SCALING ASYNCHRONOUS MULTI-PARTY COMPUTATION: A SYSTEMS PERSPECTIVE

BY

SAMARTH KULSHRESHTHA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Andrew Miller

ABSTRACT

Modern multi-party computation applications no longer have a one-time execution pattern and instead are required to be run continuously like a service. They are deployed over the Internet which is inherently asynchronous and demand an infrastructure which is end-to-end robust, fault-tolerant and scalable. Unfortunately, existing frameworks fail to satisfy all of these requirements. Hence, many MPC applications are not yet practical due to the lack of an MPC framework that meets these needs.

This work presents a scalable protocol for generating preprocessed elements required for the execution of asynchronous MPC applications with optimal Byzantine fault-tolerance (robust when one-third of the nodes are corrupt) in the *asynchronous* setting. We implement this preprocessing protocol in HoneyBadgerMPC – a scalable, robust and fault-tolerant framework designed to develop, test and benchmark MPC applications efficiently.

To my mother, for her love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Andrew Miller for providing me with a research opportunity in his group. I thank him for being immensely supportive throughout and for providing valuable academic advice. It was an amazing experience working with him.

I am grateful to other collaborators of HoneyBadgerMPC - Xavier, Rahul Govind, Rahul Mahadev, Donghang, and Tom. I would also like to thank other members of the Decentralized Systems Lab - Sanket, Kevin, Hanyun and Yunqi for all the fun that I had over the past 1.5 years. I thank my friends, Vipul, Sameksha, Isha, Siddharth and Lavanya for their support and encouragement and for making my stay at UIUC an enjoyable experience. I deeply thank my mother who is a constant pillar of support in my life.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Secure multi-party computation (MPC) allows participants to compute a public function while ensuring *privacy* of their inputs and at the same time guaranteeing the *correctness* of the computed results among other properties.

MPC has found its applications in many domains - varying from auctions in financial markets [1, 2, 3, 4], preserving privacy in Machine Learning [5, 6, 7, 8], databases [9], key management [10, 11, 12, 13], location services [14, 15], energy trading in electricity markets [16], all the way up to satellite collision detection [17, 18]. Apart from satisfying the standard MPC requirements of integrity, correctness, privacy, fairness and guaranteed output delivery, these real-world applications pose an additional set of requirements on MPC systems:

1. In order to run these long-running applications, a system must be *available* to continuously process inputs, perform computations and return results.

2. These applications are deployed over the Internet which is an *asynchronous* network where a message between two participants can be delayed arbitrarily. As a result, it is infeasible to wait for inputs from all participants.

3. In the real world, nodes may crash and the system must continue to run in the presence of crash faults. This follows from the previous point and requires a system to be *fault-tolerant* so that it does not need all the participants to be online.

4. It is infeasible for some of these applications to abort their execution in the presence of malicious inputs i.e. a system which supports such applications must be *robust* to adversaries.

5. Many applications may need to support a large number of participants so such a system must be *scalable*.

Many related works [19, 20, 21, 22, 23, 24, 25, 26] follow the "offline-online" phase paradigm. In the offline phase the parties generate preprocessed elements such as shares of triples, shares of random field elements etc. These preprocessed elements are then consumed in the online phase to operate on secret shared inputs. This thesis focuses on generating the preprocessing elements which can be used in the realization of many of the above mentioned applications.

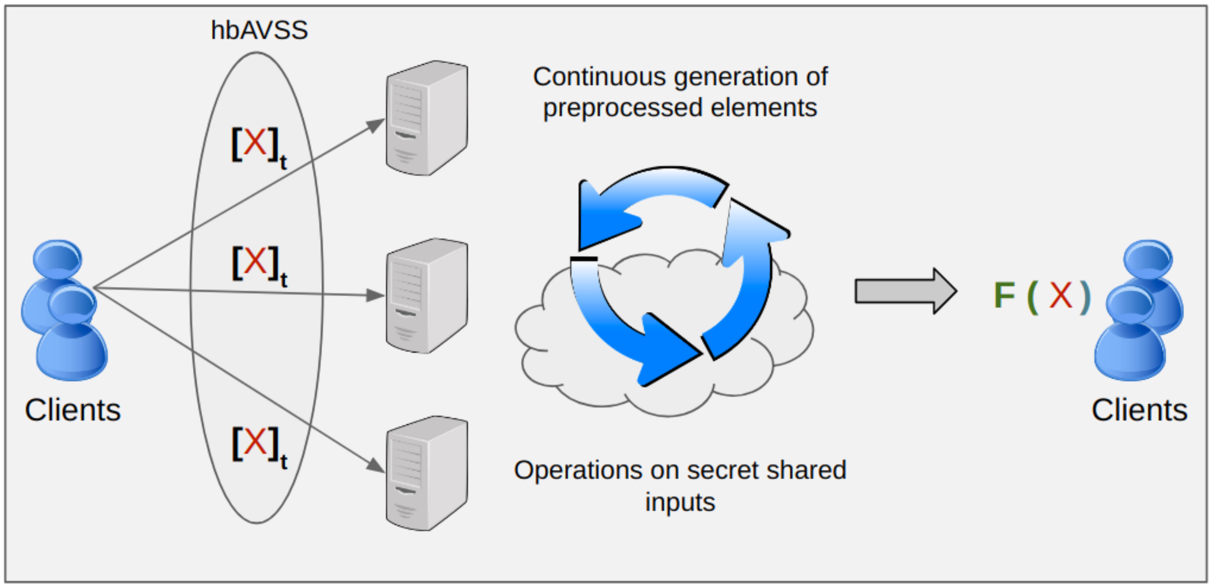

hbAVSS
$[X]_t$
$[X]_t$
$[X]_t$
Clients
Continuous generation of preprocessed elements
Operations on secret shared inputs
F (X)
Clients

Figure 1.1: Clients submit secret shared inputs, the servers operate on secret shared inputs and then return the output back to the user. The servers also generate all the preprocessed data in the background.

1.1 CONTRIBUTION OF THIS THESIS

A major consequence of the above mentioned requirements is that now we need an *infinite* supply of preprocessed elements in order to build a system which can continuously perform MPC. In this thesis, we present a protocol which allows us to generate a continuous supply of preprocessed elements in a setting where the threshold t for the number of malicious parties is less than $N/3$.

As a part of this thesis, we also propose HoneyBadgerMPC, a framework capable of performing both the preprocessing and the online phases of MPC securely in the presence of $t < N/3$ corrupt parties. HoneyBadgerMPC is asynchronous, robust, fault-tolerant and scalable. It has been designed to allow for rapid prototyping of new MPC applications. Fig. 1.1 provides a high-level overview of the entire framework. We implement and evaluate our preprocessing scheme using this framework.

As far as we are aware no other existing system achieves all of the previously stated properties. VIFF [24] is non-robust and does not scale to a large number of parties. EMP-toolkit [27], Obliv-C [28] and ObliVM [29] support only two-party computation. Sharemind [30] does not support active adversaries. SCALE-MAMBA [25] does not guarantee output in the presence of crash faults. Choudhary and Patra [26] present a work which achieves all these properties but for a corruption threshold of $t < N/4$. We are able to achieve a higher threshold of $t < N/3$ because of our hbAVSS [31] protocol.

1.2 THESIS ORGANIZATION

This thesis is organized as follows:

- Chapter 2 goes over concepts used throughout the rest of the thesis.
- In Chapter 3, we describe the design and architecture of HoneyBadgerMPC.
- Chapter 4 illustrates an example HoneyBadgerMPC program.
- In Chapter 5, we evaluate our preprocessing phase along with other algorithms and protocols implemented within HoneyBadgerMPC.
- Chapter 6 briefly describes related work.
- Chapter 7 discusses future work.

CHAPTER 2: PRELIMINARIES

2.1 SYSTEMS

2.1.1 Asynchronous execution

When a program performs I/O such as reading from or writing to a network socket, it can do so in two ways – blocking (synchronous) or non-blocking (asynchronous). A synchronous operation implies that the execution of the program is blocked until all of the data has been read or written to or from the network by the program. On the other hand, an asynchronous operation implies that the execution can continue once the program passes the responsibility of the data to the Operating System. For a write, it implies that the OS will take care of sending the data on to the network and for a read, it implies that the OS will copy the data, if any, from the network to a buffer allocated by the program. It is important to note that this is different from a multi-threaded execution which requires spawning a thread to do a read or write. Threads suffer from a synchronization overhead in order to access any shared state apart from the additional overhead of creating and scheduling them.

Another point worth noting is that this is not limited to just network communication but any operation managed by the OS, for example, spawning a separate process to execute computation in parallel, can be executed asynchronously.

2.2 PYTHON

2.2.1 asyncio

asyncio [32] is a concurrent programming framework with its syntax integrated directly within the Python [33] programming language. It uses the **async** keyword to denote a block of asynchronous code and the **await** keyword to block until an asynchronous operation finishes.

Listing 2.1 shows a sample program illustrating this framework. In this program we wish to check if two servers are up by pinging them. Line 13 creates a list of *two tasks* to ping the two servers. Note that when Line 13 is executed, the calls to function *is_server_up* are not invoked right away but are merely scheduled as *tasks* to be executed later. At Line 14, we want to wait until the two ping requests complete. As a result, these two *tasks* now get a chance to run. It is worth noting that both the tasks run concurrently i.e. while task 1 waits for its ping request to complete it gives up the control of the CPU and allows task 2 to send its ping request out.

In essence, multiple asynchronous operations (the two ping requests in this example) are executed concurrently as *tasks*. While a *task* waits on its asynchronous operation(s) to finish, other tasks have the opportunity to execute their asynchronous operations. This is possible since these tasks are not keeping the CPU busy but are simply waiting for their respective operations to complete.

```python
import asyncio


async def is_server_up(server_address):
    # Returns a None response if the server is not
    # up otherwise returns a non None response.
    response = await ping_server(server_address)
    return response is not None

server_addresses = ["172.29.30.4", "10.0.0.8"]

loop = asyncio.get_event_loop()
tasks = [asyncio.ensure_future(is_server_up(i)) for i in server_addresses]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

Listing 2.1: *asyncio* example program

asyncio.Future

The *is_server_up* method in the previous example can be written in a slightly different way to return immediately as shown in Listing 2.2. This is achieved by returning a *Future* object which will eventually resolve to the result once the ping request is completed. The caller of *is_server_up* now must **await** on the returned *Future*.

Note that the method no longer has the **async** keyword. Line 4 creates a task to execute the ping request. This task will run whenever it gets a chance. When it finishes, we set the result of the returned Future based on the ping response thus unblocking the caller.

In short, a *Future* object is a promise that it will eventually resolve to the result of an asynchronous operation that it is tied to when that operation finishes.

```python
def is_server_up(server_address):
    future = asyncio.Future()
    def callback(f): return future.set_result(f.result() is not None)
    result = asyncio.ensure_future(ping_server(server_address))
    result.add_done_callback(callback)
    return future
```

Listing 2.2: *asyncio* code snippet to demonstrate *Future*

2.3 LINEAR ALGEBRA

2.3.1 Polynomial operations

For this section, let us assume that we have a polynomial $f(x) : \mathbb{F}_p \to \mathbb{F}_p$ of the form:

$$f(x) = a_0 + a_1x + \cdots + a_nx^n \tag{2.1}$$

Here, $\mathbb{F}_p$ is a Finite Field of size p where p is a prime.

1. Interpolation

   Given a set of $n+1$ points (x_i, y_i), where $x_i, y_i \in \mathbb{F}_p$ and all x_i are distinct, interpolation is defined as the process of finding a polynomial $g(x) : \mathbb{F}_p \to \mathbb{F}_p$ of degree at most n such that:

$$g(x_i) = y_i \, \forall \, i \in [0, n] \tag{2.2}$$

2. Evaluation

   Evaluation of a polynomial at a point k refers to solving $f(x)$ for k:

$$f(k) = a_0 + a_1k + \cdots + a_nk^n \tag{2.3}$$

2.3.2 Different algorithms for performing various polynomial operations

1. Lagrange Interpolation:

   The Lagrange interpolating polynomial [34] is the polynomial $P(x) : \mathbb{F}_p \to \mathbb{F}_p$ of *degree* $<=$ (n) that passes through the $n+1$ points $(x_0, y_0 = f(x_0)), \ldots, (x_n, y_n = f(x_n))$, and is given by

$$P(x) = \sum_{j=0}^{n} P_j(x) \tag{2.4}$$

where

$$P_j(x) = y_j \prod_{\substack{k=0 \\ k\neq j}}^{n} \frac{x - x_k}{x_j - x_k} \tag{2.5}$$

written explicitly as

$$P(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)} y_0 + \cdots + \frac{(x-x_0)(x-x_1)\cdots(x-x_{n-1})}{(x_n-x_0)(x_n-x_1)\cdots(x_n-x_{n-1})} y_n \tag{2.6}$$

2. Horner's rule:

   Horner's rule [35] states that the polynomial $f(x)$ can be evaluated at a point k by solving:

   $$f(k) = a_0 + k\Big(a_1 + k(a_2 + k(a_3 + \cdots + x(a_{n-1} + k\, a_n)\cdots))\Big) \tag{2.7}$$

   This allows evaluation of a polynomial of degree n with only n multiplications and n additions.

3. Vandermonde Matrix:

   If we have the points $(x_0, y_0), \ldots, (x_n, y_n)$ where $x_i, y_i \in \mathbb{F}_p$ such that $f(x_i) = y_i \,\forall\, i \in [0, n]$, then we can interpret this as a system of linear equations in the coefficients a_k. This system of equations in the matrix form is as follows:

$$\begin{bmatrix} x_0^n & x_0^{n-1} & x_0^{n-2} & \ldots & x_0 & 1 \\ x_1^n & x_1^{n-1} & x_1^{n-2} & \ldots & x_1 & 1 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ x_n^n & x_n^{n-1} & x_n^{n-2} & \ldots & x_n & 1 \end{bmatrix} \begin{bmatrix} a_n \\ a_{n-1} \\ \vdots \\ a_0 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix} \tag{2.8}$$

or

$$V\vec{a} = \vec{y} \tag{2.9}$$

Here, V is the Vandermonde Matrix [36].

Given points $(x_0, y_0), \ldots, (x_n, y_n)$, interpolation using the Vandermonde Matrix [37] is the process of finding the coefficients a_k of $f(x)$ which is the same as solving Eq. (2.9) for a:

$$\vec{a} = V^{-1}\vec{y} \tag{2.10}$$

An interesting observation here is that if we wish to find $k+1$ interpolating polynomials $f_0(x), f_1(x), \ldots, f_k(x)$ with the same form as $f(x)$ defined above from the points:

$$\{(x_{00}, y_{00}), \ldots, (x_{0n}, y_{0n})\}, \ldots, \{(x_{k0}, y_{k0}), \ldots, (x_{kn}, y_{kn})\}$$

then we have to compute V^{-1} only once and we can rewrite Eq. (2.10) as:

$$\vec{A} = V^{-1}\vec{Y} \tag{2.11}$$

where $\vec{A}$ is an $n \times k$ matrix denoting the coefficients of $f_j(x) \, \forall \, j \in [0, k]$ and $\vec{Y}$ is an $n \times k$ matrix denoting the evaluations of $f_j(x_i) \, \forall \, j \in [0, k]$ and $\forall \, i \in [0, n]$.

NOTE: This process of finding V^{-1} only once and interpolating multiple polynomials is faster than performing $k+1$ interpolations using Lagrange's method. This is because after computing V^{-1}, finding the interpolating polynomial involves only multiplications and additions. The number of multiplications when compared with Lagrange's method is also less.

4. Fast Fourier Transform

   - Polynomial Evaluation: Let us suppose we have the coefficients $a_i \in \mathbb{F}_p \, \forall \, i \in [0, k-1]$ of a polynomial $g(x) : \mathbb{F}_p \to \mathbb{F}_p = \sum_{i=0}^{k-1} \alpha_i x^i$ and we want to evaluate $g(x)$ at any n points where $n = 2^r$. If we use Horner's rule described above in Item 2, then the algorithmic complexity of that operation is $\mathcal{O}(n^2)$.

     We can do better by using a Fast Fourier Transform in $\mathbb{F}_p$ as described in Algorithm 2.1 [38]. This way we can obtain $(g(\omega^0), \ldots, g(\omega^{n-1}))$ in $\mathcal{O}(n \, log(n))$ instead of $\mathcal{O}(n^2)$.

   - Polynomial Interpolation: Let us suppose we have the evaluations $g(\omega^i) \, \forall \, i$ in $[0, n-1]$ where $n = 2^r$ and we want to find the coefficients of the interpolating polynomial $g(x) : \mathbb{F}_p \to \mathbb{F}_p$. If we do this by using Lagrange's interpolation method then the algorithmic complexity will be $\mathcal{O}(n^2)$.

     We can do better by using an Inverse Fast Fourier Transform in $\mathbb{F}_p$ as described in Algorithm 2.2 [38] and bring down the complexity to $\mathcal{O}(n \, log(n))$.

Algorithm 2.1 Evaluating a polynomial using FFT

```
Input:
  α: coefficient vector $(\alpha_0, \ldots, \alpha_{k-1})$ such that $g(x) : \mathbb{F}_p \rightarrow \mathbb{F}_p = \sum_{i=0}^{k-1} \alpha_i x^i$
  n: $2^r$                          ▷ Number of evaluations must be a power of 2.
  ω: primitive $n^{th}$ root of unity i.e. $\omega^n = 1$
Output:
  Evaluation vector $(g(\omega^0), \ldots, g(\omega^{n-1}))$

  procedure FFT($\alpha, \omega, n$)
      $d \leftarrow n$
      if not is_power_of_two($d$) then
          $d \leftarrow$ nearest_power_of_two($d$)          ▷ Get the next higher power of 2
      $\alpha' \leftarrow \alpha + [0] * (d - len(\alpha))$          ▷ Pad with zeros to make a power of 2
      $Z \leftarrow$ FFTHelper($\alpha', \omega$)
      return $Z[: n]$          ▷ Return only first $n$ evaluations as were asked for

  procedure FFT Helper($\alpha, \omega$)
      if $len(\alpha) = 1$ then return $(\alpha_0)$
      $(\alpha_0, \alpha_1, \ldots, \alpha_{n/2-1}) \leftarrow$ FFT($(\alpha_0, \alpha_2, \ldots, \alpha_{n-2}), \omega^2$)
      $(\beta_0, \beta_1, \ldots, \beta_{n/2-1}) \leftarrow$ FFT($(\alpha_1, \alpha_3, \ldots, \alpha_{n-1}), \omega^2$)
      for $j$ in $[0, n-1]$ do
          $k \leftarrow j \mod n/2$
          $\gamma_j \leftarrow \alpha_k + \omega^j \beta_k$
      return $(\gamma_0, \ldots, \gamma_{n-1})$
```

- Polynomial Interpolation with only a subset of points: Let us suppose that we want to interpolate a degree t polynomial $f(x) : \mathbb{F}_p \rightarrow \mathbb{F}_p$ given evaluations of the polynomial at $t+1$ distinct points $(x_0, x_1, \ldots, x_t)$ where $x_i = \omega^j \, \forall \, i \in [0, t] \, \forall \, j \in [0, n-1]$. We can do this efficiently using FNT based Reed-Solomon decoding [39]. Given below is an outline of the algorithm:
  (a) Determine coefficients of polynomial $A(x) = \Pi_j (x - x_j)$. This can be done in $\mathcal{O}(t \, log^2 \, (t))$ time using a divide-and-conquer approach and FFT-based polynomial multiplication.
  (b) Differentiate polynomial $A(x)$ to obtain $A'(x)$. This can be trivially done in $\mathcal{O}(t)$ time.

Algorithm 2.2 Finding an interpolating polynomial using Inverse FFT

```
Input:
  γ: evaluation vector $(\gamma_0, \dots, \gamma_{n-1}) = (g(\omega^0), \dots, g(\omega^{n-1}))$
  ω: primitive $n^{th}$ root of unity i.e. $\omega^n = 1$
  n: $2^k$                              ▷ Length of the vector $\alpha$ must be a power of 2
Output:
  Coefficient vector $(\alpha_0, \dots, \alpha_{n-1})$ such that $g(x) : \mathbb{F}_p \to \mathbb{F}_p = \sum_{i=0}^{n-1} \alpha_i x^i$

  procedure INVERSE FFT($\gamma, \omega$)
      $\beta \leftarrow$ FFT($\gamma, 1/\omega$)
      for $j$ in $[0, n-1]$ do
          $\alpha_j \leftarrow \beta_j / n$
      return $(\alpha_0, \dots, \alpha_{n-1})$
```

(c) Evaluate $A'(x)$ on $(x_0, x_1, \dots, x_{k-1})$. Since $x_i = \omega^j \,\forall\, i \in [0, t] \,\forall\, j \in [0, n-1]$, we can evaluate $A'(x)$ at $(\omega^0, \dots, \omega^{n-1})$ and select the evaluations corresponding to $(x_0, \dots, x_{k-1})$. This can be done in $\mathcal{O}(n\, log(n))$ time.

(d) Set $n_i = y_i / A'(x_i)$. Evaluate the polynomial $N(m) = \Sigma_{j=0}^{n-1} n_j m^{z_j}$ at $\omega^{-r} \,\forall\, r \in [0, n-1]$ where $\omega^{z_j} = x_j$. This is equivalent to a single DFT and can be done in $\mathcal{O}(n\, log(n))$ time.

(e) Set polynomial $Q(x) = -\Sigma_{j=0}^{n-1} N(\omega^{-j-1}) x^j$. The interpolated polynomial $P(x)$ is given by $Q(x) * A(x)$. This can be done using FFT-based polynomial multiplication in $\mathcal{O}(n\, log(n))$ time.

5. Finding the interpolating polynomial with errors in data points

   If we have up to e erroneous points then we can use Gao's method [40] as described in Algorithm 2.3 for finding the interpolating polynomial from a set of at least $k + 2e + 1$ points where k is the degree of the interpolating polynomial.

2.4 CRYPTOGRAPHY

2.4.1 Shamir's Secret Sharing Scheme

In Shamir's Secret Sharing Scheme [41], the goal is to divide a secret S into n pieces $S_1, \dots, S_n$ such that S can be reconstructed from any k or more pieces. However, the knowledge of $k - 1$ or fewer pieces does not reveal anything about S. The pieces S_i are also referred to as *Shares* and this scheme is called (k, n) threshold scheme where $0 < k \leq n$.

For the (k, n) threshold scheme, the idea is to choose at random $k-1$ points $a_i \, \forall \, i \in [1, k-1]$ where $a_i \in \mathbb{F}_p$ and let $a_0 = S$. We then build a polynomial $f(x) = a_0 + a_1 x + \cdots + a_{k-1}x^{k-1}$. Lastly, we create the n shares by evaluating $f(x)$ at $i \, \forall \, i \in [1, n]$. Each of these shares can then be distributed to n different parties.

Algorithm 2.3 Gao's method for decoding Reed Solomon Codes

```
Input:
  A received vector $b = (b_1, b_2, ..., b_n) \in \mathbb{F}_p^n$ which comes from a codeword $c$ with $t$ errors
  where $t \leq (d-1)/2$.
Output:
  A message polynomial $m_1 + m_2 x + m_k x^{k-1}$, or "Decoding failure"

  procedure DECODING REED SOLOMON CODES $(b)$
      Step 1: (Interpolation) Find the unique polynomial $g_1(x) \in \mathbb{F}_p[x]$ of degree $\leq n-1$
  such that
                      $g_1(a_i) = b_i, 1 \leq i \leq n$
      Step 2: (Partial GCD) Apply the extended Euclidean algorithm to $g_0(x)$ and $g_1(x)$.
  Stop when the remainder, say $g(x)$, has degree $< \frac{1}{2}(n+k)$. Suppose we have at this time
                      $u(x)g_0(x) + v(x)g_1(x) = g(x)$
      Step 3: (Long division) Divide $g(x)$ by $v(x)$, say
                      $g(x) = f_1(x)v(x) + r(x),$
  where deg $r(x) <$ deg $v(x)$. If $r(x) = 0$ and $f_1(x)$ has degree $< k$ then output $f_1(x)$,
  otherwise output "Decoding failure" (which means that more than $(d-1)/2$ errors have
  occurred).
```

Reconstruction

Given a set of z shares S_i from a (k, n) threshold scheme where $k \leq z \leq n$, reconstruction is the process of computing S from S_i. We can do this easily by finding an interpolating polynomial $g(x)$ using the points (i, S_i) where $i \in [1, n]$ and then computing $g(0)$ to obtain the secret S.

NOTE: We can also employ FFT to construct the shares S_i using the coefficients a_i and use Inverse FFT to get the interpolating polynomial $g(x)$. We can then compute the secret S by evaluating $g(0)$.

Notation

We use $[x]_t$ to denote a Shamir Share of the secret x where t is the threshold.

2.4.2 Multi Party Computation

Given n parties $P_1, \ldots, P_n$ each of which have a private input $x_1, \ldots, x_n$. In a multi-party computation (MPC), the parties wish to jointly compute a function $y = f(x_1, \ldots, x_n)$ such that this computation must preserve the following security properties even if some of the parties collude and maliciously attack the protocol [42]:

1. Correctness: Parties obtain the correct output even if some parties demonstrate adversarial behaviour.
2. Privacy: Only the output is learned and nothing else.
3. Independence of inputs: Parties cannot choose their inputs as a function of other parties' inputs.
4. Fairness: If one party learns the output then all parties learn the output.
5. Guaranteed output delivery: All honest parties learn the output.

Operations on shares

- Linear Combination:
  We can compute a linear combination of the shares, $[x]_t$, $[y]_t$ and $[z]_t$ as follows:

  $$[m]_t = a \times [x]_t - b \times [y]_t + [z]_t / c$$

  Here a, b and c are elements in $\mathbb{F}_p$. This works because a linear combination of multiple degree t polynomials also results in a degree t polynomial.

- Multiplication:
  Unlike linear combination, multiplying two degree t polynomials results in a polynomial of degree $2t$. This requires us to have at least $2t + 1$ points in order to reconstruct the result of multiplication. We can multiply the shares using the following two techniques:

- We can use Beaver's Circuit Randomization technique [43]. If we have access to a set of shares $[a]_t$, $[b]_t$ and $[ab]_t$ such that $ab = a \times b$, then we can multiply two shares $[x]_t$ and $[y]_t$ as follows:

$$\begin{aligned} D &= \text{Reconstruct}([x]_t - [a]_t) \\ E &= \text{Reconstruct}([y]_t - [b]_t) \\ [xy]_t &= DE + D[b]_t + E[a]_t + [ab]_t \end{aligned} \tag{2.12}$$

  This involves a total of two reconstructions, three additions, three multiplications and three subtractions.

- We can also use a technique based on double shares. If we have access to a set of shares $[r]_t$ and $[r]_{2t}$ where r is a random element in $\mathbb{F}_p$, then we can multiply two shares $[x]_t$ and $[y]_t$ as follows:

$$\begin{aligned} [xy]_{2t} &= [x]_t \times [y]_t \\ D &= \text{Reconstruct}([xy]_{2t} - [r]_{2t}) \\ [xy]_t &= [r]_t + D \end{aligned} \tag{2.13}$$

  This requires only one reconstruction, one addition, one multiplication and one subtraction.

2.4.3 Robust Batch Reconstruction

A valid secret shared value $[x]_t$ can be trivially reconstructed by having every party broadcast their shares but this requires quadratic communication per share. We use the batch reconstruction technique from Choudhury and Patra [26] as described in Algorithm 2.4 to keep the overall communication complexity involved in reconstructing a batch of secret shared values to $\mathcal{O}(Nk)$ where k is the number of values to reconstruct and N is the total number of nodes.

In each round of the algorithm we also make use of robust decoding to tolerate potential corruptions or crashed parties. The main idea is to check that the reconstructed polynomial coincides with $2t+1$ received values, since $t+1$ of which must be honest and uniquely determine the correct degree t polynomial. We optimistically attempt to decode after receiving $2t+1$ values, but if this fails we fall back to Gao's method of Reed-Solomon decoding as additional shares arrive. We are able to leverage the speedup provided by FFT even when working with only a subset of all points because of the FNT-based Reed Solomon decoding technique described above.

Algorithm 2.4 Batch reconstruction

```
Let there be $N$ parties $P_1, \dots, P_N$. This protocol is executed at each party $P_i$.
Input: $[s_0]_t, \dots, [s_t]_t$
Output: $s_0, \dots, s_t$

  procedure BATCH OPEN( )
    Let $\phi(i, x) = \sum_{j=0}^{t} [s_j]_t^{(i)} x^j$. Evaluate $\phi(i, x)$ at $[1, N]$.
    (Round 1)
    for $j$ in $[1, N]$ do
      Send $\phi(i, j)$ to party $P_j$
    Wait to receive between $2t + 1$ and $N$ shares, robustly reconstructing $\phi(\cdot, i)$

    (Round 2)
    for $j$ in $[1, N]$ do
      Send $\phi(0, i)$ to each party
    Wait to receive between $2t + 1$ and $N$ shares, robustly reconstructing $\phi(0, \cdot)$

    for $j$ in $[1, t + 1]$ do
      $y_j \leftarrow j^{th}$ coefficient of $\phi(0, \cdot)$
    return $y$
```

2.4.4 HoneyBadgerAVSS - hbAVSS

Definition 2.1. Asynchronous Verifiable Secret Sharing (AVSS)
In an AVSS protocol, the dealer D receives input $s \in \mathbb{F}_p$, and each party P_i receives an output share $\phi(i)$ for some degree t polynomial $\phi : \mathbb{F}_p \to \mathbb{F}_p$. The protocol must satisfy the following properties:

- Correctness: If the dealer D is correct, then all correct parties eventually output a share $\phi(i)$ where ϕ is a random polynomial with $\phi(0) = s$.
- Secrecy: If the dealer D is correct, then the adversary learns no information about ϕ except for the shares of corrupted parties.
- Agreement: If any correct party receives output, then there exists a unique degree t polynomial ϕ' such that each correct party P_i eventually outputs $\phi'(i)$.

We use the *hbAVSS* protocol from [31], this protocol satisfies all the properties mentioned above and has an overall linear amortized communication overhead, linear in terms of the number of parties.

2.4.5 Asynchronous Common Subset

The Asynchronous Common Subset (ACS) primitive allows each party to propose a value, and guarantees that every party outputs a common vector containing the input values of at least $N - 2t$ correct parties where t is the threshold for the number of adversarial parties. We take the ACS implementation from HoneyBadgerBFT [44].

CHAPTER 3: HONEYBADGERMPC

3.1 SYSTEM MODEL

We consider an asynchronous distributed system model with a fixed set of N communicating parties $P_1, .., P_N$ connected to each other by pairwise authenticated channels. For a node[1] in this system, we assume that there are no bounds on execution of its processes, that its clock has an arbitrary drift rate, and that there can be an arbitrary and unbounded delay in the transmission of a message sent by it.

In addition, we assume the existence of a global adversary capable of monitoring all network communication, introducing arbitrary delays in the transmission of messages, and delivering messages out of order. Since we assume secure channels, the global adversary cannot inspect, modify or remove messages. We consider that any timeouts in the protocol will always expire and it is best if they are avoided.

Furthermore, we assume that a Byzantine threshold adversary is allowed to corrupt any t out of N nodes, such that $t < \frac{N}{3}$. This Byzantine adversary can also collude with the global adversary mentioned earlier. A node corrupted by the Byzantine adversary can behave arbitrarily. It can fail to respond to messages, respond with incorrect messages, or collude with other corrupt nodes to coordinate attacks. It is not possible to identify if a node has crashed, is slow or is simply delaying the messages because it is corrupt. We do not consider Sybil attacks, which could otherwise allow an attacker to gain control of a byzantine quorum of the network. Similarly, although the Byzantine adversary can send invalid messages to try to thwart the protocols, we consider the problem of an adversary trying to DOS the network by sending very large amounts of invalid data to be out of scope.

We assume all adversaries are computationally bounded and are unable to break cryptographic primitives.

Lastly, we assume the existence of an available and accurate public key infrastructure, which nodes can use to learn the public keys of other parties.

3.2 SYSTEM GOALS

One of the major contributions of this work is **HoneyBadgerMPC** – an MPC toolkit designed to achieve the following high-level goals for a corruption threshold of $t < N/3$:

1. Scalability

[1]The term *node* is used interchangeably with the term *party*.

We should be able to support a large number of parties (at least 100) and operate on large input sizes (at least 2^{12} field elements).

2. **Fault-tolerance**

   We should be able to successfully execute a protocol to completion even in the case when up to t nodes have crashed.

3. **Robustness**

   We should be able to successfully execute a protocol to completion as long as we have inputs from $N - t$ parties following the protocol. In other words, we should guarantee correct execution of the protocol even when up to t nodes have crashed or have been compromised.

4. **Confidentiality**

   We should never leak any information about any of the inputs even in the case when up to t nodes are corrupt.

5. **Asynchronous execution**

   MPC applications involve both computation and communication. This, coupled with the fact that we are operating in an asynchronous communication model where messages may be delayed arbitrarily, provides us with the perfect opportunity to overlap communication with computation as long as the two are independent.

   More specifically, we should be able to asynchronously trigger communication and continue further execution until we reach a point at which we need the output of previously triggered operations. Refer to Section 2.1.1 for a detailed explanation of asynchronous execution.

6. **Ease of programmability**

   We should provide an easily programmable interface to a developer interested in building MPC applications.

3.3 IMPLEMENTATION

HoneyBadgerMPC has been heavily inspired by VIFF [24]. This section goes over all the architectural decisions that have influenced the design of HoneyBadgerMPC.

3.3.1 Programming Language

HoneyBadgerMPC is implemented in the Python [33] programming language. The choice to pick Python was mainly influenced by the following factors:

- Need for a flexible language for rapid prototyping
- A syntax supported framework for writing concurrent programs
- Availability of an extensive set of packages providing various functionalities such as testing frameworks, benchmarking frameworks, implementation of cryptographic operations, modules to maintain code quality etc.

Python code is interpreted at runtime causing it to run slower when compared against code written in a language like C/C++ where the code is first compiled to native code and then executed. When building components which perform computationally intensive operations, HoneyBadgerMPC invokes compiled code written in either Rust or C++ to achieve the performance boost which Python cannot provide. Fig. 3.1 gives a perspective of the different languages which make up the HoneyBadgerMPC codebase.

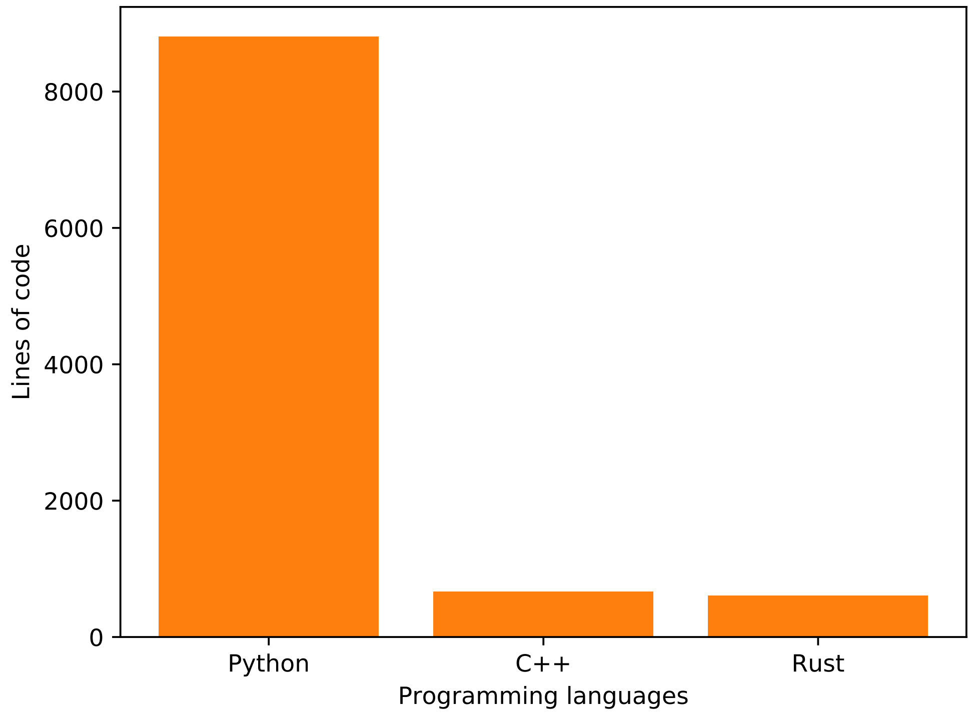


Figure 3.1: Programming languages used in HoneyBadgerMPC.

3.3.2 Asynchronous Framework

HoneyBadgerMPC is built on *asyncio* which is Python's standard framework for concurrent programming. A syntax supported concurrent framework makes development easier and results in less amount of more maintainable code. Refer to Section 2.2.1 for a brief primer on *asyncio*.

3.3.3 Finite Field

We use a finite field $\mathbb{F}_p$ where p is a prime of size 255 bits and,

$$p = \texttt{0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001}$$

p is same as the order of zk-SNARK's BLS12-381 curve [45]. We pick this particular modulus since we leverage FFT to perform multi-point polynomial evaluations and interpolations and in order to be able to that, $\mathbb{F}_p$ needs to be equipped with a large 2^r root of unity. BLS12-381 is a recent Elliptic Curve Construction with support for asymmetric pairing groups which allow us to perform elliptic curve operations with better performance.

3.3.4 Dataflow

We intend to deploy HoneyBadgerMPC in networks which are asynchronous by nature and as a consequence it can take an arbitrary amount of time for a message to reach from one node to another. asyncio's 'Future' class allows us to handle this asynchronous behaviour nicely.

In order to understand the flow of data within HoneyBadgerMPC, it is important to understand the following abstractions, each of which is implemented as a separate class:

- **GFElement**: Denotes an element within a Finite Field $\mathbb{F}_p$ where p is a prime. Every party has the same value.
- **Share**: Denotes a Shamir secret sharing [41] of a field element. Every party has a different value which corresponds to its own 'share' of the value.
- **ShareFuture**: Denotes a promise that this object will resolve to a 'Share' object when *awaited.*
- **GFElementFuture**: Denotes a promise that this object will resolve to a 'GFElement' object when *awaited.*

An ‘open()’ method can be invoked on a ‘Share’ or ‘ShareFuture’ object to publicly reconstruct the underlying secret represented by it. The return value of the ‘open()’ operation is a ‘GFElementFuture’ object which will eventually resolve to a ‘GFElement’ object once we *await* on the reconstruction to finish.

Listing 3.1 shows an example of how operations can be performed on these types without any synchronization unless required. Note the absence of *await* statements while performing multiplication of two ‘Share’ objects (lines 5 and 6, recall that a share multiplication requires at least one public reconstruction) or when *opening* a ‘Share’ (lines 7 and 8). The expression tree in Fig. 3.2 describes how this code is evaluated. A key takeaway from this expression tree is that X and Y are mutually independent and are evaluated in parallel i.e. while the evaluation of X waits on network communication from other parties, we allow Y to trigger its part of the communication. This is important for efficiency since depending on the amount of computation and communication performed within an MPC application, the communication may end up dominating the runtime.

```
1  async def mpc_prog(a, b, c, d):
2      """
3      a, b, c, d: Secret shared values of type Share.
4      """
5      x = a * b         # Share x Share => ShareFuture
6      y = c * d         # Share x Share => ShareFuture
7      X = x.open()      # Async open() of a Share => GFElementFuture
8      Y = y.open()      # Async open() of a Share => GFElementFuture
9      Z = X * Y         # GFElementFuture x GFElementFuture =>
      GFElementFuture
10     result = await Z  # Wait for result to be computed before returning
11     return result
```

Listing 3.1: HoneyBadgerMPC code snippet to demonstrate dataflow

Table 3.1 lists the relationships between various types within HoneyBadgerMPC and enlists all operations supported between them. In short:

- If a *Future* is returned then it needs to be *awaited* in order to retrieve the underlying value.

- *Future* always dominates i.e. performing operations on a *Future* always returns a *Future*.

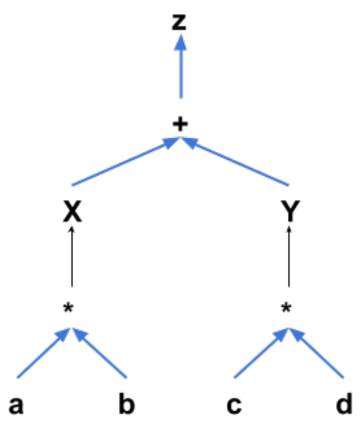


Figure 3.2: Expression tree for the code snippet in Listing 3.1.

- 'open()' always returns a 'GFElementFuture'.

The dataflow in HoneyBadgerMPC is inspired from VIFF which uses *callbacks* and *Defferds* to achieve the same. We wanted developers to have the flexibility of using different operations directly on secret shared values and field elements without putting any constraints on them to either cast or wait for values.

Operand 1 Type	**Operator**	**Operand 2 Type**	**Result Type**
GFElement	$+, -, \times, \div$	GFElement	GFElement
GFElement	$+, -, \times, \div$	int	GFElement
int	$+, -, \times, \div$	GFElement	GFElement
GFElementFuture	$+, -, \times, \div$	GFElement	GFElementFuture
GFElement	$+, -, \times, \div$	GFElementFuture	GFElementFuture
GFElementFuture	$+, -, \times, \div$	GFElementFuture	GFElementFuture
Share	$+, -$	Share	Share
Share	$\times$	Share	ShareFuture
ShareFuture	$+, -, \times$	Share	ShareFuture
Share	$+, -, \times$	ShareFuture	ShareFuture
ShareFuture	$+, -, \times$	ShareFuture	ShareFuture
GFElementFuture	$+, -, \times$	Share	ShareFuture
Share	$+, -, \times$	GFElementFuture	ShareFuture
GFElementFuture	$+, -, \times$	ShareFuture	ShareFuture
ShareFuture	$+, -, \times$	GFElementFuture	ShareFuture

Table 3.1: Summary of type relationships and operators

3.3.5 Preprocessing Phase

An MPC application involves two phases, the **preprocessing phase** and the **online phase**. The former is the one where preprocessed elements such as shares of multiplication triples, shares of random field elements, shares of random bits etc. are generated. Different kinds of MPC applications need one or more types of these preprocessed elements to perform operations on secret shared inputs. On the other hand, the online phase is the one where we compute a public function on the secret shared inputs to produce the output of MPC.

For a successful execution of an MPC application, we need to ensure that apart from conforming to our system goals, the preprocessing phase in its steady state never runs out of preprocessed elements i.e. a continuous supply of preprocessed elements is produced as they are being consumed by the online phase.

The number of preprocessed elements required by an MPC application is usually a function of the number of its inputs. Since *scalability* in the dimension of the number of inputs is one of our goals, we need to be able to generate these preprocessed elements as efficiently as possible.

Preprocessing in HoneyBadgerMPC

HoneyBadgerMPC uses hbAVSS (Section 2.4.4) to AVSS a batch of preprocessed elements, post that it uses ACS (Section 2.4.5) to agree on a batch of these unrefined preprocessed elements. Lastly, we leverage FFT to refine the agreed preprocessed elements.

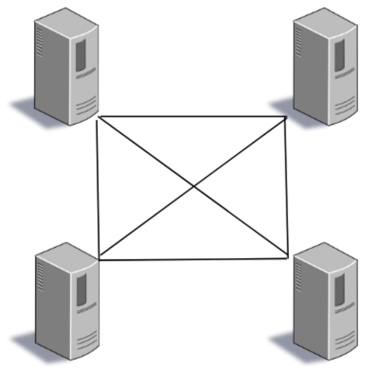
For the ease of explanation, let us say that we want to generate shares of random field elements in the preprocessing phase. The process is as follow:

1. **AVSS:**

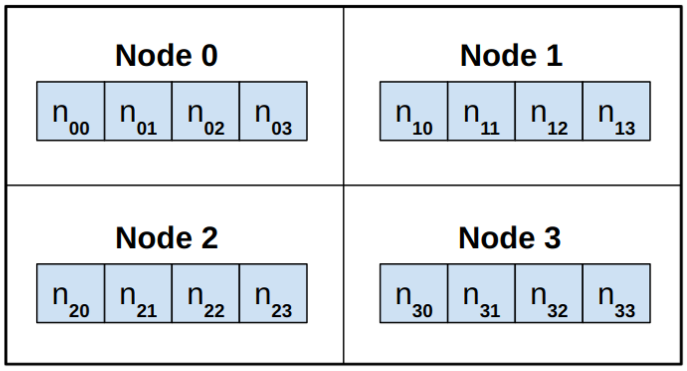
   (a) Each node P_i generates a batch b of random field elements. It then AVSSes these random field elements to all other nodes using hbAVSS. In this process, P_i retains its shares of the AVSSed values after they have been verified. P_i only waits for its own AVSS instance to finish and must not wait for AVSS instances started by other parties since they might never finish.

   (b) Each node P_i also maintains a list of count of shares of AVSSed values that it has received from each dealer P_j.

   This process is illustrated in Fig. 3.3.



(a) Each node AVSSes a batch of random field values using hbAVSS.



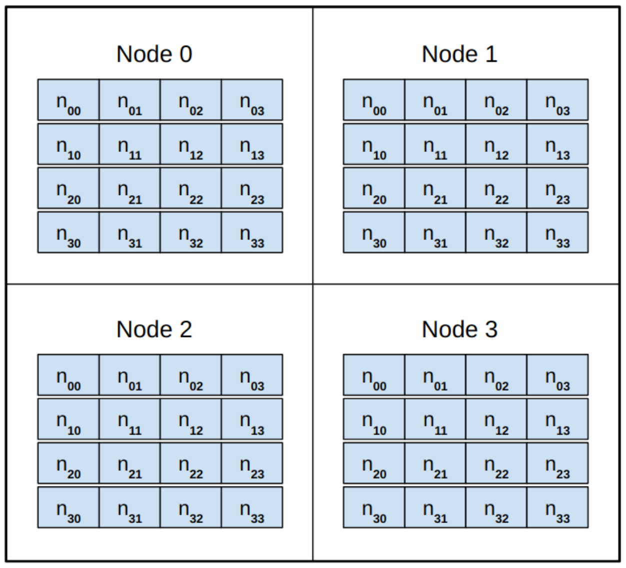
Node 0
n_{00} n_{01} n_{02} n_{03}
Node 1
n_{10} n_{11} n_{12} n_{13}
Node 2
n_{20} n_{21} n_{22} n_{23}
Node 3
n_{30} n_{31} n_{32} n_{33}

(b) Each node maintains a count of shares of AVSSed values it has received per dealer. n_{ij} is the number of AVSSed values P_i has received which have been dealt by P_j.

Figure 3.3: An example with $N = 4$ demonstrating Step 1

2. **ACS:**

   At any instant, it is possible that the list from 1.b) has different counts at some/all the nodes since they might receive the AVSS acknowledgements in different order. Thus, after every few AVSSes each node P_i needs to run an instance of ACS to agree on the count of values obtained from Step 1.b).



Node 0
Node 1
Node 2
Node 3
n_{00} n_{01} n_{02} n_{03}
n_{10} n_{11} n_{12} n_{13}
n_{20} n_{21} n_{22} n_{23}
n_{30} n_{31} n_{32} n_{33}

Figure 3.4: After ACS each node gets to know the count of AVSSed values received by all other nodes

3. **Processing of agreed values:**

(a) After ACS, each node has the same view of the matrix (Fig. 3.4).

(b) We take the transpose of the matrix such that cell $[i, j]$ now indicates the number of values received by P_j dealt by P_i. Essentially, each row i is now P_i's view of the count of values dealt by itself that all the nodes have received.

(c) We then find z_i, the t^{th} largest element (0 based index), where t is the corruption threshold in each row of the matrix. This implies that $t + 1$ nodes have seen at least z_i AVSSed values dealt by P_i. This gives us the confidence that now we can process z_i values, since if even one honest node has confirmed successful receipt of a batch of AVSSed values then all nodes will eventually receive their shares of that particular batch.

Fig. 3.5 illustrates this step.

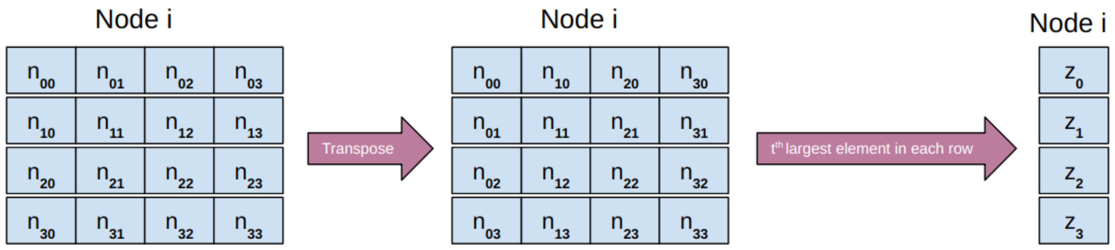

Node i
n_{00} n_{01} n_{02} n_{03}
n_{10} n_{11} n_{12} n_{13}
n_{20} n_{21} n_{22} n_{23}
n_{30} n_{31} n_{32} n_{33}
Transpose
Node i
n_{00} n_{10} n_{20} n_{30}
n_{01} n_{11} n_{21} n_{31}
n_{02} n_{12} n_{22} n_{32}
n_{03} n_{13} n_{23} n_{33}
t^{th} largest element in each row
Node i
z_0
z_1
z_2
z_3

Figure 3.5: Step 3

Triple generation: The process of generating unrefined shares of triples follows the same steps except for minor changes listed below:

(a) The input batch for AVSS now consists of multiple sets of three field elements a, b, and ab such that $ab = a \times b$.

(b) While processing the agreed values, we must ensure that the set of agreed values always includes a complete triple. This is just a note and in principle works similar to the previous case since with hbAVSS we always receive the entire batch and never receive only some of the values from a batch.

4. **Refinement:**

Up until this step, we have agreed on a set of AVSSed values at all nodes. However, some of these values may have been AVSSed by corrupt nodes and thus we need to *refine* them. We refine values in a batch such that each batch contains values dealt by at least $N - t$ different nodes, Fig. 3.6 describes the process of selecting a batch. Note that this batch comprises of values from multiple nodes whereas when we refer to a batch of AVSSed values, those correspond to a batch of values dealt by a single node. The process of refining a batch involves evaluating the polynomial represented by the values in a batch on a new set of points.

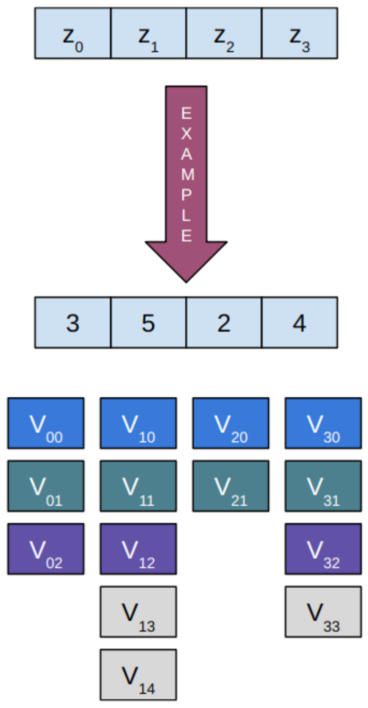


Figure 3.6: All nodes have the same view of z_i. V_{ij} represents the share of j^{th} value dealt by P_i. All the values with the same color are refined together. The values colored grey cannot be refined yet since for each refinement we need values dealt from at least $N - t$ different nodes.

The entire refinement process relies on the assumption that two batches of AVSSed values dealt by the same dealer will arrive at all the parties in the same order. This is enforced by associating a per dealer counter with each batch. This counter is incremented at the dealer after each batch is AVSSed. If any batch from a particular dealer arrives out of order at any of the parties, then we buffer it and wait for all previous batches to arrive before processing it.

Random Refinement

It is possible for a node to have crashed and as a result it can no longer AVSS more values. We can tolerate up to f such crash faults where $0 \leq f \leq t$. We will continue to proceed by

Algorithm 3.1 FFT based interpolation and extrapolation

FFT Interp Extrap takes a set of points Y and ω as input such that $\omega^{2\times len(Y)} = 1$. It treats Y such that $Y_i = f(\omega^{2i})\,\forall\, i \in [0, len(Y) - 1]$. It returns a set of points Z such that $Z_i = f(\omega^{i})\,\forall i\ \in [0, (2 \times len(Y)) - 1]$.

```
procedure FFT Interp Extrap($Y, \omega$)
    $n \leftarrow len(Y)$
    $Y' \leftarrow$ Inverse FFT($Y, \omega^2$)
    $Y'' \leftarrow$ FFT($Y', \omega, 2n$)
    return $Y''$
```

agreeing on values from all nodes which are currently online and are able to AVSS values. As a result, an input batch for random refinement can contain unrefined shares from anywhere between $N - f$ to N nodes.

Algorithm 3.2 FFT based random refinement

Random Refinement takes a set of points Y. Let $b \leftarrow len(Y)$, then each of the b points in Y come from a different party and $b \in [N - t, N]$. It treats Y such that $Y_i = f(\omega^{2i})\,\forall\, i \in [0, b - 1]$ and returns a set of $b - t$ points Z such that $Z_i = f(\omega^{2i+1})\,\forall\, i \in [0, b - t - 1]$.

```
procedure Random Refinement($Y$, N, t)
    $b \leftarrow len(Y)$
    if not is_power_of_two($b$) then
        $b \leftarrow$ nearest_power_of_two(b)          ▷ Get the next higher power of 2
    $\omega \leftarrow$ get_omega($2 \times b$)          ▷ Get $(2b)^{th}$ root of unity i.e. $\omega^{2b} = 1$
    $Y' \leftarrow Y + [0] * (b - len(Y))$          ▷ Pad with zeros to make a power of 2
    $Z' \leftarrow$ FFT Interp Extrap($Y', \omega$)          ▷ Algorithm 3.1
    return $Z'[1 : 2 \times (len(Y) - t) : 2]$          ▷ Take only first $len(Y) - t$ odd points
```

Yield: In the worst case, this input batch can contain values from all t corrupt nodes, thus we can output only $N - f - t$ refined shares. If we output more that $N - f - t$ refined shares, then after consuming $N - f - t$ random shares, the corrupt parties have enough points to define the polynomial and determine the shares of all other parties. Algorithm 3.2 describes the process of random refinement.

Algorithmic Complexity: The computational complexity per node is $\mathcal{O}(k\, log(k))$ where k is the nearest power of 2 for a batch of size m such that $m \in [N - f, N]$. The refinement of random shares does not involve any communication since the only operations that we are doing are polynomial interpolation and evaluation on the shares which we have already received.

Comparison with other refinement techniques: HyperMPC [46] employs hyper-invertible matrices for generating random double-sharings. First, every party P_i selects and double-shares a random value s_i. Then, the parties compute double-sharings of the values r_i, defined as $(r_1, \ldots, r_n) = M(s_1, \ldots, s_N)$, where M is a hyper-invertible N-by-N matrix. Then, $2t$ of the resulting double-sharings are reconstructed, each towards a different party, who verify the correctness of the double-sharings. The remaining $N - 2t$ double-sharings are outputted.

Our random refinement process can also be realized using a hyper-invertible matrix as follows:

$$M \equiv \{\lambda\}_{\substack{i=1,\ldots,N-f \\ j=1,\ldots,N}} \text{with } \lambda_{i,j} = \prod_{\substack{k=1 \\ k!=j}}^{c} \frac{\omega^{2i+1} - \omega^{2k}}{\omega^{2j} - \omega^{2k}} \tag{3.1}$$

HyperMPC is non-robust, they use the hyper-invertible matrix not just for extraction of random values but also for checking the degree of the polynomial which if it fails causes them to abort. On the other hand, we rely on AVSS to guarantee the correctness of shares. Also, instead of doing a matrix multiplication, we use the FFT-based polynomial interpolation and evaluation. This reduces the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N\,log(N))$.

Apart from hyper-invertible matrices VIFF [24] also supports PRSS based generation of shares. This does not scale with the number of parties since the local computation is exponential in N. Fig. 3.7 illustrates how PRSS compares to hyper-invertible matrices for generating a single 32-bit multiplication triple in VIFF.

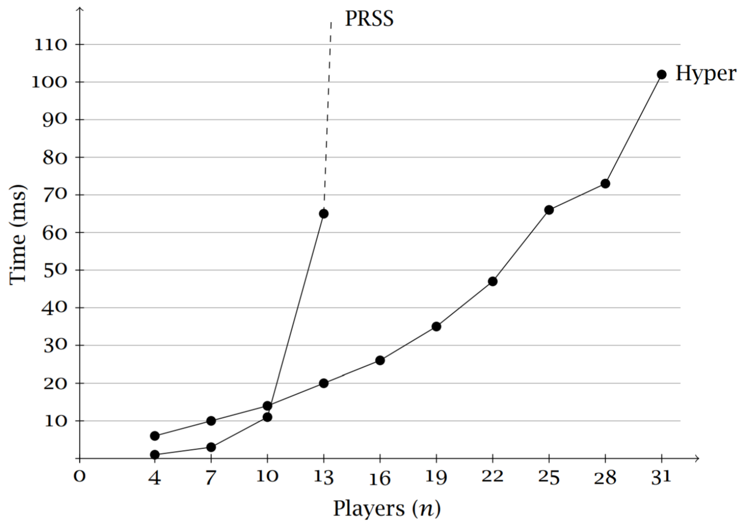


Figure 3.7: Preprocessing time needed to generate a single 32-bit multiplication triple

Triple refinement

In order to refine shares of triples A, B, and AB contributed by $N - f$ parties where $0 \leq f \leq t$ and f denotes the number of crash faults, we use the first $d + 1$ points to define all of $A()$ and $B()$, both of which are degree d polynomials, and some of $AB() = A() \times B()$ which is a degree $2d$ polynomial, here $d = 2(N - f) + 1$. We then evaluate $A()$ and $B()$ on a different set of points and multiply them in order to get more points on $AB()$ until we have enough points to fully define $AB()$. The refined triples are those points on $A(), B()$ and $C()$ that have not been used to define any of $A(), B()$ or $AB()$ and those which have not been revealed yet. Algorithm 3.3 describes how we achieve this using FFT.

Algorithm 3.3 FFT based triple refinement

```
Input:
  $N \leftarrow$ Total number of nodes
  $t \leftarrow$ Corruption threshold
  $A, B, AB \leftarrow [a_0]_t, \ldots, [a_{m-1}]_t, [b_0]_t, \ldots, [b_{m-1}]_t, [ab_0]_t, \ldots, [ab_{m-1}]_t$
  Such that $a_i \times b_i = ab_i \, \forall \, i \in [0, m-1]$ and $m \in [N-t, N]$

  procedure TRIPLE REFINEMENT($A$, $B$, $AB$, $m$, $N$, $t$)
      $d \leftarrow (m-1)/2$                                                  ▷ Let $d = 2m + 1$
      $A', B', AB' \leftarrow A[: (d+1)], B[: (d+1), AB[: (d+1)]$          ▷ First $d + 1$ values
      $X', Y', XY' \leftarrow A[(d+1) :], B[(d+1) :], AB[(d+1) :]$            ▷ Last $d$ values
      $A'' \leftarrow$ FFT Interp Extrap(FFT Interp Extrap($A', \omega^2$), $\omega$)
      $B'' \leftarrow$ FFT Interp Extrap(FFT Interp Extrap($B', \omega^2$), $\omega$)
      $P, Q \leftarrow A''[1 :: 2], B''[1 :: 2]$                             ▷ Get all values at odd indices
      $\alpha, \beta \leftarrow A''[0 :: 2], B''[0 :: 2]$                    ▷ Get all values at even indices
      $AB'' \leftarrow$ Batch Beaver($\alpha, \beta, X', Y', XY'$)    ▷ Multiply shares of $\alpha, \beta$ using $X', Y', XY'$
      $\gamma \leftarrow AB'_0, AB''_0, AB'_1, AB''_1, AB'_2, AB''_2, \ldots$
      $PQ \leftarrow$ FFT Interp Extrap($\gamma, \omega$)
      $k \leftarrow (m - 2t + 1)/2$
      return $P[: k], Q[: k], PQ[: k]$                                       ▷ Return only first $k$ points
```

Yield: Let $m = N - f$ and $d = 2m + 1$. We use the first $d + 1$ points to define $A(), B()$ and $AB()$ and now we are left with only the last d points which can be securely extracted until $A()$ and $B()$ can be fully defined. Out of these d points, in the worst case t of these could have been contributed by adversarial nodes which means we can extract only $d - t + 1$ points. When $d = t$, we still get one triple after which the polynomial is revealed. Thus, from m input triples we get $d - t + 1$ refined triples. In other words, from $N - f$ input triples we get $(N - f - 2t + 1)/2$ triples.

Algorithmic Complexity: The computational complexity per node per batch is similar to the Random Refinement process, $\mathcal{O}(k\,(log(k))$ where k is the nearest power of 2 for a batch of size m such that $m \in [N-f, N]$. The triple refinement process involves multiplication of shares using Beaver's method which involves two share reconstructions per multiplication. Thus, the communication complexity per node per batch is $\mathcal{O}(N)$.

Refining triples via Random Refinement

We can also refine triples using a $2t-$share based Random Refinement approach. In this method, each party t-shares a and b and double shares r (t and $2t$ shares). These shares are then refined using the Random Refinement technique described above. Each party then computes:

$$\begin{aligned} [ab]_{2t} &= [a]_t \times [b]_t \\ [ab]_t &= \text{Reconstruct}([ab]_{2t} - [r]_{2t}) + [r]_t \end{aligned} \tag{3.2}$$

$[a]_t$, $[b]_t$, and $[ab]_t$ then denote the shares of one refined triple.

In order to robustly interpolate a degree $2t$ polynomial, we need at least $4t+1$ points which is not possible at $N = 3t+1$. Thus, when using double shares at $t < N/3$ we can no longer be robust in the presence of faults. If we still want robustness, we have the option to reduce our threshold to $t < N/4$ in order to stay robust.

The yield and the algorithmic complexity of this approach are exactly similar to that of a $t-$share based Random Refinement.

3.3.6 Online Phase

As described earlier, in the online phase we evaluate a public function on secret shared inputs. As of this writing, HoneyBadgerMPC supports the following functionalities:

1. **Linear combination:** A linear combination of *Shares* $[x]_t$, $[y]_t$ and $[z]_t$ using coefficients p, q and r can be computed in HoneyBadgerMPC by:

```
w = p*x + q*y - r*z
```

   Here x, y, z, and w are of type *Share*.

2. **Multiplication:** Two shares $[x]_t$ and $[y]_t$ can be multiplied in HoneyBadgerMPC by:

```
xy = x*y
```

Here x and y are of type *Share* and xy is of type *ShareFuture* and thus needs to be awaited.

Mixins:

HoneyBadgerMPC supports mixins to provide additional feature plugins. We support the two ways of share multiplication as described in Section 2.4.2, each of which is implemented as a mixin. These can be passed when creating a 'ProgramRunner' (described later in Section 3.4.3).

Public reconstruction: Throughout HoneyBadgerMPC we use Shamir's secret sharing technique to encode a secret into a set of shares which are then distributed to all parties using hbAVSS. Public reconstruction is the reverse process of revealing the underlying secret encoded by a set of secret shares. This is implemented as the 'open' method in HoneyBadgerMPC. The following steps describe the interpolation process to obtain a degree t polynomial $f(x)$ which can then be evaluated at $f(0)$ to reconstruct the secret:

1. Wait for $t + 1$ points.
2. Interpolate a degree t polynomial $f(x)$ using the FNT-based algorithm as described in Section 2.3.2.
3. Evaluate $f(x)$ at all N points to compute the shares that you expect to receive from other parties.
4. As more points arrive, compare them with the corresponding expected shares.
5. If a total of $2t + 1$ points match with the expected shares, then we can be sure that there are no errors. We can then return $f(x)$.
6. If any of the $2t + 1$ points fail to match with its corresponding expected share then we invoke Gao's algorithm (Section 2.3.2) which tells us if there's a polynomial which passes through those points and the number of erroneous points which do not lie on that polynomial. We repeat this process as new points arrive until the number of non-erroneous points is $\geq 2t + 1$ and then return the polynomial passing through $2t + 1$ points.

ShareArray

MPC applications most often involve reconstruction of multiple secrets. We use the notion of a 'ShareArray' to represent a list of shares. If we wish to reconstruct a list of k shares then HoneyBadgerMPC employs the Robust Batch Reconstruction technique as described in Section 2.4.3. We use the same interpolation technique as the one described above for a single share reconstruction during the robust reconstruction process.

'ShareArrays' are also supported along with the Mixins described earlier in Section 3.3.6 to support the two variants of multiplications.

3.4 MISCELLANEOUS

3.4.1 Code optimizations

As mentioned briefly in Section 3.3.1, we have implemented certain components in different languages in order to get better performance.

- **Polynomial operations:** Public reconstruction is a building block of most MPC applications and it relies heavily on polynomial operations such as interpolation and evaluation. In order to optimize for the common case, we implemented different polynomial operations using various algorithms in C++. These include Lagrange based interpolation, Vandermonde matrix based interpolation and evaluation, FFT based interpolation and evaluation, and Gao's robust interpolation. These algorithms have been implemented using the NTL library [47] and are invoked directly from Python via Cython [48].

- **Pairing based cryptography:** hbAVSS relies heavily on pairing based cryptography for creating and verifying polynomial commitments. We create Rust [49] bindings for the zkcrpto/pairing [50] library and build a Python wrapper to call it directly from Python.

3.4.2 Development Environment

One of our design goals is the ease of programmability. We do not want the developers building MPC applications to suffer from debugging unrelated issues such as installing various toolkit dependencies, dealing with incompatible library versions, solving cross-platform/compatibility problems etc. As a result, we have containerized the entire

HoneyBadgerMPC codebase using Docker [51]. This essentially bundles all the dependencies with their correct versions in a neat package freeing the application developer from dealing with any issues in setting up the development environment.

3.4.3 Execution modes

HoneyBadgerMPC supports two execution modes:

1. Task-based: This is for rapid prototyping. This mode creates a mock router which routes the messages between various parties. It allows the developer to swiftly build and test the correctness of MPC applications without worrying about any network-related issues such as packet loss, connection management, network congestion etc.

2. Process-based: After optimizing and testing the application in the task-based mode, HoneyBadgerMPC offers a process-based mode allowing the developers to test their applications by running each party as an independent process. This gives the developers additional confidence since they can ensure that their application works as expected when receiving inputs via socket communication. Each process or party reads its configuration settings such as the IP addresses and port numbers of other parties from a configuration file. The parties then use this information to communicate with each other.

   The ideal deployment of an MPC application is that each party runs on a different server. Now to deploy onto multiple servers, the developer has to simply take the application written for the process-based mode, change the IP and ports within the configuration file and start the application processes on different servers.

3.4.4 Test suite

We believe in heavily testing the implementation of any algorithm or protocol. As of this writing, HoneyBadgerMPC has **178 test cases** which cover **74%** of the codebase. The tests are written using the *pytest* [52] framework. The availability of a testing framework and sample code to test an MPC application will allow the developers to automate the testing of their applications.

For developers who are interested in contributing to HoneyBadgerMPC, we have a Continuous Integration pipieline setup using Travis CI [53] which validates any new pull requests to ensure that all the tests pass and that the new code follows the coding practices incorporated throughout the rest of the codebase. Our code is available at `https://github.com/initc3/HoneyBadgerMPC`.

3.4.5 Amazon Web Services

HoneyBadgerMPC has a seamless integration with Amazon Web Services (AWS) [54]. This provides the application developers the capability to test, deploy and benchmark their applications on AWS using multiple servers launched in different regions.

CHAPTER 4: EXAMPLE PROGRAM

Listing 4.1 describes a complete HoneyBadgerMPC program. The program is written from a single party's perspective and all parties execute the same code. The *get_random_share* method provides an infinite supply of preprocessed random shares. When this method is called, each party receives a different value which is its share of a random element.

The *mpc_prog* collects 1000 random shares and then reconstructs all of them at once by invoking a *ShareArray.open()*.

This program runs in the process mode i.e. each party runs in a separate process. This program also demonstrates the use of the configuration file.

Listing 4.1: HoneyBadgerMPC example program

```
import asyncio
from honeybadgermpc.config import HbmpcConfig
from honeybadgermpc.ipc import ProcessProgramRunner
from honeybadgermpc.preprocessing import RandomGenerator


async def get_random_share(n, t, my_id, send, recv):
    """
    This method provides an unlimited supply of preprocessed random shares
    .
    """
    with RandomGenerator(n, t, my_id, send, recv) as random_generator:
        while True:
            yield await random_generator.get()


async def mpc_prog(context, randoms):
    """
    This is an MPC program which creates a ShareArray
    from 1000 random shares and then reconstructs them.

    This program is written from a particular party's
    pers
    """
    k, i = 1000, 0
    shares = [None] * k
```

Listing 4.1 (cont.)

```
    async for random in randoms:
        shares[i] = random
        i += 1
        if i == k:
            break
    print(await context.ShareArray(shares).open())
    await randoms.aclose()


async def run(peers, n, t, my_id):
    program_runner = ProcessProgramRunner(peers, n, t, my_id)
    await program_runner.start()
    send, recv = program_runner.get_send_and_recv("random_generator")

    program_runner.add(
        "mpc",
        mpc_prog,
        randoms=get_random_share(n, t, my_id, send, recv),
    )
    await program_runner.join()
    await program_runner.close()


if __name__ == "__main__":
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(
        run(
            HbmpcConfig.peers,
            HbmpcConfig.N,
            HbmpcConfig.t,
            HbmpcConfig.my_id,
        )
    )
```

CHAPTER 5: EVALUATION

In this section, we evaluate the performance of various algorithms and protocols within HoneyBagerMPC.

5.1 EXPERIMENTAL SETUP

- We deployed HoneyBadgerMPC on Amazon EC2 [55].
- We used *c5.xlarge* instances for all our distributed benchmarks. Each instance had the following specifications:
  - **CPU:** Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz
  - **Memory:** 8 GiB
  - **Cores:** 4
- For all our local benchmarks, we used a *c5.large* instance with the following specifications:
  - **CPU:** Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz
  - **Memory:** 4 GiB
  - **Cores:** 2
- We used a finite field $\mathbb{F}_p$ where:

  $p =$ `0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001`

  The size of p is 255 bits.
- We repeated each experiment three times and report the average of the three runs.
- All experiments were conducted in the *us-east-1* region unless specified otherwise. The ping latency and the bandwidth between two servers was measured to be **0.1 ms** and **9.6 Gbits/sec** respectively.
- All algorithms involving polynomial interpolation and evaluation were implemented using the FFT based techniques described in Section 2.3.2.

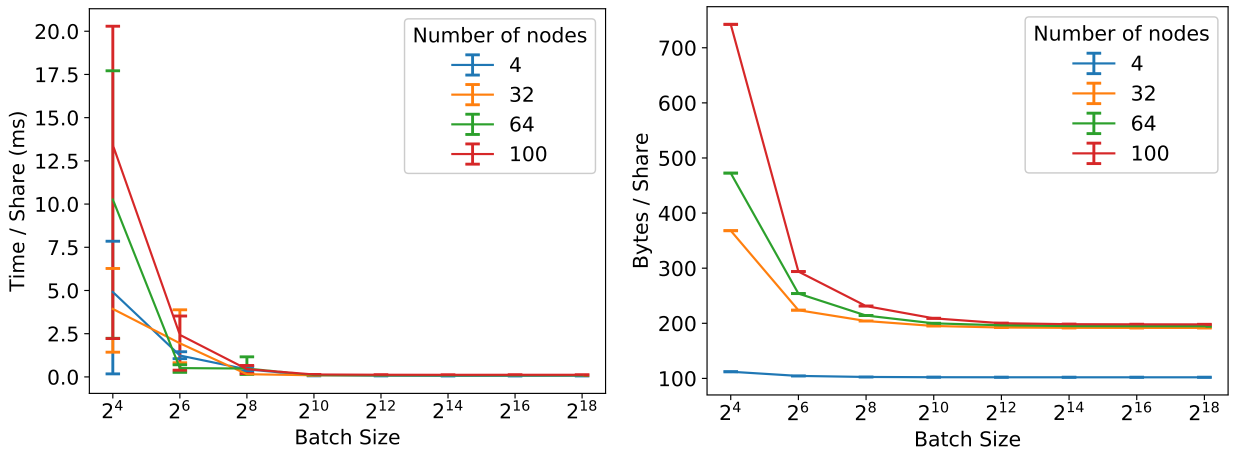
- We use *pickle*, Python's default serialization protocol, for sending a share to another node. A single serialized share value has a size of ≈ 34 bytes.
- We use $t = \lfloor (N-1)/3 \rfloor$ for the $t < N/3$ setting and $t = \lfloor (N-1)/4 \rfloor$ for the $t < N/4$ setting.

5.2 ONLINE PHASE: ROBUST BATCH RECONSTRUCTION

The Robust Batch Reconstruction algorithm described in Section 2.4.3 has a communication complexity of $\mathcal{O}(k)$ per node where k is the batch size. We evaluate our implementation in the following two settings.

5.2.1 Single AWS Region

We demonstrate that HoneyBadgerMPC can successfully reconstruct a batch of $\mathbf{2^{18}}$= **262144 shares** on **100 nodes** in **32.135 seconds** i.e. $\approx$ **8154 shares/second** with a communication cost of $\approx$ **198 bytes/share** which is an overhead of $6\times$. Fig. 5.1 and Table 5.1 describe the cost of batch reconstruction as the number of nodes and the batch sizes are varied in the same AWS region.



(a) Computation cost per node (b) Communication cost per node

Figure 5.1: Batch reconstruction costs in the *us-east-1* region

N	**2^4**	**2^6**	**2^8**	**2^{10}**	**2^{12}**	**2^{14}**	**2^{16}**	**2^{18}**
4	79	80	112	86	268	984	3993	16509
32	63	125	41	97	385	1515	6138	24288
64	165	33	124	111	399	1533	6295	25362
100	215	156	115	144	512	1980	8035	32136

Table 5.1: Time in milliseconds to reconstruct batches of different sizes across varying number of nodes in the *us-east-1* region.

5.2.2 Multiple AWS Regions

Setup

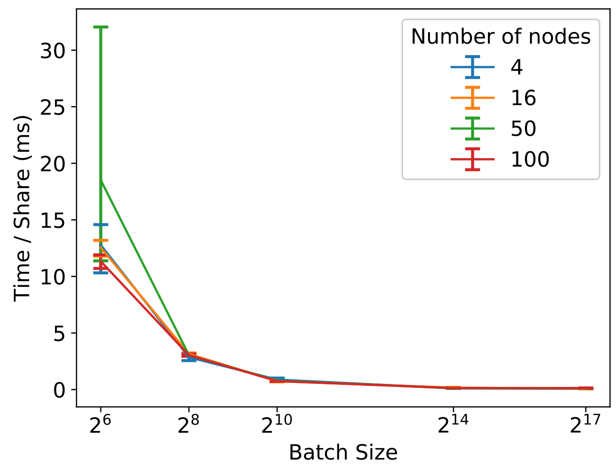
In the multi-region setting, the minimum and maximum ping latency between any two servers was 0.327 ms and 328 ms respectively and the minimum and maximum bandwidth between two servers was 15 Mbits/sec and 968 Mbits/sec respectively. We used the *t2.medium* instance type for these experiments. Table 5.2 describes the configuration for this experiment.

N	**Configuration**
4	1: us-east-1, 1: ap-south-1, 1: ap-northeast-1, 1: sa-east-1
16	2: us-east-1, 1: us-east-2, 2: us-west-1, 1: eu-central-1, 2: ap-northeast-1, 1: ap-south-1, 2: sa-east-1, 1: ca-central-1, 2: eu-west-2, 2: eu-west-3
50	5: us-east-1, 5: us-east-2, 5: us-west-1, 5: eu-central-1, 5: ap-northeast-1, 5: ap-south-1, 5: sa-east-1, 5: ca-central-1, 5: eu-west-2, 5: eu-west-3
100	10: us-east-1, 10: us-east-2, 10: us-west-1, 10: eu-central-1, 10: ap-northeast-1, 10: ap-south-1, 10: sa-east-1, 10: ca-central-1, 10: eu-west-2, 10: eu-west-3

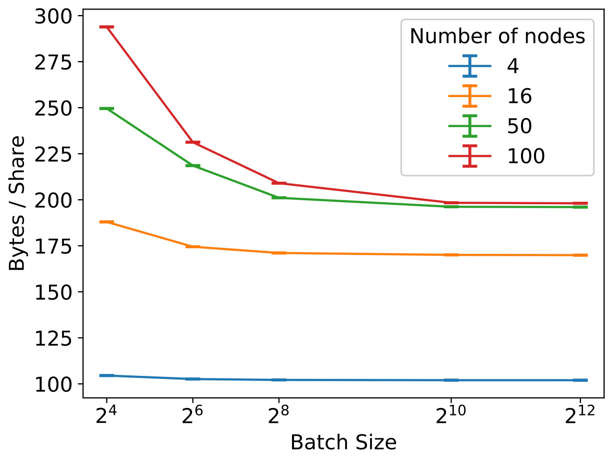
Table 5.2: Server configurations for the multi-region setting

Results

Fig. 5.2 and Table 5.3 describe the cost of reconstructing batches of different sizes across a varying number of nodes in the multi-region setting. We observe that for $N = 100$ and $k = 2^{14}$, the increased latency (from 0.1 ms to 328 ms in the worst case) between the servers in the multi-region setting makes the reconstruction time go up by just 8.7%. This is because we have to wait for only $2t + 1$ points and not all. We are able to reconstruct a batch of



(a) Computation cost per node



(b) Communication cost per node

Figure 5.2: Batch reconstruction costs in the multi-region setup

N	$\mathbf{2^6}$	$\mathbf{2^8}$	$\mathbf{2^{10}}$	$\mathbf{2^{14}}$	$\mathbf{2^{17}}$
4	818	731	902	1705	8689
16	804	807	753	2768	9887
50	1186	779	814	2003	15533
100	726	779	770	2153	17093

Table 5.3: Time in milliseconds to reconstruct batches of different sizes across varying number of nodes in the multi-region setting.

$\mathbf{2^{17}}$ = **131072 shares** on **100 nodes** in **17.093 seconds** i.e. ≈ **7668 shares/second**. The increased latency causes a 5.9% decrease in the throughput.

5.2.3 Communication Cost Validation:

In the following steps, we verify the communication cost for $N = 100$, $t = 33$ and $k = 2^{18}$ for the same region setting.

- The total number of bytes sent by one node on average was ≈ 51911697.33 bytes.
- For one round of batch reconstruction, we send out $k/(t+1)$ shares to $N-1$ parties, making it a total of $\frac{k(N-1)}{t+1}$ bytes per round. There are two such rounds, therefore:

$$\text{Total bytes sent by one node} = \frac{\text{Size of one share} \times 2k(N-1)}{t+1} \tag{5.1}$$

$$\text{Size of one share} = \frac{\text{Total bytes sent by one node} \times (t+1)}{2k(N-1)} \tag{5.2}$$

- If we use Eq. (5.2) to calculate the size of one share then it comes out to be ≈ 34.0047 bytes which is about right.
- We can also verify the per node per share communication cost by dividing Eq. (5.1) with k making it equal to $6 \times$ *Size of one share* $=$ 204 bytes/share, which is pretty close to the observed value of 198 bytes/share.
- As expected, we have an overhead of $6\times$ even in the multi-region setup.

Based on the results from this experiment, we fix the batch size to be $2^{11} = 2048$ for the rest of our experiments since that's when both the communication and computation costs start to amortize.

5.3 ONLINE PHASE: SHARE MULTIPLICATION

We evaluated the following two techniques described in Section 2.4.2 for share multiplication in the online phase at $t < N/3$:

1. Beaver's circuit randomization technique
2. Multiplication using double shares

For the share reconstructions involved in the two approaches we use our Robust Batch Reconstruction algorithm. Note that the double share based multiplication in the $N/3$ setting is non-robust i.e. a single fault in this setting will cause the program to hang because we will not have enough shares to reconstruct the polynomial robustly.

Fig. 5.3 describes how these two approaches compare against each other. We took a batch of 8192 triples for this experiment in order to get stable results. At **N=100**, we can multiply **6400 shares/second** using **double shares** and **3312 shares/second** using **Beaver's method**.

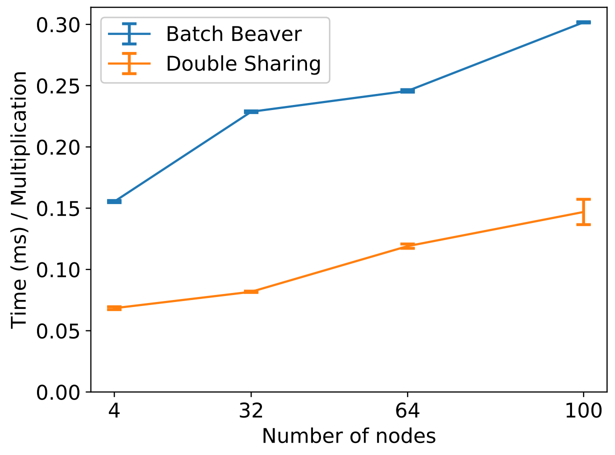
5.3.1 Computation Cost

We need to perform three multiplications, three additions and two subtractions per share multiplication using Beaver's method whereas if we were to use a double share then we need to perform only one multiplication, one addition and one subtraction per share.

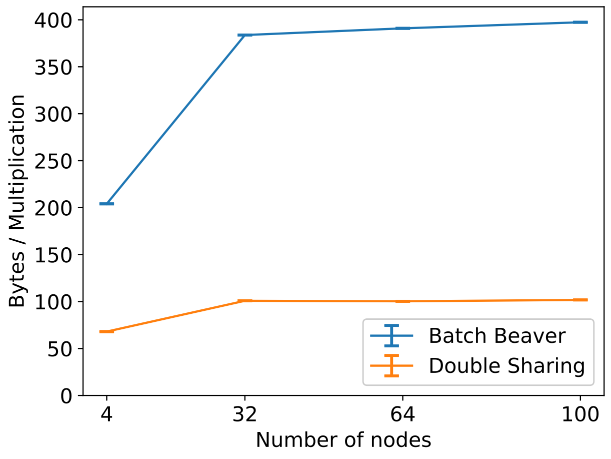
For Robust Batch Reconstruction we have to perform $k/(t+1)$ polynomial interpolations of a degree t polynomial and $k/(t+1)$ $N-$point polynomial evaluations in each of the two rounds. Since we are working with a degree $2t$ polynomial, we need to perform half the number of interpolations with double the number of values and only half the number of evaluations. As a result, we expect the double sharing technique to perform better as is evident from Fig. 5.3a.

5.3.2 Communication Cost

Beaver's method requires two share reconstructions for each share multiplication, on the other hand using double shares requires only one reconstruction. Since we are working with a degree $2t$ polynomial for double shares, we send out a chunk of $(N-1)/(2t+1)$ shares per round as compared to a chunk of $(N-1)/(t+1)$ in the case of Beaver's method. This implies that we send roughly $2\times$ more bytes in the case of Beaver's method for a single batch reconstruction. Since there are twice as many reconstructions for the Beaver's method, each node ends up sending $4\times$ times more bytes in comparison with the double sharing case. This is evident from Fig. 5.3b.



(a) Computation cost per node



(b) Communication cost per node

Figure 5.3: Cost of 8192 multiplications in the online phase at $t < N/3$

5.4 PREPROCESSING PHASE: GENERATION OF UNREFINED PREPROCESSED VALUES

We evaluate the performance of our preprocessing scheme as described in Section 3.3.5 by AVSSing 128 random values from all nodes simultaneously. Table 5.4 and Fig. 5.4 demonstrate the performance of one iteration of our preprocessing scheme. One iteration implies one execution of *hbAVSS* $\rightarrow$ *ACS* $\rightarrow$ *Processing of agreed values.* These are the total costs inclusive of all the three phases. For hbAVSS, our implementation uses linear-sized polynomial commitments instead of constant-sized ones and we skip the implicate phase. In our implementation, we parallelize each of the below mentioned steps:

- Computation of polynomial commitments for all values in the batch.
- Computation of witnesses for all values in the batch.
- Computation of shares for all values in the batch.
- Computation of shared keys for each batch for all parties at the dealer.
- Encryption of all dealer messages.
- Decryption of values in a batch at each recipient.
- Verification of all shares and their corresponding witnesses at each recipient.

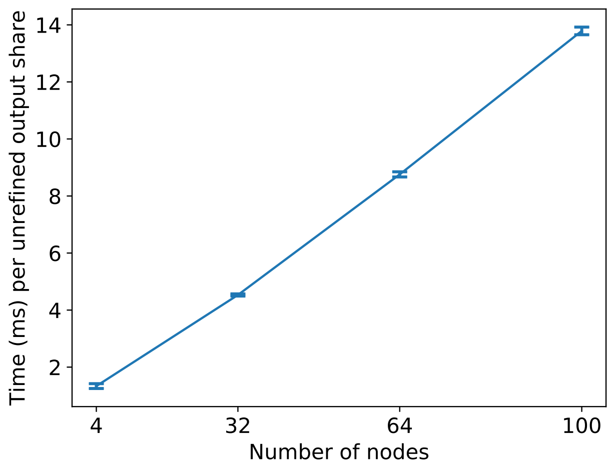
There is still a lot of room for improvement in our implementation such as moving the polynomial commitment computation to C++, use of constant-sized commitments, moving the Merkle tree creation to C++ etc.

5.4.1 Computation Cost

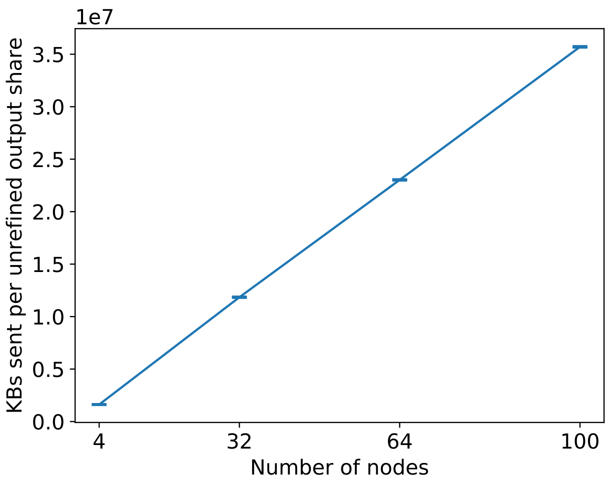
Let k be the number of values in a batch, then for our implementation of hbAVSS, the computational complexity for creating the witnesses is $\mathcal{O}(kN^2)$. Fig. 5.4a illustrates the computation cost per node per output share as the number of nodes are increased, this is expected to be linear since we get kN unrefined output shares.

5.4.2 Communication cost

As we can observe from Fig. 5.4b, the communication cost per node per output share has a linear cost as the number of nodes increases, this is expected since we are using linear-sized polynomial commitments. With the use of constant-sized polynomial commitments, we expect the communication cost per node to be constant for a large enough batch size.



(a) Computation cost per node



(b) Communication cost per node

Figure 5.4: Cost of AVSSing a batch of 128 random shares from all nodes simultaneously

N	**Count of generated unrefined random shares per node**	**Total Time (s)**
4	512	0.683712
32	4096	18.55580833
64	8192	71.74694167
100	12800	176.4862573

Table 5.4: Time taken to hbAVSS a batch of 128 random shares from all nodes

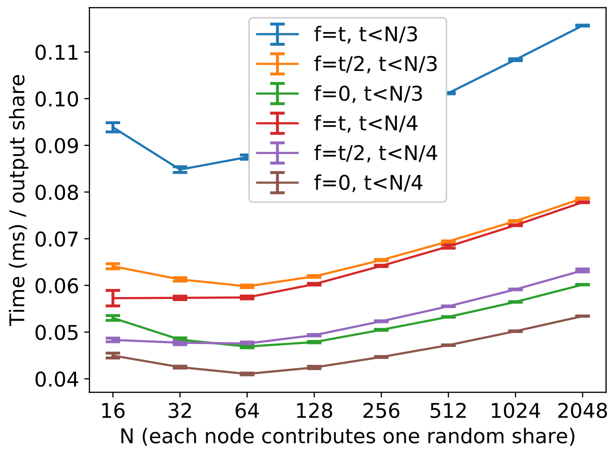
5.5 REFINING SHARES

After the *Processing of agreed values* phase each node receives a batch of shares from anywhere between $N - f$ to N nodes where f is the number of crash faults. Just to reiterate, f is not an additional parameter to the protocol and is only a setting to simulate the number of failed nodes. As described earlier in Section 3.3.5, these shares need to be refined. In this section, we evaluate different techniques for refining shares of random field elements and shares of triples.

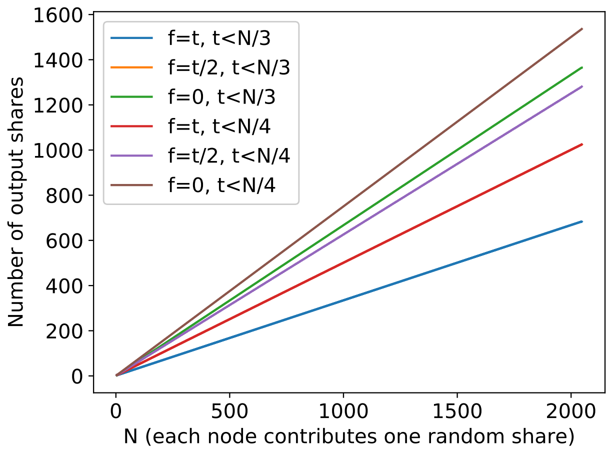
5.5.1 Random shares

We performed an experiment to observe the cost of refining a single batch of random $t-$shares for different threshold settings by varying the number of nodes and the number of crash faults. Note that the process of refining random shares involves only local computation at each node and does not involve any communication since all we do is interpolate and evaluate a polynomial on a set of points.

If there are f crash faults, then we need to refine $N - f$ shares, where $f \in [0, t]$ and for $N - f$ unrefined input shares we get $N - f - t$ refined output shares. Fig. 5.5b illustrates how the number of output refined shares varies theoretically as the input size changes for different f and t values.



(a) Computation cost per node



(b) Theoretical yield

Figure 5.5: Random Refinement of $t-$shares

The operation at the core of Random Refinement is FFT which has a computational complexity of $\mathcal{O}(N\,(log(N))$. The time taken per output share is expected to be of the order of $log(N)$. In Fig. 5.5a, for small batch sizes, the cost is dominated by other factors. As the batch size grows, we can see that the plot tends to be linear as expected (this is a lin-log plot). The amount of computation performed per output share correlates perfectly with the theoretical yield in Fig. 5.5b.

5.5.2 Shares of triples

We compare the following two different techniques for refining triples as described in Section 3.3.5:

1. $t-$share Triple Refinement

2. $2t-$share Random Refinement

Just to recall, when using double shares at $t < N/3$ we can no longer be robust in the presence of faults. In order to robustly interpolate a degree $2t$ polynomial, we need at least $4t + 1$ points which is not possible at $N = 3t + 1$.

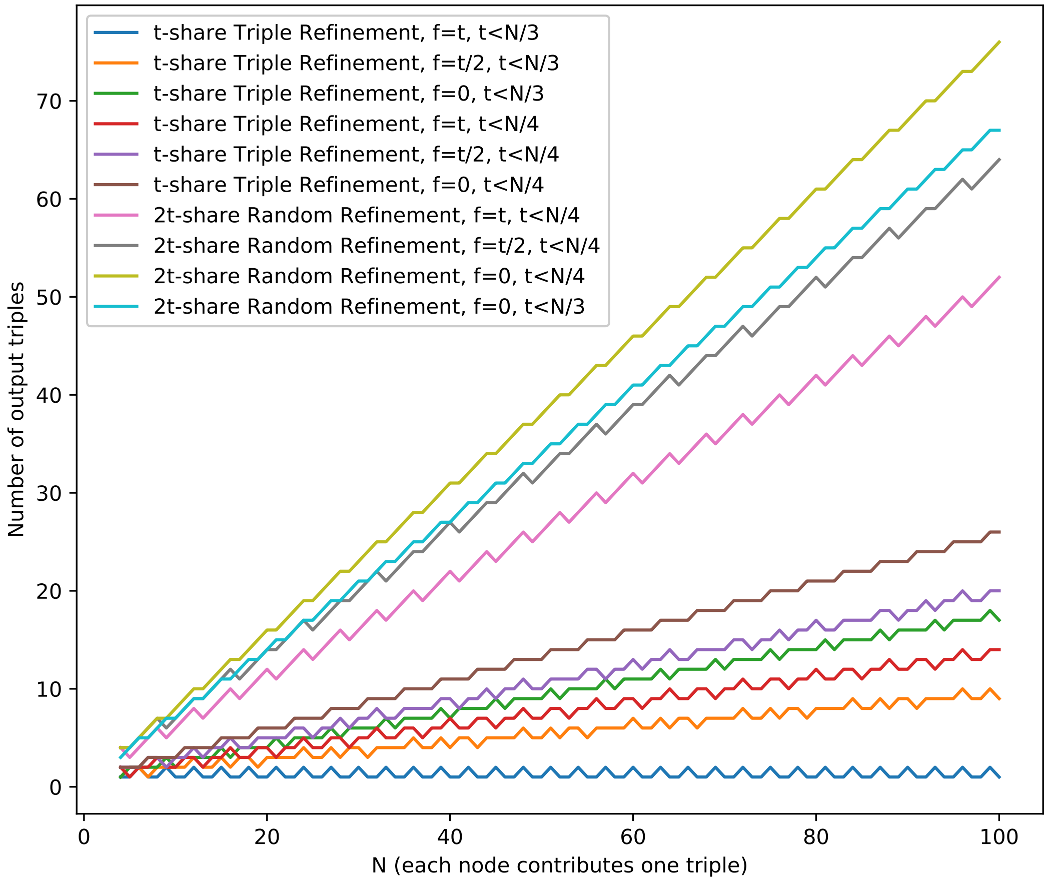


Figure 5.6: Triple Refinement: Theoretical Yield

The number of refined output triples for a batch of $N - f$ unrefined input triples for the two different techniques are given by the following expressions:

1.
$$t - \text{share Triple Refinement} = \frac{N - f - 2t + 1}{2} \tag{5.3}$$

2.
$$2t - \text{share Random Refinement} = N - f - t \tag{5.4}$$

Refer to Section 3.3.5 for an explanation.

Fig. 5.6 illustrates how the number of output triples vary theoretically as the threshold, the number of crash faults, and the number of nodes contributing the input triples change. This plot is jagged because we compute t by taking floor of $(N-1)/3$ or $(N-1)/4$.

How much does it cost to be $t-$robust?

Fig. 5.8a illustrates the actual computation cost for different threshold settings as the number of faults is varied. We fixed the number of nodes to be $N = 32$ for this experiment. We ran this experiment with a batch size of 4096 where each batch contained $N - f = 32 - f$ triples. For a given t, as the number of faults increase, the decrease in yield is much more than the decrease in the amount of total computation causing the cost per output triple to go up. If we fix f, then increasing the threshold t has a similar impact on the yield and the total computation as it does when we fix t. These relations can be realized perfectly with the help of a theoretical computation cost that we plot in Fig. 5.7 using the following equation:

$$\text{Computation cost per output triple} = \frac{c\,(N-f)\,log(N-f)}{\text{Number of output triples}} \tag{5.5}$$

Extracting triples using a Random Refinement approach based on $2t-$shares does much better because of a significantly higher yield and the fact that we need to do only one polynomial interpolation and extrapolation. This increased yield is also evident in Fig. 5.6.

'The theoretical communication cost per output triple is also very similar to the theoretical computation cost. It just does not have the logarithmic component which is anyway pretty small, as a result the graph in Fig. 5.8b looks very similar to Fig. 5.7.

Fig. 5.9 shows the cost of Triple Refinement for three different approaches as the number of nodes are varied. For this experiment, we use 2048 batches of triples wherein each batch contains exactly N triples. We can justify the computation cost by correlating with the yield in Fig. 5.6. The $2t-$share approach involves only half the number of reconstructions compared to the $t-$share approach.

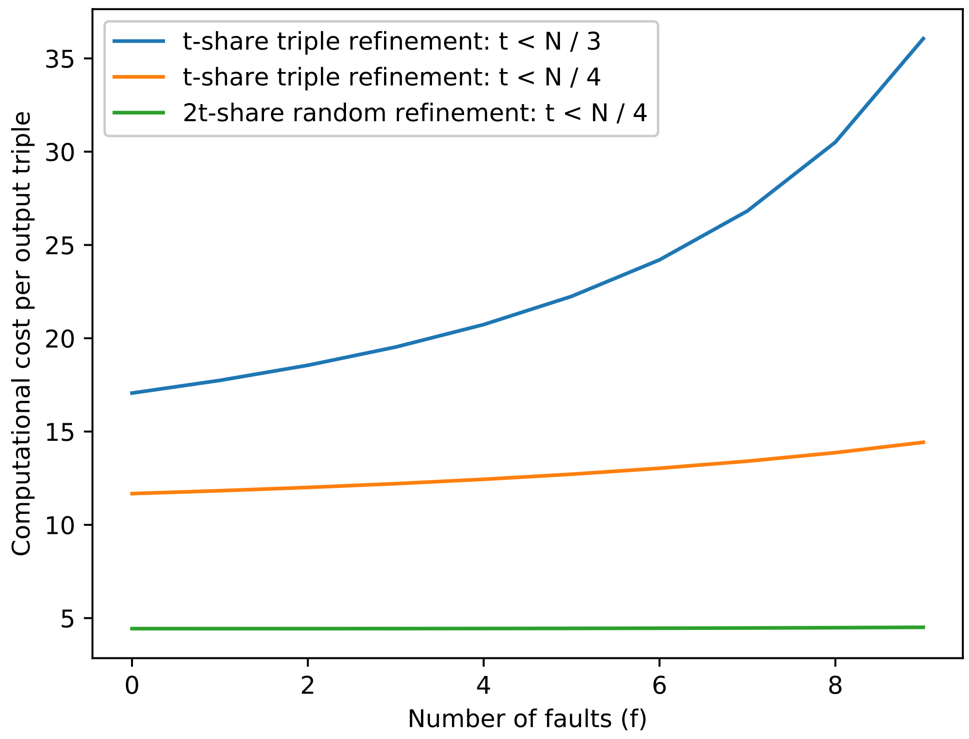
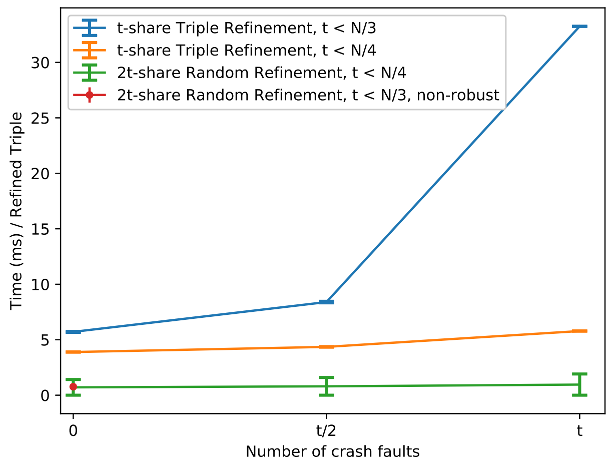
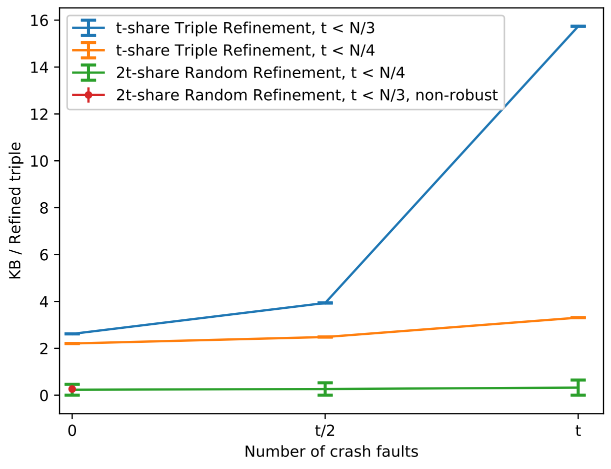


Figure 5.7: Triple Refinement: Theoretical Cost

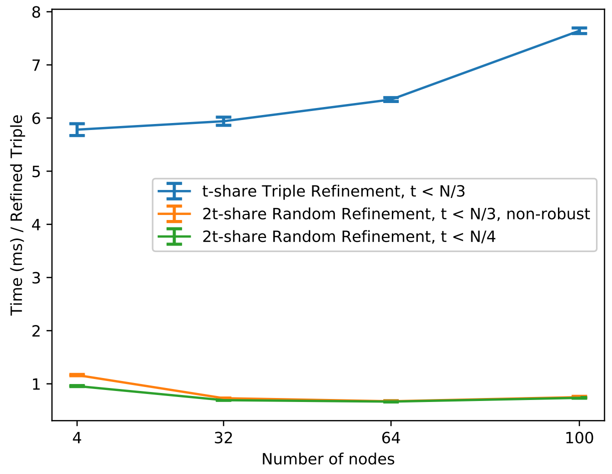


(a) Computation cost per node

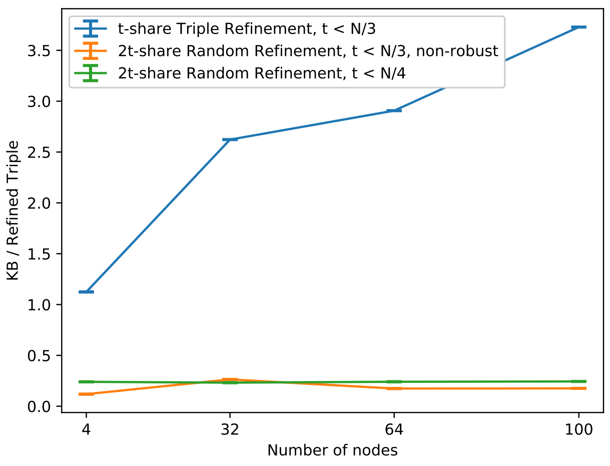


(b) Communication cost per node

Figure 5.8: Cost of Triple Refinement at $N = 32$ for 4096 batches



(a) Computation cost per node



(b) Communication cost per node

Figure 5.9: Cost of Triple Refinement with zero crash faults for 2048 batches

5.5.3 Code optimization

Fig. 5.10 illustrates the performance benefit of implementing the same algorithm in C++ and invoking it via Python versus implementing it purely in Python. We can see that the C++ implementation invoked via Python is orders of magnitude faster than the pure Python implementation.

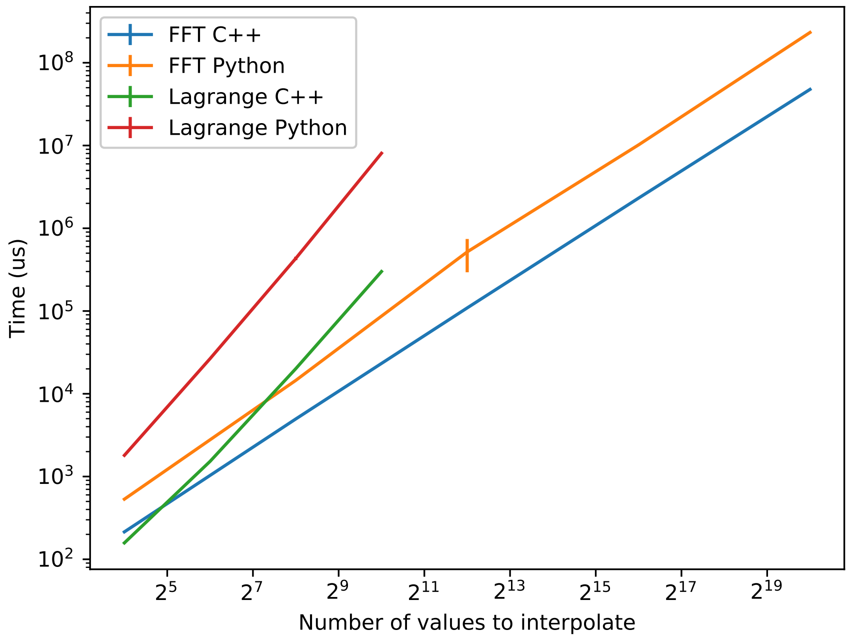


Figure 5.10: C++ vs Python

CHAPTER 6: RELATED WORK

VIFF [24] is secure against an active adversary in the $t < N/3$ setting. It is, however, non-robust since an adversary can halt the protocol. VIFF's preprocessing phase is also not scalable since it uses an approach based on Hyperinvertible Matrices which has a per node computation and communication cost of $\mathcal{O}(N^2)$. Although they do talk about *ShareList* to denote a collection of shares, the *ShareList* abstraction does not employ any techniques to reduce the communication overhead for reconstructing a batch of shares when compared to our *ShareArray* abstraction which uses the Robust Batch Reconstruction technique.

Damgård et al. [23] describe a protocol with robust preprocessing for a threshold of $N/3$ but it is synchronous since it relies on a synchronous VSS scheme. They are able to maintain a threshold of $N/3$ with the use of double sharings even for the online phase since they rely on a synchronous model. They make use of the Berlekamp-Welch algorithm for robust decoding of shares which has a computational complexity of $\mathcal{O}(N^3)$ whereas we use Gao's [40] algorithm which when used along with FFT has a complexity of $\mathcal{O}(n(log\, n)^l)$ for some small constant l.

Chida et al. [56] describe a method in which they propose techniques to verify that the adversary has not cheated. Their protocol is secure in the presence of static malicious adversaries who control at most $t < N/2$ corrupted parties. Their protocol however does not achieve fairness, implying that the adversary may receive output while the honest parties do not. It is also non-robust since a malicious party can cause the protocol to abort.

Choudhury and Patra [26] describe an asynchronous protocol resilient to $t < N/4$ adversarial nodes. However, we can run in the $t < N/3$ setting since we make use of the *hbAVSS* protocol based on polynomial commitments. The *hbAVSS* protocol has a linear amortized overhead in the $N/3$ setting which is similar to the linear overhead of their AVSS protocol in the $N/4$ setting.

HyperMPC [46] aims to achieve the same goal of supporting continuously running MPC applications for a threshold of $N/3$. However, it is non-robust. It relies on double sharings for the preprocessing phase followed by a polynomial reduction for a share multiplication in the online phase. For the refinement of double shares, HyperMPC uses Hyperinvertible matrices, this approach has a computation complexity of $\mathcal{O}(N^2)$ as against $\mathcal{O}(N(log\, N))$ for our FFT based refinement. They do mention running the preprocessing in parallel in order to generate a batch of random shares. However, in contrast to our hbAVSS protocol HyperMPC does not employ any techniques to reduce the communication overhead for generating a batch of shares.

Other frameworks that we evaluated do not realize many-party computation or are not robust to active adversaries. SCALE-MAMBA [25] is a production system which has evolved from SPDZ [57], BDOZ [58] and TinyOT [59] but cannot guarantee output if some nodes crash. MPyC [60] is an MPC framework which came out of VIFF but is secure only against semi-honest adversaries. The EMP-Toolkit [27] supports 2PC and MPC with garbled circuits but is not fault-tolerant. Obliv-C [28] and ObliVM [29] support only 2PC. Sharemind [30] does not support active adversaries. Choudhary and Patra [26] present a work which achieves all our goals but for a corruption threshold of $t < N/4$.

CHAPTER 7: FUTURE WORK

HoneyBadgerMPC is an ongoing project and there are some important research directions that we wish to explore. We want to see if we can get the benefit of cheap dot products using double sharing while still maintaining a corruption threshold of $t < N/3$ with the help of pairing friendly polynomial commitments. This will be essential in getting good performance on linear regression and federated learning. In order to be able to scale to $N = 10000$ nodes, we want to support committees where we would sample a constant number of nodes which together make up a committee such that all committees work concurrently. We also intend to investigate constant round MPC using MPC-friendly symmetric encryption along with committees to generate garbled circuits. These garbled circuits can be evaluated in public, and only the input/output and wire label mappings need to be stored in secret shared form. Lastly, we would also like to look into publicly auditable MPC such that the output of MPC also includes a zero-knowledge proof or correctness. In terms of the codebase, for performance reasons it would be better to have the entire implementation in C++ which is then wrapped in Python to still allow rapid prototyping of applications.

REFERENCES

[1] J. Cartlidge, N. P. Smart, and Y. T. Alaoui, "Mpc joins the dark side," Cryptology ePrint Archive, Report 2018/1045, 2018, https://eprint.iacr.org/2018/1045.

[2] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft, "Secure multiparty computation goes live," in *Financial Cryptography and Data Security*, R. Dingledine and P. Golle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 325–343.

[3] C. S. Jutla, "Upending stock market structure using secure multi-party computation," Cryptology ePrint Archive, Report 2015/550, 2015, https://eprint.iacr.org/2015/550.

[4] "Discover tax and vat fraud easier with sharemind," 2019. [Online]. Available: https://sharemind.cyber.ee/tax-vat-fraud/

[5] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," Cryptology ePrint Archive, Report 2017/396, 2017, https://eprint.iacr.org/2017/396.

[6] M. Dahl, J. Mancuso, Y. Dupis, B. Decoste, M. Giraud, I. Livingstone, J. Patriquin, and G. Uhma, "Private machine learning in tensorflow using secure computation," *CoRR*, vol. abs/1810.08130, 2018. [Online]. Available: http://arxiv.org/abs/1810.08130

[7] V. Chen, V. Pastro, and M. Raykova, "Secure computation for machine learning with SPDZ," *CoRR*, vol. abs/1901.00329, 2019. [Online]. Available: http://arxiv.org/abs/1901.00329

[8] P. Mohassel and P. Rindal, "Aby3: A mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243760 pp. 35–52.

[9] D. W. Archer, D. Bogdanov, L. Kamm, Y. Lindell, J. I. Pagter, K. Nielsen, N. P. Smart, and R. N. Wright, "From Keys to DatabasesReal-World Applications of Secure Multi-Party Computation," *The Computer Journal*, vol. 61, no. 12, pp. 1749–1771, 09 2018. [Online]. Available: https://doi.org/10.1093/comjnl/bxy090

[10] I. Damgård, T. P. Jakobsen, J. B. Nielsen, and J. I. Pagter, "Secure key management in the cloud," in *Cryptography and Coding*, M. Stam, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 270–289.

[11] "Curv," 2019. [Online]. Available: https://www.curv.co/technology/

[12] "Sepior," 2019. [Online]. Available: https://sepior.com/

[13] "Unbound tech," 2019. [Online]. Available: https://www.unboundtech.com/solutions/blockchain-key-management/

[14] "Use sharemind to build location services without breaking privacy laws," 2019. [Online]. Available: https://sharemind.cyber.ee/location-services/

[15] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell, "Privacy-preserving shortest path computation," *CoRR*, vol. abs/1601.02281, 2016. [Online]. Available: http://arxiv.org/abs/1601.02281

[16] A. Abidin, A. Aly, S. Cleemput, and M. A. Mustafa, "An mpc-based privacy-preserving protocol for a local electricity trading market," *IACR Cryptology ePrint Archive*, vol. 2016, p. 797, 2016.

[17] L. Kamm and J. Willemson, "Secure floating point arithmetic and private satellite collision analysis," *International Journal of Information Security*, vol. 14, no. 6, pp. 531–548, Nov 2015. [Online]. Available: https://doi.org/10.1007/s10207-014-0271-8

[18] B. Hemenway, S. Lu, R. Ostrovsky, and W. W. IV, "High-precision secure computation of satellite collision probabilities," Cryptology ePrint Archive, Report 2016/319, 2016, https://eprint.iacr.org/2016/319.

[19] Z. Beerliová-Trubíniová and M. Hirt, "Efficient multi-party computation with dispute control," in *Theory of Cryptography*, S. Halevi and T. Rabin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 305–328.

[20] Z. Beerliová-Trubíniová and M. Hirt, "Simple and efficient perfectly-secure asynchronous mpc," in *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT'07. Berlin, Heidelberg: Springer-Verlag, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1781454.1781486 pp. 376–392.

[21] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-secure mpc with linear communication complexity," in *Theory of Cryptography*, R. Canetti, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 213–230.

[22] E. Ben-Sasson, S. Fehr, and R. Ostrovsky, "Near-linear unconditionally-secure multi-party computation with a dishonest minority," in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 663–680.

[23] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *Advances in Cryptology - CRYPTO 2007*, A. Menezes, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 572–590.

[24] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *Public Key Cryptography – PKC 2009*, S. Jarecki and G. Tsudik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 160–179.

[25] "Scale-mamba software," 2019. [Online]. Available: https://homes.esat.kuleuven.be/~nsmart/SCALE/

[26] A. Choudhury and A. Patra, "An efficient framework for unconditionally secure multiparty computation," *IEEE Transactions on Information Theory*, vol. 63, no. 1, pp. 428–468, Jan 2017.

[27] "Emp-toolkit: Efficient multiparty computation toolkit," 2019. [Online]. Available: https://github.com/emp-toolkit

[28] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1153, 2015.

[29] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015. [Online]. Available: https://doi.org/10.1109/SP.2015.29 pp. 359–376.

[30] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *Computer Security - ESORICS 2008*, S. Jajodia and J. Lopez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 192–206.

[31] A. Kate, A. Miller, and T. Yurek, "Brief Note: Asynchronous Verifiable Secret Sharing with Optimal Resilience and Linear Amortized Overhead," *arXiv e-prints*, p. arXiv:1902.06095, Feb 2019.

[32] "Asynchronous i/o in python," 2019. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[33] "Python programming language. release 3.7," 2019. [Online]. Available: https://docs.python.org/3/library/asyncio.html

[34] B. Archer and E. W. Weisstein, "Lagrange interpolating polynomial." 2019. [Online]. Available: http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html

[35] "Horner's rule," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Horner%27s_method

[36] "Vandermonde matrix," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Vandermonde_matrix

[37] "Constructing the interpolation polynomial," 2019. [Online]. Available: https://en.wikipedia.org/wiki/Polynomial_interpolation#Constructing_the_interpolation_polynomial

[38] R. Fateman, "The (finite field) fast fourier transform," 2019. [Online]. Available: https://pdfs.semanticscholar.org/7a2a/4e7f8c21342a989d253704cedfb936bee7d7.pdf

[39] A. Soro and J. Lacan, “Fnt-based reed-solomon erasure codes,” in *2010 7th IEEE Consumer Communications and Networking Conference*, Jan 2010, pp. 1–5.

[40] S. Gao, *A New Algorithm for Decoding Reed-Solomon Codes.* Boston, MA: Springer US, 2003, pp. 55–68. [Online]. Available: https://doi.org/10.1007/978-1-4757-3789-9_5

[41] A. Shamir, “How to share a secret,” *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[42] R. C. T. A. University), “Secure multiparty computation: Introduction,” 2019. [Online]. Available: https://www.cs.tau.ac.il/~iftachh/Courses/Seminars/MPC/Intro.pdf

[43] D. Beaver, “Efficient multiparty protocols using circuit randomization,” in *Advances in Cryptology — CRYPTO ’91*, J. Feigenbaum, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 420–432.

[44] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of bft protocols,” Cryptology ePrint Archive, Report 2016/199, 2016, https://eprint.iacr.org/2016/199.

[45] S. Bowe, “Bls12-381: New zk-snark elliptic curve construction,” 2019. [Online]. Available: \url{https://z.cash/blog/new-snark-curve}

[46] A. Barak, M. Hirt, L. Koskas, and Y. Lindell, “An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243801 pp. 695–712.

[47] “Ntl: A library for doing number theory,” 2019. [Online]. Available: https://www.shoup.net/ntl/

[48] “Cython: C-extension for python,” 2019. [Online]. Available: https://cython.org/

[49] “Rust,” 2019. [Online]. Available: https://www.rust-lang.org/

[50] “Rust,” 2019. [Online]. Available: https://github.com/zkcrypto/pairing

[51] “Docker,” 2019. [Online]. Available: https://www.docker.com/

[52] “pytest: helps you write better programs,” 2019. [Online]. Available: https://docs.pytest.org/en/latest/

[53] “Travis ci - test and deploy your code with confidence,” 2019. [Online]. Available: https://travis-ci.org/

[54] “Amazon web services,” 2019. [Online]. Available: https://aws.amazon.com/

[55] “Amazon ec2,” 2019. [Online]. Available: https://aws.amazon.com/ec2/

[56] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, “Fast large-scale honest-majority mpc for malicious adversaries,” in *Advances in Cryptology – CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 34–64.

[57] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, “Multiparty computation from somewhat homomorphic encryption,” in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 643–662.

[58] R. Bendlin, I. Damgrd, C. Orlandi, and S. Zakarias, “Semi-homomorphic encryption and multiparty computation,” Cryptology ePrint Archive, Report 2010/514, 2010, https://eprint.iacr.org/2010/514.

[59] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, “A new approach to practical active-secure two-party computation,” Cryptology ePrint Archive, Report 2011/091, 2011, https://eprint.iacr.org/2011/091.

[60] “Mpyc: Secure multiparty computation in python,” 2019. [Online]. Available: https://github.com/lschoe/mpyc