MPC Gender-Based Credit Utilization

Courtney Duquette
The George Washington University
April 28, 2021

Overview

- Application Description
- Protocol
  - Input Data
  - Code Overview
  - Supporting Scripts
  - MPC Protocols Utilized
- Experiments & Results
- Challenges & Lessons Learned

Overview

- Application Description
- Protocol
  - Input Data
  - Code Overview
  - Supporting Scripts
  - MPC Protocols Utilized
- Experiments & Results
- Challenges & Lessons Learned

Application Description

- We want to calculate an average monthly credit utilization across a population and see if it differs by gender

- However, each individual person does not want to share either their credit limit or how much they routinely spend on their credit cards

- Therefore, we are use MPC to perform this calculation and the results can be shared back to the participants

Overview

- Application Description
- Protocol
  - Input Data
  - Code Overview
  - Supporting Scripts
  - MPC Protocols Utilized
- Experiments & Results
- Challenges & Lessons Learned

Protocol Steps

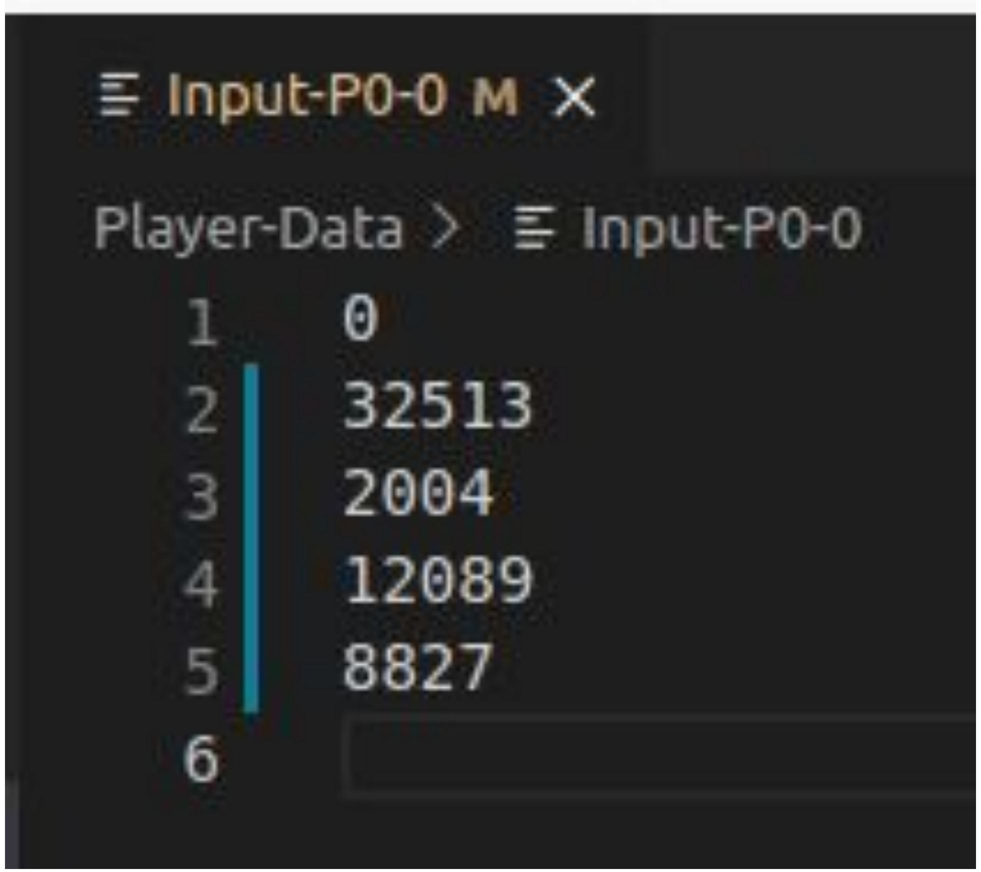
1. Read participant/player data and store gender for later use
2. Parse the creditline and monthly spend amounts
3. Determine monthly credit utilizations by dividing each month spend amount by credit line
4. Calculate individual average utilization by summing the utilizations and dividing by total number of months
5. Sum the individual average utilization for all participants by gender
6. Divide sum by total players of the gender
7. Print each average for the different genders

Input Data

- Each participant will create an input text file

- Format:
  - Line 1: Gender [0, 1, 2]
  - Line 2: Creditline
  - Line 3+: Monthly Spend

- The same number of months must be provided by all parties



Code Overview

- All input it read into a `SFIX` matrix

- All summations and divisions are done securely

- Only reveals the final average when printing results

average_by_gender_all_secured.mpc

Programs > Source > average_by_gender_all_secured.mpc

```
# Initializing constants
NUM_PLAYERS=2
NUM_MS_GIVEN=3
GENDER_INDEX=0
CL_INDEX=1
MS_START_INDEX=2

NUM_INPUT=NUM_MS_GIVEN+2

sfix.set_precision(16, 32) # Setting the precision

# Inputing Data in Matrix
print_ln('Inputing Data')
input_data = Matrix(NUM_INPUT, NUM_PLAYERS, sfix)

for p in range(NUM_PLAYERS):
    for i in range(NUM_INPUT):
        input_data[i][p] = sfix.get_input_from(p)

# Calulating Monthly Utilizations per Player
print_ln('Calculating Monthly utilization')
monthly_utlization_data = Matrix(2, NUM_PLAYERS, sfix)

for p in range(NUM_PLAYERS):
    monthly_utlization_data[GENDER_INDEX][p] = input_data[GENDER_IND
```

Code Overview

- Optimized by allowing one of the divisions to be done in the clear
- The total count of each gender and the summations of the monthly utilizations are reveal

average_by_gender_optimized.mpc

Programs > Source > average_by_gender_optimized.mpc

```
            gender_data[1][1] = gender_data[1][1] + monthly_

        @if_((monthly_utlization_data[0][p] == 2).reveal())
        def _():
            gender_data[0][2] = gender_data[0][2] + 1
            gender_data[1][2] = gender_data[1][2] + monthly_


# divide sum by total gender count
print_ln('Calculating Averages')

clear_gender_data = gender_data.reveal_nested()

for g in range(3):
    clear_gender_data[2][g] = clear_gender_data[1][g] /


# print out final average by gender
print_ln('Output:')
for g in range(3):
    print_ln('For gender, %s, the average credit utiliza
```

Code Overview

- Next optimization was updating the MPC application to allow for pre-processing
- Wrote a python script that would perform each individuals computations before calling the MPC protocol
- Script requires the range of players to process

preprocess_code.py

Scripts > preprocess_code.py > ...

Set as interpreter

```python
#! /usr/bin/python
import sys
from datetime import datetime

start_time = datetime.now()

start_index = 0
end_index = 2

if len(sys.argv) > 1:
    start_index = int(sys.argv[1])
    end_index = start_index + 1


for P in range(start_index, end_index):
    fileName = "../Player-Data/Input-P{0}-0".format(P)
    input_file = open(fileName, "r+")
    Lines = input_file.read().splitlines()

    creditline = Lines[1]

    UTILIZATION = 0
```

Code Overview

average_by_gender_preprocessing_all_secured.mpc

Programs > Source > average_by_gender_preprocessing_all_secured.mpc

```
# Initializing constants
NUM_PLAYERS=2
NUM_INPUT=2

sfix.set_precision(16, 32) # Setting the precision

# Inputing Data in Matrix
print_ln('Inputing Data')

# [
#    gender, sumOfThatPlayersUtilizations
# ]
input_data = Matrix(NUM_INPUT, NUM_PLAYERS, sfix)

for p in range(NUM_PLAYERS):
    for i in range(NUM_INPUT):
        input_data[i][p] = sfix.get_input_from(p)

# sum the secret shares for all players by gender
print_ln('Summing data by gender')

# [
#    numOfThatGender, sumOfUtilizations, averageForThatGender
# ]
```

average_by_gender_preprocessing_optimized.mpc

Programs > Source > average_by_gender_preprocessing_optimized.mpc

```
            gender_data[0][1] = gender_data[0][1] + 1
            gender_data[1][1] = gender_data[1][1] + input_data[1][

        @if_((input_data[0][p] == 2).reveal())
        def _():
            gender_data[0][2] = gender_data[0][2] + 1
            gender_data[1][2] = gender_data[1][2] + input_data[1][


# divide sum by total gender count
print_ln('Calculating Averages in Clear')

clear_gender_data = gender_data.reveal_nested()

for g in range(3):
    clear_gender_data[2][g] = clear_gender_data[1][g] / clear_


# print out final average by gender
print_ln('Output:')
for g in range(3):
    print_ln('For gender, %s, the average credit utilization i
```

Compilation Comparisons

average_by_gender_all_secured.mpc

```
Program requires:
           5 integer inputs from player 0
           5 integer inputs from player 1
       11235 integer bits
        3360 integer triples
         165 virtual machine rounds
```

average_by_gender_optimized.mpc

```
Program requires:
           5 integer inputs from player 0
           5 integer inputs from player 1
        7890 integer bits
        2340 integer triples
         132 virtual machine rounds
```

abg_preprocessing_all_secured.mpc

```
Program requires:
           2 integer inputs from player 0
           2 integer inputs from player 1
        3777 integer bits
        1206 integer triples
          77 virtual machine rounds
```

abg_preprocessing_optimized.mpc

```
Program requires:
           2 integer inputs from player 0
           2 integer inputs from player 1
         432 integer bits
         186 integer triples
          44 virtual machine rounds
```

Supporting Scripts

Creating Random Inputs

- Python script that creates random input files in proper format

- Takes in two different inputs:
  - Number of months per player
  - Number of player files to generate

create_input.py ×

Scripts > create_input.py > ...

Set as interpreter

```python
#! /usr/bin/python
import sys, os, random

os.chdir('../Player-Data/')

num_stmts = int(sys.argv[1])
end_index = int(sys.argv[2])

for P in range(end_index):
    fileName = "Input-P{0}-0".format(P)
    input_file = open(fileName, "w")

    input_file.write(str(random.randint(0,2)) + "\n")

    creditline = random.randint(1000, 50000)
    input_file.write(str(creditline) + "\n")

    for s in range(num_stmts):
        input_file.write(str(random.randint(0, creditline)) +

    input_file.close()

```

Supporting Scripts

Validating Output

- Reads all player files from Player-Data
- Performs the protocol steps and out prints the results
- Is used to validate the result of the MPC application

validate_output.py ×

Scripts > validate_output.py > ...

Set as interpreter

```python
#! /usr/bin/python
import sys
from datetime import datetime

start_time = datetime.now()

end_index = int(sys.argv[1])

GENDER_0 = 0
GENDER_1 = 0
GENDER_2 = 0
UTILIZATION_0 = 0
UTILIZATION_1 = 0
UTILIZATION_2 = 0
AVERAGE_0 = 0
AVERAGE_1 = 0
AVERAGE_2 = 0

for P in range(0,end_index):
    input_file = open("../Player-Data/Input-P{0}-0".forma
    Lines = input_file.read().splitlines()
```

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

MPC Protocol: MASCOT

- A protocol improving on SPDZ that focuses on the offline phase, while keeping the online phase as it was
  - The offline phase has been updated to use Oblivious Transfer instead of Somewhat Homomorphic Encryption to generate the triples

- In MASCOT/SPDZ, addition and constant gates are done locally and multiplication requires authenticated triples

- A semi-honest security of this protocol will strip what is needed for malicious security checks
  - E.G. MAC generation, OT correlation checks

MPC Protocol: Shamir

- A participant, *P1*, wants to share *x* between *n* parties s.t. any *t* parties can construct the value
  - P1 constructs a t-1 degree polynomial, p(x), s.t. p(0) = x
  - For i $\in$ [n], P1 computes $[x]_i = p(i)$ and sends this value to P_i
  - Doubly-shared random values ($[r]^t$, $[r]^{2t}$) for each multi. gate [Damgård-Nielsen '07]

- In the online stage
  - + gate: Parties locally compute [c] = [a] + [b]
  - x gate: Parties use a pair ($[r]^t$, $[r]^{2t}$) to produce [c] = [a] x [b]

- Malicious security involves running the protocol twice [Chida et al. 2018]
  - Once with real inputs, *x, y* and the second time with inputs αx, αy for a random α
  - After the protocol, reveal α and for each multiplication gate, ensure that *([w], [u])*, check that $u - \alpha w = 0$

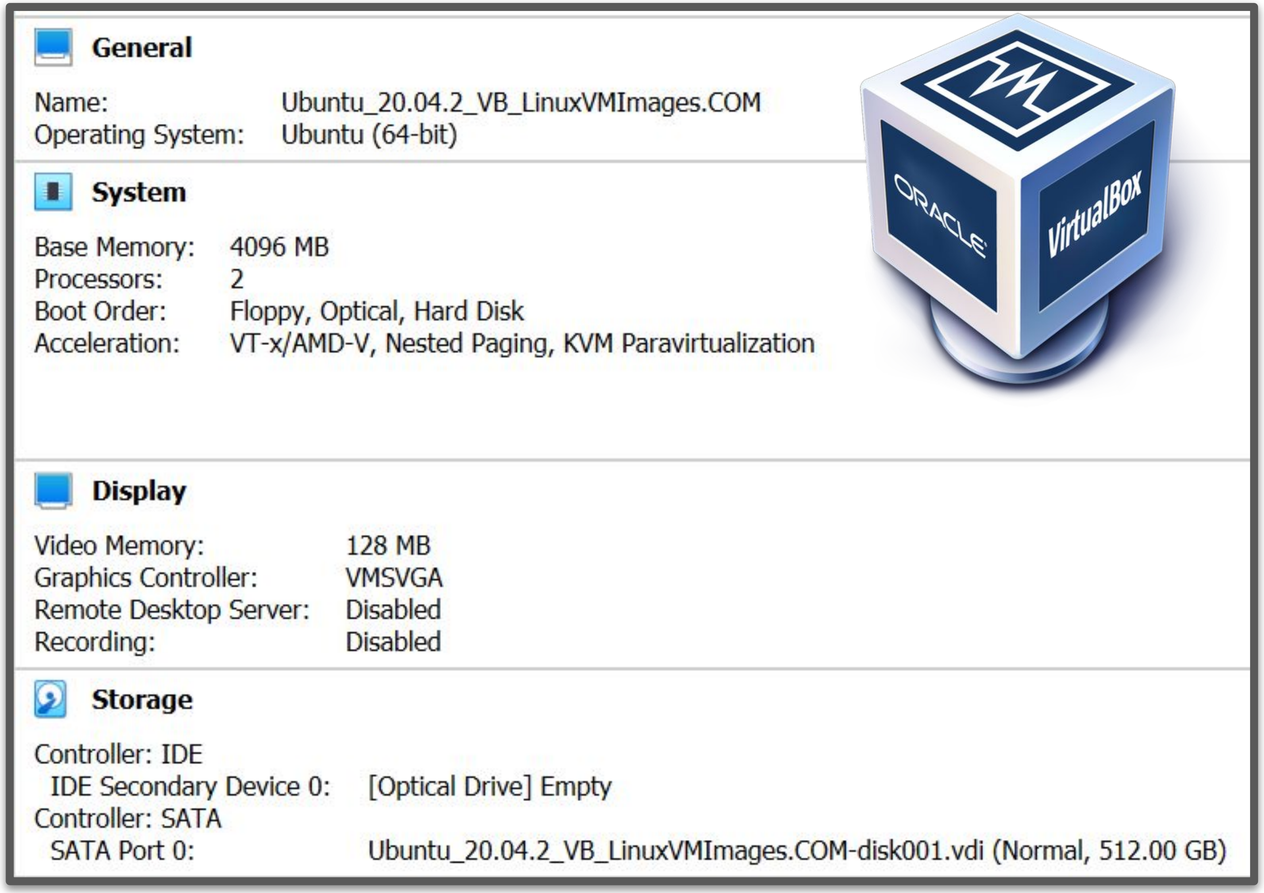
Overview

- Application Description
- Protocol
  - Input Data
  - Code Overview
  - Supporting Scripts
  - MPC Protocols Utilized
- Experiments & Results
- Challenges & Lessons Learned

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

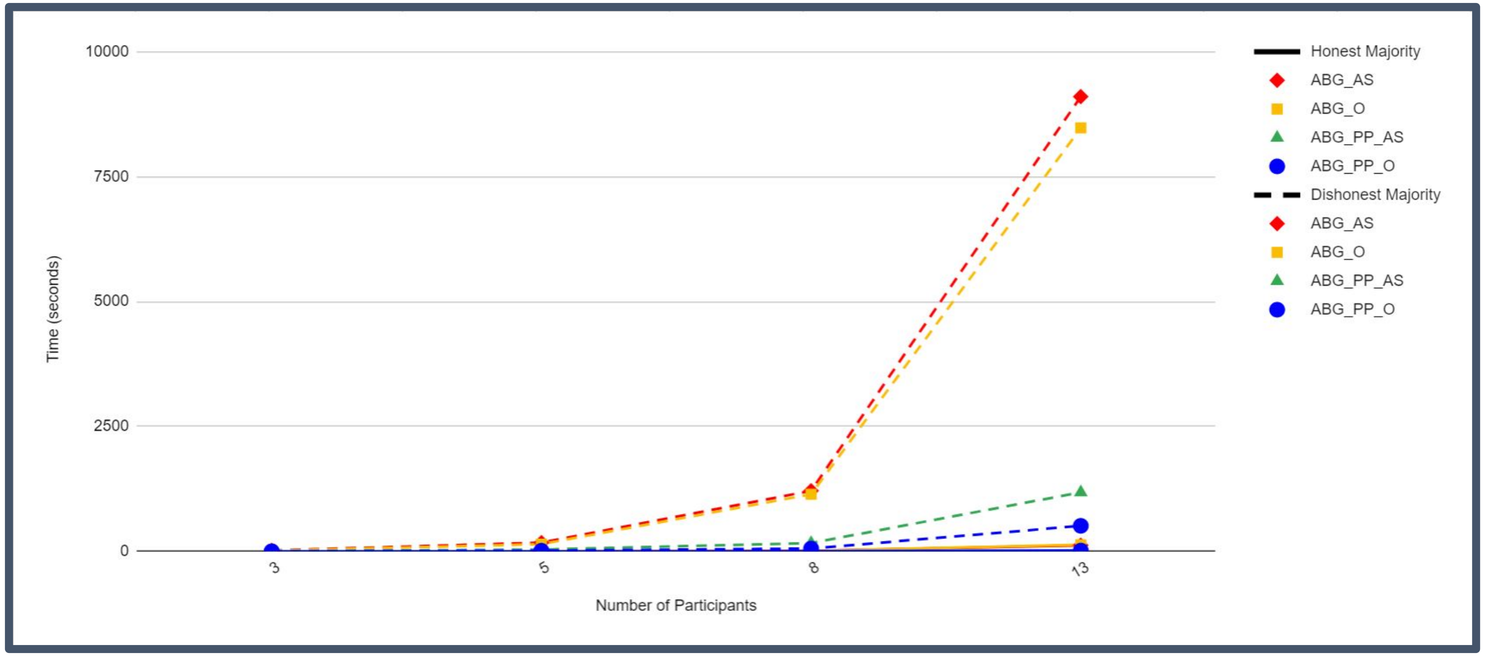
Machine Specs

- Ubuntu 20.04 running on a VirtualBox
- 6 processors
  - CPU speed 1608.002
- 4096 MB of memory



Experiment Setup

- Each experiment will run the 4 different variations of the code
  - Input will have 3 months of credit spend per participate
- Will measure the timing of the protocol with incrementing total participates
  - Comparing between 3 and 25 different participants
  - The time for the preprocessing script to run will be factored into the total time for those variations
- Finally, we will see how the Shamir protocol performs with varying the input data size
  - Comparing 3, 6, 9, 12 months of data with 3 participants, maliciously secured

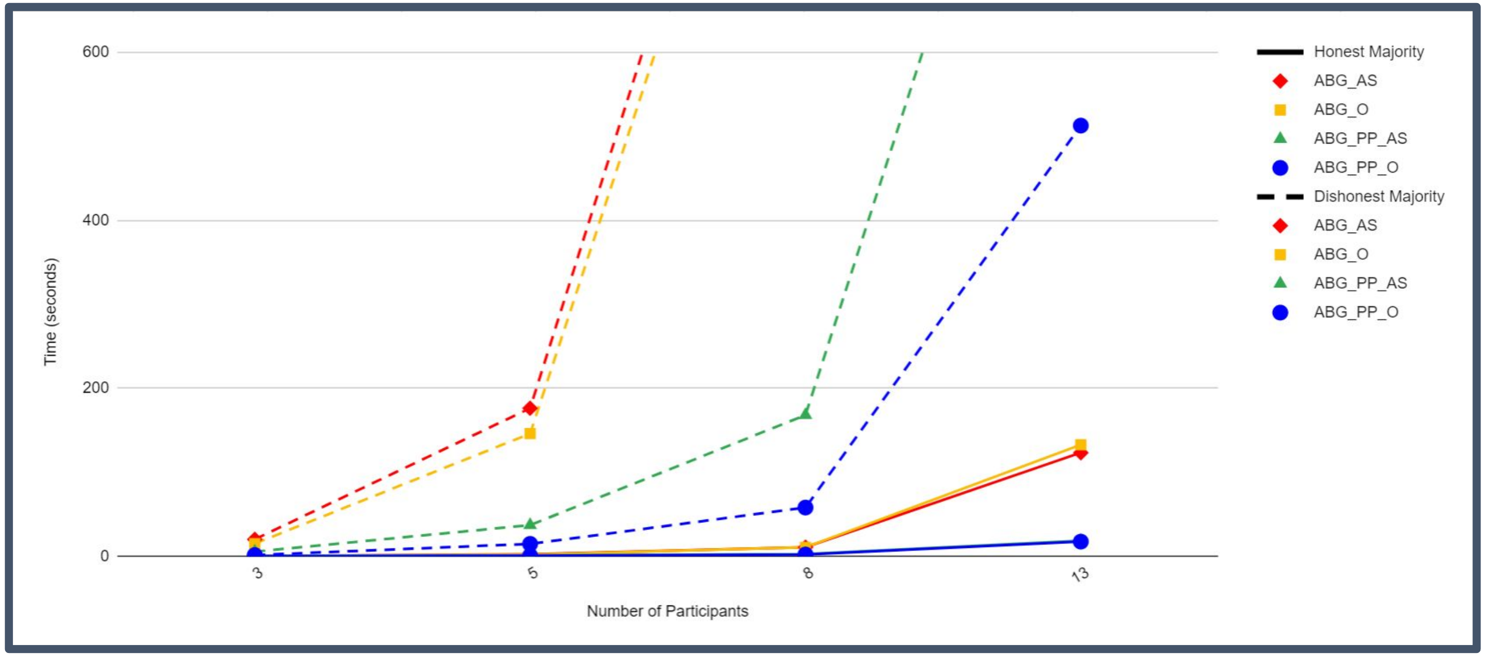


Number of Participates	3	5	8	13
Honest Majority	0			
ABG_AS	0.770233	2.83085	11.1971	123.676
ABG_O	0.701304	2.40429	11.1502	133.041
ABG_PP_AS	0.69959	1.508741	3.021501	18.976679
ABG_PP_O	0.598663	1.114001	2.584691	17.883079
Dishonest Majority	0			
ABG_AS	20.6485	176.488	1213.62	9101.84
ABG_O	15.5033	146.308	1143.84	8474.98
ABG_PP_AS	5.950353	37.581221	168.401621	1183.105779
ABG_PP_O	1.914263	15.216321	58.402621	512.614779

Honest vs Dishonest Majority (Semi-Honest)

https://github.com/cdduquette/CSCI6907-FinalProject/tree/main/Experiments/Honest_Dishonest_Semi

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

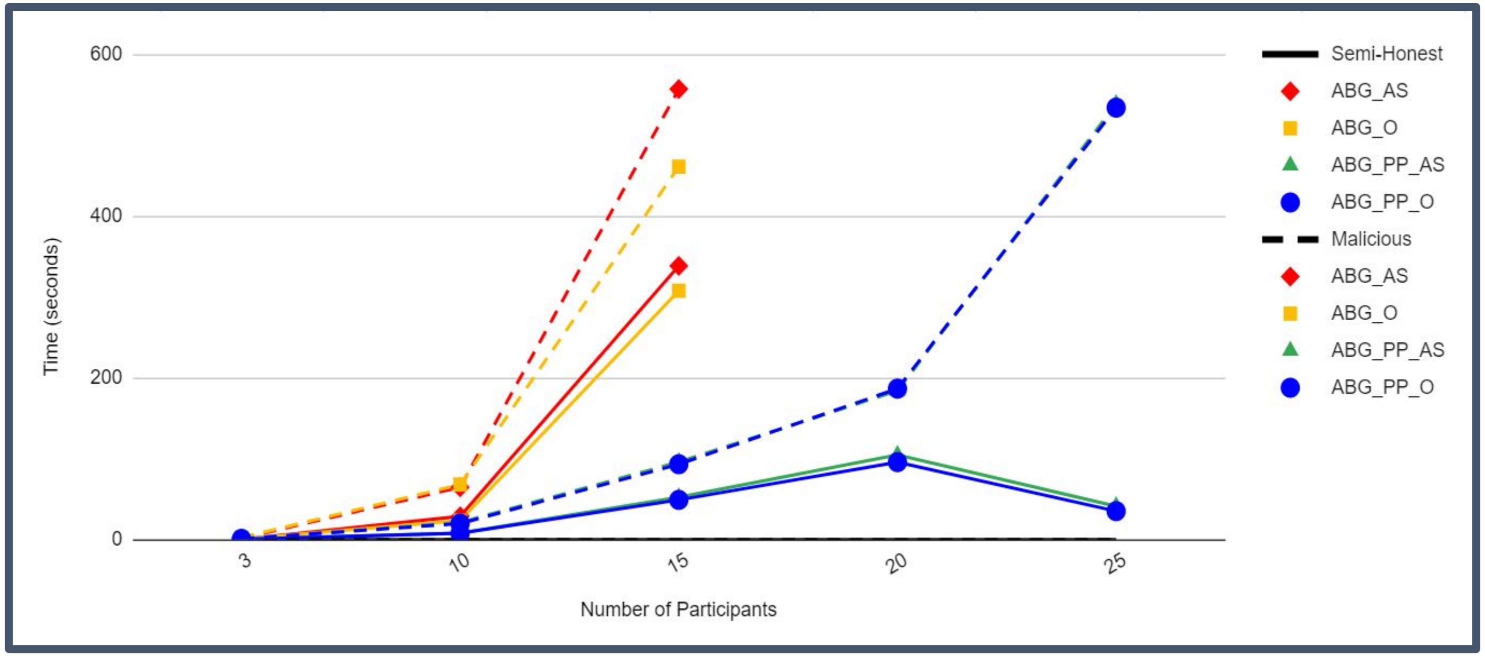


Number of Participates	3	5	8	13
Honest Majority	0			
ABG_AS	0.770233	2.83085	11.1971	123.676
ABG_O	0.701304	2.40429	11.1502	133.041
ABG_PP_AS	0.69959	1.508741	3.021501	18.976679
ABG_PP_O	0.598663	1.114001	2.584691	17.883079
Dishonest Majority	0			
ABG_AS	20.6485	176.488	1213.62	9101.84
ABG_O	15.5033	146.308	1143.84	8474.98
ABG_PP_AS	5.950353	37.581221	168.401621	1183.105779
ABG_PP_O	1.914263	15.216321	58.402621	512.614779

Honest vs Dishonest Majority (Semi-Honest)

https://github.com/cdduquette/CSCI6907-FinalProject/tree/main/Experiments/Honest_Dishonest_Semi

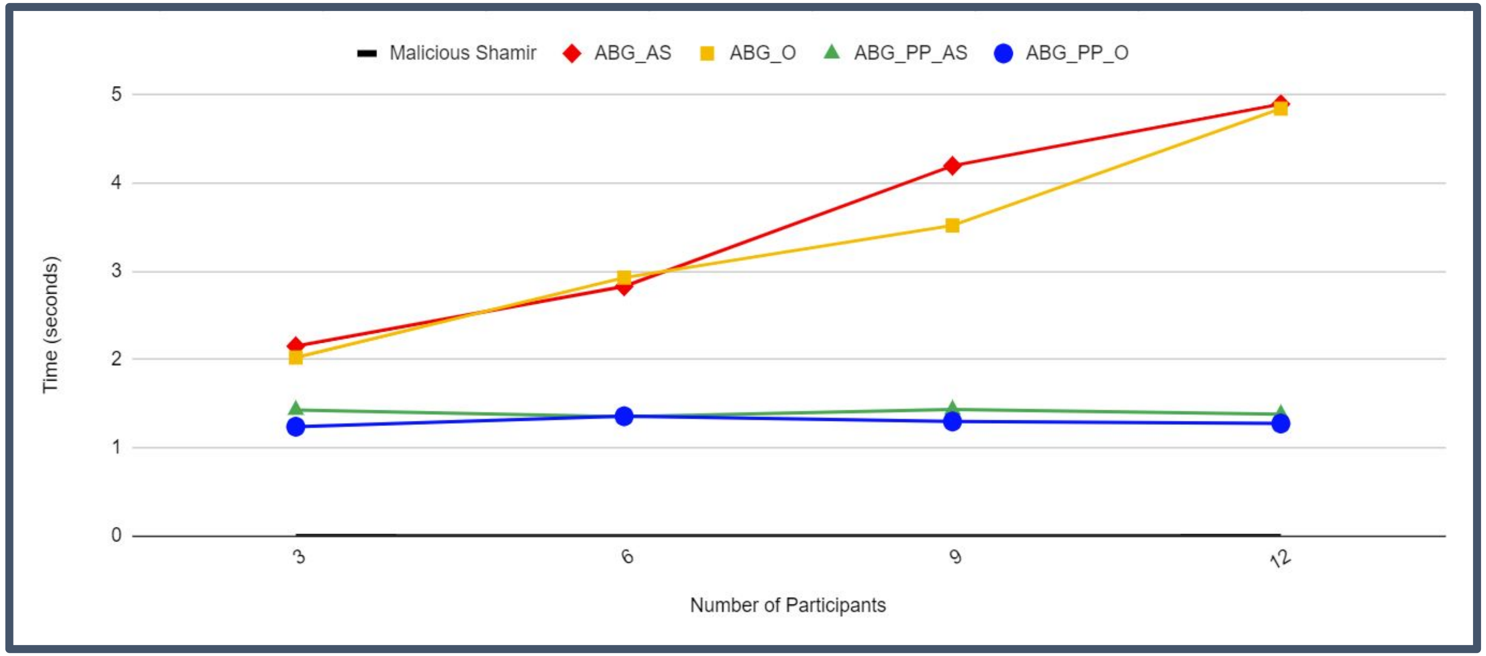
THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC



Number of Parti	3	10	15	20	25
Semi-Honest	0	0	0	0	0
ABG_AS	0.406746	28.6869	338.181	[redacted]	[redacted]
ABG_O	0.73117	23.4092	307.661	[redacted]	[redacted]
ABG_PP_AS	0.716832	7.994972	52.248946	104.69208	41.226073
ABG_PP_O	0.165694	7.822542	49.028246	95.70438	35.114973
Malicious	0	0	0	0	0
ABG_AS	1.43943	64.7595	557.064	[redacted]	[redacted]
ABG_O	2.29654	67.9681	460.982	[redacted]	[redacted]
ABG_PP_AS	0.895998	20.987522	95.806646	184.07908	538.329273
ABG_PP_O	0.871688	19.836322	93.134146	186.58908	533.965273

Semi-Honest vs Malicious Adversary (Shamir)

https://github.com/cdduquette/CSCI6907-FinalProject/tree/main/Experiments/SemiHonest_Malicious



Months Provide	3	6	9	12
Malicious Sham	0	0	0	0
ABG_AS	2.14885	2.82783	4.19503	4.89383
ABG_O	2.02092	2.9268	3.51835	4.84274
ABG_PP_AS	1.426963	1.348615	1.430834	1.378502
ABG_PP_O	1.236393	1.355015	1.295774	1.272572

Input Size: 3 Participants, Shamir*

https://github.com/cdduquette/CSCI6907-FinalProject/tree/main/Experiments/InputSize

Takeaways

- Do as much preprocessing as possible (i.e. set your application up to do as little in the MPC that you can allow)
  - This will save you about 6.5-8x amount of time
- Having the assurance that you will have an honest majority can save you a lot in computation time
- Maliciously secure for a honest majority only increases your computation time by 3x
- Input size will make the issues seen with party size worse (except in the case of preprocessing)

Overview

- Application Description
- Protocol
  - Input Data
  - Code Overview
  - Supporting Scripts
  - MPC Protocols Utilized
- Experiments & Results
- Challenges & Lessons Learned

Challenges

- Updating dishonest majority code to support multiple players
- VirtualBox Machine would crash with too many players
- High Run Times

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Challenges

The Lost Experiment

- Honest vs. Dishonest Majority with Malicious Security
- 3 parties were able to run with the MASCOT protocol
- 5 parties got a “Guru Meditation” error in VirtualBox
- Dishonest Majority with Malicious security had more steps/overhead than my machine could support

Lessons Learned

- Building an application in an abstracted fashion to run on any protocol
- Debugging Linux errors and supporting a virtual machine
- Analyzing various data points produced from a program to understand execution behavior
- The importance of minimizing “overhead” and “communication” when scaling an application to work with many parties

Thank you for your time!

Any Questions?

Contact: Courtney Duquette; cdduquette@gwu.edu; @Courtney Duquette

References

- Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20). Association for Computing Machinery, New York, NY, USA, 1575–1590. DOI:https://doi.org/10.1145/3372297.3417872
  - https://github.com/data61/MP-SPDZ
- M. Keller and E. Orsini and P. Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. Cryptology ePrint Archive, Report 2016/505, 2016.

Github: https://github.com/cdduquette/CSCI6907-FinalProject