with good enough performance for practical use.

References

[1] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

[3] Dan Bogdanov. Sharemind: programmable secure computations with practical applications. 2013. `https://cyber.ee/research/theses/dan_bogdanov_phd.pdf`.

[4] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, Oct 2011.

[5] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Report 2011/277, 2011. `https://eprint.iacr.org/2011/277`.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 505–524, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, pages 316–334, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[9] Ivan Damgard, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. Cryptology ePrint Archive, Report 2012/642, 2012. `https://eprint.iacr.org/2012/642`.

[10] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multi-party computation from somewhat homomorphic encryption. In *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*, pages 643–662, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438, Oct 1987.

[12] Tommy Färnqvist. Number theory meets cache locality - efficient implementation of a small prime fft for the gnu multiple precision arithmetic library. 01 2005.

[13] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 101–111, New York, NY, USA, 1998. ACM.

[14] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[15] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. Cryptology ePrint Archive, Report 2013/340, 2013. `https://eprint.iacr.org/2013/340`.

[16] Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. Cryptology ePrint Archive, Report 2018/244, 2018. `https://eprint.iacr.org/2018/244`.

[17] NingBo Li, TanPing Zhou, XiaoYuan Yang, YiLiang Han, Longfei Liu, and WenChao Liu. Two round multiparty computation via multi-key fully homomorphic encryption with faster homomorphic evaluations. Cryptology ePrint Archive, Report 2018/1249, 2018. `https://eprint.iacr.org/2018/1249`.

[18] Adriana Lopez-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. Cryptology ePrint Archive, Report 2013/094, 2013. `https://eprint.iacr.org/2013/094`.

[19] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[20] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 24–43, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[21] Sophia Yakoubov. A gentle introduction to yao ' s garbled circuits. 2017.

[22] A. C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, Oct 1986.