The Interdisciplinary Center, Herzlia
Efi Arazi School of Computer Science
M.Sc. program

Usage of Regenerating Codes in Secure Multi-Party Computation

by
Ori Chen

Final project, submitted in partial fulfillment of the requirements
for the M.Sc. degree, School of Computer Science
The Interdisciplinary Center, Herzliya

September 2017

This work was carried out under the supervision of Dr. Elette Boyle from the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya.

Abstract

In this project we implement existing secure MPC protocols while utilizing recent results in the field of regenerating codes to decrease the bandwidth of the computation. In a secure Multi-Party Computation (MPC) a set of n parties each holding a secret input wish to compute some functionality on their inputs. For the information theoretic semi-honest scenario, various protocols were designed allowing unconditional perfect security in the presence of an honest majority [BGW88, GMW87, DIK10]. A secret sharing scheme, such as Shamir's secret sharing scheme, is often in the foundation of many of these protocols – the secret inputs are shared, some joint computation is made on these shares and finally the output is reconstructed from the processed shares. In parallel, in the field of Coding Theory and regenerating codes in particular, recent results by Guruswami and Wootters show that even for Reed-Solomon codes, lower bandwidth algorithms for the exact-repair problem exist [GW15]. As Shamir's secret sharing scheme reconstruction procedure is in essence an exact repair procedure over Reed-Solomon codes, these results imply lower bandwidth reconstruction procedures which in turn imply lower bandwidth MPC protocols.

In our work we implement some of these MPC protocols and show how these recent results translate to lower communication protocols. The project's code is written in pure Python without the use of any 3rd-party dependencies, resulting a generic and portable code for experimenting with MPC protocols, an efficient procedure for the exact-repair problem for Reed-Solomon codes, and various mathematical implementations such as finite fields, polynomials, linear transformations and matrices used as tools.

Table of Contents

1 Introduction

In a Secure Multi-Party Computation (MPC) [Yao86, GMW87], a set of parties wish to securely perform a joint computation without compromising the privacy of their inputs, in the presence of adversarial behavior. Loosely, an MPC is secure if every party receives the correct output while nothing about its input is learned by possibly cooperating adversaral parties (aside from the function output). This problem is motivated by real-life applications such as privacy-preserving data mining, where a number of parties want to compute some desired data mining algorithm on the union of their databases, without revealing their data. The core of many suggested MPC protocols makes use of coding theory techniques - dividing a secret between the parties which can later be recovered using some decoding algorithm.

Independent of MPC, the field of Coding Theory, and Regenerating Codes in particular, is gaining attention due to the increased use of distributed storage systems, intended to reliably store data for long periods of time using a distributed collection of possibly unreliable nodes. In these systems, in order to ensure reliability, some form of redundancy is needed - accomplished by using error correcting codes that ensure that failed nodes may be restored. Except storage efficiency and reliability, another important property required from such codes is the repair bandwidth required to restore some failed node. In particular, Regenerating Codes, introduced by Dimakis *et al.*, allow systems to recover a failed node with low communication [DGW$^+$10, RSKR10]. Recent results show that even for Reed-Solomon codes, regeneration bandwidth can be lowered below the bandwidth required by the naive algorithm [GW17].

The connection between these recent results and MPC has yet to be fully explored. In this project, we show a proof of concept by implementing an MPC protocol that utilizes regenerating codes to lower the amount of communication sent between the parties.

1.1 Multi-Party Computation

More formally, in typical MPC settings n parties $P_1, ..., P_n$ hold secret inputs $x_1, ..., x_n$ respectively and wish to compute some public $f(x_1, .., x_n) = (y_1, .., y_n)$ such that party P_i learns exactly y_i. An adversary attempting to attack the protocol may corrupt a subset of the parties in attempt to force incorrect output or to learn something about the inputs of the remaining parties. In addition to correctness and privacy, other properties of security are considered, such as guaranteed output delivery, fairness and independence of inputs. The security definition is formalized by comparing the outcome of a real protocol to an "ideal computation" of the function by an external trusted party receiving all of the inputs, calculating the desired functionality and sending each party its corresponding output. That is, for any adversary attacking the real protocol, there exists an adversary who could have learned the same information in the ideal execution. Security of an MPC protocol is proved by showing the corrupt parties *view* can be simulated given only its input and output, in a way that is indistinguishable from its view in a real execution (where a party's view is, loosely, its input, randomness, messages seen and output).

There are various settings in which MPC is considered. The adversaries may be either semi-honest (i.e., who follow the protocol but try to learn more information than they should) or malicious (i.e., may use an arbitrary strategy). Assumptions on adversaries' computational power are also made: limited to polynomial-time in the computational setting or computationally unbounded in the information theoretic setting. Adversaries may either statically corrupt a subset of the parties, or adaptively choose to corrupt parties during the protocol's execution. Other considerations are the channels that exists between the parties (private channels versus broadcast channel).

The core of existing MPC protocols is often based on *secret sharing*. The basic idea of secret sharing is dividing a secret into shares, where each party receives one share in a way that only certain subsets of the parties are able to recover the secret by performing some joint computation. MPC protocols often use a secret sharing scheme to share the parties' inputs, calculating the desired functionality locally or interactively on the shares, and finally performing some secure reconstruction of the outputs from the processed shares. Results from the late 1980s have shown the *feasibility* of multi-party computation in both computational and information theoretic scenarios [GMW87, BGW88, CCD88]. It is known that when less than one third of the parties are corrupted by a malicious adversary, MPC can be done in the information theoretic setting for any functionality. In the semi-honest setting, similar results have been shown as long as an honest majority exists [RB89]. Ben-Or *et al.* [BGW88] have used Shamir's secret sharing [Sha79] to achieve perfect information theoretic security.

1.2 Regenerating Codes

An Error Correcting Code is a set C of codewords in Σ^n (for alphabet Σ), associated with a encoding function that injects messages of some smaller domain Σ^k into codewords ($k < n$). Decoding is the mapping between elements of Σ^n to messages from Σ^k, mapping the element to the nearest element in C and then applying the inverse of the encoding function.

Regenerating Codes, introduced by Dimakis *et al.* [DGW$^+$10], are error correction codes that have an efficient algorithm for the *exact repair problem*: Namely, assuming each symbol of the codeword is stored in some node, such codes can regenerate a failed node with low communication. For this reason, regenerating codes are commonly found in distributed storage systems. In these scenarios, since storage efficiency and reliability are also highly important, maximum distance separable (MDS) codes have been used, such as the Reed-Solomon code and different variations of it [REG$^+$03, BTC$^+$04]. MDS codes hold the property that any k out of n nodes can recover the entire data and specifically the data held by the other $n-k$ (possibly failed) nodes. Regenerating Codes are built to allow systems to recover a failed node with low total communication from the other nodes by sending partial information from possibly more than k nodes, instead of of full symbols from k nodes.

There is a trade-off between the amount of storage required by each node and the repair bandwidth where both minimum-storage, minimum-communication and other more balanced codes were introduced [DGW$^+$08, RSK11].

Although new regenerating codes are known, in practice Reed-Solomon codes and variations of it are still being used in distributed storage systems. Recent results show that even for Reed-Solomon codes, regeneration bandwidth can be lowered below the bandwidth required by the naive algorithm [GW17].

1.3 Related Work

We now discuss in greater technical detail relevant existing results.

Guruswami and Wootters introduced in [GW17] new efficient ways for solving the exact repair problem for any arbitrary node of a Reed-Solomon codeword over a field $\mathbb{F}$ in various settings. In certain settings the required bandwidth may be lowered to $O(k)$ bits, as opposed to traditional $\Omega(k \log k)$ solutions (where k is the number of bits needed to represent an element of $\mathbb{F}$). Loosely, this is done by choosing some subfield $B \leq \mathbb{F}$ and in order to restore a failed node P_i, each node P_j sends its evaluations of some carefully chosen B-linear transformations from $\mathbb{F}$ to B on its symbol. These linear transformations are crafted such that as few of them are needed, but enough information is sent for restoring P_i's symbol.

We are interested in MPC protocols with low communication and computation. Original feasibility results in MPC [BGW88, GMW87, CCD88] had communication and computation complexity of $\Omega(|C| \cdot n^2)$ (Where $|C|$ is the circuit's size and n number of parties). Work in "Scalable MPC" [DI06] has yielded protocols with improved complexities. We present the best known results of three main scenarios in which MPC is considered.

- The first scenario assumes static adversaries and allows a small probability of error that decreases with the number of parties [DKMS14, BCP15, ZMS14]. In this setting Dani *et al.* [DKMS14] achieve information theoretic security against static (active) corruption of $\leq 1/8$ of the parties. The communication complexity of their protocol is $\tilde{O}(|C|)+poly(n)$ (where $\tilde{O}$ hides polylogarithmic factors in $|C|$ and n).
- In the second scenario, adaptive security is achieved with computational assumptions. Assuming a "computationally simple" pseudo-random generator, Damgard *et al.* [DIK+08] introduce a protocol that deals with an adaptive adversary corrupting $\leq 1/2$ of the parties and is computationally bounded. The protocol's complexity is $\tilde{O}(|C| \cdot poly(k)) + poly(k, n)$, where k is a security parameter.
- The last scenario we consider guarantees unconditional perfect security. Damgard *et al.* [DIK10] define a protocol coping with an adaptive adversary corrupting $< 1/3$ of the parties or an semi-honest adversary corrupting $< 1/2$ of the parties. The amortized communication and computational complexity of their protocol is $\tilde{O}(|C|) + D^2 \cdot poly(n, \log |C|)$ where D is the circuit's depth. The protocol assumes the circuit's multiplicational width is $\Omega(n)$, allowing to perform $\Omega(n)$ multiplications simultaneously with low amortized cost.

Our main focus on this project is on this last scenario requiring unconditional perfect security. The use of packed secret sharing has been introduced in previous works, allowing reduction of the amount of *times* a secret sharing subprotocol is used [FY92, DIK10]. Some of these works assume certain properties hold on the circuit or the inputs' field size.

1.4 Project Overview

Our goal in this project is to show that utilizing regenerating codes and specifically efficient exact repair algorithms can decrease the concrete communication bandwidth needed for a secure multi-party computation. In this project we implement the BGW protocol for the semi-honest case described in [BGW88]. It allows an honest majority perform a secure MPC over some arbitrary finite field. In addition to the traditional BGW protocol, it may also be configured to use the MPC multiplication protocol described in [DIK10] (see Section 2.3.2), allowing lower communication complexity for the computation of multiplication gates.

The second aspect of this project is the use of regenerating codes. In particular, we implemented a lower bandwidth protocol for reconstructing a secret shared by Shamir's secret sharing scheme (compared to the naive reconstruction). This reconstruction procedure is based on the exact repair protocol for the Reed-Solomon codes introduced by Guruswami and Wootters in [GW17]. Since shares in Shamir's secret sharing scheme are in essence elements of a Reed-Solomon codeword, an improved low-bandwidth algorithm for the exact repair problem for Reed-Solomon codes imply a low-bandwidth algorithm for secret share reconstruction. As opposed to using completely new types of codes, this strategy of using the same the secret sharing scheme's sharing procedure guarantees that MPC protocols that are based on Shamir's secret sharing will still have their correctness and security properties preserved.

The project was written in pure Python 2.7.8 using only built-in modules. The project spans over more than 4000 lines of code in 20 different files. We chose not to rely on existing 3rd-party modules both to ensure that the code would be extremely portable and easy to use, and for the educational purpose of learning about field extensions, irreducible polynomials, trace functionals and many more in a project-based-learning manner.

2 The Project

In this section we describe the project in greater detail. Our code may be found at:
https://github.com/oriorio/pyMPC

2.1 Overview

The project allows a variable number of parties to perform a secure MPC of some user-defined circuit over an arbitrary ore-specified finite field $\mathbb{F}$, in the presence of semi-honest adversaries. The MPC protocol is based on the protocol defined in [BGW88], but also supports efficient multiplication using the protocol described in [DIK10] (see Section 2.3). The number of parties, circuit size and field size may be arbitrary, and are limited only by the system's resources.

The circuits supported are arithmetic circuits over $\mathbb{F}$ - containing linear gates such as addition or scalar multiplication, as well as multiplication gates. In the current version, only a single output circuit may be used. These circuits may be defined freely by the user (See Section 2.2.3). The secret sharing scheme underlying the MPC may also be easily changed, as long as needed properties hold for correctness of the BGW protocol. For linear circuits, any linear secret sharing scheme may be used. For circuits containing multiplication gates, a more strict scheme is needed. The project contains implementations of three secret sharing schemes which all may be used for any circuit as defined (See Section 2.2.7).

The project was written from scratch in Python 2.7.8 without the use of any 3rd-party packages. All of the code was written entirely by the author except for parts of the matrix manipulation module (See Section 2.2.6). Currently, the entire program runs in a single-threaded process, but the project was designed in a way that in future implementations it could be easily converted to a scenario where nodes sit on different threads or even machines.

The execution of the program may be divided into three logical steps:

1. Set-up step - where the parameters of the MPC are generated and processed *without interaction between parties* and even without the knowledge of the circuit. This includes calculating the needed transformations that will be used during the computation, setting up the irreducible polynomials when working in field extensions, finding good parameters for the secret sharing scheme and more. These procedures depend only on the basic parameters of the MPC: the number of parties, the size of the field, the type of the secret sharing scheme and so on. The setp-up step may be precomputed for a variety of parameters in advance.
2. MPC preprocessing step - where parties interact in order to minimize the computation and communication complexities that will be needed later in the circuit evaluation step. This step includes all the computations that may be precomputed independently of the parties' inputs (and in particular before sharing their inputs). This step ends when the parties share their inputs.
3. MPC circuit evaluation step - the "core" step of the MPC, interactively evaluating the circuit gate-by-gate and finally reconstructing the circuit's output.

In Figure 1 the structure of the project is introduced. An elaborated explanation about each of the different parts is given in Section 2.2.

2.2 Detailed Structure

In this section we will describe in greater detail the different parts of the program and their roles. The main.py file shows an example of how to run the program: it does the necessary set-up and then runs the whole MPC using the Master object.

2.2.1 Master

The Master object is the "conductor" of the MPC protocol. A single Master object initializes and runs the whole MPC protocol. Its role is to initialize the different "Node"-s (party objects) and to synchronize between them. It has no interest in or use of the parties' messages. It merely instructs the parties what is the next step, allowing easier synchronization from the coding perspective. It assumes that all the necessary steps of the set-up step are finished - a field, a circuit, a specific secret sharing scheme and so on are all defined. After initializing a master object the set-up step is finished. By invoking the Master's *init* method the required MPC preprocessing steps run. Invoking the *run* method results in the circuit evaluation step to take place.

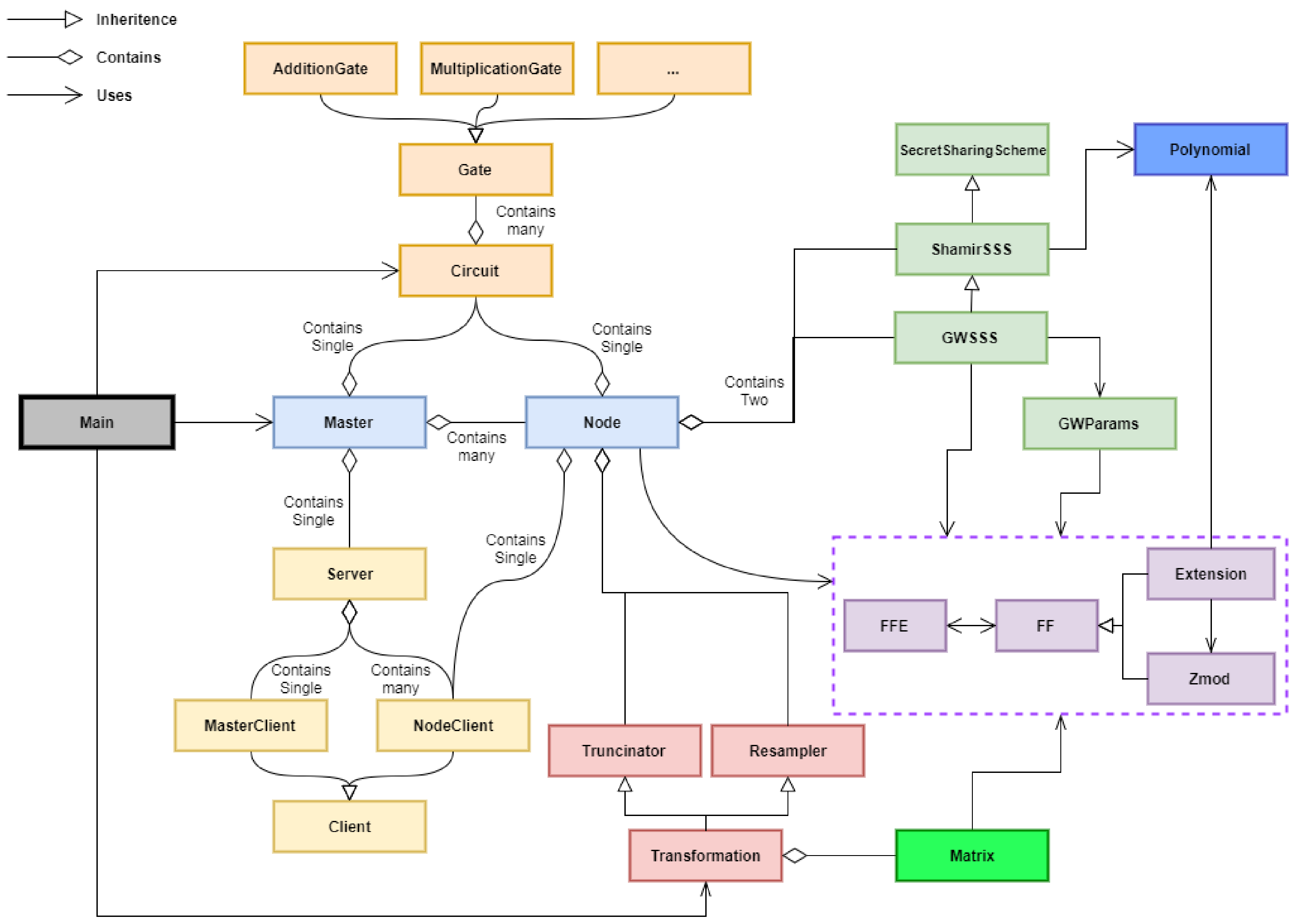

Figure 1: Program's Structure

In the current version of the Master-Node interface, all messages sent by the parties go through the Master and then sent to their proper destination. This implementation decision was made from the development's perspective - a cleaner a simpler interface. From the MPC point of view - this means that the computation is not secure, as an adversary controlling this Master may see all of the messages sent between the Nodes. But this design decision does not affect the protocol itself by any means and it could be easily changed in the future by setting $n(n-1)$ direct private peer-to-peer channels.

The Master also allows logging to be triggered when certain messages are received, as well as "traffic counters" for auditing use by the user. Also, a breakpoint debugging interface is in place allowing an interactive shell to be opened on certain triggers, for easier development and user intervention during the computation. These features may be turned off from the configuration.

2.2.2 Node

A Node object represents a party in the MPC. Each Node holds a copy of the circuit, has some secret input and knows the parameters of the MPC (number of parties, secret sharing scheme etc). The set-up, MPC preprocessing step and circuit evaluation step are "paced" by the Master using control messages that are passed between it and the Nodes. This allows the Node's implementation to be a state-machine that transitions between its states according to its current state, control messages received from the Master and data messages received from the Nodes. This state machine is depicted in Figure 2.

The messaging protocol implemented for Node-Node and Node-Master communication uses serialized Python objects encoded as byte strings, which allows them to be transferred over any type of communication channel in future implementations. A detailed explanation of the Node's state machine follows.

When a Node starts its state is *UNINITIALIZED*. This state is the initial state of a Node, which can always be reset to when receiving the *Reset* control message. The control messages *SetCircuit*, *SetInput* and *SetSecretSharing* are used to set the Node's circuit, secret input and secret sharing scheme's parameters (such as the position of the secret in the scheme (see Section 2.2.7) and more - see Section 2.2.8), respectively. The Node's input may be either set to some specific input or to be randomly chosen by the Node to some element in the field. Each node holds two active secret sharing schemes initialized by *SetSecretSharing* - one for "low dimensional" computation (for threshold t) and one for "high dimensional" computation used in multiplication (for threshold $2t$). After all three messages are received, the Node transitions to the *INITIALIZED* state. All of the operations done by the Node during the *UNINITIALIZED* state require no interaction with other Nodes. This phase is part of the set-up step of the run.

While *INITIALIZED*, MPC preprocessing is allowed. In the current version of the project, three types of operations are allowed during preprocessing:

1. When the multiplication protocol of the MPC is done using the protocol described in [DIK10], random mask generation may be triggered by receiving the control message *GenBulkMulMasks*. See Section 2.3.2 for an detailed explanation of this protocol.

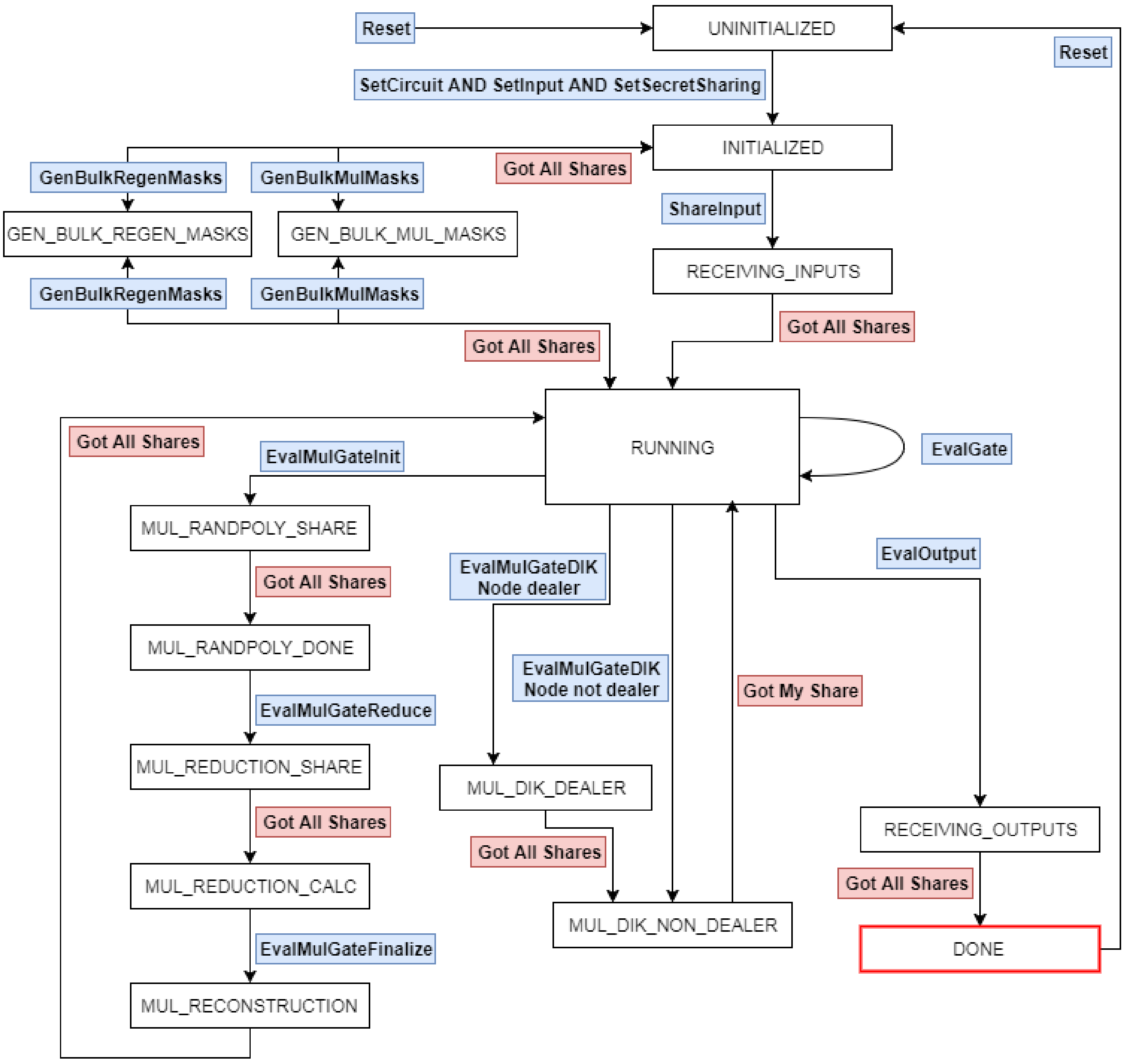

Figure 2: Node State Machine

During this process the Node's state transitions to *GEN_BULK_MUL_MASKS*, and after computation is done the Node transitions back to *INITIALIZED*.

2. Similarly, a process for generating random masks for different positions than the secret's position may be triggered by receiving the control message *GenBulkRegenMasks*. This feature is for future implementation of regenerating a fallen Node during the circuit evaluation step. During this process the Node's state transitions to *GEN_BULK_REGEN_MASKS* and later back to *INITIALIZED*.

3. Finally, and in order to move on from the *INITIALIZED* state, when the control message *ShareInput* is received, the Node shares its input using the "low-dimensional" secret sharing scheme and transitions to *RECEIVING_INPUTS*. After all input shares are received from the other Nodes, the Node transitions to the main *RUNNING* state, ending the MPC preprocessing step of the run.

The *RUNNING* state is the default state for a Node during the MPC circuit evaluation step. In this state, receiving the *EvalGate* control messages instructs the party to perform a local computation on its shares corresponding to a specific linear gate of the circuit, such as an addition gate. As communication between the parties for this computation is not required, the Node stays in the *RUNNING* state during the entire process. Receiving *GenBulkMulMasks* or *GenBulkRegenMasks* is also allowed during the *RUNNING* state, where the behavior is the same as if they were received in the *INITIALIZED* state with the only difference that after the computation is done the Node transitions back to *RUNNING* state. Another two types of valid control messages during the *RUNNING* state are messages that start a multiplication protocol between the parties. As both the multiplication protocol of [BGW88] and the multiplication protocol of [DIK10] are implemented, Nodes transition to different states depending on the type of protocol used. From now on we will refer to these multiplication protocols as BGW-multiplication and DIK-multiplication respectively. See Section 2.3 for a overview of these two protocols.

The BGW-multiplication protocol is triggered when receiving the *EvalMulGateInit* control message. The message specifies some specific gate to be evaluated interactively, for which all of the Nodes currently hold shares of its two inputs.

1. First, a random mask is generated interactively by the Nodes. Each Node picks a random polynomial of degree $2t$ over $\mathbb{F}$ that evaluates to zero at the secret's position, and sends shares to the other Nodes (the appropriate evaluation of the polynomial to each Node). The Node transitions to *MUL_RANDPOLY_SHARE*.

2. When all n shares of the previous step are received by the Node (all are evaluations of random polynomials at some specific point), they are all combined to single random mask share of some coordinated polynomial. The local multiplication of the gate's input shares is calculated and then the random mask share is added to the result. Node transitions to *MUL_RANDPOLY_DONE* and waits for further control messages.

3. When the *EvalMulGateReduce* control message is received, the masked local multiplication is shared to the parties. The Node transitions to state *MUL_REDUCTION_SHARE*

4. When all of the new shares from the previous step are received, the Node transitions to state *MUL_REDUCTION_CALC* and waits for further control messages.

5. When the *EvalMulGateFinalize* control message is received, the precomputed Truncinator linear transformation is locally applied to the shares vector. Results are then sent back to the parties (the component i of the resulting vector is sent back to party i). The Node transitions *MUL_RECONSTRUCTION* and waits for all of its new shares to be received.

6. The final step, after all shares are received, is straightforward reconstruction. The n received shares are reconstructed to our new desired degree t share of the multiplication. The Node transitions back to *RUNNING* state and waits for further control messages.

The DIK-multiplication protocol is triggered when receiving the *EvalMulGateDIK* control message. The message specifies some specific gate to be evaluated interactively, which all of the Nodes currently hold shares of its two inputs. It also specifies some specific party, for which will be the "dealer" of this interactive protocol. The dealer may be any party per multiplication gate (see Section 2.3.2).

1. Upon receival of the *EvalMulGateDIK* control message, the Node takes the next unused mask generated in the MPC preprocessing step previously generated by the *GenBulkMulMasks* control message (if none are left, an additional bulk of masks may be generated during the computation - but in current implementation this responsibility is on the Master). This unused mask is a pair of degree t and degree $2t$ shares. The local multiplication of the input shares is computed, and the higher degree mask is added to the result. The Node then sends this masked multiplication to the dealer party. If the node is the dealer for this multiplication it transitions to *MUL_DIK_DEALER*, or else it transitions to *MUL_DIK_NON_DEALER*.

2. Only the dealer, upon receival of all shares, reconstructs the masked secret multiplication in the clear. The dealer then shares this masked multiplication in the "low-dimension", so each party receives back a single share. The dealer transitions to the same state as the rest of the Nodes *MUL_DIK_NON_DEALER*.

3. When the Node receives back a share from the dealer, it subtracts its "low-dimensional" mask from the received share, resulting in a share of the desired multiplication in dimension t. The Node transitions back to *RUNNING* state and waits for further control messages.

The last control message allowed during the *RUNNING* state is *EvalOutput* which causes parties to reveal their final shares and reconstruct the output of the circuit. Upon receival of the *EvalOutput* control message, the Node sends its final share of the output to all of the parties. The Node then transitions to *RECEIVING_OUTPUTS*. After all shares are collected - final reconstruction of the output is done. The Node transitions to *DONE* and stays in that state until a *Reset* control message is received.

2.2.3 Circuit

A general implementation of a circuit and a gate are defined in the circuit module. A Gate object has a variable number of inputs and a single output. Each gate has a label and it may be evaluated given inputs. Specific types of gates may be defined, implementing the needed API of a Gate. The program currently defines the arithmetic AdditionGate, ScalarMultGate and MultiplicationGate (which add inputs, multiply an input by a constant scalar and multiply inputs respectively).

A Circuit is made of a set of gates and the connections between them. A connection may be from one gate's output to another gate's input, from the inputs of the circuit to some gate's input, and from specific gate's output to the output of the circuit. These gates and connections define the circuit. It may be viewed as an directed acyclic graph (DAG) with gates as vertices and connections are edges (directed from one gate's output to another's input). A circuit may be evaluated directly, or construct a list of its gates in some valid evaluation order (an ordered list of gates $(G_1, G_2, ..., G_{|C|})$ such that G_i may be evaluated using only the circuit inputs or outputs of G_j for $j < i$). This list is built by traversing the DAG in post-order starting from the output of the circuit.

2.2.4 Polynomials

The implementation of polynomials in the code is by the Polynomial data type. A polynomial is represented by its list of coefficients, which in turn could be from any type supporting arithmetic operations and a minimal API (for returning the "zero" and "one" elements for an example). In this project, polynomials were used over finite fields, with FFE objects as their coefficients (see Section 2.2.5). But a Polynomial may be used in a different context with coefficients from a different type with small or no effort.

The polynomial implementation supports numerous operations - starting from basic arithmetic operations such as addition and subtraction, through multiplication and exponentiation, modulus, finding the GCD of two polynomials, and even inversion and checking irreducibility. Algorithmic efficiency was in mind when implementing these operations. Fast exponentiation is done by using the exponentiation by squaring method. Inversion is done applying the extended Euclidean algorithm. Checking irreducibility was implemented using Rabin's irreducibility test [Rab80] stating that a polynomial f of degree n over a finite field $GF(q)$ is irreducible if and only if the following holds: $gcd(f, x^{q^{n_i}} - x) = 1$ for all $n_i = q/p_i$ where p_i is a prime factor of n, and $(x^{q^n} - x) \, mod \, f$ is not zero.

In the project, polynomials are needed in three different scenarios. First, finite fields which are not simply $\mathbb{Z}_p$ for some prime p, are implemented as polynomials over such $\mathbb{Z}_p$. More generally, the elements of a finite field extension are implemented in general as a polynomial over the subfield. The second scenario where polynomials are used are in the implementation of Shmair's secret sharing scheme. And last, polynomials are used in the set-up phase for finding good parameters for the [GW17] reconstruction scheme.

2.2.5 Finite Fields

The finite fields (GF) module implements three important types: Zmod, Extension and FFE. Zmod and Extension are types for representing finite fields. With these two types, any prime-power sized finite field may be constructed and used. To avoid the hassle of using these two types directly, the helper function $GF(q, n)$ builds and returns a finite field of size q^n (for some prime power q). These two types support various operations such as inversion of elements or finding a generator (a primitive element). Finding a generator for $GF(q)$ is done by repeatedly picking some random non-zero element $g \in GF(q)$ and checking if $g^{(q-1)/p_i} \neq 1$ for all prime factors p_i of $q - 1$.

A Zmod object represents the finite field whose elements are represented as $\{0, ..., p-1\}$ where p is prime - also denoted as $\mathbb{Z}_p$.

Extension is the extension field of some other field (a Zmod or some other Extension). It is used for representing $GF(p^n)$ for $n > 1$ (and prime p) where the subfield is then defined to be $\mathbb{Z}_p$ (as a Zmod object), or for defining an arbitrary extension of some given $GF(p^m)$ subfield. Elements of an Extension are represented as polynomials over the subfield, and operations are done modulo an irreducible polynomial of degree n (where n is the extension degree). The generation of such irreducible polynomials over the subfield is due to the Polynomials module, as well as the implementation of all of the arithmetic operations. Extensions of extensions may be defined with no limitations beyond the machine's resources.

Finally, the FFE type (finite field element) implements a single finite field element and all the necessary arithmetic operations on it. A complete abstraction is made for the user using advance pythonic programming no matter if this element is in Zmod and represented by some $\{0, ..., p-1\}$ integer, or if this element is in some Extension of some other Extension and thus represented by polynomials with coefficients that are polynomials themselves. The depth of this recursion is only limited by the machine's resources, while maintaining a clean and simple implementation. Efficiency was in mind when designing this basic type - fast exponentiation done by using the exponentiation by squaring, and multiple exponents may be supplied and computed in parallel.

2.2.6 Matrix

A simple and yet strong and generic module for matrix manipulation, based on the open source implementation [IV14]. The data contained in the matrix could be of any type supporting arithmetic operations (such as the built-in float or FFE). Basic operations on a matrix or between two matrices are implemented, as well as more advance operations such as Gaussian elimination or inversion.

Another specific type of matrix implemented is the VanderMatrix - allowing easy definition and some faster operations on Vandermonde matrices. Fast inverse calculation is due to [MS12]. In this project the matrices are used for computing transformations (during the set-up step) that are used later when handling multiplication gates in the MPC (during the MPC preprocessing step or during the evaluation of the circuit, see Section 2.3).

2.2.7 Secret Sharing Schemes

A secret sharing scheme in the program has to implement a few basic methods: sharing a secret, preprocessing a share before reconstruction and a reconstruction method. This is the API used by the Nodes during the MPC, allowing the secret sharing schemes to be changed or added without requiring further change of the code or logic of other modules (as long as the MPC requirements from the secret sharing scheme still hold).

Three secret sharing schemes are implemented: NullSSS, ShamirSSS, GWSSS. The NullSSS is a simple scheme where "shares" are simply the secret itself. It may be simply defined as Shamir's secret sharing scheme where the threshold is set to zero. ShamirSSS is the implementation of Shamir's secret sharing scheme where the secret is embedded at an arbitrary user defined position σ of some random polynomial (We refer to this σ as *the position of the secret*). Reconstruction is done by Lagrange interpolation at the specified position. The last scheme supplied is GWSSS which is Shmair's secret sharing scheme with low-bandwidth reconstruction implemented according to the algorithm introduced by [GW17]. Reconstruction is done by parties sending elements of the subfield, and then building back the secret (which is an element of the extension field) by making some pre-calculated linear transformation with these subfield elements. A more elaborate explanation follows.

2.2.8 GWSSS and GWParams

In this section we will explain how [GW17] low-bandwidth reconstruction works, and how GWParams generates parameters for this type of scheme. We will start with a quick overview on notation and how the scheme works.

Let $\mathbb{F} = GF(q^m)$ be some field extension of $B = GF(q)$ for some prime power q and some $m > 1$. Let $A = \{a_1, ..., a_n\} \subseteq \mathbb{F}$ be a set of evaluation points, and some $a^* \in \mathbb{F}$, $a^* \notin A$ a position to be recovered. The secret sharing procedure chooses some random polynomial f over $\mathbb{F}$ of degree t (where t is the threshold) such that $f(a^*)$ evaluates to the secret, and the shares $f(a_1), ..., f(a_n)$ are given out to the parties. Denote P_j the party receiving $f(a_j)$. We want to find a low-bandwidth linear reconstruction algorithm that takes some subfield elements calculated from each of $f(a_1), ..., f(a_n)$ and reconstructs $f(a^*)$ from them. In order to do that, we make use of the B-linear transformations from $\mathbb{F}$ to B which are exactly all the trace functionals $L_\lambda(\beta) = Tr_{F/B}(\lambda\beta)$, where $Tr_{F/B}$ is the trace field of $\mathbb{F}$ over B defined by:

$$Tr_{F/B}(\beta) = \beta + \beta^q + \beta^{q^2} + ... + \beta^{q^{m-1}}$$

For building some specific repair scheme, we can pick any basis $Z = \{z_1, ..., z_m\}$ for $\mathbb{F}$ over B and $\mu_{i,j} \in \mathbb{F}$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$ such that for any polynomial f over $\mathbb{F}$ of degree less than t:

$$z_i \cdot f(a^*) = \sum_{j=1}^{n} \mu_{i,j} \cdot f(a_j) \qquad \forall i = 1, ..., m \tag{1}$$

By taking traces from both sides we get that for any polynomial f over $\mathbb{F}$ of degree less than t:

$$Tr_{F/B}(z_i \cdot f(a^*)) = Tr_{F/B}(\sum_{j=1}^{n} \mu_{i,j} \cdot f(a_j)) = \sum_{j=1}^{n} Tr_{F/B}(\mu_{i,j} \cdot f(a_j)) \qquad \forall i = 1, ..., m \tag{2}$$

by the B-linearity of the trace.

Now, let there be some specific f for which we want to reconstruct $f(a^*)$. Denote $M_j = \{Tr_{F/B}(\mu_{i,j} \cdot f(a_j)) \mid 1 \leq i \leq m\}$. If $\forall P_j$ will publish their M_j, then all $Tr_{F/B}(z_i \cdot f(a^*))$ may be computed according to (2). If we denote the dual basis of Z as $\{v_1, ..., v_m\}$, we can now reconstruct $f(a^*)$ by:

$$f(a^*) = \sum_{i=1}^{m} Tr_{F/B}(z_i \cdot f(a^*)) \cdot v_i$$

But party P_j doesn't actually need to send all of M_j, it can rather send $\{Tr_{F/B}(\mu \cdot f(a_j)) \mid \mu \in Q_j\}$ for some minimal spanning over B set Q_j of the party's $\{\mu_{i,j} \mid 1 \leq i \leq m\}$. Then, for reconstruction, M_j may be built as a linear combination of this received data, then from all M_j sets $\{Tr_{F/B}(z_i \cdot f(a^*)) \mid 1 \leq i \leq m\}$ may be computed and finally $f(a^*)$ may be recovered.

This defines a general scheme which works for any basis Z of $\mathbb{F}$ over B and $\mu_{i,j}$ as long as the conditions of (1) hold. The bandwidth of the scheme is:

$$\sum_{j=1}^{n} |Q_j| = \sum_{j=1}^{n} dim_B(\{\mu_{i,j} \mid 1 \leq i \leq m\})$$

of B-symbols. In the worst case scenario, $\forall j : |Q_j| = m$, meaning that at most nm of B-symbols are sent. But as $\mathbb{F}$ is a degree m extension of B, this amount of bandwidth is equivalent to n $\mathbb{F}$-symbols, which is the bandwidth required by classic reconstruction of Shamir's secret sharing in our MPC.

But bandwidth may be much lower if these parameters are more carefully selected. Guruswami and Wootters show in [GW17] that finding parameters that yield a low bandwidth scheme is equivalent to finding some nice polynomials. For a set of m polynomials $\mathcal{P}$ over $\mathbb{F}$, such that $\{p(a^*) \mid p \in \mathcal{P}\}$ is a basis of $\mathbb{F}$ over B it is shown that a scheme with:

$$|Q_j| = dim_B(\{p(a_j) \mid p \in \mathcal{P}\}) \qquad \forall j = 1, ..., n$$

may be built, explicitly showing how to compute the basis Z and $\mu_{i,j}$ values from $\mathcal{P}$. The degree of each of these polynomials has to be less than $n - t$.

The module GWParams is used to find and construct good parameters for the GWSSS module. It receives as input the fields $B \leq \mathbb{F}$, the set of points A and the scheme's threshold t. For any position to be recovered a^*, the module searches for "nice polynomials" yielding a low-bandwidth scheme. The search is made on polynomials that have lots of their roots from A, while always requiring that $\{p(a^*) \mid p \in \mathcal{P}\}$ is a basis (by first "normalizing" each polynomial by an appropriate scalar). The search algorithm is based on the code supplied by Wootters [Woo15].

After finding good enough polynomials the module computes the dual basis of Z, the $\{\mu_{i,j}\}$ values and a basis B_j for $\{\mu_{i,j} \mid 1 \leq i \leq m\}$ for each $j = 1, ..., n$. The method for calculating the dual basis is implemented according to [McE87]. These are the parameters loaded into GWSSS. For reconstruction, when a party P_j has to send its part for recovering $f(a^*)$, it sends $Q_j = \{Tr_{F/B}(\mu \cdot f(a_j)) \mid \mu \in B_j\}$, and from these the GWSSS object computes the reconstructed $f(a^*)$ as described earlier.

Correctness of this scheme is proved in [GW17]. Security follows from the fact that a party P_j sharing Q_j for some specific polynomial f hands out information that is only dependent on $f(a_j)$ (as B_j and the trace functional are publicly known). Meaning that an adversary holding some party's Q_j can infer no more but $f(a_j)$ itself - which is what P_j sends in the classic reconstruction of Shamir's secret scheme protocol, that was proved as safe to use [BGW88].

The process in which the "nice" polynomials are constructed gives an upper limit to the number of B-symbols the reconstruction protocol requires. As we already mentioned, the worst case bandwidth of this scheme is nm when $\forall j : |Q_j| = m$, meaning when $dim_B(\{p(a_j) \mid p \in \mathcal{P}\}) = m$. But building the polynomials such that they have many factors of the form $\{(X - \alpha) \mid \alpha \in A\}$ we can decrease these dimensions dramatically. Specifically, if each $p \in \mathcal{P}$ is some $(n - t - 1)$ degree polynomial made by multiplying $(n - t - 1)$ distinct factors from $\{(X - \alpha) \mid \alpha \in A\}$ an upper limit on our bandwidth would be:

$$\sum_{j=1}^{n} dim_B(\{p(a_j) \mid p \in \mathcal{P}\}) \leq nm - |\mathcal{P}|(n - t - 1) = nm - m(n - t - 1) = m(t + 1)$$

This approximately matches the naive Shamir's secret sharing scheme reconstruction bandwidth of t $\mathbb{F}$-symbols (which are mt B-symbols). But in practice, even better results are achieved by randomly picking such polynomial sets (as this worst case is not very common), sometimes requiring even half of the needed mt bandwidth. We also note that a trivial lower bound for a reconstruction protocol of $\mathbb{F}$ over B is $(m+t+1)$ B-symbols. We refer to [GW17] for more details and discussion.

2.2.9 Transformation

The Transformation module is used to perform linear transformations defined by a matrix. Its main use is for representing linear transformations that are defined by a complex computation that could be calculated once but later applied multiple times. In this project, the two types transformations used concern multiplication in the MPC protocol. When the [BGW88] multiplication method is used, a transformation for re-sampling some high degree polynomial after truncating it to a lower degree is needed (see Section 2.3.1). When [DIK10] multiplication is is used, a transformation for re-sampling a polynomial defined by a set of points is needed in the random masks generation procedure (see Section 2.3.2). These two transformations involve Vandermonde matrices and their inverses, which their computation and multiplication are time consuming (especially as the number of parties increase). However, they only need to be computed once for any given set of parameters (number of parties, secret sharing scheme etc). The module ensures this calculation is only done once (in our case - during the set-up step), while allowing later to apply the transformation relatively fast.

2.3 MPC multiplication protocols

In this section we describe in more detail how the two MPC multiplication protocols implemented in our project are interactively computed. In both cases, our starting point is two secret inputs a, b shared between the parties using Shamir's secret sharing with threshold t.

When the protocol terminates, we require that parties hold shares of ab in a scheme with threshold t. Denote the position of the secret as σ. Denoting the parties $P_1, ..., P_n$, each party P_i holds the shares $f(x_i),\ g(x_i)$ where f, g are random polynomials such that $f(\sigma) = a$ and $g(\sigma) = b$. Local multiplication of $f(x_i), g(x_i)$ indeed yield a share of some polynomial that evaluates to ab at σ, but its degree is $2t$ and it's not random - as we know it has the property of being some multiplication other polynomials. So a protocol for generating shares in lower degree is needed. We describe the two protocols we've implemented in this project.

2.3.1 BGW multiplication

We first note that this protocol is correct only when $\sigma = 0$, as in the "classic" scenario of Shamir's secret sharing where the secret is stored as the polynomial's free term. The approach taken by Ben-Or *et al.* [BGW88] is to first locally multiply the shares so parties hold shares from the polynomial $f \cdot g$ of degree $2t$, then interactively randomize this polynomial by adding some uniformly chosen polynomial of degree $2t$ that has a zero free term, and finally perform some linear transformation on the vector of shares to reduce the degree of the polynomial to t. More specifically, the protocol is as follows:

1. Each party locally multiplies the input shares.
2. The parties interactively generate shares of some $2t$ degree random polynomial that has a zero free term. Each party P_i picks a random polynomial r_i of degree $2t$ such that $r_i(0) = 0$, and sends $r_i(x_j)$ to P_j for all $1 \leq j \leq n$.
3. Denote by h the degree $2t$ polynomial:

   $$h(x) = f(x) \cdot g(x) + \sum_{j=1}^{n} r_j(x)$$

   Each party P_i calculates $h(x_i)$ by summing all of its received shares $\sum_{j=1}^{n} r_j(x_i)$ and adding it to their locally multiplied shares $f(x_i) \cdot g(x_i)$. Since h has ab as its free term, the parties now hold shares of ab generated by some uniformly random polynomial of degree $2t$.
4. Now the parties interactively perform a degree reduction from $2t$ to t of their shares. The parties currently together hold n evaluations of a random polynomial of degree $2t$, and want to hold n new shares of some random polynomial of degree t which still has ab as its free term. One way to do that is by calculating the evaluations of h truncated to degree t from the evaluations they currently hold. Let $Q(x) = q_0 + q_1 x + ... + q_{2t} x^{2t}$ be a polynomial of degree $2t$ and denote $\hat{Q} = trunc_t(Q(x)) = q_0 + ... + q_t x^t$ the truncation of the polynomial to degree t. Define:

   $$F_{reduce}(Q(x_1), ..., Q(x_n)) = (\hat{Q}(x_1), ..., \hat{Q}(x_n))$$

   This is in fact a linear transform that may be computed by the matrix $V \cdot P_t \cdot V^{-1}$ where V is the square Vandermonde matrix for $(x_1, ..., x_n)$ and P_t the projection matrix from

$2t$ to t (a square matrix of order $2t$ where $P_t(i,j) = 1$ if and only if $i = j$ and $i \leq t$, else zero). The intuition is that V^{-1} performs interpolation on the points to recover the coefficients of Q, then P_t zeros out the coefficients of terms of degree greater than t and finally V evaluates this truncated polynomial.

Applying this transformation on the shares may be computed interactively as a "mini-MPC" with the current n shares as inputs. Therefore:

(a) Each party P_i shares its "secret" $h(x_i)$ value over some new random degree t polynomial (creating "sub-shares"). Each party receives a sub-share from each party.

(b) Next, each party locally evaluates the linear degree reduction transformation on its sub-shares. Results are returned to the parties: $\forall i, j$ party P_i sends component j of its resulting vector to party P_j.

(c) Finally, each party reconstructs its new share $\hat{h}(x_i)$ from its received sub-shares. Since $\hat{h}(0) = h(0) = ab$, parties now hold a degree t sharing of ab as needed.

This linear transformation is a constant transformation throughout the entire MPC, as it only depends on t and the evaluation points. Therefore the calculation of this transformation $V \cdot P_t \cdot V^{-1}$ should only be done once during the set-up step of the run. This transformation is implemented by the Truncinator object and may be precomputed in advance or during the set-up step.

2.3.2 DIK multiplication

The idea behind the multiplication protocol introduced by Damgard *et al.* in [DIK10] and earlier in [DIK$^+$08] has two parts. The first, during the MPC preprocessing step, is generating in bulk degree t and $2t$ random masks that will be used during the evaluation step. The second part, when evaluating some multiplication gate, is using one of these masks for randomizing ("masking") $f \cdot g$, then sending the new masked shares to some chosen "dealer" party which reconstructs the masked multiplication in the clear and finally reshares it in degree t. Parties receiving their new share of *masked ab* use their degree t mask share to locally compute the desired ab share. We now describe in more detail these two parts.

During the preprocessing step parties interactively construct $n - t$ random masks by computing some carefully chosen linear combinations of random polynomials that the n parties generated. This is as opposed to the BGW protocol where only a single mask was generated from such n polynomials. These carefully chosen linear combinations are defined by the $F_{resample}$ transformation, ensuring that all of the $n - t$ output polynomials are affected by the (at least) $n - t$ honest parties. This procedure may be repeated until enough random masks are generated:

1. Each party P_i chooses some random field element R^i and shares it both in degree t and in degree $2t$. More specifically, it randomly picks polynomials r^i_t of degree t and r^i_{2t} of degree $2t$ such that $r^i_t(\sigma) = r^i_{2t}(\sigma) = R^i$ and sends $(r^i_t(x_j),\ r^i_{2t}(x_j))$ to P_j.

2. Next, each party P_i locally computes a linear transformation $F_{resample}$ on each of their received vectors $(r_t^1(x_i), ..., r_t^n(x_i))$ and $(r_{2t}^1(x_i), ..., r_{2t}^n(x_i))$. We denote:

$$F_{resample}((r_t^1(x_i), ..., r_t^n(x_i))) = (m_t^1(x_i), ..., m_t^{n-t}(x_i))$$

$$F_{resample}((r_{2t}^1(x_i), ..., r_{2t}^n(x_i))) = (m_{2t}^1(x_i), ..., m_{2t}^{n-t}(x_i))$$

   Where the $\{m_t^1, ..., m_t^{n-t}\}$ polynomials are fixed linear combinations of the $\{r_t^1, ..., r_t^n\}$ polynomials defined by the rows of matrix of $F_{resample}$. If we denote by $\vec{f_1}, ..., \vec{f}_{n-t}$ the rows of this matrix, then for all $1 \leq j \leq (n-t)$:

$$m_t^j(\sigma) = \langle \vec{f_j} , (r_t^1(\sigma), ..., r_t^n(\sigma)) \rangle = \langle \vec{f_j} , (R^1, ..., R^n) \rangle = \langle \vec{f_j} , (r_{2t}^1(\sigma), ..., r_{2t}^n(\sigma)) \rangle = m_{2t}^j(\sigma)$$

3. For all $1 \leq j \leq (n-t)$ each one of these pairs $(m_t^j(x_i),\ m_{2t}^j(x_i))$ is considered a *random mask pair* of P_i. The degree t shares $\{m_t^j(x_i) \mid \forall i\}$ and the degree $2t$ shares $\{m_t^j(x_i) \mid \forall i\}$ are shares of a "secret" fixed linear combinations of of the R^i values defined by $\vec{f_j}$.

This procedure may be repeated until enough random mask pairs are generated. The $F_{resample}$ linear transformation is defined in the following way: Let $y = (y_1, ..., y_n)$ be some vector over the field and let Q be the polynomial of degree at most n such that $\forall i\ Q(x_i) = y_i$. Note that this is a deterministic definition as exactly one exists over the field. For some fixed $n-t$ distinct points $(\hat{x}_1, ..., \hat{x}_{n-t})$ we define $F_{resample}$ by:

$$F_{resample}((y_1, ..., y_n)) = (Q(\hat{x}_1), ..., Q(\hat{x}_{n-t}))$$

This is a linear transformation which can be computed by $V_{\hat{x}} \cdot V_x^{-1}$ for the two appropriate Vandermonde matrices. This matrix is *hyper-invertible* - for any subset R of its rows and C of its columns, the sub-matrix consisting of only the rows R and the columns C is invertible whenever $|R| = |C| > 0$.

It follows that the $n-t$ generated polynomials $m_t^1(x), ..., m_t^{n-t}(x)$ defined by the linear combination of $r_t^1(x), ..., r_t^n(x)$ according to the rows of $F_{resample}$ are independent linear combinations that are fully randomized by (at least) $n-t$ honest parties. This is true as long as $(\hat{x}_1, ..., \hat{x}_{n-t})$ and $(x_1, ..., x_n)$ are all distinct points. This claim is holds for $m_{2t}^1(x), ..., m_{2t}^{n-t}(x)$ by the same set of considerations.

We've defined how $n-t$ random mask pairs may be generated in bulk using $F_{resample}$. As this functionality only depends on t, the evaluation points and some set of other evaluation points $(\hat{x}_1, ..., \hat{x}_{n-t})$, the linear transformation matrix may be computed once and used multiple times. In our project, this transformation is implemented by the Resampler object and may be precomputed in advance or during the set-up step.

We now continue to describe the second step of the protocol. When evaluating some specific multiplication gate:

1. Each party P_i takes its next unused (j-th) random mask pair $(m_t^j(x_i), m_{2t}^j(x_i))$ and computes $s_i = f(x_i) \cdot g(x_i) + m_{2t}^j(x_i)$. This masked multiplication share of degree $2t$ is sent to some predefined "dealer" party.

2. The dealer party receives $s_1, ..., s_n$ and reconstructs from them $ab + m_{2t}^j(\sigma)$ in the clear. Then he reshares this value in the low-dimension t: for some random degree t polynomial h such $h(\sigma) = ab + m_{2t}^j(\sigma)$, the dealer sends $h(x_i)$ to P_i.

3. Each party P_i computes $h(x_i) - m_t^j(x_i)$ resulting in the desired share of some degree t polynomial $h - m_t^j$ that evaluates to ab at σ.

We note that the dealer may be any party chosen at random per multiplication, but as far as correctness and security it might as well be some constant party - as the dealer collects no additional information about the secrets. Because the reconstructed multiplication is masked by some random value that as a random variable is independent of all previously used random masks, the dealer gains no information about ab or the shares of the parties.

We also note that in the original papers the authors defined a more complex protocol for calculating multiple multiplication gates in parallel by using packed secret sharing [DIK10, DIK$^+$08]. In our project, we implemented only the basic multiplication protocol described above.

3 Results and Conclusion

In this project we've shown how regenerating codes, and more specifically efficient protocols for the exact repair problem, could be used in MPC protocols to decrease communication bandwidth. Although our results do not asymptotically improve the communication complexity, in tests we ran, the amount of bandwidth required by schemes generated by GWParams consistently required as little as 50% bandwidth compared to the required bandwidth by the naive reconstruction algorithm. Further, we showed that the upper limit on the bandwidth of generated schemes as implemented asymptotically matches the bandwidth of the naive reconstruction as the number of parties increases.

For practical uses, for a medium scale MPC with a few tens of parties, a full run of the MPC (including the set-up step) takes a matter of seconds when using DIK-multiplication. Distributing the parties over different machines could improve the results dramatically.

The project allows experimenting with the variable number of parties, arbitrary arithmetic circuit and field size, three different types of secret sharing reconstruction schemes and even two implementations of secure multiplication protocols.

From the coding perspective, the project contributes to anyone wishing to implement MPC protocols or regenerating codes, or even implementations requiring the use of arithmetic circuits, finite fields, polynomials, matrices etc in pure Python. The fact that the whole project is written without any dependencies and in pure Python makes it portable and easy to reuse.

Further development of the project may allow fast regeneration of a party's state in the case it goes down during the computation, using the low-bandwidth exact repair protocols. Other directions of expanding the project would be extending it to support multi-threading or distributing the parties over different machines allowing practical usage of the program in certain scenarios. Yet another option is implementing the full MPC protocol by [DIK10] or using different regenerating codes as the foundation of a secret sharing scheme. An additional research direction might be extending our results to other settings other than the semi-honest setting.

On the theoretical side, it is interesting to ask whether regenerating codes may be used to design asymptotically better MPC protocols. This direction is yet to be fully explored, and recent results in the regenerating codes field still need to be studied further to understand their implications on secure MPC.

References

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 742–762, 2015.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BTC$^+$04] Ranjita Bhagwan, Kiran Tati, Yuchung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, pages 337–350, 2004.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.

[DGW$^+$08] Alexandros G. Dimakis, Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *CoRR*, abs/0803.0632, 2008.

[DGW$^+$10] Alexandros G. Dimakis, Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Trans. Information Theory*, 56(9):4539–4551, 2010.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, pages 501–520, 2006.

[DIK$^+$08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam D. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology*

Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings, pages 241–261, 2008.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 445–465, 2010.

[DKMS14] Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Quorums quicken queries: Efficient asynchronous secure multiparty computation. In *Distributed Computing and Networking - 15th International Conference, ICDCN 2014, Coimbatore, India, January 4-7, 2014. Proceedings*, pages 242–256, 2014.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 699–710, 1992.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[GW17] Venkatesan Guruswami and Mary Wootters. Repairing reed-solomon codes. *IEEE Trans. Information Theory*, 63(9):5684–5698, 2017.

[IV14] Ismael-VC. https://github.com/ismael-vc/pymatrix, 2014.

[McE87] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers*, pages 110–111. Kluwer Academic Publishers, 1987.

[MS12] H. Moya-Cessa and F. Soto-Eguibar. Inverse of the Vandermonde and Vandermonde confluent matrices. *ArXiv e-prints*, 2012.

[Rab80] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.*, 9(2):273–280, 1980.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 73–85, 1989.

[REG$^+$03] Sean C. Rhea, Patrick R. Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y. Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003.

[RSK11] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, 2011.

[RSKR10] K. V. Rashmi, Nihar B. Shah, P. Vijay Kumar, and Kannan Ramchandran. Explicit and optimal exact-regenerating codes for the minimum-bandwidth point in distributed storage. In *IEEE International Symposium on Information Theory, ISIT 2010, June 13-18, 2010, Austin, Texas, USA, Proceedings*, pages 1938–1942, 2010.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Woo15] Mary Wootters. http://sites.google.com/site/marywootters/exhaust_fb.sage, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.

[ZMS14] Mahdi Zamani, Mahnush Movahedi, and Jared Saia. Millions of millionaires: Multiparty computation in large networks. *IACR Cryptology ePrint Archive*, 2014:149, 2014.