SMPCEngine: An N-Party Implementation of the Secure Multiparty Private Computation Protocol

Nicholas D. McKinney

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2018

Reading Committee:

Dr. Anderson C.A. Nascimento, Chair

Dr. Martine de Cock

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

SMPCEngine: An N-Party Implementation of the Secure Multiparty Private Computation Protocol

Nicholas D. McKinney

Chair of the Supervisory Committee:
Dr. Anderson C.A. Nascimento
Institute of Technology

In this thesis, we implement a framework for secure multiparty computation. Our framework works in the commodity-based model, where the players running the distributed computation receive pre-distributed data from a trusted source during a setup phase. The framework allows secure multiplications of field elements, secure multiplications of matrices and matrix inversions. Differently from previous proposals, in our framework, the running times do not increase significantly as more players are added to the protocol.

We illustrate the power of our solution by applying it to the problem of privately estimating driver's drowsiness based on EEG data.

We show that our private solution is practical and achieves a similar accuracy as a solution in the clear, where there are no privacy concerns.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Anderson C.A. Nascimento and Dr. Martine de Cock, to whom I credit with my success.

DEDICATION

to my dear wife, Vanessa, for putting up with me

Chapter 1
INTRODUCTION

1.1 The Problem

We are awash in data. Data is generated at exponential rates. These data are of increasing value to researchers, with applications from the mundane commercial to the medical community. But these data are valuable for a good reason. With the right features, a skilled researcher can extract personal information, even if that information is anonymized. [1]

Some of this data is, of course, sensitive. Medical data, for instance, is protected by laws and regulations globally. In the United States a number of laws protects private information: the Privacy Act of 1974[3], the Health Insurance Portability and Accountability Act (HIPPA) of 1996[2], and others protect consumers in the United States. Most recently, the European Union has enacted regulations that strictly protect consumer data, including personally identifiable information, financial information, and medical information[1].

The challenge at hand is thus: how do we allow grant research access to the data in a way that preserves the privacy of the data owner. Here, there are three possibilities: grant the researchers direct access under some legal protections; grant the researchers indirect access through a data broker; or, grant the researchers indirect access through some privacy preserving technique. In this paper, we will explore this third option.

1.2 Motivation

The electroencephalogram (EEG) is one of the core measurement tools in neuroscience. It uses an array of sensors to measure a range of electrical impulses in the brain.[23] This tool allows researchers to experiment with real subjects, in real time, and gain significant insights

[1] http://www.nytimes.com/2006/08/09/technology/09aol.html

into the underlying workings of the human brain.

In the same way that the EEG and other tools are revolutionizing neuroscience, machine learning (ML) is revolutionizing the way researchers can approach data. Combined with advances in parallel and distributed computing, machine learning allows researchers to extract features and patterns from huge datasets.

Applying ML to EEG data has enormous potential to revolutionize neuroscience.[22] But having the tools to compute on EEG data is a necessary but not sufficient consideration. EEG data is highly sensitive; in the right hands, researchers can extract all sorts of features and neurological diagnoses from the same data.[23] Furthermore, data privacy laws, especially in the healthcare arena, further constrict the ability of researchers to use big-data techniques to extract gains.

To open this data securely, we must apply some form of privacy-preserving technique. In this work, we will focus on secure multiparty computation

1.3 Results

In order to fill this void, the proposes a distributed, privacy-preserving computational framework that is capable of computing on an arbitrary number of parties' data. In my implementation, we are able to grant access to an arbitrary number of parties.

Our proposed framework will enable the following computations in a privacy preserving way:

- Privacy preserving addition and multiplication of finite precision numbers.
- Privacy preserving multiplication of matrices
- Privacy preserving matrix inversion

We then proceed to apply our computational framework to the problem of estimating drivers drowsiness based on electroencephalogram (EEG) data. This problem can be solved

by using linear regression, which in turn can be shown to be reducible to privately computing matrix inversions.

Compared to previously proposed solutions for linear regression, our proposed framework scales horizontally with an increased number of parties. That is running time remains about the same as the number of involved parties increase. This contrasts with previous solutions where the running time increases quadratically with the number of computing parties. Finally, to the best of our knowledge, this thesis also presents the very first privacy preserving analysis of EEG signals by using techniques from secure multiparty computatiion.

Chapter 2

PRELIMINARY BACKGROUND

This chapter will provide the reader with basic background information necessary to understand the SMPC engine. This includes a brief description of the notation used, background mathematical concepts, and the mathematics behind additive-blinding secure multi-party computation.

2.1 Abstract Algebra Constructs

Because we will use a finite integer field as the basis security construct, it is important to remind the user of the basic properties of fields. Those unfamiliar with these topics are encouraged to consult [20] or another primer.

2.1.1 Groups

A group is a set G with some operation $\cdot$ that adheres to following axioms:

1. $\cdot$ is associative

2. There is some element e in G such that $\forall a \in G, a \cdot e = a$ and $e \cdot a = a$.

3. $\forall a \in G$, there is some element a^{-1} such that $a \cdot a^{-1} = e$ and $a^{-1} \cdot a = e$

4. If the property $a \cdot b = b \cdot a | \forall a, b \in G$, the group is said to be commutative or abelian.

A finite group is a group with a finite number of elements.

2.1.2 Rings

A ring is a set R with two operations, denoted addition ($+$) and multiplication ($\cdot$), but need not be arithmetic addition or multiplication, that adhere to the following axioms:

1. R over addition is abelian

2. Multiplication is associative.

3. $\forall a, b, c \in R, a(b+c) = ab + bc$ and $(b+c)a = ba + ca$.

4. A ring is said to be commutative if it is also associative over multiplication.

A ring has unity if there exists a neutral element 1 such that $\forall a \in R, a \cdot 1 = a$ and $1 \cdot a = a$.

An element $a \in R$ may have a multiplicative inverse a^{-1} if $a \cdot a^{-1} = a^{-1} \cdot a = 1$.

A finite ring is a ring with a finite number of elements.

2.1.3 Fields

A field is a commutative ring with unity F where every element $a \in F$ has a multiplicative inverse.

A finite field is a field with a finite number of elements.

2.2 Notation and Usage

1. $\mathbb{Z}_p$: represents a prime integer field of p elements.

2. $\mathbb{Q}_{\langle k,f \rangle}$ represents the set of real numbers with resolution 2^{-f}.

3. Square brackets $[[X]]_i$ denote an indexed set of elements coming from a finite field that add up to some value X such that $X = \sum_i [[X]]_i$.

4. Unless otherwise defined, we denote e as the integer range used when mapping into the field.

5. Unless otherwise defined, we denote f to be the fixed decimal precision of the field.

2.3 Multiparty Private Computation

We now informally introduce secure multiparty computation. For a detailed and rigorous set of definitions we refer to [12]. In a secure multiparty computation protocol, several parties $\{P_1, P_2 \ldots, P_n\}$ engage in a protocol such that each party holds an input $x_i, i \in \{1, \ldots, n\}$ and they jointly want to compute a function $f(x_1, x_2, \ldots, x_n)$ so that at the end of the protocol any subset C of corrupt players, of size at most $t < \frac{n}{2}$, learns no information beyond $x_j, y_j Pj \in C$ from executing the protocol, regardless of their computing power. The protocol is said to be correct if all players receive outputs that are correct based on the inputs supplied with overwhelmingly probability. In our work, we assume that the players are honest-but-curious. That is, they will stick to the protocol instructions but will otherwise try to obtain as much knowledge on the honest players' inputs as possible.

2.3.1 Dealing with non-integers

Inputs in secure multiparty computations usually come from finite fields or rings. However, inputs to realistic machine learning problem might come from continuous, unbounded sets, such as the real numbers. To solve this problem, I use the method put forward by [10] to map fixed-precision decimals to our field. We accomplish this by mapping a floating point decimal to a fixed precision decimal with the function $field : \mathbb{Q} \to \mathbb{Q}_{\langle k,f \rangle}, field(x) = x2^{-f} \mid x \in \mathbb{Z}_{\langle k \rangle}$. That is, we sample elements in $\mathbb{Q}$ at 2^{-f} intervals.

2.3.2 Dealing with negative integers

To handle negative numbers, we must "wrap around" our negative values to the top of the field. We do this by the mapping function $fld : \mathbb{Z}_{\langle k \rangle} \to \mathbb{Z}_p, fld(x) = x \mod p$ [10].

2.3.3 Field Size

Because will map our numbers into the field, we must ensure that we have sufficient space to ensure that we do not overflow our values when we multiply values. Thus, the range e must be sufficiently large to ensure this condition. If this is not done, we will inadvertently map our desired value to another, unexpected value in the finite field, and this will cause a cascade failure across all forward computations.

In addition to selecting a sufficiently large range, we must ensure that we have sufficient decimal space to avoid underflow. As we will show below, we introduce error when we (1) convert our number from a floating point decimal to a fixed precision decimal, and (2) when we truncate. If the amount of error exceeds the available decimal precision, we will underflow our field and introduce an unstable condition. This will cause our computations to fail.

Having defined our decimal and integer requirements, we choose a value $p \in \mathbb{Z} \mid p > 2^{e+f}, \forall a, b \in \mathbb{Z} : p \nmid a \vee p \nmid b$. That is, we choose any prime number p larger than 2^{e+f}. In practice, this will be the first prime that meets this criteria.

2.3.4 Shares

We divide a secret value X into shares X_i distributed over the parties engaged in a secure multiparty computation protocol, such that each share X_i gives no information about X. Moreover, the players can recover X by adding their shares $X = \sum_i [[X]]_i \mod p$. We denote the act of computing and distributing shares to each party P_i by $[[X]]_i$.

2.3.5 Secure Addition

One of the benefits of this system is that addition is "free" to us. That is, since we have defined each party to have shares of some variable X such that $X = \sum_i [[X]]_i$, we can conduct any additions locally, since $X + Y = \sum_i [[X]]_i + \sum_i [[Y]]_i = \sum_i [[X]]_i + [[Y]]_i$. This reduces the overall communications complexity and will not require the use of additive sharing.

2.4 Trusted Initializer

To provide secure resources without having to resort to a "Byzantine Generals" solution as found in [17], we use the commodity-based secure function evaluation technique described by Beaver in [5]. These parameterized, stateless client-server calls produce pre-distributed random data that can be used for speeding up distributed secure multiplication protocols as demonstrated by Beaver in [5].

Secure multiplication is known to be impossible with just two-parties if no further assumptions are in place. Thus, the pre-distributed data generated by the trusted initializer will be used to enable secure multiplication protocols. The trusted party is assumed to do not collude with players but is free otherwise to grab knowledge on the players' inputs.

Chapter 3
RELATED WORKS

3.1 MPC

There is a large body of work describing how to securely jointly compute a common answer among parties who do not wish to share their inputs [12]. Given that all the players engaged in an MPC protocol are connected by pairwise secret channels, MPC is possible if more than 2/3 of the players are honest. If a broadcast channel is available, MPC is possible with any dishonest minority. In case of a dishonest majority, further assumptions are necessary to enable MPC. In our case, we work in the so-called commodity-based model, first introduced in [6] and [5]. The commodity-based model assumes that a trusted third party creates commodity-sharing secrets to distributed among parties, providing a relatively straightforward and flexible tool for distributed computation. The work in [10] provides us the techniques to map decimal values into a finite ring in order to do the same computations.

In [11], we see a complete structure on how to use these tools to perform linear regressions among two parties. We build on this work using the generalizations found in [14] to extend this technique to n parties.

The other major technique, found in [25], creates "garbled" logic gates, where the parties construct logic gates for computation without disclosing their inputs. These circuits have the distinct advantage of being highly flexible, but this comes at a cost of performing poorly and difficult to extend beyond two parties.

3.2 Implementations of MPC

There are a handful of implementations found in the literature. Some systems, such as Fairplay, build on garbled circuits. A few, such as Sharemind, use additive sharing increase

performance of linear operations.

3.2.1 Fairplay and FairplayMP

There are only two feature-complete frameworks that implement secure multiparty computation. The first system, Fairplay (and its descendant FairplayMP)[7] is a robust platform for conducting privacy-preserving computations using garbled circuits. As noted elsewhere, garbled circuits grow in complexity quite rapidly, and research in the field related to > 2 party cases is still its infancy. While there have been recent advances[8], FairplayMP is still an academic project.

FairplayMP uses a constant round (8) computation to provide for n parties to construct gate tables. There are three categories of parties: input players, who contribute input values for comparison; computation players, who build the garbled circuit and emulate a third party; and result players, who receive some portion of output. The protocol is secure so long as an adversary cannot corrupt a majority of the computation players.

FairplayMP uses the Secure Function Definition Language (SFDL) to construct boolean circuits for computations among how the parties. SFDL is intended to be implementation-agnostic; a party does not need to know the implementation of a trusted third party.

3.2.2 Sharemind

The other system, Sharemind, is a commodity-based SMC system developed by Bogdanov, Laur and Willemson in [9]. This system, however, is also unsuited for the application I propose for two important reasons. First, Sharemind is restricted to three parties. Secondly, it maps data to a fixed ring of order $\mathbb{Z}_{2^{32}}$, which does not allow for the level of precision required to work with this data.

Sharemind derives its security from a number of techniques. First, all inputs are divided into shares among three processing units, denoted "miners". These miners receive as input shares of data from all inputs. Assuming that an adversary has not corrupted a majority of

the miners, we assume the inputs to be secure. Thus, while the performance gains found in Sharemind are great, they come at the cost of

3.2.3 Chameleon

Chameleon is a novel hybrid framework for secure function evaluation (SFE) that combines additive secret sharing and SFE to provide for performance gains over Sharemind.[21]. Chameleon is a two-party system that is based on additive secret sharing over a ring $\mathbb{Z}_{2^32}$. Chameleon is a hybrid system; it uses additve shares for computing linear functions and garbled circuits for computing non-linear functions. Thus, it is the most flexible system available, albeit limited in the number of parties that can use it.

Chapter 4

THE SECURE MULTIPARTY COMPUTATION ENGINE (SMPCENGINE)

4.1 Overview

SMPCEngine is an additive secret-sharing based framework system used to perform computations over a prime integer field $\mathbb{Z}_p$. The intended use case for this system is for training machine learning models, particularly linear regression.

Unlike other MPC systems based on additive secret sharing, SMPCEngine provides two distinct advantages. First, it supports an arbitrary number of parties without any significant loss in performance. Secondly, it is designed to conduct matrix operations using integer elements in a prime field, guaranteeing the existence of multiplicative inverses. The work in this chapter derives from the scheme proposed in [11].

Contrasting to [11], our solution introduces an additional third-party to provide a common data store, called the *Broadcast Agent*, that reduces the communications complexity among the parties from $O(n^2)$ to $O(n)$. Reducing the communications complexity thus allows a more practical scaling of the number of parties involved without violating our security model.

4.2 Framework Components

4.2.1 Clients

There are several players, referred to in this paper as clients, who hold private data that serves as input to distributed functions. Clients provide as input shares of variables and a public instruction set which defines their role. Because the protocol is asymmetric (i.e., not all clients perform the same computation), we assume that there is a leader who initiates distributed computations, but there are no restrictions on how that leader is chosen or on

which client may serve as a leader. In our implementation, we choose a leader arbitrarily prior to beginning any computation.

4.2.2 Trusted Initializer

To facilitate the generation of random data, we employ a "trusted initializer" to coordinate the generation of random data. We assume the trusted initializer exhibits the following properties:

1. The trusted initializer generates uniformly random data that is accurate

2. Data generated through the trusted initializer is privately communicated to each of the parties

The trusted initializer in this implementation runs online (*i.e.*, the random values are generated on-demand), but this need not be the case. The values used for the trusted initializer may be generated prior to computation. In any case, the values generated by the trusted initializer must not be disclosed to any party.

4.2.3 Broadcast Agent

One of my contributions to the existing models is the addition of a second party to facilitate communications between the parties. In the protocol described in [11], the parties share the results of their computations pairwise. In this framework, I replace this communication with a "bulletin board" type function I call the "Broadcast Agent". The Broadcast Agent servers two functions: (1) collect and sort information from all the parties, and (2) collate and broadcast the results with all the parties.

The principal benefit to using the Broadcast Agent it to reduce the communications complexity among the parties. In a two-party system, there is obviously no benefit; however, as the number of parties grows, I see the communications complexity scale from $O(n^2)$ using a traditional broadcast channel to $O(n)$ using the Broadcast Agent.

An authenticated broadcast agent can be implemented by using digital signatures.

4.3 Computations

4.3.1 Distributed Multiplication

Distributed multiplication uses the Beaver triples previously described to implement secure distributed multiplication. We use a modified form of the technique outlined in [14] to multiply matrices X, Y among n parties.

We begin the protocol by signaling to the trusted initializer the need for matrices U, V, and W. U and V are chosen uniformly at random of the form

$$U \in \mathbb{Z}_p^{\mathcal{N}_1 \times \mathcal{N}_2}, U \in \mathbb{Z}_p^{\mathcal{N}_2 \times \mathcal{N}_3}$$

, where $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ are the dimensions of our matrices to be multiplied, and $W = UV$. The trusted initializer distributes n shares of U, V, W to the parties.

Upon receipt of their shares, the parties locally compute $[[D]]_i = [[X]]_i - [[U]]_i$ and $[[E]]_i = [[Y]]_i - [[V]]_i$ then publish the values of D, E to the broadcast agent. Once it receives shares from all parties, the broadcast agent shares D, E with all parties.

Upon receipt of D, E, the parties compute $[[XY]]_i = [[W]]_i + E[[U]]_i + D[[V]]_i + DE$, thus completing the multiplication protocol.

4.3.2 Distributed Truncation

Multiplication this way introduces a scaling problem whenever we multiply two fixed precision real numbers that have been mapped to our field. Suppose we have values $x, y \in \mathbb{Z}_q = \tilde{x}2^f, \tilde{y}2^f \in Z_k$. If we add x and y, we see that $z = x + y = (\tilde{x} + \tilde{y})2^f$; however, when we multiply x and y, we see that $z = x + y = \tilde{x}2^f\tilde{y}2^f = \tilde{x}\tilde{y}2^{2f}$. That is, our result is scaled up by a factor of 2^f.

If multiplication is local, this is a simple matter of computing $z' = z2^{-f}$. In the secure multiparty world, we must do something else.

Given an integer field of size q and an input matrix of dimensions $\ell_1 \times \ell_2$

1. The trusted initializer selects a uniformly random matrix $R' \in \mathbb{Z}_q^{\ell_1 \times \ell_2}$ with elements in $\{0, \ldots, 2^f - 1\}$ and a uniformly random matrix $R'' \in \mathbb{Z}_q^{\ell_1 \times \ell_2}$ with elements in $\{0, \ldots, 2^{e+f} - 1\}$

2. The trusted initializer computes a matrix $R = R''2^f + R'$

3. The trusted initializer distributes shares $[[R]]_q$ and $[[R']]_q$ to all parties.

4. Given their share $[[W]]_q$, all parties locally compute $[[Z]]_q \leftarrow [[W]]_q + [[R]]_q$ and open Z to all parties.

5. Compute $C = Z +_{2^{e+2f}} 1$ and $C' = C \bmod 2^f$, where the modulo function is performed element-wise. Compute the secret share $[[S]]_q \leftarrow [[W]]_q + [[R']]_q + C'$

6. Recognizing that the truncated multiplicative inverse $x^{-1} = x(\frac{q+1}{2})^f, x \in \mathbb{Z}_q$, locally compute $[[T]]_q \leftarrow (\frac{q+1}{2})^f \cdot [[S]]_q$

4.3.3 Matrix Inversion

The most essential, and computationally most expensive, operation necessary for linear regression is matrix inversion. Recall that, given a matrix $\mathcal{M}$ and its additive shares $\mathcal{M}_i = \sum_i \mathcal{M}_i$, we have that $\mathcal{M}^{-1} \neq \sum_i \mathcal{M}_i^{-1}$. That is, the inverse of the sum of matrices is not the sum of the inverses. Thus, we must resort to other methods.

To compute the inverse of the matrix, I use the Newton-Raphson method to find the multiplicative inverse using the technique described in [11]. This technique is restricted to positive, semi-definite matrices, so the covariance matrix is perfectly suited for this application.

In order to invert a matrix A, we iteratively compute X_i, $i \in \{1, \ldots, n\}$ according to the following formula: form:

$$c = trace(A)$$

$$X_0 = c^{-1}I$$

$$X_{s+1} = X_x(2 - AX_s)$$

X_i converges quadratically to the inverse of A. The trace of A is chosen as a proxy for the eigenvalue of A. Because A is a positive, semi-definite matrix, the trace is a good proxy; unlike the eigenvalue, it can be computed without leaking information and can be computed locally by all parties.

The challenge with this method is determining when convergence occurs (i.e., $X_{s+1} = X_s$. Because parties only receives as an output of the protocol shares of A^{-1}, we must estimate when convergence occurs. An upper bound on the number of iterations required for convergence is $2f$.[11].

4.3.4 Rounding Errors

There are two sources of rounding error that are introduced in SMPC as a result of the design. First, there is error introduced in the conversion of floating-point decimals to fixed-precision decimals, which can potentially cause a loss of information. Second, the truncation protocol described above will introduce error. This error, unlike the rounding error in our first case, is self-correcting, and will introduce up to two bits of error from the expected fixed-precision value.

4.3.5 Initializing a computation

The current implementation of SMPCEngine assumes that there is a "leader" who will be responsible for initializing protocols where the actions are not symmetric across all parties, such as distributed multiplication and distributed truncation. The implementation also assumes that the same player will always initiate these computations due to the implementation of the message passing system.

Chapter 5

IMPLEMENTATION DETAILS AND BENCHMARKING

The following is a description of the design of SMPCEngine, including the specific classes used to implement its features. We implement SMPCEngine using a combination of a client-server architecture (for commodity-based services) and event-driven architecture (for internal message handling, and sending messages to and from the servers).

The system consists of two or more Clients, one Broadcast Agent, and one Trusted Initializer. There are no direct communications between Clients; all communications exist between the Clients and Trusted initializer, and the Clients and Broadcast Agent (see Figure 5.1). The Broadcast Agent and Trusted Initializer may exist on one more network servers. There is no requirement for a client to exist on a single machine; in practice, a Client represents a single dataset, so it is possible that multiple Client machines may be under the control of a single user.

5.1 Interprocess Communications

All communications both internally and among the system components are handled by passing messages between the clients and either the BroadcastAgent or the TrustedInitializer servers. Each message consists of a few basic components: an operation code, a list of operands, an optional byte array payload, a message sequence number (counter), and an upper limit if the operand takes a counter range. A full listing of the available opcodes can be found in Appendix A

Messages received by the client are first checked for timeliness. If the operation ascribed by the message can be computed, it is immediately moved into a work queue for computation; if not, it is placed into a waiting queue until the appropriate message(s) to arrive.

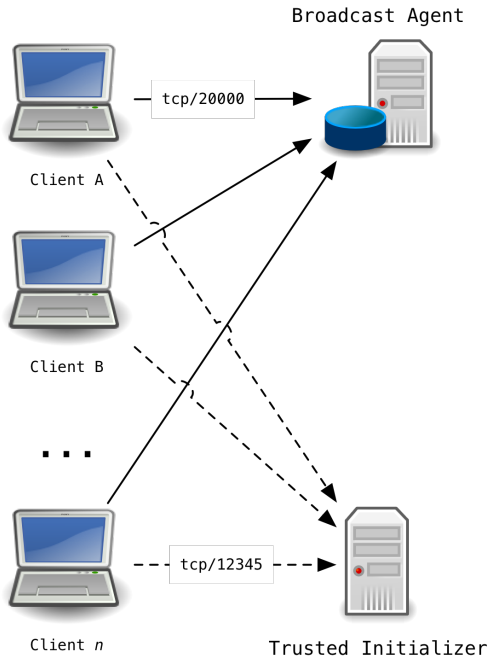

Figure 5.1: Major Components for SMPCEngine

5.2 Function Decomposition

Functions, or their decomposed parts, are loaded by the Client class through instruction files. This list of instructions corresponds to the desired function, as indicated by an opcode.

5.3 Client Structure

5.3.1 Client Class

Each client class initializes the top-level components necessary for the client: a datastore (the MathEngine class), a message processor (the ServerThread class), and the supporting class (the Protocol class). It also preloads variables and instructions from given directory paths. We initialize the Client class with the decimal precision and integer range from which it computes the approprate prime for $\mathbb{Z}_p$ using the Java BigInteger class's BigInteger.nextProbablePrime() method. This uses the Miller-Rabin primacy test to compute the

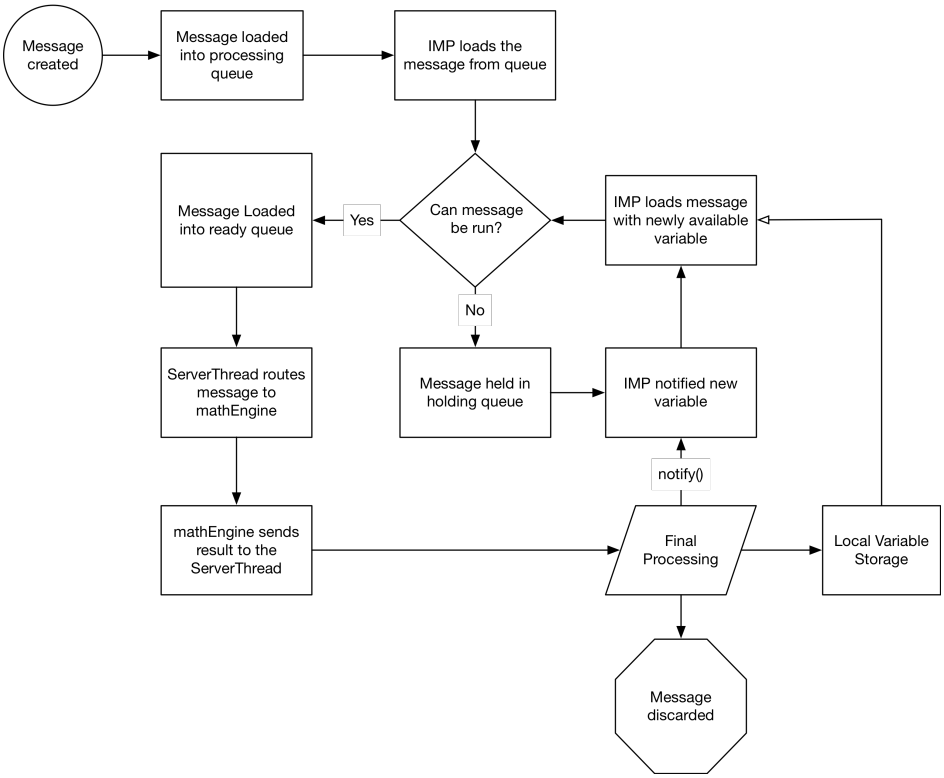

Figure 5.2: Lifecycle of an SMPCEngine Message

next larger prime to 2^{e+f} to a probability of 2^{-100}. [19]

The client class also loads variables from external files into the MathEngine class prior to starting. These files are simple: they consist of headerless comma-separated value (CSV) files, with the file name representing the variable name.

5.3.2 ServerThread Class

The heart of the Client class is the ServerThread class. This object is responsible for coordinating all activity for the client, including opening, maintaining, and closing the network connection to the TrustedInitializer and BroadcastAgents, processing all messages, controlling I/O, and managing the message handling queues. See Figure 5.4.

ServerThread also simplifies the problem by decomposing complex functions into smaller,

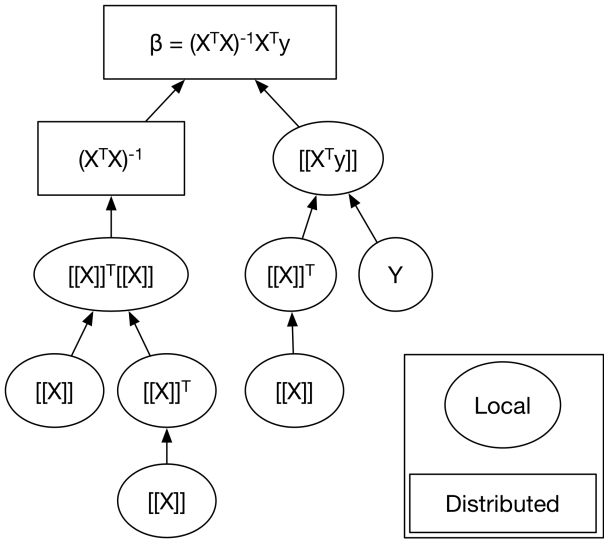

Figure 5.3: Decomposing a function into component operations

more manageable functions, and generates new Messages for each. For example, the linear regression function$\hat{\beta} = (X^T X)^{-1} X^T y$ can be decomposed into smaller operations: $(X^T X)^{-1}$ and $X^T y$, and these operations can be decomposed further, until we reduce the problem to its most fundamental building blocks. The outputs of these smaller programs then serve as inputs to the larger function, allowing highly complex computations to be both simplified and parallelized. See Figure 5.3 for a complete example.

ServerThread operates by maintaining two message handling queues. All incoming messages are placed into a message processing queue, which ServerThread will process in sequence. A Message object will consist of one or more inputs and up to one output. If any input(s) are not yet available, they are moved to a holding queue, which tracks the unsatisfied inputs of for each message. When a message is fully processed by ServerThread, it notifies the holding queue that its output (if any) is now available as an input for other messages. The holding queue will update its dependency tables accordingly and release any messages now available for processing back to that queue. Thus, every message will be in the processing queue no more than twice and in the holding queue no more than once.

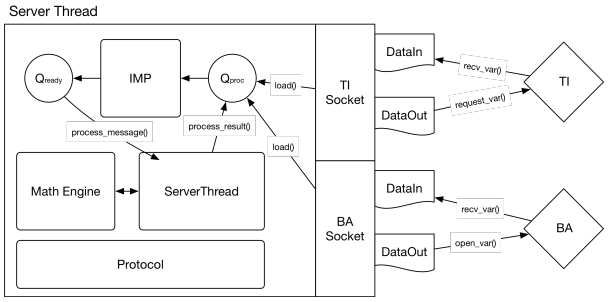

Figure 5.4: ServerThread high-level view

ServerThread does not interact with data directly; instead, it will use variable names as metadata, and pass these requests to the MathEngine for actual processing.

As ServerThread processes Message objects, it may create additional Messages for its own consumption or as requests from the external server resources. For example, the distributed trunction requires commodity variables from the TrustedInitializer, so it will first create a Message that will inform the TrustedInitializer of the need for a new commodity. ServerThread will also create messages for each of the individual operations necessary for the trunctation protocol. Finally, ServerThread will send a message requesting all other clients initiate their part of the protocol. In total, initiating distributed truncation creates no less than nine new Messages for processing; it will create additional constant variables if they do not yet exist locally, for a maximum value of 11 Messages. These threads are the put on the processing queue.

When ServerThread has no messages in its processing queue, the thread will sleep. When it receives a new message from either the trusted initializer or the broadcast agent, it will wake up, look for new lively messages, and continue processing messages.

5.3.3 MathEngine Class

The MathEngine class organizes all data for each ServerThread (as well as within the TrustedInitializer and BroadcastAgent classes). It can create new SMPCVariables, initiates all

computations on variables, controls variable input/output, holds the parameters about the field in which we are operating, and holds information about the client itself, including the unique identifier for the client.

The MathEngine is also responsible for reporting the availability of a required variable. If a variable is not yet ready for processing, MathEngine will report the variable is not ready for computation (i.e., would be a null reference) to the ServerThread, which should then place the requesting message into the holding queue until the variable becomes available.

5.3.4 SMPCVariable Class

The SMPCVariable class is the structure for storing and computing data. It holds the values for each variable and the includes all the aritmetic operation methods.

5.3.5 Protocol Class

The Protocol provides common functionality to across all other classes. These are described below:

Opcode definitions

The most common usage for the Protocol class is opcode mapping. The class provides a static naming definition for every opcode (see Appendix A) and the number of expected arguments for each.

Message I/O

The Protocol class also the common methods for handling message input and output. It is the common step used for moving data to or from the network sockets created by the ServerThread and the MathEngine datastore.

```
public class Message implements Serializable {
    int msg;
```

```
    String[] vars;
    byte[] data;
    private int counter;
    private int limit;
}
```

Listing 5.1: Objects within a *Message* object

Object Serialization

Finally, the Protocol class provides common for object serialization of Message objects used for inter-process communication. Using a Java-native object exchange simplifies the overall code base and avoids using another absstraction serialization format such as JSON.

Object serialization is used not only to package and transmit messages to other clients, but to serialize messages for storage in other messages. This technique is not used much, but it is useful when broadcasting instructions to all clients through the broadcast agent, such as a shutdown message.

5.3.6 TrustedInitializer Class

The TrustedInitializer class is the first of two servers used to support the commodity-based system described in [5] (i.e., Beaver triples). It structured similarly to the ServerThread class, but has a reduced set of instructions available and only uses one processing queue. The TrustedInitializer takes as input a request from a client (by convention, a "leader"), and returns to all clients shares of the requested variables.

On startup, the TrustedInitializer is given the field parameters (integer range, decimal precision, and the number of parties in the computation group. TrustedInitializer then produces Beaver triples accoring to the requestor's input. The request consists of a list of variables names (including sequence number tags) and pairs of dimensions for the matrices to be created. It also matches an opcode for the type of triple required: distributed

multiplication requires a different set of triples than does distributed truncation. [14]

5.3.7 BroadcastAgent Class

The BroadcastAgent class is the simpler of the supporting servers and is overall the simplest system. It consists of a highly simplified ServerThread, but it need only maintain a simple data store that adds shares of a given variable. When it has received a share from each client, BroadcastAgent will send a specially formatted (and expected) message to all clients with the value of the aggregate number, then discard the variable from its local datastore.

5.3.8 Containerization

To simplify the deployment of the application, we encapsulated each functional unit (client, trusted initializer, and broadcast agent) into a Docker container `https://www.docker.com`. These provide a lightweight, portable virtualized platform through which to deploy the application. The reader is encouraged to read more about containers generally at the link provided above.

5.4 Test Environment

For testing, we ran all clients, along with the trusted initializer on a MacBook Pro (2015) with 2.5GHz Intel Core i7 process and 16GB RAM, running MacOS 10.13.3.

5.5 Benchmarking

5.5.1 Distributed Multiplication

As expected, we observer that matrix multiplication exhibits cubic growth as a function of the matrix size, ranging from 180ms for a 10×10 matrix to 410ms for a 100×100 matrix. This calculation is made on a local network without any network latency; if we are to incorporate latency, we would expect this to be an additional sum of up to $3t_{latency}$, where $t_{latency}$ represents round trip time to the slowest network resource. See Figure 5.5.

5.5.2 Matrix Inversion

We also observe that matrix inversion exhibits cubic growth as a function of matrix size. This is nature result of distributed multiplication protocol; however, in addition to two distributed multiplication calls per iteration, matrix inversion also includes local multiplication, putting additional pressure on computational resources.

5.6 Practical Considerations

5.6.1 Overflow/Underflow

As mentioned above, there is a risk of overflow/underflow when working with number in the field. In order to avoid the possibility of this condition, sufficient space must be allotted for both the integer range and decimal precision. This requires some prior knowledge of the data set and the number of parties involved.

5.6.2 Inversion Round Count

We use the number of rounds defined in [11] as $2f$ to set a guaranteed upper bound for the number of rounds necessary for the matrix inversion protocol to reach convergence. However, through empirical testing, we find that the lower bound for this problem may be as low as f.

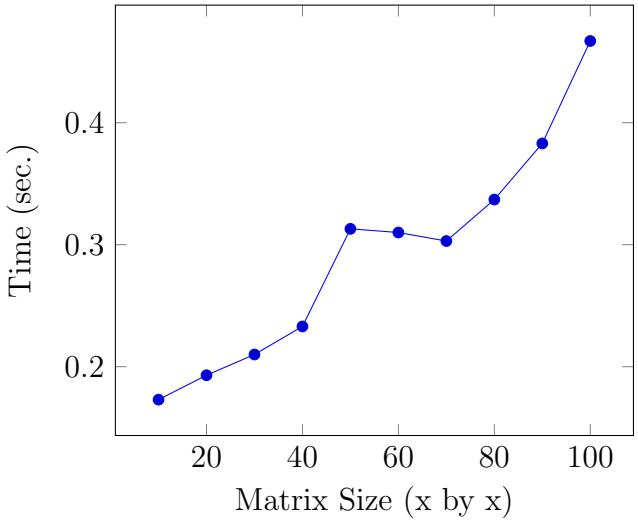

Figure 5.5: Distributed Matrix Multiplication, 64-bit range, 64-bit precision, 2-parties

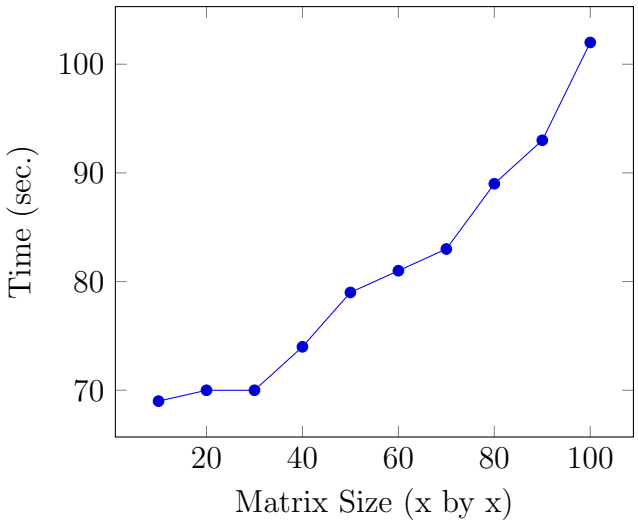

Figure 5.6: Matrix Inversion Runtime, 64-bit range, 64-bit precision, 2-parties

Chapter 6

USE CASE: PRIVACY PRESERVING MACHINE LEARNING ON BRAIN-MACHINE INTERFACES

We now look to use to apply SMPCEngine to solve problems in a secure manner. We have at our disposal all the tools necessary to perform linear regression against large matrix datasets. In the case described below, we will privately compute a model to estimate drivers' drowseness based on EEG sygnals by using a linear regression model. We also and compare this result to the linear regression model computed in the cleat. We show that SMPCEngine is capable of securely building a regression model without ever disclosing the private data of our test subjects.

6.1 Description of Problem

Electroencephalogram (EEG) data is highly feature-rich, highly sensitive data. It is the gold standard for a number of medical research disciplines, from neuroscientists to behavioral psychologists. The practical application of real-time systems is of extraordinary value. It is simple to measure, portable, and feature-rich.

In [24] researchers used EEG data to test drivers's drowsiness. They were also interested in knowing if they could apply high-performance algorithms to solve a very real problem in Brain-Computer Interface research: how to transition from laboratory settings to near-real time applicaitons under ordinary circumstances? In our paper, the question we ask is simple: can SMPCEngine help with this transition?

The original experiment looks to apply transfer learning techniques, specifically domain adaptation, to reduce the size of the training set and reduce the overall amount of classification data required to train the model, thus reducing the overall computational requirement

of the model and increase the real-world applicability. The authors propose a set of novel algorithms to reduce the data input requirements for regression. These algorithms are beyond the current capabilities of our system. Instead, we turn to one of the baseline cases (BL1) employed by the authors to see if we can develop a model in a way that is still computationally practicable while producing a "good enough" result.

We show that training a model in the clear for predicting drivers' drowsiness gives results that are close to privately training the model with our secure multiparty computational framework.

6.2 Dataset

The 15 datasets used in [24] represent preprocessed EEG data with 30 features extracted from patients. These drivers were placed in a driving simulator for 60 to 90 minutes, providing 1200 data points of labeled data. These features are used to determine if a driver is drowsy, expressed as a probability function.

We are provided 15 sets of feature domains *thetaPower* from 15 different subjects[1], each row in the table labeled with a known drowsiness level RT. Each domain consists of measurements from 32 EEG channels (30 input channels plus 2 earlobe return channels) representing the voltage differential between each channel and the earlobe returns for a total of 30 features. *thetaPower* is the collection of measurements collected as measured over a 500Hz cycle.[24]. The dataset is processed by the original authors in to reduce noise and amplifiy the signal. See Listing 6.1 for a data sample.

```
4.00127769693, 4.12568638432, 2.8456941294, 3.18665221943,
   3.68743154192, 5.44557858687, 7.8558961827, 8.60644031941,
   6.76531376511, 9.49035499228, 5.70401852654, 6.23603187607,
   7.98725327573, 15.3111976776, 15.4983511004, 2.80814473211,
   3.71214650279, 8.27875064525, -0.0885832661153, 0.553904343206,
```

[1]In this experiment, we only work with 14 datasets due to a technical formattting error with one of the datasets provided.

```
 0.182328683092, 2.20087604485, 2.16613156093, 2.69974016048,
1.02112473911, 0.793920019019, 0.696538094853,
-0.0773147041433, 0.367153644364, 0.233021090173
```

Listing 6.1: Sample Data

We take the labeled data from the subjects and thus integrate data from the source domain with data from the target domain. We assume that the features selected by the original authors are appropriate. Our objective is to compare the accuracy of the model using SMPCEngine to perform the same computations.

6.3 Description of Testing

For simplicity, we only look at the experiment BL1 from the paper: combine data from 14 parties and score against the 15th. If we cannot make this computation in a reasonable time, it is unlikely that we would be able to achieve the line-speed performance of the more sophisticated models proposed in [24].

We modify BL1 slightly from the original work. In that case, they train their linear regression with a ridge regression model ($\sigma = 0.01$). For simplicity, we train our model with an ordinary least square (OLS) regression model. The performance difference between the two in SMPCEngine is minimal, as all addition operations are local. We use the first 14 subject domains as our source domain with the intent of integrating the 15th with our target domain.

Similar to [11], we are interested in using our EEG input matrix $thetaPower$ and known drowsiness vector RT to compute an estimated regression parameter vector β and independent term ϵ. We want to obtain the regression model:

$$RT = thetaPower \cdot \beta + \epsilon$$

It can be show that the following estimation $\hat{\beta}$ for the regression vector β minimizes the

empirical risk function:

$$\hat{\beta} = (thetaPower^T thetaPower)^{-1} thetaPower^T RT$$

. Where ϵ is already included in the vector $\hat{\beta}$. We will compute $\hat{\beta}$ in the clear and using SMPCEngine, using 14 of the datasets. We will then compare the predictions by computing the root mean squared error (RMS) between the predicted values and the measured results provided.

SMPCEngine provides the tools for computing this regression model: we have the ability to compute basic matrix operations, conduct matrix multiplication distributed among user shares, and perform matrix inversion distributed among user shares. We now map ths model into our framework.

6.3.1 Instruction Set

To begin, we must model the linear regression in a form that is well represented in our model. We gain a computational advantage by recognizing that for a co-variance matrix

$$\mathcal{M} = \sum_{i=1}^{n} \mathcal{M}_i = \sum_{i=1}^{n} \mathcal{M}_i^T \mathcal{M}_i$$

. Thus, each party can compute locally their own share of $\mathcal{M}_i^T \mathcal{M}_i$. This is a substantial computational saving, reducing the total number of distributed multiplication protocols by a factor of $2f$.

Thus if we decompose or model, we develop the following expressions:

$$\hat{\beta} = (thetaPower^T thetaPower)^{-1} thetaPower^T RT$$

$$c_1 = thetaPower^T thetaPower; c_2 = c_1^{-1}; c_3 = thetaPower^T RT, c_4 = c_2 c_3$$

These are then loaded into our model using the expressions shown in Listings 6.2 and 6.3. Note that opcode *15* is the command to shutdown and write the listed variables to disk upon completion.

```
// Compute transpose of thetaPower and store as X_tr
23,thetaPower, X_tr
// Locally multiply X_tr*thetaPower and store as XTX2
21,X_tr, thetaPower, XTX2
// Locally scale down XTX2 by a factor of 2^{-f} and store as XTX
31,XTX2, XTX
// Computed distributed inverse of XTX and store as XTX_inv
40,XTX, XTX_inv
// Locally multiply X_tr and RT, store it as GAMMA2
21,X_tr, RT, GAMMA2
// Locally scale down GAMMA2 by a factor of 2^{-f} and store as XTX
31,GAMMA2, GAMMA
// Compute distributed product of XTX_inf * GAMMA, truncate, and store
   ↪ as BETA
34,XTX_inv, GAMMA, BETA
// Output values to file and quit
15,XTX_inv,thetaPower,RT,XTX_inv,XTX,XTX2,X_tr,GAMMA,BETA
```

Listing 6.2: Coordinator Instructions

```
// Compute transpose of thetaPower and store as X_tr
23,thetaPower, X_tr
// Locally multiply X_tr*thetaPower and store as XTX2
21,X_tr, thetaPower, XTX2
// Locally scale down XTX2 by a factor of 2^{-f} and store as XTX
31,XTX2, XTX
// Locally multiply X_tr and RT, store it as GAMMA2
21,X_tr, RT, GAMMA2
// Locally scale down GAMMA2 by a factor of 2^{-f} and store as
```

```
    ↪ XTX
31,GAMMA2, GAMMA
// Output values to file and quit
15,XTX_inv,thetaPower,RT,XTX_inv,XTX,XTX2,X_tr,GAMMA,BETA
```

Listing 6.3: Responder Instructions

To test the performance of SMPCEngine, we run the linear regression in two scenarios. The first test, we run against an incresing number of patients (2 - 13) to test the horizontal scaling of the system. In the second test, we select a fixed number of drivers (in this case, 7 is chosen arbitrarily) and increase the decimal precision in increments of 8-bits to test how well it scales. All clients run on the same local machine, so network latency is negligible ($< 0.1ms$).

6.4 The Results

The results from training are consistent with what we would expect. In the case of scaling the number of drivers, we see a slight increase in the running time as the number of drivers increase. This is mostly an artifact of the testing suite. Each container is started in sequence on the same machine, and the overhead of starting a new container amounts to approximately 2-3 seconds per driver. See figure 6.2

In the second case, where we scale the decimal precision, we see a linear increase in the running time. This is also to be expected. Recall that the most costly operation in our linear regression system is matrix inversion. Each iteration consists of two distributed multiplications, and this must be iterated $2f$ times to ensure convergence. Thus, we would expect this term to dominate and be reflected in the runtime.

Of important note is a 10-second cooldown period hard-coded in the ServerThread class. This is an arbitrary timeout designed to ensure that each Client class is able to write any requested values to disk before terminating, artificially increasing the runtime performance by approximately 10 seconds. In summary, we note that we are able to train our regression

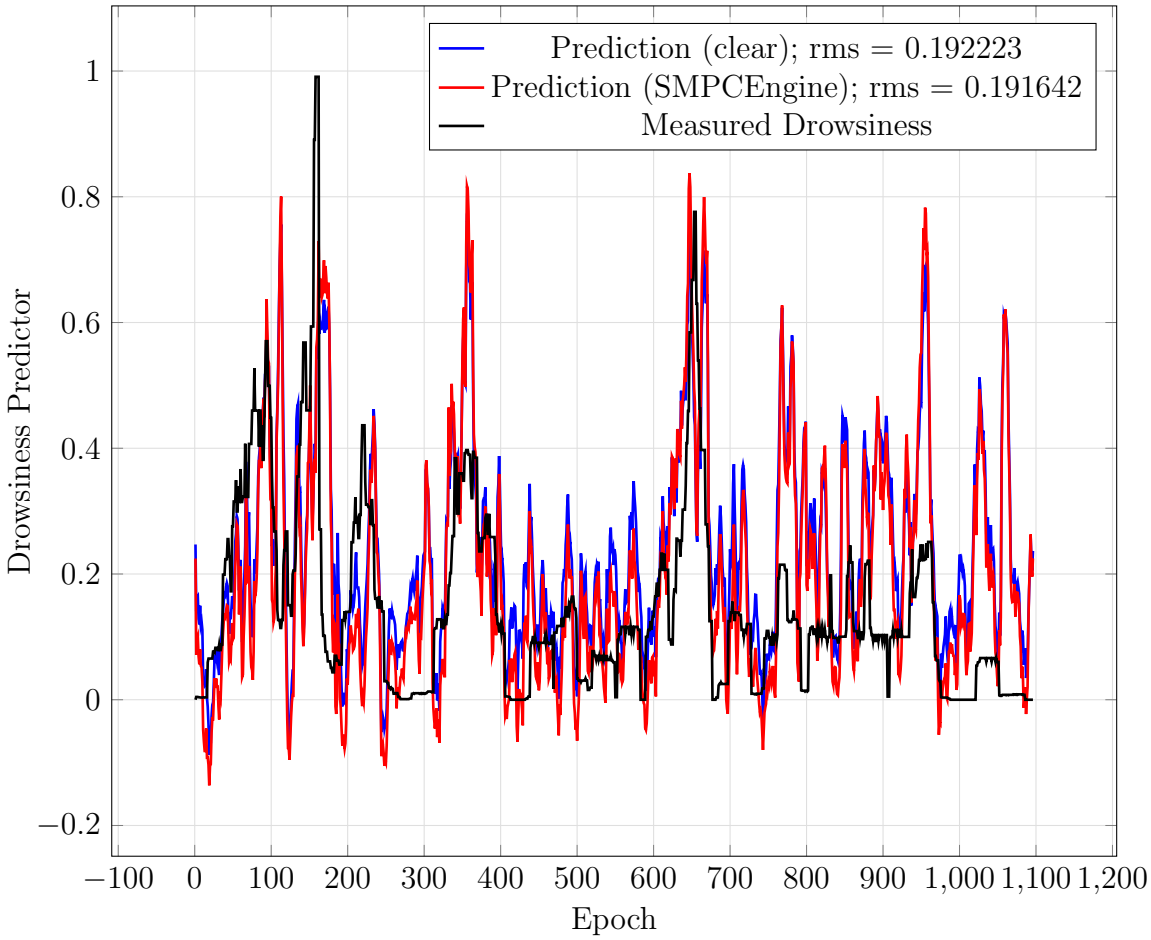

Figure 6.1: Comparing SMPCEngine to unsecured computations, trained with 13 parties of data, 64-bit range, 64-bit precision

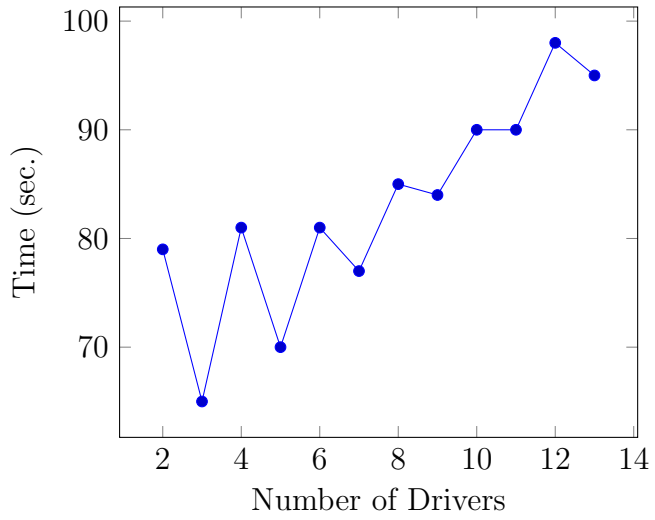

Figure 6.2: Number of Players v. Runtime, 64-bit range, 64-bit precision (Sequential Startup)

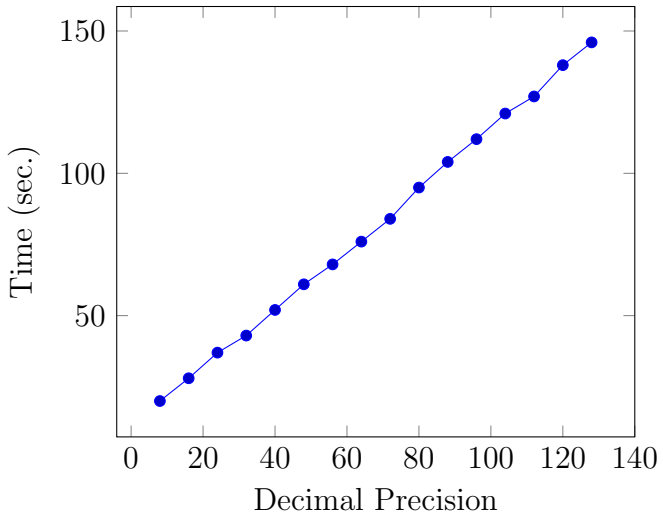

Figure 6.3: Decimal Precision v. Runtime, 7-players, 64-bit range

model within a couple of minutes using the entire dataset, although there is certainly always room to improve performance.

Having computed $\hat{\beta}$ using both through unsecure local computations and using SMPCEngine, we now compare our results to those computed in the clear. As we in Figure 6.1, both the prediction using the clear computation method and using SMPCEngine both have substantial error in their prediction, but this value is consistent with the results found in [24]. Futhermore, the rms for both methods is measured at 0.19, indicating that they are providing comparable results.

6.5 Observations

One important note on the size of the field. As discussed above, it is of utmost importance to ensure to select sufficient size to avoid overflow/underflow conditions. This requires the parties involved to come to some agreement when defining the parameters used. This could be as simple as defining a sufficiently large space for a single party based on expected values, then linearly scaling the problem for n parties.

Chapter 7

CONCLUSIONS

7.1 Utility of the Model

In this work we have implemented a highly scalable framework that allows us to run secure multiparty computations for the following tasks:

- Secure multiplications over elements of a finite field
- Secure multiplications of matrices defined over the same field.
- Secure inversion of matrices

We then applied our framework for the problem of estimating driver's drowsiness based on EEG data. We approached this problem by privately implementing a linear regression model. We showed that our privately learned model is almost identical to a model learned in the clear. We tested our results with real data.

There are, of course, some limitations worth considering. When developing a training experiment, a researcher must take into consideration the expected range of data prior to defining the range and precision parameters. While there are no technical limitations on scaling, large number of parties imply that the size of data sets will grow, requiring a large decimal precision for scale.

7.2 Future Work

My implementation of MPC shows great promise for future work. Aside from the normal programing optimizations, there are a several areas ripe for improvement. In the current implementation, matrix multiplication is done naively, with an runtime performance of $\mathcal{O}(N^3)$.

When working with large covariance matricies, this will lead to a significant reduction in performance. We can reduce this even futher using the techniques in [16] and achieve a performance of approximately $\mathcal{O}(N^2.3754770)$. This will help reduce matrix multiplication time, especially during the iteratively expensive matrix inversion protocol.

The most expensive computation I use in this framework is matrix inversion, which requires a maximum $2f$ iterations of Newton-Raphson root finding in order ensure convergence. Obviously, there is substantial benefit to reducing the cost of this operation. There is already general server-assisted method described in [4], but it is yet to be applied to the additive-sharing MPC I use here.

BIBLIOGRAPHY

[1] General data protection regulation (gdpr), regulation (eu) 2016/679.

[2] Health insurance portability and accountability act of 1996, pub.l. 104191, 110 stat. 1936.

[3] Privacy act of 1974, 5 u.s.c. 552a.

[4] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 201–209, New York, NY, USA, 1989. ACM.

[5] Donald Beaver. Commodity-based cryptography (extended abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 446–455, New York, NY, USA, 1997. ACM.

[6] Donald Beaver. Server-assisted cryptography. In *Proceedings of the 1998 Workshop on New Security Paradigms*, NSPW '98, pages 92–106, New York, NY, USA, 1998. ACM.

[7] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: A system for secure multi-party computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 257–266, New York, NY, USA, 2008. ACM.

[8] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 578–590, New York, NY, USA, 2016. ACM.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Octavian Catrina and Amitabh Saxena. *Secure Computation with Fixed-Point Numbers*, pages 35–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[11] Martine de Cock, Rafael Dowsley, Anderson C.A. Nascimento, and Stacey C. Newman. Fast, privacy preserving linear regression over distributed datasets based on pre-distributed data. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, AISec '15, pages 3–14, New York, NY, USA, 2015. ACM.

[12] R. Cramer, I. Damgård, and Jesper Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[13] Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In *Proceedings of the 8th Australasian Conference on Information Security and Privacy*, ACISP'03, pages 350–364, Berlin, Heidelberg, 2003. Springer-Verlag.

[14] Rafael Baião Dowsley. *Cryptography Based on Correlated Data: Foundations and Practice Cryptography Based on Correlated Data: Foundations and Practice*. PhD thesis, Karlsruhle Institute of Technology, 2016.

[15] Martin Franz. *Secure Computations on Non-Integer Values Secure Computations on Non-Integer Values*. PhD thesis, Technische Universität Darmstadt, 2011.

[16] François Le Gall. Powers of tensors and fast matrix multiplication. *CoRR*, abs/1401.7714, 2014.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[19] Oracle. Java BigInteger, 2017.

[20] Charles C. Pinter. *A Book of Abstract Algebra*. Dover Publications, 1990.

[21] M. Sadegh Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. *ArXiv e-prints*, January 2018.

[22] B. T. Skinner, H. T. Nguyen, and D. K. Liu. Classification of eeg signals using a genetic-based machine learning classifier. In *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 3120–3123, Aug 2007.

[23] D. Puthankattil Subha, Paul K. Joseph, Rajendra Acharya U, and Choo Min Lim. Eeg signal analysis: A survey. *J. Med. Syst.*, 34(2):195–212, April 2010.

[24] D. Wu, V. J. Lawhern, S. Gordon, B. J. Lance, and C. T. Lin. Driver drowsiness estimation from eeg signals using online weighted adaptation regularization for regression (owarr). *IEEE Transactions on Fuzzy Systems*, PP(99):1–1, 2016.

[25] Andrew C. Yao. Protocols for secure computations. In *27th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1986.

Appendix A

PROTOCOL OPCODES

```
// Local arithmetic operations
public static final int MATRIX_EYE = 16;  // Generate identity matrix
public static final int MATRIX_MUL_NO_SCALE = 17;  // Local matrix multiplication without scale-down

// Secure matrix inversion protocol (Newton-Raphson)
static final int R_OUTPUT_VAR = 2;  // Remotely output variable to remote console
static final int COPY_VAR = 3;  // Copy variable to another space
static final int PUT_VAR = 4;  // Store an object as a local variable
static final int POP_VAR = 5;  // Pull any messages from the processing queue that might be waiting
static final int WRITE_ZQ_FILE = 13;  // Write the specified variable to a file as field element
static final int WRITE_REAL_FILE = 14;  // Write the specified variable to a file as a decimal
static final int SHUTDOWN = 15;  // Shutdown server

// Variable operations
static final int OUTPUT_ZQ_VAR = 1;  // Output variable to local console as field element
static final int OUTPUT_REAL_VAR = 2;  // Output variable to local console as decimal
static final int MATRIX_ADD_STACK = 18;  // Add a set of variables at once
static final int MATRIX_ADD_MOD = 19;  // Local matrix modulo addition
static final int MATRIX_SUB_MOD = 20;  // Local matrix modulo subtraction
static final int MATRIX_MUL_MOD = 21;  // Local matrix modulo multiplication
static final int MATRIX_TR = 22;  // Local matrix trace
static final int MATRIX_TRANS = 23;  // Local matrix transpose
static final int MATRIX_MUL_BY_EYE = 24;  // Multiply local variable by identity matrix
static final int MATRIX_MODULO = 25;  // Element-wise matrix modulo
static final int MATRIX_ADD = 26;  // Local matrix addition
static final int MATRIX_SUB = 27;  // Local matrix subtraction
static final int MATRIX_SCALE = 31;  // Scale matrix by a factor of 2^f
static final int MATRIX_INV = 0x30;  // Local matrix inversion

// Secure Multiplication protocol over Integers Z_p
static final int S_MAT_DMM_INIT = 48;  // Code for initiating party to intiate DMM protocol
static final int S_MAT_DMM_RESP = 49;  // Code for party to respond to DMM protocol

// Secure Multiplication then truncation protocol
static final int S_MAT_DMM_TRNC_INIT = 34;  // Code for initiating party to initiate DMM with truncation
static final int S_MAT_DMM_TRNC_RESP = 35;  // Code for party to respond to DMM with truncation

// Compute inverse of trace of a covariance matrix
static final int S_MAT_TR_INV_INIT = 38;  // Code to initiate distributed truncation protocol
static final int S_MAT_TR_INV_RESP = 39;  // Code to respond to distributed truncation protocol
static final int S_MAT_INV_Q_INIT = 40;  // Code to initiate matrix inversion
static final int S_MAT_INV_Q_RESP = 41;  // Code to responde to matrix inversion

// Broadcast agent communication
static final int REGISTER_BA = 160;  // Require the client to register with the broadcast agent
static final int REG_RESP_BA = 161;  // Client response to registration request
```

```
static final int PUBLISH = 162;  // Publish local shares over broadcast
static final int BROADCAST = 163;  // Send variable to all clients
static final int BA_FLUSH = 164;  // Clear tracking flag
static final int BCAST_MSG = 165;  // Relay a remote command to all other agents

// Shared operations
static final int MATRIX_ADD_OPEN = 176;  // Local matrix addition with broadcast
static final int MATRIX_SUB_OPEN = 177;  // Local matrix subtraction with broadcast
static final int MATRIX_MUL_OPEN = 178;  // Local matrix multiplication with broadcast
static final int MATRIX_ADD_MOD_OPEN = 179;  // Local matrix modulo addition with broadcast
static final int MATRIX_SUB_MOD_OPEN = 180;  // Local matrix modulo subtraction
static final int MATRIX_MUL_MOD_OPEN = 181;  // Local matrix modulo multiplication

// Trusted initializer interface
static final int REQUIRE_TI = 208;
static final int REQUEST_TI = 209;
static final int RESPOND_TI = 210;
static final int GEN_DMM_TI = 211;
static final int GEN_TRNC_TI = 212;
static final int GEN_INV_TI = 213;
static final int REQUEST_R = 214;  // Request invertible matrix R from the trusted initializer
static final int RESPOND_R = 215;  // Response packet containing share of invertible matrix R

// Data transfer operations
static final int REQUIRE_VAR = 224;
static final int REQUEST_VAR = 225;
static final int RESPOND_VAR = 226;
public static final int DELETE_VAR = 227;  // Cleanup old, unnecessary temporary variables

// Error messages
public static final int VAR_NOT_RDY = 240;
```