Blindly Follow: SITS CRT and FHE for DCLSMPC of DUFSM

(Preliminary Version)

Shlomi Dolev and Stav Doolman

Department of Computer Science, Ben-Gurion University of the Negev, Israel

Abstract. A Statistical Information Theoretic Secure (SITS) system utilizing the Chinese Remainder Theorem (CRT), coupled with Fully Homomorphic Encryption (FHE) for Distributed Communication-less Secure Multiparty Computation (DCLSMPC) of any Distributed Unknown Finite State Machine (DUFSM) is presented. Namely, secret shares of the input(s) and output(s) are passed to/from the computing parties, while there is no communication between them throughout the computation. We propose a novel approach of transition table representation and polynomial representation for arithmetic circuits evaluation, joined with a CRT secret sharing scheme and FHE to achieve SITS communication-less within computational secure execution of DUFSM. We address the severe limitation of FHE implementation over a single server to cope with a malicious or Byzantine server. We use several distributed memory-efficient solutions that are significantly better than the majority vote in replicated state machines, where each participant maintains an FHE replica. A Distributed Unknown Finite State Machine (DUFSM) is achieved when the transition table is secret shared or when the (possible zero value) coefficients of the polynomial are secret shared, implying communication-less SMPC of an unknown finite state machine.

Keywords: Secure Multiparty Computation, Replicated State Machine, Chinese Remainder Theorem

1 Introduction

The processing of encrypted information where the computation program is unknown is an important task that can be solved in a distributed fashion using communication among several participants, e.g., [14]. Unfortunately, this communication reveals the participants to each other and requires a non-negligible overhead concerning the communication between them. Computational secure communication-less approaches can also be suggested, either for the case of known automaton and global inputs, e.g., [12] or for the case of computational security alone. Here, we present the first communication-less solution that is statistical information-theoretical secure, with (FHE-based) computational secure scheme.

A major contribution to the area of distributed computing is the replicated state machine introduced by Lamport [24]. The implementation of such a state machine is usually based on a distributed consensus [23].

We propose a sharing scheme that is based on a secret shared transition function or a unique polynomial over a finite ring for implementing e.g., Boolean function, state machine transition, control of RAM, or control of Turing Machine. Specifically, for any state machine, this polynomial encodes the information of all the transitions from a state x and input y to the next state z. The information may also contain the encoding of the output. Once the polynomial is adjusted, it is described by an arithmetic circuit that can be evaluated distributively by the SMPC participants. Each participant evaluates the arithmetic circuit using the CRT SITS secret sharing scheme where the shares are FHE encrypted. Consequently, the possibility for (value secured) additions and multiplications with no communication is achieved. In the scope of this polynomial representation of the transition function, the actual computed function is kept private by using secret shares for all (zero and non-zero) coefficients of the polynomial, revealing only a bound on the maximal degree k of the polynomial.

The CRT representation allows independent additions and multiplications of the respective components of two (or more) numbers over a finite ring. In that manner, we can compute arithmetic circuits in a distributed fashion, where each participant performs calculations over a finite ring defined by the (relatively) prime number they are in charge of. Thus, we accomplish a distributed polynomial evaluation, where several participants do not need to communicate with each other.

The transition function of a state machine may be represented by a bi-variate polynomial from the current state and the input to the next state (and output). Namely, a bi-variate polynomial can be defined by the desired points that define the transition from the current state (x) and the input (y) to the next state (z), which may encode the output too. Alternatively, a univariate polynomial can be defined by using the most significant digits of $(x + y)$ to encode the state (x) and the least significant digits, to encode the input (y). The output state (z) occupies the same digits of (x) that serve to encode the next state, while the rest of the digits in (z) are zeros. Thus, the next input can be added to the previous result and be used in computing the next transition, and so forth.

By using our suggested scheme, one can implement a more efficient version of the famous replicated state machine (and Blockchain) with only a logarithmic-sized memory compared to the legacy replicated state machine.

Naturally, several known error correction techniques that rely on features of the Chinese Remainder Theorem (depicted in [20, 21]) can eliminate the influence of Byzantine participants. These schemes are not designed to preserve the fully homomorphic property of CRT secret sharing, just as the CRT threshold secret sharing does not support additions and multiplications as the values can exceed the global maximal value the (original, with no additional error-correcting values) mutual primes can represent. Still, when using FHE, a computation can be designed to never exceed this maximal value and be error corrected. Note that FHE has a significant complications when executed over a single server, as the server can be Byzantine, not following the algorithm it should execute on the encrypted values, thus, a distributed secure multiparty computation is preferred.

Related Work and Our Contribution. Recently, extensive work on computationally secure communication-less computation has been done, see [15, 9] and in references therein. However, the computation security is only based on the belief that one-way functions exist [11]. Several other works in the scope of perfect information-theoretically secure schemes were presented in [14, 16, 13, 5, 17]. Unfortunately, neither of them can compute all possible functions, and they require either communication or a need for exponential resources to maintain continuous functioning. Function secret sharing (FSS) is described in-depth in [8], and provides an efficient solution for MPC. However, the suggested scheme relies on secret keys and random masks that are applied to the inputs. While both techniques reflect computational difficulty, there is no backup in the form of SITS, in case the secret key is revealed or the Pseudo Random Generator (PRNG) is not sufficient.

In this paper, we present an alternative to a replicated state machine with no communication, while improving the communication overhead of the secret shared random-access machine presented in [18] and the secret shared Turing machine presented in [14].

This SITS within FHE approach can also be used in implementations for distributed, efficient, databases [3], Accumulating Automata with no communication [16] or even for ALU operations in the communication-less RAM implementation [18]. Another important application is in the scope of SMPC of machine learning queries [10].

Paper Organization. The rest of the paper starts with background and settings in the next section. Then in Sec. 3, we demonstrate the use of the CRT to implement communication-less DUFSM that uses logarithmic memory with relation to an equivalent replicated state machine, where each replica receives the full input and executes a globally known transition-function/program. We show that our CRT implementation is SITS, and can further prevent information leakage by employing FHE for executing additions and multiplications. In Sec. 4 we present an implementation of DUFSM based on polynomial representation, where the transition function is encoded by a polynomial and kept private, revealing only a bound on the polynomial maximal degree. The implemented FSM is private in terms of SITS within FHE. Demonstration of the polynomial-based FSM, in the scope of a string matching problem, appears in Sec. 5. Lastly, concluding remarks are found in Sec. 6 and more details appear in Appendix A.

2 Background and Settings

The focus of this section is to briefly review some of the key topics that serve as the base of this work, see Appendix A for comprehensive details.

CRT Arithmetic. Let $p_1 < p_2 < \ldots < p_k$ where p_i are relatively prime and a set of congruence equations $a \equiv a_i \pmod{p_i}$ for $1 \leq i \leq k$ for $k > 0$ and where a_i are remainders. The original form of the Chinese Remainder Theorem (CRT) states that this given set of congruence equations always has one and *exactly one* solution modulo $\prod_1^k p_i$.

The most important feature of the Chinese Remainder Theorem for our interest, is the possibility of adding and multiplying two vectors of congruence values independently. Namely, for performing fully homomorphic (addition and multiplication) operations on CRT-based secret shares. Unlike perfectly secure secret sharing (such as the schemes of Shamir [28] and Blakley [6]), CRT-based secret sharing that supports homomorphic additions and multiplications (unlike [2]) is only statistically secured. We use FHE to computationally mitigate information leakage from the individual CRT share.

Secure Multiparty Computation. The effectiveness of a joint secure operation is detailed in [22], introducing a series of arithmetic calculations, done over a finite field. The solution is perfect information-theoretic secure but requires communication among the participants to support polynomial degree reduction after a multiplication. Our CRT-based Secure Multiparty Computation is only statistical information-theoretic secure, but at the same time uses significantly less memory per participant and enables communication-less operations.

As mentioned in the CRT overview, the calculation results of each participant can be collected and recovered into a unique result in $\mathbb{Z}_K$ where $K = \prod_{i=1}^k p_i$. The task of reducing all the results into a single solution can be performed by some known algorithms, such as Garner's Algorithm [25], which we explicitly use in our proposed algorithms.

Consider two n-bit numbers x, y that are multiplied distributively among k parties. In this case, the series of calculations is only 1-multiplication-long, so the result is bounded by 2^{2n}. We first find k primes whose product is large enough (line 4), so the CRT recovery is possible. Then, we distribute all k primes to the parties such that every party holds a different prime modulus (line 11). In this example, we collect the calculation results synchronously. However, in a different scenario, we might do it otherwise. Ultimately, we recover actual multiplication results using Garner's algorithm (line 13).

Example 1. The square of $x = 14$ can be distributively computed by a group of 3 parties. Each party calculates the multiplication of x with itself such that:

$$\left.\begin{aligned} x \pmod{5} &= 4 \\ x \pmod{7} &= 0 \\ x \pmod{11} &= 3 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} 4 \cdot 4 &\equiv 1 \pmod{5} \\ 0 \cdot 0 &\equiv 0 \pmod{7} \\ 3 \cdot 3 &\equiv 9 \pmod{11} \end{aligned}\right\} \Rightarrow x \cdot x \equiv 196 \pmod{385} \tag{1}$$

Indeed, the result of Garner's algorithm is $y = 196 \in \mathbb{Z}/385\mathbb{Z}$ where:

$$y = 196 = 1 + 4 \cdot 5 + 5 \cdot (5 \cdot 7) \tag{2}$$

Which satisfies the following as expected:

$$y \equiv 1 \pmod{5}, \quad y \equiv 0 \pmod{7}, \quad y \equiv 9 \pmod{11}, \tag{3}$$

We note that the result of the calculation did not exceed the size of $\mathbb{Z}/385\mathbb{Z}$. Specifically, in the case the result of a calculation does overflow the ring bounds

Algorithm 1: Distributed Multiplication Example

```
   input  : integers $x$, $y$, number of parties $n$
   output: the result of $x \cdot y$
1  $K \leftarrow 2^{|x|+|y|}$ ;                                     // define highest bound
   // find $k$ prime numbers whose product is greater than $2^{2^n}$
2  for $bound \leftarrow 2$ to $K$ do
3      $primes$ = findPrimes($k, bound$) ;                          // find $k$ primes up to $bound$
4      if production($primes$) > $K$ then
5          break;
6      end
7      $bound \leftarrow bound + k$ ;                               // try find primes with higher $bound$
8  end
9  for $i \leftarrow 1$ to $k$ do
10     $m \leftarrow primes[i]$;
11     $results[i] \leftarrow$ sendParty($m$, multiply, $x$, $y$) ;  // $x \cdot y \pmod m$
12 end
13 $result \leftarrow$ garner($primes, results$);
14 return $result$
```

(the maximal product value of the moduli, $5 \cdot 7 \cdot 11 = 385$ in this example), we might experience an unexpected result in the recovery step. This result is guaranteed to be the modulo reduction of the correct one in respect to the moduli product value, thus it might still be useful. Nevertheless, this problem can be resolved by adding more parties to the computation, or by choosing larger primes. While both options increase the ring's bounds, thus preventing a calculation overflow, the first option is preferable as it has no penalty on the total memory usage.

Therefore, in the general case where a distributed calculation is carried out for some operator/function, one may follow the dealer-worker scheme. In this scheme, there is a *single* party that is responsible for the assignment of jobs and collection of the results while other parties, however, have no responsibility besides the calculation itself. We denote the first party as the "dealer" in this scheme and the other parties as "workers". The following Alg. 2 and Alg. 3 respectively describe their procedures.

Initially, the dealer generates the appropriate primes (line 1) and distributes them to the workers (line 5). Throughout the computation, the dealer manages a queue that is shared with the workers in such a manner that every time an input arrives, it is pushed to the queue (line 8) and popped in turn by the workers. Later, we replace this method with a more complex secret sharing method. Yet, thanks to this queue, the dealer can start and stop each worker asynchronously (line 11), as opposed to the example before, and by that can be more efficient. The dealer ultimately recovers the result using a recovery function of their choice (line 13).

Unlike the dealer, the worker has a more straightforward procedure to run. They merely receive the required parameters such as the unique modulus, the operation to carry out, the value to begin with, and an input queue. Then, they run indefinitely, while trying to apply the operation on every new input

Algorithm 2: Dealer Procedure

```
   input  : initial value $x$, an operation op, and a stream of inputs $stm$
   output: result of op applied on $x$ with all the inputs in $stm$
   // generate primes for all the workers, K is chosen in advance to guarantee the
      result does not overflow
 1 $primes \leftarrow$ genPrimes($K$);
   // initialize an empty queue
 2 $q \leftarrow$ queue();
   // start all workers with their modulo and the queue
 3 for $i \leftarrow 1$ to $K$ do
 4     $m \leftarrow primes[i]$;
 5     $workers[i] \leftarrow$ startWorker($m$, op, mod($x, m$), $q$);
 6 end
 7 while hasNext($stm$) do
       // start pushing inputs to the queue
 8     $q.push$(next($stm$));
 9 end
   // stop all workers and collect their results
10 for $i \leftarrow 1$ to $K$ do
11     $results[i] \leftarrow$ stopWorker($workers[i]$);
12 end
   // recover the final result
13 return recover($primes, results$)
```

Algorithm 3: Worker Procedure

```
   input  : modulus $m$, operation op, initial value $x$, and inputs queue $q$
   output: result of $op$ applied on $x$ and all the inputs
 1 while notSignaled() do
 2     $y \leftarrow$ tryPop($q$);
 3     $x \leftarrow$ op($x, y$) (mod $m$);
 4 end
 5 return $x$
```

(line 3), until they are signaled to stop by the dealer (line 1). The operation is always executed with respect to the unique modulus, such that there is no risk of overflow, or exceeding the finite field by the computation. The computation's limit is defined by the maximal number that the CRT shares represent, thus keeping the whole memory footprint small during the process.

3 Replicated State Machine vs. CRT DFSM or DUFSM

In this section, we explore the different aspects of a CRT based SMPC that utilizes the features mentioned before. We introduce our DFSM approach that copes with several of the Replicated State Machine (RSM) drawbacks. Also, to increase the privacy of the computation implied by this approach, we suggest using a local Fully Homomorphic Encryption (FHE) based arithmetic circuit that keeps the efficiency of memory while protecting the data.

Implementing the Transition Function with Secret Sharing. An Arithmetic Circuit is based on additions and multiplications which support the implementation of any Finite State Machine (FSM) transition function or table.

One convenient way to do so is by representing each bit in the circuit as a vector of two different bits (just as a quantum bit is represented). Namely, the bit 0 is represented by 01, and the bit 1 by 10. Consider each directed edge in the transition function graph tuple representation being represented as $\langle CurrentState, Input \rightarrow NextState, Output\rangle$. Then, given a (possibly secret shared) transition function, this structure allows us to secret share the table among different participants, possibly even padding it with additional never-used tuples. $CurrentState$, $Input$, and $NextState$ are represented by a sequence of 2-bits vectors. Thus, we double the logarithmic number of bits needed for the binary representation, rather than using (optimized for small degree polynomial, secret shares, and multiplication outcome) a linear number of bits in the unary representation as used in [14].

Now, to blindly compute the next state and output, given the current state and input, a participant *multiplies* each bit of the shared secret (2-bits vector representation) with the bits of each line of the transition table. Then, they sum up the resulting 2-bits vector into a single bit. For example, for the binary representation of the current state 110, the 2-bits vector representation is 101001. Consider this example of transition function representation:

$$\begin{aligned}
&\langle 010101, 01 \rightarrow 010101, 442\rangle\\
&\langle 010101, 10 \rightarrow 101001, 065\rangle\\
&\langle 101001, 01 \rightarrow 101001, 542\rangle\\
&\langle 101001, 10 \rightarrow 010101, 324\rangle
\end{aligned} \tag{4}$$

Note that only two inputs are possible in the example here, either 0, represented by 01, or 1, represented by 10. Furthermore, the output can be agreed to be represented in binary, and express a number inside the finite ring of the CRT secret sharing. For example, when three participants are using the primes $3 < 11 < 19$, then the finite ring being used for the secret sharing is 627. While the states and inputs representations are optimized for logical matching through arithmetic operations, the output representation can benefit from being memory efficient.

In case the current secret shared state and inputs are 101001 and 01 respectively, we find the next state and output by multiplication of every bit of the 2-bits vectors with each line of the table. Namely, the first two bits 10 of the current state are multiplied by the first two bits 01 in the table, resulting in $1 \cdot 0 = 0$ and $0 \cdot 1 = 0$, obtaining together 00. Then, by summing the two resulting bits, we get 0, which (blindly) indicates that there is *no match*. However, the third line in the table yields a match, as a sum of 1 is obtained from the first two bits (10 in the current state and 10 in the table), same for the next two bits 10 and the last two bits 01, altogether yielding the desired output. Finally, if the input is 01, then the third line matches completely, as also the input matches by yielding 1. Therefore, only the third line of the table yields results consisting of only 1 bits that when are (blindly) multiplied among themselves result in 1.

In Alg. 4, we can see that by multiplying the resulting bits with the state and input encoding (lines 9, 11), we ensure that only the fitting transition is chosen as the rest become 0. Blindly summing up all results of all next states and outputs (line 12) results in the desired (secret shared) next state and output.

Algorithm 4: Blindly Matching a Transition Tuple

```
input  : transition table $T$, state $current$ with length $L$ and input $i$
output: next state and output
$x \leftarrow$ encode($current$) ;                     // encode in the redundant form
$y \leftarrow$ encode($i$);
$result \leftarrow 0$
foreach $line$ in $T$ do
    $xT \leftarrow line[0]$ ;                          // unpack each tuple
    $yT \leftarrow line[1]$;
    $sum \leftarrow 1$;
    for $i \leftarrow 0$ to $2L$ do
        // perform multiplication with previous and current sum (i += 2)
        $sum \leftarrow sum \cdot (x[i] \cdot xT[i] + x[i+1] \cdot xT[i+1])$;
    end
    $sum \leftarrow sum \cdot (y[0] \cdot yT[0] + y[1] \cdot yT[1])$;
    $result \leftarrow result + sum \cdot T[line]$ ;   // accumulate conditioned next state
end
return decode($result$)
```

Utilization of an FHE Mechanism. Nowadays, the concept of Fully Homomorphic Encryption (FHE) has become highly popular in the field of modern cryptography. In a nutshell, FHE scheme is an encryption scheme that allows the evaluation of arbitrary functions on encrypted data. The problem was first suggested by Rivest, Adleman, and Dertouzos [26], and thirty years later, implemented in the breakthrough work of Gentry [19]. A major application of FHE is in cloud computing. This is because nowadays a user can store data on a remote server that has more storage capabilities and computing power than theirs. However, the user might not trust the remote server, as the data might be sensitive, so they send the encrypted data to the remote server and expect it to perform some arithmetic operations on it, without learning anything about the original raw data. In our case, an FHE scheme is employed to preserve the privacy among the participants, each being a remote server, blindly following the computation process.

The dealer's procedure described in Alg. 5 is extended hereby to support FHE behavior. The dealer now initializes an FHE context with which they encrypt both the initial value and the incoming inputs (lines 6,11). From this point, they continue in the same way as before (line 7,12), except for a decryption step at the end (line 16) and scheduled bootstrapping steps during the computation. For the sake of generality, the bootstrapping step is omitted but can be regarded as the assignment of the first share of the input to be the share of the initial state. After completing all of the decryptions, the results are reassembled by the CRT into a single solution as shown before.

Equally, the participants (workers) are dealt with a plaintext modulus in which they operate. By keeping the modulus in the clear, we do not leak any meaningful information and aid the participant in carrying out the computation with respect to their finite field. As before, after a worker is initialized, they start receiving encrypted inputs and apply the operator to them (line 3). As opposed to the operator application in a general field, these blind applications are expected

Algorithm 5: Dealer Extended FHE Procedure

```
input  : initial value $x$, an operation op, and a stream of inputs $stm$
output: result of op applied on $x$ with all the the inputs in $stm$
$context \leftarrow$ initFHE() ;                          // context allows encryption+decryption
$primes \leftarrow$ genPrimes($K$);
$q \leftarrow$ queue();
for $i \leftarrow 1$ to $K$ do
    $m \leftarrow primes[i]$;
    $xEncrypted \leftarrow$ encrypt(mod($x$), $context$) ;      // encrypt $x \pmod{m}$
    $workers[i] \leftarrow$ startWorker($m$, op, $xEncrypted$, $q$);
end
while hasNext($stm$) do
    $y \leftarrow$ next($stm$);
    $yEncrypted \leftarrow$ encrypt($y$, $context$) ;          // encrypt incoming input
    $q.push(yEncrypted)$;
end
for $i \leftarrow 1$ to $K$ do
    $rEncrypted \leftarrow$ stopWorker($workers[i]$);
    $results[i] \leftarrow$ decrypt($rEncrypted$, $context$) ;   // decrypt result
end
return recover($primes$, $results$)
```

Algorithm 6: Worker Extended FHE Procedure

```
input  : modulus $m$, operation op, encrypted initial value $x$, and encrypted inputs queue $q$
output: encrypted result of $op$ applied on $x$ and all the encrypted inputs
while notSignaled() do
    $y \leftarrow$ tryPop($q$);
    $x \leftarrow$ op($x$, $y$);
    $x \leftarrow$ blindMod($x$, $m$) ;          // can be implemented in several ways
end
return $x$
```

to be done in a finite field that is typically different from the binary field in computers (e.g., 8 bits for BYTE or 32/64 for a computer WORD). Therefore, the worker performs a dedicated balancing step after each iteration (line 4). Namely, they perform a *blind* modulo reduction to the result, thus keeping it inside the field. This step is possible due to a unique feature of FHE bitwise calculations that allows a blind conditioned output. One popular library that supports this feature is IBM's HELib [29]. The idea behind this implementation is based on an aggregation of the condition results. Namely, if one wishes to blindly increment a number i by 1 in case it is negative, or otherwise, blindly decrement it, they should first implement an indicator function:

$$I(x) = \begin{cases} 1 & x < 0 \\ 0 & otherwise \end{cases} \tag{5}$$

Then, use this function in a context such that i's value changes correctly:

$$F(x) = x + 1 \cdot I(x) - 1 \cdot (1 - I(x)) \tag{6}$$

A suggested implementation is outlined in the following algorithm. Note that line 3 creates an unknown bit and line 5 reflects a conditioned output based on that bit. The subtraction is aggregated by using the differences computed in line 4.

Algorithm 7: Blind Modulo Reduction

```
input  : encrypted integer $x$, modulus $m$
output: result of encrypted $x \pmod m$
1 $levels \leftarrow$ maxLevels($m$) ;            // max possible value i.e $\lfloor \frac{(m-1)^2}{m} \rfloor \approx m$
2 for $j \leftarrow 0$ to $levels$ do
3     $i \leftarrow$ compare($x, m$);
4     $d \leftarrow x - i$;
5     $x \leftarrow xi \cdot d + (1 - i) \cdot x$;
6 end
```

Utilizing this feature is essential during the procedure of a worker in our CRT based approach as the worker should be oblivious to the fact they carry out the same procedure only on encrypted data. As long as they know how to perform homomorphic operations such as additions and multiplications, while staying within the boundaries of the computer's binary representation, the homomorphism of the operations over the CRT secret shares is preserved.

4 Polynomial Based CRT DUFSM

We present an alternative to further improve the secrecy of the transition function of the FSM that is based on polynomial representation.

Polynomial Interpolation in Finite Rings. It is often useful (e.g., [27]) to estimate the value of a function $y = f(x)$ at a certain point x based on some known values of the function, i.e., $f(x_0), f(x_1), \dots, f(x_n)$. These values are evaluated at a set of $n+1$ points $a = x_0, x_1, \dots, x_n = b$ in the range $\{a \dots b\}$. One way to carry out this operation is to approximate the function $f(x)$ by an n-th degree polynomial:

$$f(x) \approx P_n(x) = a_n x^n + a_{n-1}x^{n-1} +, \dots, a_2 x^2 + a_1 x + a_0 = \sum_{j=0}^{n} a_j x^j \qquad (7)$$

Where the coefficients $a_0, \dots, a_n$ are obtained based on the $n+1$ given points. Specifically, to find the coefficients of $P_n(x)$, we require the polynomial to pass through all the points: $\{(x_i, y_i) = f(x_i) | i = 0, \dots, n\}$ so that the following $n+1$

linear equations hold:

$$
\begin{aligned}
P_n(x_0) &= \sum_{i=0}^{n} a_i x_0^i = f(x_0) = y_0 \\
P_n(x_1) &= \sum_{i=0}^{n} a_i x_1^i = f(x_1) = y_1 \\
&\vdots \\
P_n(x_n) &= \sum_{i=0}^{n} a_i x_n^i = f(x_n) = y_n
\end{aligned}
$$

Fortunately, this polynomial $P_n(x)$ can be obtained by the interpolation polynomial in the Lagrange form. This polynomial is defined as a linear combination denoted by:

$$L_n(x) = \sum_{i=0}^{n} y_i l_i(x) \tag{8}$$

Where $l_i(x)$ is the Lagrange basis polynomial of degree n, that together with the other basis polynomials span the space of all n-th degree polynomials:

$$l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)\dots(x - x_{i-1})(x - x_{i+1})\dots(x - x_n)}{(x_i - x_0)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)} \tag{9}$$

Namely, when $x = x_j$ for $j = 0, \dots, n$, we get:

$$l_i(x_j) = \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases} \tag{10}$$

This way, the polynomial $L_n(x)$ passes through all $n + 1$ points, because:

$$
\begin{aligned}
L_n(x_0) &= \sum_{i=0}^{n} y_i l_i(x_0) = \sum_{i=0}^{n} y_i \delta_{i0} = y_0 = f(x_0) \\
L_n(x_1) &= \sum_{i=0}^{n} y_i l_i(x_1) = \sum_{i=0}^{n} y_i \delta_{i1} = y_1 = f(x_1) \\
&\vdots \\
L_n(x_n) &= \sum_{i=0}^{n} y_i l_i(x_n) = \sum_{i=0}^{n} y_i \delta_{in} = y_n = f(x_n)
\end{aligned}
$$

We observe that this polynomial is only an approximation of $f(x)$ in certain points. So, in fact, at any other point $x \neq x_j$ for $j = 0, \dots, n$, the polynomial value is *unpredictable*.

Conceptually, polynomial interpolation in finite rings should not differ from polynomial interpolation in general rings such as $\mathbb{Z}$. That is because we can use modular arithmetic instead of regular arithmetic, thus following a standard interpolation algorithm. Nevertheless, there are some concerns to bear in mind. First, in case we use the Lagrange interpolation, it is essential to choose the parameter $M > 0$ of the ring $\mathbb{Z}/M\mathbb{Z}$ wisely. Otherwise, the interpolation fails. Since is not guaranteed that every number $x \in \mathbb{Z}/M\mathbb{Z}$ is invertible (e.g, zero), and the denominators in the basis polynomials are comprised of differences between two numbers, the different divisions might not be possible. Therefore, for a set of points $\{(x_i, y_i)|i = 0, \ldots, n\}$, it is crucial to choose such ring $\mathbb{Z}/M\mathbb{Z}$, where all the differences $x_i - x_j$ are invertible.

Consider a set of points in the finite ring $\mathbb{Z}/K\mathbb{Z}$, such that $K = \prod_i^n p_i$ for relatively primes p_i. To successfully interpolate this set of points using Lagrange's method, we need to verify that neither of the differences has a common factor in $\{p_1, \ldots, p_n\}$.

Lemma 1. *If there exist $i \neq j$ such that the difference $d = x_i - x_j$ has a prime factor $p_d \in \{p_1, \ldots, p_n\}$ then Lagrange's polynomial interpolation is not possible.*

Proof. By contradiction, if such p_d exists, then $d = {p_{i_1}}^{k_1}, \ldots, {p_d}^{k_d}, \ldots, {p_{i_n}}^{k_n}$ for $k_d > 0$ and $k_i \geq 0$. In other words, the distance d can be represented as $d = {p_d}^{k_d} \cdot N$ for some $N \geq 1$, and $d = {p_d}^{k_d} \cdot N \pmod{p_d} \equiv 0$ holds. Thus, d is not invertible although we assume it is. □

We use the following Alg. 8 to choose the relatively primes $p_1, \ldots, p_k$ before starting the interpolation process. We first find all the differences that might not be invertible and factorize them (line 5). Once we have all the factors we have to avoid, we find such primes that are coprime to these factors (line 12). Lastly, we return the prime set whose product is large enough (line 15).

Consider a given FSM that is represented by a truth table. Namely, we are interested in the relations between the different states and the possible inputs or outputs. We suggest a (non-perfect) encoding scheme that allows us to represent this FSM completely by polynomials. First, we encode the different states and transitions in some grid-compatible representation, where a transition in that context, is a 2-tuple $e = (u, v)$ such that the state u has a valid input that leads to v. One simple encoding is through positive integers representation. Given a set of states V, and a set of transitions E, the 2-D point *unique* encoding of them is calculated as follows in Alg. 9.

Since the y value of a point is comprised only of a state encoding, the decoding process is simple. It is however not guaranteed for the x value, as it is comprised of an encoded summation that might overlap other encoded values. One possible way to deal with this, is to simply work on different scales, more specifically, we use a factor $f = 10^t$ where $t > 0$ to choose the integers in line 1 from the range $\{f+1 \ldots |V| \cdot f\}$. Also, considering that there might be many transitions to cause an overflow between the scales, the parameter t needs to be bounded such that $t > \log |E|$ and $f = 10^t > |E|$ holds.

Algorithm 8: Finding Interpolation Parameters in $\mathbb{Z}/K\mathbb{Z}$

```
input  : set of points $s = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ and a parameter $k$
output: sequence of appropriate $k$ prime integers $p_1, \ldots, p_k$ or $null$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        $difference \leftarrow$ abs($s[i] - s[j]$);
        foreach $f \in$ factorize($difference$) do
            $factors.append(f)$;
        end
    end
end
for $bound \leftarrow 2$ to $K$ do
    $primes \leftarrow$ findPrimes($k, bound$);
    if length($primes$) $< k$ $or$ length($factors \cap primes$) $> 0$ then
        continue;
    end
    if production($primes$) $> K$ then
        return $primes$
    end
    $bound \leftarrow bound + k$;
end
return $null$
```

Algorithm 9: FSM Encoding Procedure

```
input  : $V, E$
output: A list of points $P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n)$ where $n = |E|$
$rangeV \leftarrow$ randRange($1, |V|$) ;                // generate numbers for each state
$rangeE \leftarrow$ randRange($1, |E|$) ;           // generate numbers for each transition
$points = [\,]$;
for $i \leftarrow 1$ to $|E|$ do
    $u, v \leftarrow$ unpack($E[i]$) ;                   // unpack the states in transition
    $x \leftarrow rangeV[$indexOf$(u, V)] + rangeE[i]$;
    $y \leftarrow rangeV[$indexOf$(v, V)]$;
    $points.append((x, y))$;
    /* each point $P$ is calculated such that $P = (r_u + r_e, r_v)$ where $r_u, r_v$ are the
       random numbers chosen for states $u, v$ and $r_e$ is the random number chosen for
       transition $e$                                                                */
end
return $points$
```

Evaluating Polynomial within FHE. Since the polynomials are both encrypted and already evaluated in a specific field, the only information a participant can learn stems from the encryption parameters and the finite field modulus assigned to him beforehand. By keeping the modulus clear, we simplify the assignment process while not revealing any meaningful data to the participants, as all the other data they receive is encrypted. The encryption parameters, however, including the public key, might hint at the computational security of the scheme, in case the participant is interested in breaking it. The Homomorphic Encryption Standard [1] may assist in choosing recommended parameters for implementation.

In practice, the suggested process provides the participants with a reduced polynomial in some finite field, but the actual operation does not consider that fact. Fortunately, we can maintain the result in the respected finite field by applying a *blind* modulo operation on each polynomial evaluation. This can be done by the previous method described in Alg. 7.

Moreover, to successively evaluate the polynomial without consuming all noise budget of the FHE scheme, one can utilize a bootstrapping method, thus allowing the computation to carry on endlessly.

5 Blind String Matching via DUFSM Example

For the sake of simplicity and readability we consider the task of searching the word "nano" in a (possibly unbounded streaming) large text. In fact any state machine computation, where the current state is maintained only in (FHE CRT) secret shared form can be supported, thus, eliminating the need of the computation delegating client, to protect the current state security and privacy (avoiding single point of failure) and carrying the computation of the transition function. This problem can be presented as a distributed computation problem, where we send each participant text characters one after the other, and expect them to yield a positive result if and only if the sequence "nano" was detected among the received characters. We collect the negative or positive results from all participants and decide the correct result by using the majority, thus eliminating any Byzantine errors. We observe that in this scenario, both the string to search and the text itself are shared in clear-text for all participants, along with the string-matching algorithm. The RSM solution reflects a "naive" (as repetition codes) approach for error correction, where there are codes with equivalent Hamming distance, such that in total they use a smaller number of bits [4]. In turn, we consider a different scenario, where all inputs are kept secret and the algorithm is *unknown* to the parties participating in the distributed computation.

One can build a simple automaton for that specific task, disregarding any preprocessing operations as done in the Boyer Moore algorithm [7]. In this simple automaton, there are only five states - an "empty" state denoted by ϵ, and four other states, each of them representing a valid substring of "nano". For the sake of clarity, the transition table is detailed in Tab. 1.

Table 1: Transition Table

	n	a	o	other
ϵ	n	ϵ	ϵ	ϵ
n	n	na	ϵ	ϵ
na	nan	ϵ	ϵ	ϵ
nan	n	na	$nano$	ϵ
$nano$	$nano$	$nano$	$nano$	$nano$

For the sake of convenience, the state machine is also detailed in Fig. 1. Following this data, we encode all the possible letters (inputs) as integers and later use them to simulate a transition from one state to another. The most straightforward method to use in this encoding is a map of each character to its real ASCII value. However, there are a couple of concerns to note. First, the English alphabet values are located in the ASCII table in a non-continuous manner. Namely, the values have undesired gaps between them. This is a disadvantage for us, mainly in case we create an interpolating polynomial for the state machine. Also, by using the ASCII table, we limit ourselves to a single text encoding, solely for the English language. We can work around this limitation by creating a map of characters to values for every possible encoding. However, this solution might be unscalable for different texts, as texts are encoded differently, and creating as many tables as text encoding requires undesired preprocessing work.

A different hybrid method is to use these two techniques together. Namely, we can define an encoding, where each character is mapped to an integer v from the respected text encoding table, only this time, it is subtracted by an offset value *off*, such that the minimum possible value v_{min} becomes $v_{min}-$ off $= 1$. In this way, although we do not eliminate the unwanted gaps, we minimize each character value while allowing an application of this method for any text encoding.

Once we complete the encoding of the input, we move to encode the states of this machine as integers. We choose to do so using a set of integers, with a constant distance d between them, such that the highest value of an input v_{max} holds $v_{max} < d$. In that way, we guarantee that there are no overlaps between states while computing the transitions (recall the algorithm of state and transition encoding, where we sum together the encoding of a state with the encoding of an input). Finally, based on the simple state machine from before, we can build an encoded FSM as portrayed in Fig. 2.

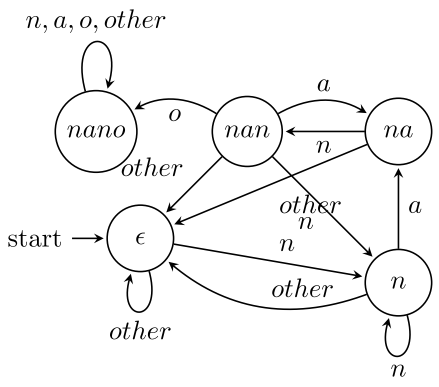

Fig. 1: Simple nano State Machine

In regards to the range and factor parameters from Alg. 9 and for the sake of readability, we demonstrate the automaton using a decimal base, while in practice, binary base is more efficient. The five states above are encoded as integers with a factor of $f = 10^2$ and a similarly a range of $\{1 \ldots 5\}$, such that: $1 \cdot 10^2 = 100,\ 2 \cdot 10^2 = 200,\ 3 \cdot 10^2 = 300,\ 4 \cdot 10^2 = 400,\ 5 \cdot 10^2 = 500$. Also, all inputs are encoded as integers in the range $\{1 \ldots 29\}$ to represent the English alphabet and the punctuation signs such as spaces, dots, and newlines.

Again, following Alg. 9, we build a list of points, each representing a transition in the state machine from one state to another with some input. We note that this list is sparse, namely, it does not include invalid or non-existent transitions. Therefore, when we apply the polynomial interpolation, invalid values will result

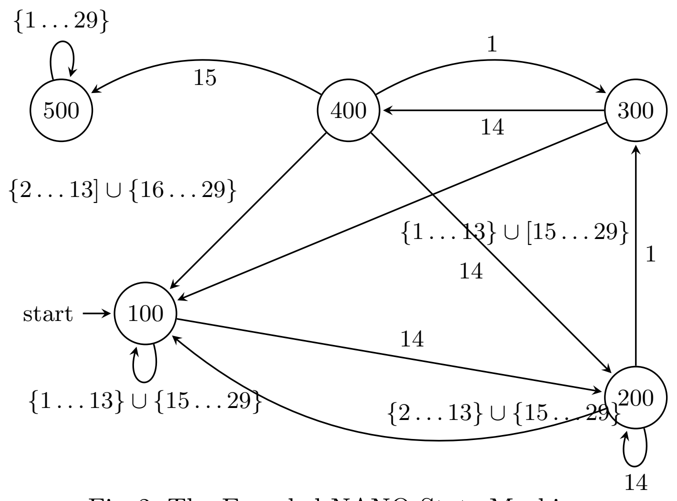

Fig. 2: The Encoded NANO State Machine

in invalid or unexpected results. See Appendix B for the complete list of encoded points.

Next, following Alg. 8, we build an interpolating polynomial $P(x)$ such that $P(x) \in \mathbb{Z}/K\mathbb{Z}[X]$. Namely, besides that all the points detailed above fit into the polynomial, all the polynomial's coefficients are in $\mathbb{Z}/K\mathbb{Z}$ for some relatively primes $p_1, \ldots, p_k$ and a product $K = \prod_i^k p_i$.

As a result of a large number of encoded points, this polynomial has a high degree. However, this is acceptable as it is only evaluated under some finite field and there is no risk of overflowing or exceeding memory resources. As soon as we finish the interpolation step, we immediately apply modulo reduction for each participant and distribute the reduced polynomial to start the computation.

6 Concluding Remarks

Communication-less secure multi-party computation, where the servers that perform the computation are not aware of the other servers' identity and location, introduces a new facet of security, where colluding is much harder to be coordinated. Efficiency is an obvious additional benefit as in many cases the (typical quadratic) communication overhead is significantly more expensive than the local computation. One more important aspect is that we no longer need to synchronize the actions of the servers, as they may process their own (secret shared) inputs whenever convenient before the eventual output collection. Our various implementations demonstrate the practicality of our scheme.

Lastly, it is possible to integrate an error-correction encoding into our scheme that copes with Byzantine participants. This includes increasing the size of the mutual primes $p_1 < p_2 < \ldots < p_k$ to avoid overflow of the computation with respect to $p_1 \cdot p_2, \ldots \cdot p_k$, and adding more mutual primes $q_1 < q_2 < \ldots < q_l$ larger than p_k, where l is a function of the maximal number of errors.

References

[1] Martin Albrecht et al. 2018. URL: `https://eprint.iacr.org/2019/939.pdf`.

[2] Charles Asmuth and John Bloom. "A modular approach to key safeguarding". In: *IEEE Trans. Information Theory* 29.2 (1983), pp. 208–210. DOI: `10.1109/tit.1983.1056651`. URL: `https://doi.org/10.1109/TIT.1983.1056651`.

[3] Hillel Avni, Shlomi Dolev, Niv Gilboa, and Ximing Li. "SSSDB: Database with Private Information Search". In: Feb. 2016, pp. 49–61. ISBN: 978-3-319-29918-1. DOI: `10.1007/978-3-319-29919-8_4`.

[4] Bharath Balasubramanian and Vijay K. Garg. "Fault tolerance in distributed systems using fused state machines". In: *Distributed Computing* 27.4 (Aug. 2014), pp. 287–311. ISSN: 1432-0452. DOI: `10.1007/s00446-014-0209-4`. URL: `https://doi.org/10.1007/s00446-014-0209-4`.

[5] Dor Bitan and Shlomi Dolev. "Optimal-Round Preprocessing-MPC via Polynomial Representation and Distributed Random Matrix (extended abstract)". In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 1024. URL: `https://eprint.iacr.org/2019/1024`.

[6] G.R. Blakley. "Safeguarding cryptographic keys". In: *Proceedings of the 1979 AFIPS National Computer Conference*. Monval, NJ, USA: AFIPS Press, 1979, pp. 313–317.

[7] Robert S. Boyer and J. Strother Moore. "A Fast String Searching Algorithm". In: *Commun. ACM* 20.10 (Oct. 1977), pp. 762–772. ISSN: 0001-0782. DOI: `10.1145/359842.359859`. URL: `https://doi.org/10.1145/359842.359859`.

[8] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. *Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation*. Cryptology ePrint Archive, Report 2020/1392. `https://eprint.iacr.org/2020/1392`. 2020.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. *Secure Computation with Preprocessing via Function Secret Sharing*. Cryptology ePrint Archive, Report 2019/1095. `https://eprint.iacr.org/2019/1095`. 2019.

[10] Philip Derbeko and Shlomi Dolev. "PolyDNN Polynomial Representation of NN forCommunication-less SMPC Inference". In: *Cyber Security Cryptography and Machine Learning - Fifth International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*. Vol. 12716. Lecture Notes in Computer Science. Springer, 2021.

[11] Hagar Dolev and Shlomi Dolev. *Toward Provable One Way Functions*. Cryptology ePrint Archive, Report 2020/1358. `https://eprint.iacr.org/2020/1358`. 2020.

[12] Shlomi Dolev, Karim Eldefrawy, Juan Garay, Muni Venkateswarlu Kumaramangalam, Rafail Ostrovsky, and Moti Yung. "Brief Announcement: Secure Self-Stabilizing Computation". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. Podc '17. Washington, DC, USA: Association for Computing Machinery, 2017, pp. 415–417. ISBN:

9781450349925. DOI: 10.1145/3087801.3087864. URL: https://doi.org/10.1145/3087801.3087864.

[13] Shlomi Dolev, Juan Garay, Niv Gilboa, and Vladimir Kolesnikov. "Secret Sharing Krohn-Rhodes: Private and Perennial Distributed Computation." In: Jan. 2011, pp. 32–44.

[14] Shlomi Dolev, Juan A. Garay, Niv Gilboa, Vladimir Kolesnikov, and Muni Venkateswarlu Kumaramangalam. "Perennial secure multi-party computation of universal Turing machine". In: *Theor. Comput. Sci.* 769 (2019), pp. 43–62. DOI: 10.1016/j.tcs.2018.10.012. URL: https://doi.org/10.1016/j.tcs.2018.10.012.

[15] Shlomi Dolev, Juan A. Garay, Niv Gilboa, Vladimir Kolesnikov, and Yelena Yuditsky. "Towards efficient private distributed computation on unbounded input streams". In: *J. Mathematical Cryptology* 9.2 (2015), pp. 79–94. DOI: 10.1515/jmc-2013-0039. URL: https://doi.org/10.1515/jmc-2013-0039.

[16] Shlomi Dolev, Niv Gilboa, and Ximing Li. "Accumulating automata and cascaded equations automata for communicationless information theoretically secure multi-party computation". In: *Theor. Comput. Sci.* 795 (2019), pp. 81–99. DOI: 10.1016/j.tcs.2019.06.005. URL: https://doi.org/10.1016/j.tcs.2019.06.005.

[17] Shlomi Dolev, Limor Lahiani, and Moti Yung. "Secret swarm unit: Reactive k-secret sharing". In: *Ad Hoc Networks* 10.7 (2012), pp. 1291–1305. ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2012.03.011. URL: http://www.sciencedirect.com/science/article/pii/S1570870512000613.

[18] Shlomi Dolev and Yin Li. "Secret Shared Random Access Machine". In: *Algorithmic Aspects of Cloud Computing*. Ed. by Ioannis Karydis, Spyros Sioutas, Peter Triantafillou, and Dimitrios Tsoumakos. Cham: Springer International Publishing, 2016, pp. 19–34.

[19] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices". In: STOC '09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178. ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[20] Oded Goldreich, Dana Ron, and Madhu Sudan. "Chinese remaindering with errors". In: *IEEE Trans. Information Theory* 46.4 (2000), pp. 1330–1338. DOI: 10.1109/18.850672. URL: https://doi.org/10.1109/18.850672.

[21] R. Jaiswal. "Chinese Remainder Codes : Using Lattices to Decode Error Correcting Codes Based on Chinese Remaindering Theorem". In: 2007.

[22] Artur Jakubski. "Selected application of the Chinese Remainder Theorem in multiparty computation". In: *Journal of Applied Mathematics and Computational Mechanics* 2016 (Jan. 2016), pp. 39–47. DOI: 10.17512/jamcm.2016.1.04.

[23] Leslie Lamport. “Fast Paxos”. In: *Distrib. Comput.* 19.2 (Oct. 2006), pp. 79–103. ISSN: 0178-2770. DOI: 10.1007/s00446-006-0005-x. URL: https://doi.org/10.1007/s00446-006-0005-x.

[24] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[25] Yongnan Li, Limin Xiao, Aihua Liang, Yao Zheng, and Li Ruan. “Fast Parallel Garner Algorithm for Chinese Remainder Theorem”. In: Sept. 2012, pp. 164–171. ISBN: 978-3-642-35605-6. DOI: 10.1007/978-3-642-35606-3_19.

[26] R.L. Rivest, L. Adleman, and M.L. Dertouzos. “On data banks and privacy homomorphisms”. In: *Foundations on Secure Computation, Academia Press*. 1978, pp. 169–179.

[27] Tomas Sauer. “Polynomial interpolation of minimal degree”. In: *Numerische Mathematik* 78 (Nov. 1997), pp. 59–85. DOI: 10.1007/s002110050304.

[28] Adi Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176. URL: http://doi.acm.org/10.1145/359168.359176.

[29] Huiyong Wang, Yong Feng, Yong Ding, and Shijie Tang. “A Homomorphic Arithmetic Model via HElib”. In: *Journal of Computational and Theoretical Nanoscience* 14 (Nov. 2017), pp. 5166–5173. DOI: 10.1166/jctn.2017.6690.

Acronyms

ASCII	American Standard Code for Information Interchange.
CRT	Chinese Remainder Theorem.
DCLSMPC	Distributed Communication-less Secure Multiparty Computation.
DFSM	Distributed Finite State Machine.
DUFSM	Distributed Unknown Finite State Machine.
FHE	Fully Homomorphic Encryption.
FSM	Finite State Machine.
FSS	Function Secret Sharing.
MPC	Multiparty Computation.
PRNG	Pseudo Random Generator.
RSM	Replicated State Machine.
SITS	Statistical Information Theoretic Secure.
SMPC	Secure Multiparty Computation.

A Appendix A

Background. The focus of this section is to briefly review some of the key topics that serve as the base of this work.

Finite Rings and Finite (Galois) Fields. We first introduce the type of field that has a finite number of elements. This number is a power p^n of some prime number p. In fact, for any prime number p and any natural number n there exists a unique field of p^n elements that is denoted by $GF(p^n)$ or by $\mathbb{F}_{p^n}$. Conveniently, all standard operations such as multiplication, addition, subtraction, and division (excluding division by zero) are defined and satisfy the rules of arithmetic, just as the corresponding operations on rational and real numbers do. This type of field helps quite often to think and calculate, in terms of integers, modulo another number.

Definition 1. *Fix an integer n. The integers modulo n denoted $\mathbb{Z}/n\mathbb{Z}$, is the set of numbers $0, 1, 2, \ldots, n-1$, with addition and multiplication defined by taking the remainder upon division by n.*

We are specifically interested in reducible integers n such that $n = p_1, \ldots, p_k$ for some $k > 0$ primes. A key observation here is that when all the p_i are co-prime (referred to as relatively prime sometimes) then any number modulo n that is divided by these p_i gives us a *unique* sequence of residues. Formally, this observation is a restatement of the Chinese Remainder Theorem which is explained below.

CRT Arithmetic. Let $p_1 < p_2 < \ldots < p_k$ where p_i are relatively prime and a set of congruence equations $a \equiv a_i \pmod{p_i}$ for $1 \leq i \leq k$ for $k > 0$ and where a_i are remainders. The original form of the Chinese Remainder Theorem (CRT) states that this given set of congruence equations always has one and *exactly one* solution modulo $\prod_1^k p_i$.
Similar to our previous notation, this theorem is often restated as:

$$\mathbb{Z}/a\mathbb{Z} \cong \mathbb{Z}/p_1\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k\mathbb{Z} \tag{11}$$

This means that by doing a sequence of arithmetic operations in $\mathbb{Z}/a\mathbb{Z}$ one may do the same computation independently in each $\mathbb{Z}/p_i\mathbb{Z}$ and then get the result by applying the isomorphism from the right to the left. We refer to this operation as the *recovery* process later on.

Example 2. The integer $m = 14$ can be represented as a set of these congruence equations:

$$\begin{aligned} 14 &\equiv 2 \pmod{3}, \quad 14 \equiv 4 \pmod{5}, \\ 14 &\equiv 0 \pmod{7}, \quad 14 \equiv 3 \pmod{11} \end{aligned} \tag{12}$$

More significantly, $m = 14$ is the exact and only solution modulo $3 \cdot 5 \cdot 7 \cdot 11 = 1155$.

This feature of CRT allows us to represent big numbers using a small array of integers. Namely, when performing arithmetic operations on big numbers this feature assists in preserving memory resources.

Byzantine Fault Tolerance. The assumption that every participant in a distributed system correctly follows the algorithm may not hold in reality. Either because of faults or malicious takeovers. Therefore, distributed systems are often designed to tolerate a part (typically less than $\frac{1}{3}$) of the participants that are acting as if controlled by a malicious adversary. These participants are called *Byzantine* participants.

Secure Multiparty Computation. To further demonstrate the effectiveness of a joint operation as detailed in [22], consider a series of arithmetic calculations, done over t-bits numbers in the form of $b_1, \ldots, b_t$. The additions in this series might add up to 1 bit between each calculation, but the multiplications might double the number of bits. Thus, if the whole operation is completed individually by a single party, it might require up to t^2 bits in the worst case. However, if we use the CRT representation and split the moduli into several parties, we keep performing each calculation individually but achieve a bounded number of bits per party (bound as large as implied by the largest modulus). Namely, to support calculations of up to t^2 bits, i.e numbers up to 2^{t^2}, it is sufficient to setup k parties and k primes (moduli) $p_1, \ldots, p_k$ that:

$$2^{t^2} < \prod_{i=1}^{k} p_i \tag{13}$$

This observation leads to the conclusion that we can decide whether to use a few large primes or many small primes to carry out the same series of calculations. This decision might change according to the availability of more parties and the number of memory resources we wish to consume.

As previously explained, the calculation result of each participant can be collected and recovered into a unique result in $\mathbb{Z}_K$ where $K = \prod_{i=1}^{k} p_i$.

B Appendix B

Polynomial Points.

Table 2: Encoded Points

(101,100)	(201,300)	(301,100)	(401,300)	(501,500)
(102,100)	(202,100)	(302,100)	(402,100)	(502,500)
(103,100)	(203,100)	(303,100)	(403,100)	(503,500)
(104,100)	(204,100)	(304,100)	(404,100)	(504,500)
(105,100)	(205,100)	(305,100)	(405,100)	(505,500)
(106,100)	(206,100)	(306,100)	(406,100)	(506,500)
(107,100)	(207,100)	(307,100)	(407,100)	(507,500)
(108,100)	(208,100)	(308,100)	(408,100)	(508,500)
(109,100)	(209,100)	(309,100)	(409,100)	(509,500)
(110,100)	(210,100)	(310,100)	(410,100)	(510,500)
(111,100)	(211,100)	(311,100)	(411,100)	(511,500)
(112,100)	(212,100)	(312,100)	(412,100)	(512,500)
(113,100)	(213,100)	(313,100)	(413,100)	(513,500)
(114,200)	(214,200)	(314,400)	(414,200)	(514,500)
(115,100)	(215,100)	(315,100)	(415,500)	(515,500)
(116,100)	(216,100)	(316,100)	(416,100)	(516,500)
(117,100)	(217,100)	(317,100)	(417,100)	(517,500)
(118,100)	(218,100)	(318,100)	(418,100)	(518,500)
(119,100)	(219,100)	(319,100)	(419,100)	(519,500)
(120,100)	(220,100)	(320,100)	(420,100)	(520,500)
(121,100)	(221,100)	(321,100)	(421,100)	(521,500)
(122,100)	(222,100)	(322,100)	(422,100)	(522,500)
(123,100)	(223,100)	(323,100)	(423,100)	(523,500)
(124,100)	(224,100)	(324,100)	(424,100)	(524,500)
(125,100)	(225,100)	(325,100)	(425,100)	(525,500)
(126,100)	(226,100)	(326,100)	(426,100)	(526,500)
(127,100)	(227,100)	(327,100)	(427,100)	(527,500)
(128,100)	(228,100)	(328,100)	(428,100)	(528,500)
(129,100)	(229,100)	(329,100)	(429,100)	(529,500)