Dissertation Type: Research - GG1K

UNIVERSITY OF BRISTOL

DEPARTMENT OF COMPUTER SCIENCE

Secure Two Party Computation

A practical comparison of recent protocols

Author : Nicholas Tutte

Supervisor : Prof. Nigel Smart

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Monday 4th May, 2015

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of **GG1K** in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Nicholas Tutte, Monday 4th May, 2015

Prelude

Executive Summary

Abstract

We present implementations of several recently proposed Secure Two Party Computation protocols and perform experiments for the purpose of comparison. We also give and implement a novel variant combining two of the aforementioned protocols.

For several of these protocols our implementation is, to the best of our knowledge, the first. As such until now we have had only theoretical comparisons of these protocols, making it difficult to know which approach is the most promising and deserving of further research.

In particular we have implemented the protocols described in [3], [5] and [6] and additionally we experiment with modifying [5] to use [6] instead of [3] as a sub-protocol.

[RESULTS SUMMARY HERE]

Summary of Achievements

- We implemented the protocols described in [3, 5], to the best of our knowledge these are the first implementations of these protocols.
- We implemented the protocol described in [6]. Huang et al. have produced an implementation in Java this cannot be fairly compared to my C implementations of the other protocols. Furthermore they only performed preliminary experiments, we provide more extensive results.
- We experimented with modifying the sub-protocol used for the Secure computation to detect cheating in [5], exchanging the use of [3] for [6] and making other changes as necessary for this approach to work.
- We have argued informally for the security of the modified protocol.
- We have run practical comparisons of all the implemented protocols on a variety of circuits/computations and provided some analysis of the results.

Supporting Technologies

- Unless otherwise stated all tests have been run upon the Bristol Cryptography Group's Diffie and Hellman machines. These machines are identical and have dedicated network cards for communications between each other.
- All code is in either C or C++, using the OpenMP library for parallelism in the shared memory paradigm. Furthermore AES-NI support is enabled.
- Extensive use has been made of the GNU Multi Precision Arithmetic Library.
- The net code was provided by my supervisor Prof. Nigel Smart.

- The AES implementation I use was mostly provided by my supervisor Prof. Nigel Smart (coded by Dr. Dan Page). Though I have extended this as it did not provide non-AES-NI decryption.
- The SHA-2 implementation used was taken from [INSERT CITATION HERE].
- For much of the random number generation we have used the implementation of ISAAC provided by [32].

Notational Glossary

$\mathbb{G} = (G, g, q) \leftarrow \zeta(1^n)$ informally speaking this indicates choosing a group such that the 'security' of the group is n bits. We define the group as the tuple (G, g, q) where G is the set of all elements of the group, g is a set that generates G (we deal primarily in Cyclic groups so usually this will be a single element) and finally q which is the order of the group.

$\|$ indicates concatenation. $\oplus$ denotes XOR.

Throughout take S to be a statistical security parameter.

Contents

Chapter 1

Introduction

Secure multi-party computation(SMC) is a long standing problem in Cryptography. We have a set of parties who wish to cooperate to compute some function on inputs distributed across the parties. However, these parties distrust one another and do not wish their inputs to reveal their inputs to the other parties. Using SMC we can perform the desired computation without any party ever knowing the other's inputs.

A commonly used example is the Millionaires problem. A group of rich persons wish to find out who among them is the richest, but do not wish to tell each other how much they are worth. Here the parties are the rich individuals, each party's inputs is their net worth and the function will return the identifier of the individual with the greatest input. Additionally, at the end of the computation no party should be able to divine anything about another party's inputs, apart from what can be inferred from their own input and the output.

For many years Yao's protocol [15] has been the most attractive avenue of theoretical research, mainly due to its conceptual simplicity and constant round nature. In particular recent work has endeavoured to produce variants of Yao's protocol that can provide security in the presence of malicious adversaries ([1], [3], [5], [6], [12], [13], [14]) and to improve the efficiency of the original protocol itself ([10], [11]).

The purpose of this Dissertation is to provide practical implementations for several of the more recent protocols, in some cases to first implementations, in order to run experiment to measure and compare the performance of the protcols.

Our contributions are as follows,

- To the best of our knowledge we provide the first implementations of the protocols of [3] and [5].
- We also provide an implementation of the protocol described in [6].
- We put forward and implement a modification of [5] using our implementation of [6] for the sub-computation rather than [3] as originally proposed. Further we informally argue this modification maintains security.
- We measure the performance of each protocol on several of the classic SMC benchmark computations and give analysis of the results.

Chapter 2

Background to Secure Multiparty Computation

2.1 Security Properties

There are three main properties that we wish to achieve with any SMC protocol,

- Privacy, the only knowledge parties gain from participating is the output.
- Correctness, the output is indeed that of the intended function.
- Independence of inputs, no party can choose its inputs as the function of other parties inputs.

In this sense we define the goal of an adversary to compromise one or more of these properties.

We compare any protocol to the *ideal* execution, in which the parties submit their inputs to a universally trusted and incorruptible external party via secure channels. This trusted party then computes the value of the function and returns the output to the relevant parties.

Informally we say that the protocol is secure if no adversary can attack the protocol with more success than they can achieve against the ideal model.

It is worth noting that some functions inherently leak information about the inputs of the other parties. For example in a two-party addition both parties can easily recover the other party's input after the computation has been run by subtracting their own input from the result. In these cases SMC is not at fault so we do not concern ourselves greatly with this scenario.

Occasionally a fourth property is proposed, namely *fairness*, meaning if one party gets their output then all parties get their output. However, generally this is ignored due being thought to be impossible outside a synchronous communication model as any party can stop participating in the protocol at any time.

2.2 Security levels

Having established the goals of the adversary and how we can measure if said adversary has a valid attack, we next deal with the capabilities of the adversary. We use three main models to describe the capability of the adversary.

2.2.1 Semi-honest Adversary

The Semi-honest adversary is the weakest adversary, with very limited capabilities. The Semi-honest adversary has also been referred to as "honest but curious", because in this case the adversary is not allowed to deviate from the established protocol (i.e. they are honest), but at the same time they will do their best to compromise one of the aforementioned security properties by examining the data they have legitimate access to. This is in some ways analogous to the classic "passive" adversary.

Example

At first it can be difficult to think of applications where only Semi-Honest security is required, but such applications do exist. Semi-Honest security is of use when in situations where it is not in the interest of either party to cheat.

So take the example of parties who wish to decide whether they should cooperate on a particular project. More concretely maybe two drug companies are considering cooperating in a particular area of research, but first need to establish that they have the combined expertises required. To do this without unnecessarily revealing information about their capabilities to the other company they might run a legally binding Secure Computation.

In this case undetected cheating could lead to the parties committing to a project they do not have the expertises to complete, this is clearly not in the interests of the parties so it is reasonable to assume that both parties will act honestly.

2.2.2 Malicious Adversary

The Malicious adversary is allowed to employ any polynomial time strategy and is not bounded by the protocol (they can run arbitrary code instead), furthermore the Malicious adversary does not care if it is caught cheating so long as it achieves its goal in the process. This is in some ways analogous to the classic "active" adversary.

Example

Security in the presence of malicious adversaries is much sought after, and is useful in many more scenarios. Suppose a pair of persons wish to compute the intersection of sets they each hold but only wish to reveal those elements in both sets, keeping the rest secret.

A malicious adversary might wish to reveal all of the elements in the other party's set. If the adversary can rig the condition in the computation checking whether an element is in both sets they can get all elements returned. Clearly in this case the adversary has something to gain and so we cannot count on the adversary being honest.

2.2.3 Covert Adversary

The Covert adversary model is very similar to the Malicious model, again bounded by polynomial time with freedom to ignore the protocol. However, in this case the adversary is adverse to being caught cheating and is therefore slightly weaker than the Malicious adversary. A Covert adversary will accept a certain probability of detection, this probability represents the point at which the expected benefit of cheating successfully outweighs the expected punishment for getting caught, effectively a game theory problem [17].

We call the probability that a Covert adversary will be caught the "deterrent probability", usually denoted using ϵ. Often protocols providing security against Covert adversaries take a Security parameter which varies the probability of detecting cheating.

Example

This model can be thought of as a compromise between practicality and malicious security and is usually appropriate when there are tangible consequences to a party being caught cheating. For example consider a consortium of companies who wish to cooperate in some way that benefits participants and that if one is caught cheating in the computations they are publicly expelled from the consortium.

In this case a sufficiently high deterrent probability will mean the chance of being caught is so high that the risk of being caught outweighs the benefits to be gained by cheating.

2.3 Applications of SMC

Here we take time to motivate the study of SMC by giving several actual or proposed applications.

2.3.1 Secret Auctions - Danish Beets

In Denmark a significant number of farmers are contracted to grow sugar beets for Danisco (a Danish bio-products company). Farmers can trade contracts amongst themselves (effectively sub-contracting the production of the beets), bidding for these sub-contracts is done via a "double auction".

Farmers do not wish to expose their bids as this gives information about their financial state to Danisco and so refused to accept Danisco as a trusted auctioneer. Similarly all other parties (e.g. Farmer union) already involved are in some way disqualified from playing the role of a universally trusted party. Rather than rely on a completely uninvolved party like an external auction house (an expensive option) the farmers use an SMC-based approached described in [7]. Since 2008 this auction has been ran multiple times.

As far as team behind this auction are aware this was the first large scale application of SMC to a real world problem, this application example in particular is important as it is a concrete practical example of SMC being used to solve a problem demonstrating this is not just a Cryptological gimmick.

2.3.2 Distributed secrets

Consider the growing use of physical tokens in user authentication, e.g. the RSA SecurID. When each SecurID token is activated the seed generated for that token is loaded to the relevant server (RSA Authentication Manager), then when authentication is needed both the server and the token compute 'something' using the aforementioned seed. However, this means that in the event of the server being breached and the seed being compromised the physical tokens will need to be replaced. Clearly this is undesirable, being both expensive both in terms of up front clean up costs and reputation.

In the above scenario we clearly need to store the secret(the seed) somewhere. If we can split the seed across multiple servers and have these servers perform the computation as an SMC problem we can remove the single point of failure and increase the cost to an attacker. As the secret is now distributed an attacker will now have to

compromise multiple servers.

Such a service is in development by Dyadic Security who provide a technical primer on applying SMC to this problem [8] (full disclosure, my supervisor Prof. Nigel Smart is a co-founder of Dyadic).

2.3.3 PROCEED - Computation on encrypted data

Recently US Defence Advanced Research Projects Agency (DARPA) ended a programme called PROCEED. The eventual goal being the ability to efficiently perform computations on encrypted data without knowledge of the data. This could be used by companies such as Google to continue to provide services requiring computation on personal data without intruding on the privacy on their users.

The PROCEED program is not restricted to SMC, it also considers Fully Homomorphic Encryption. At present DARPA claim that SMC slows the computation by at least 2 orders of magnitude whilst FHE slows it by nearly 10 orders of magnitude [9].

Chapter 3

Technical Background

Here we detail two of the two main common components of a Yao Garbled Circuits based system. The Yao Garbled Circuits themselves and Oblivious transfer.

3.1 Oblivious Transfer

Oblivious Transfers are vitally important for SMC and in particular Yao's Protocol that we shall be looking at later. Oblivious Transfers protocols allow for one party(the Receiver) to get one out-of two values from another party (the Sender). The Receiver is oblivious to the other value, and the Sender is oblivious to which value the Receiver received.

We shall first talk abstractly about what functionality Oblivious Transfers should have before then giving two concrete examples of how to perform Oblivious Transfer.

Oblivious Transfers were first suggested by Rabin in [18]. We define the functionality of a 1-out-of-2 OT protocol in Figure 3.1. As we shall see later, Oblivious Transfers are vital to Yao Garbled Circuits.

Receiver	**Sender**
Inputs : $b \in \{0,1\}$	Inputs : $x_0, x_1 \in \{0,1\}^l$
Outputs : x_b	Outputs : $\emptyset$

Figure 3.1: Formal definition of the functionality of a one-out-of-two OT protocol.

The security of Oblivious Transfers is defined in a similar way to that of SMC, the focus is on Semi-honest(passive) and Malicious(active) adversaries. Security against these adversaries is usually either computational or statistical.

A protocol is considered secure with regards to Semi-honest adversaries if neither a Semi-honest adversary in the sender role cannot learn anything about which value the receiver requested, nor can a Semi-honest adversary in the role of the Receiver learn anything about values other than the one it requested. The protocol being secure against Malicious adversaries is defined by the obvious extension of the Semi-honest case.

We primarily use OTs based on the Peikert-Vaikuntanathan-Waters OT (PVW-OT) from [21] or more precisely the modifications of the PVW-OT suggested in [3] and expanded on in [5]. However, we also use the Naor-Pinkas (NP-OT) from [22] for the protocol in [6].

3.1.1 Naor-Pinkas Oblivious Transfer

Here we describe the Naor-Pinkas Oblivious Transfer as put forward in [6] that is used in the Huang, Katz and Evans protocol later implemented, for a full description includ-

ing proofs of security see [22].

We assume that we have the usual OT inputs and parties. That is a Sender S who holds two input bit strings denoted $x_0, x_1 \in \{0,1\}^*$ and a Receiver who has a $b \in \{0,1\}$ representing the input that the Receiver wishes to uncover.

On top of these inputs the parties share a group $\mathbb{G}$ as an auxiliary input. We denote the group by $(\mathbb{G}, g, q)$ where $< g >= \mathbb{G}$ and q is the order of the group.

See Figure 3.2 for the functionality of the Naor-Pinkas OT.

Shared Auxiliary Input

$\mathbb{G}$ a group for which the CDH assumption is believed to hold. C, an element of G generated by the Sender.

Receiver	**Sender**
Inputs : $b \in \{0,1\}$	Inputs : $x_0, x_1 \in \{0,1\}^l$
Outputs : x_b	Outputs : $\emptyset$

Figure 3.2: Formal definition of the functionality of The Naor-Pinkas Oblivious Transfer.

The Naor-Pinkas OT is known to be simulatable against a malicious Sender assuming the CDH holds in the group. However, it is only known to provide *privacy* against a malicious Receiver, the question of whether it is simulatable against such an adversary is as yet unanswered.

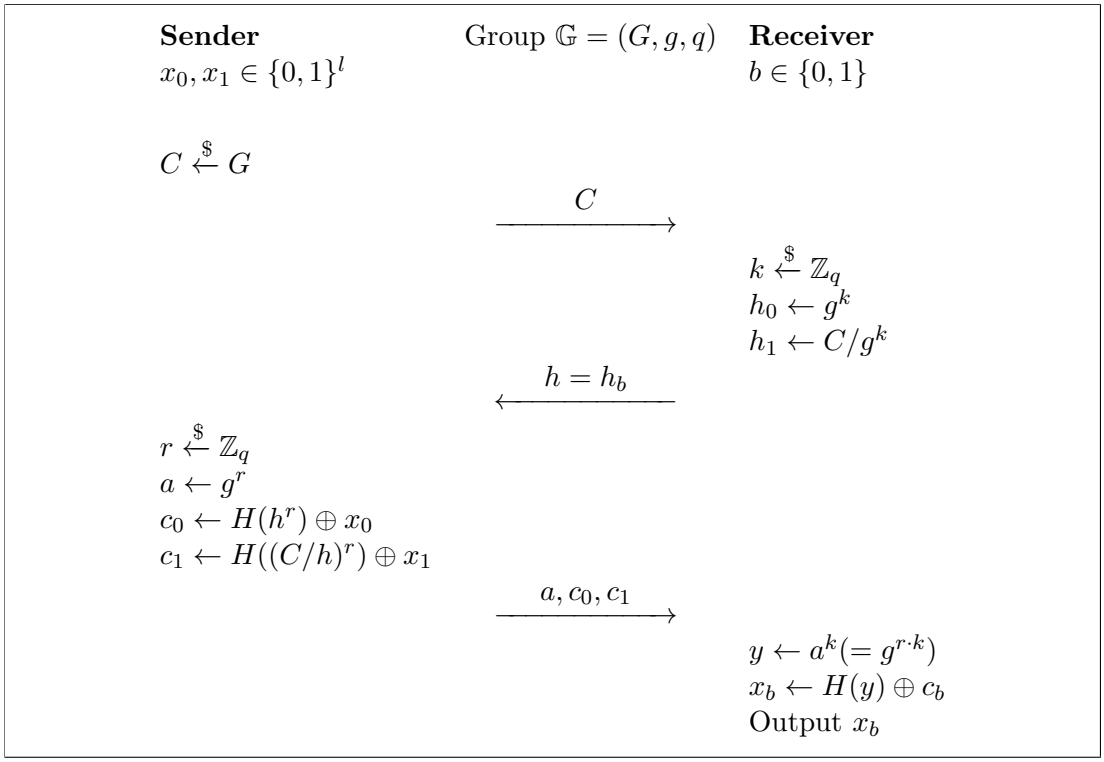

Figure 3.3: The Naor-Pinkas Oblivious Transfer protocol. Note that the same C can be used for multiple OTs.

3.1.2 Peikert - Vaikuntanathan - Waters Oblivious Transfer

The basis of the Oblivious transfer protocol we shall be using comes from [21], in particular we shall be using the realisation of the dual-mode cryptosystem based on

Decisional Diffie-Hellman problem. Whilst I shall not go into depth on this protocol we shall give a broad overview of the dual-mode cryptosystem.

High level concepts

The Peikert-Vaikuntanathan-Waters (PVW) OT has at its core the concept of a messy key. This is a key such that under encryption by this key all information about the plaintext is lost, moreover messy keys are indistinguishable from normal valid (*neat*) keys that do not obliterate the plaintext. It does not take much to see how these could be useful for an Oblivious Transfer scheme.

The PVW OT is constructed in such a way we can ensure one of the keys will be a messy key, whilst the other will be a neat key. Furthermore the Receiving party can control which key will be messy and which will be neat, allowing the Receiving party to choose which input to uncover.

Dual-Mode Encryption

In [21] Peikert-Vaikuntanathan and Water's describe a new abstraction, a Dual-mode cryptosystem. This system requires a setup phase in which the parties produce a public *Common Reference String* and potentially a trapdoor. Peikert et al. state that this trapdoor information is only needed for the security proof as such we will mostly ignore these details.

The setup also chooses one of two modes (*messy* and *decryption*).. Once this setup is complete this cryptosystem is very similar to a normal Public Key system, with one major difference, Peikert et al. introduce the concept of encryption branches.

The key generation algorithm takes a parameter $\sigma \in \{0, 1\}$, and returns a public/secret key pair. Similarly when encrypting using the public key produced by the key generation one must also specify a $b \in \{0, 1\}$.

Plaintexts can be decrypted if encrypted with $b = \sigma$ (the decryptable branch of pk), but plaintexts encrypted with $b \neq \sigma$ cannot be decrypted (we call this the messy branch of pk). Additionally when carrying out an encryption using a public key provided by the other party you cannot tell which branch is decryptable.

Depending on which mode is selected during setup the trapdoor returned allows subversion of one of these properties. If the system is in messy mode the trapdoor allows the encrypting party to distinguish when the branch input to the key generation that produced a public key was. If the system is in decryptable mode the trapdoor allows the decryption of both branches.

In Figure 3.4 we more formally define the abstract system and in particular what functions are required.

Dual-mode encryption using DDH

Having described the abstract form of a Dual-mode cryptosystem we now give a concrete realisation. This realisation requires a group, as usual we define this group as $\mathbb{G} = (G, g, q)$ where g generates G and $|g| = q$. Further we require that the the group is chosen such that we believe the Decisional Diffie-Hellman problem be hard for this group.

- **Setup**(1^n, μ) - This function takes a security parameter 1^n and a bit $\mu \in \{0,1\}$ which defines which mode (messy or decryptable). The function should output the CRS and trapdoor information (crs, t). All other functions take this crs as an implicit parameter.

  In order to ease notation later we define two separate functions depending on μ. **SetupMessy**(1^n) := **Setup**$(1^n, 0)$ and **SetupDec**(1^n) := **Setup**$(1^n, 1)$.

- **KeyGen**(σ) - This function takes a single input of a bit $\sigma \in \{0,1\}$ and outputs (pk, sk) where pk is a public key for encryption and sk is a secret key that allows decryption of plaintexts encrypted using pk on the branch σ.

- **Enc**(m, pk, b) - This function takes a message $m \in \{0,1\}^l$, a public key pk and a bit $b \in \{0,1\}$. It returns the encryption of m under pk on branch b.

- **Dec**(c, sk) - This function takes a ciphertext c and a secret key sk. It outputs a message $m' \in \{0,1\}^l$.

- **FindMessy**(pk, t) - This function takes a public key pk and a messy mode trapdoor t. The function then outputs a bit $b \in \{0,1\}$ indicating which branch of pk is messy.

- **TrapKeyGen**(t) - This function takes decryptable mode trapdoor t and is an alternative key generation. The function outputs (pk, sk_0, sk_1), note that it outputs two secret keys, one for each branch. These secret keys allow the decryption of both branches of pk.

Figure 3.4: The abstract functions defining a Dual-mode cryptosystem.

Before giving concrete definitions of the functions we need a few primitives relating to DDH cryptosystems.

Randomisation Take G to be an arbitrary group, we shall use multiplicative notation, such that the group is of order p where p is prime. We then define $DLOG_G(x) = \{(g, g^x) : g \in G\}$. Put another way $DLOG_G(x)$ is the set of all pairs of elements in G such that the discrete log of the second over the first is x.

We define a probabilistic algorithm *Randomise* that takes generators $g, h \in G$ and elements $g', h' \in G$. The algorithm then outputs $(u, v) \in G^2$ such that the following properties hold,

- If $(g, g'), (h, h') \in DLOG_G(x)$ for some $x \in \mathbb{Z}_p$ then (u, v) is chosen from $DLOG_G(x)$ uniformly at random.
- If $(g, g') \in DLOG_G(x)$ and $(h, h') \in DLOG_G(y)$ for some distinct $x, y \in \mathbb{Z}_p$ then (u, v) is chosen uniformly at random from G^2.

In particular we define $Randomise(g, h, g', h')$ as follows, choose $s, t \xleftarrow{\$} \mathbb{Z}_p$ independently of one another, then let $u = g^s \cdot h^t$ and $v = (g')^s \cdot (h')^t$.
A full proof that this instantiation of *Randomise* is given in [21], suffice to say the main idea of the proof is to re-write h as a power of g which we can do as g generates G.

Having defined the function *Randomise* Peikert et al. next defined a simple asymmetric cryptosystem based on it.

DDH-Randomise Cryptosystem As with all asymmetric cryptosystems we need to define three algorithms namely key generation, encryption and decryption.

- **DDH-KeyGen**(1^n) - This function takes a security parameter and chooses a group $\mathbb{G} = (G, g, q) \leftarrow \gamma(1^n)$, this group G is the message space. For our purposes this group will be an Elliptic curve group of size $\sim 2^{2n}$.

  Then choose another generator of the group $h \in G$ and an exponent $x \in \mathbb{Z}_p$. Then set $pk = (g, h, g^x, h^x)$ and $sk = x$.

- **DDH-Enc**$(pk,\ m)$ - This function takes a message $m \in \{0,1\}^l$, a public key pk. The public key should be parsed as (g, h, g', h').

  The function computes $(u, v) \leftarrow \mathit{Randomise}(g, h, g', h')$ and then outputs the ciphertext $(u, v \cdot m)$.

- **DDH-Dec**$(sk,\ c)$ - This function takes a ciphertext c and a secret key sk, parse c as (c_0, c_1). Output a decryption $m' = c_1/c_0^{sk}$.

Figure 3.5: A simple asymmetric cryptosystem based on *Randomise* in a DDH group. It is important to note here that this system is messy if

Dual-mode Cryptosystem based on Randomise Finally Peikert et al. give instantiations of the functions specified in 3.4, using the DDH cryptosystem just defined. These instantiations can be seen in Figure 3.6

- **Setup**$(1^n, \mu)$ - Recall that for notational purposes we split this function depending on the value of μ. However, both branches of this function begin by choosing a group $\mathbb{G} = (G, g, p) \leftarrow \zeta(1^n)$. Then the Decryption and Messy Setup functions diverge.

  SetupDec(1^n) - Choose a random generator $g_0 \in G$, a random *non-zero* exponent $y \in \mathbb{Z}_p$ and let $g_1 = g_0^y$. Then take another random *non-zero* exponent $x \in \mathbb{Z}_p$ and let $h_b = g_b^x$ for $b \in \{0,1\}$. The outputs are then $(crs, t) = ((g_0, h_0, g_1, h_1), y)$.

  SetupMessy(1^n) - Choose a pair of random generators $g_0, g_1 \in G$ and a pair of random *distinct and non-zero* exponents $x_0, x_1 \in \mathbb{Z}_p$. Let $h_b = g_b^{x_b}$ for $b \in \{0,1\}$. The outputs are then $(crs, t) = ((g_0, h_0, g_1, h_1), (x_0, x_1))$.

- **KeyGen**(σ) - Firstly choose $r \xleftarrow{\$} \mathbb{Z}_p$. Then set $g = g_\sigma^r$ and $h = h_\sigma^r$. Finally set $pk = (g, h)$ and $sk = r$ and output (pk, sk).

- **Enc**$(m,\ pk,\ b)$ - Parse pk as (g, h). Let $pk_b = (g_b, h_b, g, h)$ where g_b, h_b are taken from the crs. Then output **DDH-ENC**(pk_b, m)

- **Dec**$(sk,\ c)$ - This function just outputs **DDH-Dec**(sk, c).

- **FindMessy**$(pk,\ t)$ - Parse pk as (g, h) and a messy mode trapdoor t as x_0, x_1. If $h \neq g^{x_0}$ then output 0 (as the pk provided is for branch 1, so branch 0 is the messy branch). Else output 1.

- **TrapKeyGen**(t) - Parse t as $y \in \mathbb{Z}_p$, check that y is indeed non-zero and a member of $\mathbb{Z}_p$. Pick a random $r \xleftarrow{\$} \mathbb{Z}_p$, compute $pk = (g_0^r, h_0^r)$ and output $(pk, r, r/y)$

Figure 3.6: The realisation of the Dual-mode cryptosystem based on the DDH cryptosystem defined.

3.2 Yao's Protocol

3.2.1 Overview

Yao garbled circuits are one of the primary avenues of research into Secure multi-party computation. Yao first proposed garbled circuits in [15]. The two parties are designated the Builder and the Executor. The Builder then constructs a circuit representing the function the parties wish to compute, this circuit is "garbled" in such a way that it can still be executed.

This garbled circuit, hardcoded with the Builder's input data, is sent to the Executing party who then obtains the data representing its input from the Builder via Oblivious Transfer (for details on OT see Section 3.1). The Executor then evaluates the circuit and obtains the output of the function.

3.2.2 Yao Garbled Circuits

As noted above we first represent the function to compute as a binary circuit. Denote the two parties as P_1 and P_2, we will denote the party building the circuit by P_1 and the executing party by P_2.

Take a single gate of this circuit with two input wires and a single output wire. Denote the gate a G_1 and the input wires as w_1 and w_2 and let w_3 be the output wire. Let $b_i \in \{0,1\}$ be the value of w_i. Here we will take the case where w_i is an input wire for which P_i provides the value. Define the output value of the gate to be $G(b_1, b_2) \in \{0,1\}$. We now garble this gate in order to obscure the inputs and outputs.

P_1 garbles each wire by selecting two random keys of length l, for the wire w_i call these keys k_i^0 and k_i^1. The length of these keys (l) can be considered a security parameter, and should correspond to the length of the key needed for the symmetric encryption scheme we'll be using later.

P_1 also generates a random permutation $\pi_i \in \{0,1\}$ for each w_i. Note that this can be represented by XORing the bit to be permuted and the permutation. We define $c_i = \pi_i(b_i)$. The garbled value of the i^{th} wire is then $k_i^{b_i} \| c_i$, we then represent our garbled truth table for the gate with the table indexed by the values for the c_1 and c_2.

$$c_1, c_2 : E_{k_1^{b_1}, k_2^{b_2}}(k_3^{G(b_1,b_2)} \| c_3)$$

This table is referred to as the *encrypted truth table*.

Where $E_{k_i,k_j}(m)$ is some encryption function taking the keys k_i and k_j and the plaintext m. Since the advent of AES-NI and the cheapness of using AES we will use AES with 128 bit keys to make this function. This is particularly convenient since AES-128 takes in inputs of size 128 bits and produces an output of size 128 bits.

Suppose that $AES_k(m)$ denotes the AES encryption of the plaintext m under the 128 bit key k and $AES_k^{-1}(c)$ denotes the decryption of ciphertext c under key k. We define E_K (and its inverse D_K) as follows,

$$E_K(m) = AES_{k_1}(AES_{k_2}(m)), \text{ where } K = \{k_1, k_2\}$$

$$D_K(m) = AES_{k_2}^{-1}(AES_{k_1}^{-1}(m)), \text{ where } K = \{k_1, k_2\}$$

This is the intuitive extension of AES to multiple keys, chaining the encryption under all of the keys in a set order.

Then P_1 sends this garbled version of the circuit to P_2. P_1 should send the garbling key for its input bit ($k_1^{b_1}$), the full encrypted truth table and $c_1 = \pi(b_1)$. P_1 should also send the permutations for each input wire owned by the Executor and for every output wire owned by P_2.

Then P_2 needs to get $k_2^{b_2} \| c_2$ from P_2 without revealing the value of b_2. This is done by an Oblivious Transfer (see Section3.1) where P_1 inputs k_2^0 and k_2^1 and P_2 inputs b_2. P_2 receives the output $k_2^{b_2} \| c_2$ from the OT and learns nothing about $k_2^{(1-b_2)}$, P_1 gets no output and learns nothing about the value of b_2.

P_2 can then look up the entry in the encrypted truth table indexed by c_1 and c_2 and decrypt it using $D_{k_1^{b_1}, k_2^{b_2}}(\cdot)$. This will give P_2 a value for $k_3^{G(b_1,b_2)} \| c_3$. If this is an output gate then P_2 can extract a value for $G(b_1, b_2)$ by using π_3^{-1}.

This can be extended to a full circuit, the input wires belonging to the circuits builder are hard coded and their garble keys and permuted values are sent to the executor. The values for the input wires belonging to the executor are obtained by the executor via Oblivious transfer with the builder. The executor is only given the permutations for the output wires, and therefore the intermediate wire bit values are protected.

Free XOR Improvement

Over the years many improvements have been made to the original Yao Garbled Circuits to make them quicker to evaluate. One of these improvements is called the Free XOR technique and at its most simple level it reduces the cost of evaluating an XOR gate in the garbled circuit to virtually nil. This is why one of the key measures of a binary circuits optimisation for Yao Garbling is the number of non-XOR gates.

The Free XOR method works by introducing a relationship between the 0-key(k_0) and 1-key(k_1) for each wire. In particular an R is chosen at random for each Yao Garbled Circuit and whilst k_0 is generated randomly as usual we take $k_1 := k_0 \oplus R$.

Then if we have an XOR gate that takes two input gates. Suppose the output keys of the input wires to be $\{X_0, X_1 = X_0 \oplus R\}$ and $\{Y_0, Y_1 = Y_0 \oplus R\}$. Then we take the output keys of the XOR gate to be $Z_0 := X_0 \oplus Y_0$ and $Z_1 := Z_0 \oplus R$.

Then to evaluate a 2-to-1 XOR gate one only need to XOR the input keys. This removes any need for symmetric decryption when evaluating a gate.

To understand why this works it may be helpful to think of the R factor as being the indicator of a 1. Therefore if both inputs are lacking the R factor or both have it then the XOR eliminates it and the output also lacks the R factor.

However, if the input bits are different then only one of input key contains the R factor, as such the XOR preserves it in the output. This can be seen in Figure 3.7.

3.2.3 Security of Yao Garbled Circuits

A naive implementation of a protocol using Yao Garbled Circuits provides only Semi-honest security. For a formal proof of Semi-honest security see [16], we shall briefly give an intuitive explanation of why naive Yao Garbled Circuits are not secure in the presence of Malicious or Covert adversaries.

Consider the case where P_1 is Malicious, at no point does a naive P_2 verify that the garbled circuit provided by the Builder actually computes the function the builder

X	Y	$X \oplus Y$
X_0	Y_0	$X_0 \oplus Y_0 = Z_0$
X_0	Y_1	$X_0 \oplus (Y_0 \oplus R) = Z_1$
X_1	Y_0	$(X_0 \oplus R) \oplus Y_0 = Z_1$
X_1	Y_1	$(X_0 \oplus R) \oplus (Y_0 \oplus R) = Z_0$

Figure 3.7: A tabulation of an XOR gate demonstrating that if all keys are of the Free-XOR form then an XOR gate can be evaluated by simply XORing the input keys.

claims it does.

Whilst the Executor can check that the garbled circuit has the correct "shape" (number of gates, wires between gates etc.) the Executor cannot verified that each gate has the correct output. This clearly breaks the Correctness requirement and depending on the function being computed and the structure of the circuit corresponding to it, the Builder can craft a garbled circuit to undermine the Privacy or Independence of Input properties.

Additionally, the Executor has no way to check that the key it received from the OTs actually corresponds to the request key in the circuit, the Builder could use the same key for both X_0 and X_1 and thus alter the key used by the Executor for a given input wire.

3.2.4 Cut and Choose - Security against Malicious and Covert Adversaries

Concept

Several extensions of Yao's original protocol have been proposed in order to achieve security against Malicious and Covert adversaries. Many depend on an approach dubbed "cut and choose" which provides statistical security (detects cheating with a certain probability).

This relates to the old solution to dividing a cake fairly between two parties. Neither trusts the other so neither is willing to let the other cut a piece for themselves first as they will cut an unfairly large slice. The solution is to have one party cut the cake, then the other party chooses a slice. As the cutting party goes second it is in their interest to ensure the two available slices are of equal size else the chooser will pick the bigger slice and leave the cutter with the smaller.

In our case the Builder builds S many garbled circuits and sends them to the Executor. A subset of these circuits are chosen to be opened for the purpose of checking if they are correct. The remaining circuits are then referred to as Evaluation circuits.

If all check-circuits pass then the Executor evaluates the remaining circuits as usual. If the Executor receives differing outputs from the Evaluation Circuits this indicates cheating, furthermore if any check circuits fail during correctness testing this is also taken to indicate cheating.

As the check-set is unknown to the Builder till they have already committed to the circuits the Builder must guess at which circuit will be in the check-set before the Executor has chosen the check-set.

The number of garbled circuits built (S) acts as a security parameter and the probability of detecting cheating is expressed in terms of S. For example cheating in

the protocol proposed in [5] goes undetected with probability 2^{-s}.

Issues

Cut and Choose seems very simple conceptually, but it creates several subtle new problems to be solved.

Firstly whilst evaluating the many circuits we must now also ensure that both parties' provide the same inputs to each circuit, else they might be able learn many outputs. Aggregating these results might then reveal something about the input of the Building party.

In [1] the example is given of computing the inner product of two binary strings, in this situation the Executing party could give many different inputs each with a single bit set to 1. The output of the circuit would then give the Executor the value of the Builder's input bit corresponding to the high bit in the Executor's input.

Secondly, in order to open the check circuits the Executor needs obtain both keys for each of its input wires for the check circuits without revealing its input. To this end after the Oblivious Transfers have taken place the Executor reveals the check-set and requests all input keys for these circuits.

Note that as the OT has already taken place the Executor can now check that the Builder provided the correct inputs to the OT by comparing its output from the OT with the keys it now receives.

Thirdly, given all keys for the inputs wires how does one go about check the correctness of a circuit? The obvious method would be to fully decrypt each gate, checking to make sure it is the correct gate type(e.g. AND gate).

A simpler alternative though would be for the Builder to seed the randomness used for each circuit differently and then send the seed for each circuit identified as a check circuit. The Executor can then fully re-build each check circuit using this seed and the full inputs sets and check that the resulting circuit is equal to the check circuit. This is the approach we have taken.

Fourthly, how should the Executor react to differing outputs from the evaluation circuits? Whilst it is tempting to simply abort immediately this opens the Executor up to an attack referred to as a *selective failure* attack. This is where the Builder crafts one (or more) of the circuits to fail in some way if the input from the Executor fulfils some condition (e.g. if the first bit is 1). Then the Executor aborting due to differing outputs from Evaluation Circuits leaks information about whether the Executor input satisfies the condition or not.

The original approach to this has the Executor return the majority output on each output wire. The if we have S many circuits, t of which are selected as check circuits the output will only be corrupted if half or more of the Evaluation circuits are corrupted.

This means that the Builder will need to submit at least $\frac{S-t}{2}$ many corrupted circuits else the bad circuits will certainly be outvoted when it comes to decided the majority output. However, the Builder also requires that none of the corrupted circuits are selected as Check circuits, else their cheating will be detected.

A final note, the Builder cannot hardcode its input anymore. Whilst assessing the correctness of the check circuits the Executor is given both keys for each input wire, including those belonging to the Builder. So if the Builder has hardcoded the keys

the Executor can tell which key he was given hardcoded and know the Builder's input. Instead the Builder must wait until after the check-set is revealed to send its inputs for only the evaluation circuits.

Chapter 4

Summary of Protocols to be implemented

We now give an overview of each of the protocols we have implemented. This is not intended to be a full blow-by-blow explanation of the protocols, instead we intend on giving the reader a high-level intuition of the key points of each protocol. We will dig a little deeper into a few points of each protocol, particularly where we feel the original papers were not as clear as they could be.

4.1 Lindell and Pinkas 2011

4.1.1 Overview

The protocol proposed in [3] is a significant improvement on their previous proposal [1] both in terms of performance and conceptual simplicity.

This protocol gives an improved deterrent probability of $\epsilon = 1 - 2^{-0.311S}$, further the work in [4] showed how to achieve a slightly improve deterrent probability of $\epsilon = 1 - 2^{-0.32S}$.

A key improvement is the removal of the very large number of commitments entailed in [1]. These commitments formed one of the main costs in the Lindell-Pinkas-Smart implementation of the previous protocol [2]. Indeed Lindell and Pinkas comment in their introduction to [3] that the previous protocol, when running on a circuit with input size 128, required $6,553,600$ commitments.

Another big improvement is this protocol does not require the preprocessing of the circuit that vastly inflates the number of input wires for the Executor and thus the number of Oblivious transfers needed.

4.1.2 Cut and Choose Oblivious Transfer

The main new idea in this protocol is a modification of the PVW-OT from [21]. We refer to this new OT as the "Cut and Choose OT"(CnC OT). The Receiver generates a random $J \subset [1, ..., S]$ during the setup such that $|J| = \frac{S}{2}$, this set is kept secret from the Sender till later). The represents a subset of the S circuits to be opened in order to check for correctness, we call this the *J-set*.

This set J is then used to generate S many CRSs, each CRS to be used for the OTs to obtain inputs for a different circuit that the Builder sent. For the j^{th} CRS if $j \in J$ then an OT using this CRS will reveal *both* values input by the sender rather than the usual 1-out-of-2 values, otherwise the usual OT functionality holds.

The Executor can then reveal for which circuits it received both values for each input wire, in doing so proves those circuits at least belong to the J-set. The Builder can then reveal all information required to fully decrypt these check circuits, allowing the Executor to test the correctness. The keys representing the Builder's input for each wire for the evaluation circuits are then sent, allowing the Executor to evaluate all the non-check circuits.

A subtle detail that may have passed the reader by is that we require the Executing party be able to prove that at most $\frac{S}{2}$ many of the CRSs allow the recovery of both inputs.

This is achieved via a Zero Knowledge Proof detailed in Appendix B of [3], we will not dwell upon it other than to say it uses a secret sharing scheme to modify the classic Sigma-protocol for proving a Diffie-Hellman tuple to prove that at most $\frac{S}{2}$ of the CRSs allow recovery of both inputs.

4.1.3 Consistency of Builder's inputs

Lindell and Pinkas present a conceptually elegant method for ensuring the consistency of the builder's inputs. Before building the circuits the builder takes a group $\mathbb{G}$ in which the Discrete Log problem is hard. It then generates $\{a_i^0, a_i^1\}_{i=1}^l$ where l is the number of builder's input wires and $\{r_j\}_{j=1}^S$ where S is the number of circuits.

The Builder then computes $\{g^{a_i^0}, g^{a_i^1}\}_{i=1}^l$ and $\{g^{r_j}\}_{j=1}^S$ which are sent to the Executor as commitments to the secret keys. Then $K_0 = H(g^{r_j^{a_i^0}})$ is used as the 0-key on the i^th Builder input wire in the j^{th} circuit. Similarly the Builder uses $K_1 = H(g^{r_j^{a_i^1}})$ as the 1-key on the same wire.

Then inputs for circuit j can be revealed to the Executor by sending r_j, with which the Executor can compute the keys using the commitments to the a values. This reduces the bandwidth needed for revealing the check-circuit keys significantly.

Lindell and Pinkas note this means the keys are *Pseudo-Random Synthesizers* [23], so if some of the keys are revealed (a necessity for checking correctness of circuits) the rest remain pseudo random.

Now note that $(g, g^{r_j}, g^{a_i^0}, g^{r_j^{a_i^0}})$ and $(g, g^{r_j}, g^{a_i^1}, g^{r_j^{a_i^1}})$ are both Diffie-Hellman tuples. Further note that if we take K to be the Builder's input key (either K_0, K_1 then only one of $(g, g^{r_j}, g^{a_i^0}, K)$ and $(g, g^{r_j}, g^{a_i^1}, K)$ is a Diffie-Hellman tuple.

This property is the basis for proving the consistency of the Builder's inputs using the Zero Knowledge Proof for an Extended Diffie-Hellman tuple put forwards in Appendix B of [3].

Extended Diffie-Hellman Tuples

The last detail we shall touch upon for the Lindel-Pinkas 2010 protocol is the Zero Knowledge proof of an Extended Diffie-Hellman(DH) tuple. As already mentioned this is used to prove the consistency of the Builder's inputs to the evaluation circuits.

An Extended DH tuple is a tuple of elements $(g, h, u_1, ..., u_l, v_1, ..., v_l)$ such that each $\{(g, h, u_i, v_i)\}_{i=1}^l$ is a DH Tuple. When proving the consistency of the i^{th} Builder input the parties input $(g, g^{a_i^0}, g^{a_i^1}, U, V)$ where $U = \{g^{r_j}\}_{j \notin J}$ and V is the Builder's inputs in Group element form.

The Parties then engage in a Zero Knowledge Proof that either $(g, g^{a_i^0}, U, V)$ is an Extended DH tuple or $(g, g^{a_i^1}, U, V)$ is.

Lindell and Pinkas provide a pre-processing step that reduces the Extended DH tuple problem to a standard DH tuple problem.

4.2 Lindell 2013

4.2.1 Overview

In [5] Lindell proposed further improvements on his work with Pinkas in [3].

Lindell uses a Secure Computation to determine the output that will produce the correct output if even only one of the evaluation circuits produces the correct output.

This means that to successfully cheat a malicious builder will need to corrupt every evaluation circuit guess *exactly* which circuits will be selected as check circuits. If the guess made by the malicious builder is wrong on even one circuit the cheating will either be detected (if it corrupts a check circuit) or mitigated (if it fails to corrupt every evaluation circuit).

Lindell suggest this Secure Computation be carried out using the protocol he authored with Pinkas in [3] using a small circuit he provides. The hope is that, especially for large circuits, this small secure computation will be relatively cheap.

In order to take full advantage of this improved output determination Lindell modifies the Cut and Choose Oblivious Transfer in [3]. The modification removes the requirement that exactly half the circuits are selected as check circuits. Instead each circuit is selected with probability $\frac{1}{2}$.

This modification of the OT requires a series of Zero knowledge proofs. However, as we shall see it also allows a significant reduction in the number of circuits needed and so the number of OTs needed. One of the purposes of our implementation is to find out if this exchange is worth it.

As each circuit is chosen to be a check-circuit with probability $\frac{1}{2}$ this is effectively requiring a malicious adversary to guess at a random element in the set $\{0,1\}^s$ in order to cheat successfully. Therefore such a builder can only successfully cheat with probability 2^{-S}. (It is a worth noting here that as at least one circuit needs to be checked and at least one needs to be evaluated that there are really 2^{-S+1} sets).

4.2.2 Secure Computation to detect cheating

The Builder constructs all the circuits so that the keys for output wires are the same across all circuits, call these consistent output keys $\{b_i^0, b_i^1\}_{i=1}^h$ (where there are h many output wires). Further we denote the input of the Builder to the circuit as x.

Then if any of the circuits evaluated by the Executor give different outputs on any output wire (say output wire i) the executor will obtain both b_i^0 and b_i^1, these will then be used as input to the cheating detection. If all circuits produce the same output then the Executor randomly generates this input to the cheating detection.

The parties then perform a Secure Computation to detect cheating (here on in, the Sub-computation) where the Builder inputs x (its original input to the main computation) and we call the Executor's inputs b. The Secure Computation returns x to the Executor if its input b indicates it knows both b_i^0 and b_i^1 for some i, otherwise it returns

garbage.

We need to be sure the Builder inputs the same x to the sub-computation as the main computation. This can be done by using the same consistent input style as in the Lindell-Pinkas 2010 protocol.

Lindell suggests using the Lindell-Pinkas protocol for this secure computation and gives several iterations of optimisations for the circuit to compute the function. We skip some of these optimisations, particularly those rendered obsolete by later optimisation. For the full history of optimisation see Lindell's paper.

First let the builder choose $\{b_i^0, b_i^1\}$ and some $\delta \in \{0,1\}^{128}$ such that $b_i^0 \oplus b_i^1 = \delta$ for all i. We can then check if they Executor knows δ in rather than checking to see if they know b_i^0 and b_i^1 for each pair, reducing the size of hte circuit (and bandwidth for sending it) significantly. Given only one of the pair the Executor gains no knowledge of δ so security is maintained. The Executor's input to the sub-computation is then δ'.

This optimisation requires an extra check, after the Executor has committed to his guess at δ the Builder must then revealed δ to prove that the $\{b_i^0, b_i^1\}$ set is consistent with a single δ.

Secondly, as we are aiming for statistical security of 2^{-S} we only need to check S many bits of the δ, reducing the number of inputs and so the number of OTs required.

The Oblivious Transfer Optimisation

Lindell finally describes a method to eliminate completely the gates for comparison between δ and δ' with an elegant use of the OTs we were already going to have to perform. Suppose that the Builder construct the check circuit with the Executor's input as a single bit, indicating knowledge of δ.

Then take the keys for the single Executor input wire for the j^{th} circuit to be $\{k_0^j, k_1^j\}$. All the k_0^j can be simply sent to the Executor. Then for each circuit j construct the set $\{Y_i^j\}_{i=1}^S$ such that $Y_1^j \oplus ... \oplus Y_S^j = k_1^j$.

The parties then run a CnC OT over S many pairs where for the j^th circuit in the i^th pair $P_{\delta_i} = Y_i^j$ while ${}_{\delta_i}P$ is random (where δ_i is the i^{th} bit of δ). For each input i the Executor asks for the δ_i' member of the pair.

If $\delta = \delta'$ then the Executor will be able to use the outputs of the OT to compute k_1^j for every j. If, however, the Executor gets even one bit wrong then the value of k_1^j will be completely hidden. The Cut and Choose nature of the OT allows the Executor to recover k_1^j for every check circuit once δ is revealed.

4.3 Huang, Katz and Evans 2013

Overview

Concurrently to Lindell's work in [5] Huang, Katz and Evans produced a protocol also based along the same cut and choose paradigm. However, in their protocols the parties symmetrically generate a set of circuits and then evaluate each others circuits.

Output determination for each output wire is such that the value for an output wire is only taken if the partner obtained the same value for that output wire in at least one of their evaluation circuits.

The observant reader might question what one does if a party gets both 0 and 1 on some output wire in different circuits, and likewise for the other party. We claim this situation is only possible if both parties cheated, in which case we care little for either party's plight.

If at least one party is honest then this party will provide honest circuits and will only provide the keys required to get one output from these circuits. As such at least one party will have the correct value for every output wire in all their evaluation circuit.

The probability of a malicious adversary successfully cheating is stated as $2^{-S+log(s)}$ where S is the number of circuits created by *each* party. Note this means that we actually need to create $\sim 2S$ many circuit so this protocol requires a factor of about $3/2$ less circuits for the same security level as [3] (we ignore the log factor here but include it in our implementation).

4.3.1 Consistency of party's inputs

The Lindell-Pinkas approach for ensuring inputs from parties are consistent involves expensive zero knowledge proofs. Furthermore, in the symmetric paradigm this approach is problematic as P_1 (resp. P_2) needs to know that P_2 (P_1) gave consistent inputs both to the circuits P_2 (P_1) created and to the circuits P_1 (P_2) created. Furthermore, this must be accomplished without leaking any knowledge of the either party's input bits to the other.

The solution to this problem presented by Huang et al. is an elegant one, based on the form of the queries sent by the receiver in the Naor-Pinkas OT and the 'hardness' of the Discrete Logarithm problem.

Clearly P_2 will need to engage in an OT with P_1 to get its inputs for the circuits P_1 has sent to it. Recall in a Naor-Pinkas OT both parties generate a C in the group at random, and send this to their partner. Each party then refers to the C received from its partner as $\tilde{C}$. Then the query sent by P_2 for its i^{th} input bit will be of the form,

$$h_i = \begin{cases} g^{k_i}, & x_i = 0 \\ \tilde{C}/g^{k_i}, & x_i = 1 \end{cases}, \text{ where } k_i \text{ is the key for } P_1\text{'s } i^{th} \text{ input bit.}$$

These queries are used for Naor-Pinkas OTs for all the circuit built by P_1 and as such P_2 obtains consistent input keys from the OT stage. Effectively the queries commit a party to its input bit string.

This deals with ensuring each party uses consistent keys for the circuits it executes, next we ensure those keys are further consistent with the ones given to the party's partner for executing the circuits it built. Huang et al. propose that when building their circuits each party make their input keys be of the form

$$k_{i,j}^0 = g^{a_0^i}$$
$$k_{i,j}^1 = \tilde{C}/g^{a_1^i}$$

Now consider the value of $A = h_i/k_{i,j}^b$ for any j and some $b \in \{0,1\}$. If $b = x_i$ when x_i is the input bit used to generate h_i then the $\tilde{C}$s cancel and using the laws of exponentiation the querying party can compute the discrete logarithm of A over g.

However, $b \neq x_i$ if there will be some factor of $\tilde{C}$ in A. As $\tilde{C}$ was generated by the other party the querying party does not know the discrete log of $\tilde{C}$ and so cannot compute the discrete log of A over g. Therefore if the querying party can demonstrate knowledge of the discrete logarithm of A over g its partner can take this as proof of the consistency of the querying party's inputs to both sets of circuits.

4.3.2 Output determination

Huang et al. use a verifiable secret sharing scheme for the output determination. For each output wire in the circuit representing the function the parties each randomly generate two secrets. One secret represents a 0 output on that wire, the other represents a 1 output. For P_1 label these (S_i^0, S_i^1) where i is the output wire. In the case of P_2 label them (T_i^0, T_i^1).

From here on in we look from one party's perspective, the other party mirrors the behaviour we specify.

P_1 creates a secret sharing scheme for each S_i^b with S many shares and a threshold such that $\frac{S}{2} + 1$ many shares are needed to reconstruct the secret. Label these shares $W_{i,j}^b$, b being the output bit value on the i^{th} output wire for the j^{th} circuit.

Then when sending the secrets required to assess the correctness of the check circuits P_2 also sends $\{W_{i,j}^0, W_{i,j}^0\}, \forall j \in J, \forall i$. This means P_1 is now in possession of $\frac{S}{2}$ many shares for S_i^b, as such P_1 only needs one more share to uncover the secret.

P_2 evaluates the remaining circuits and for each output wire i if any evaluation circuit outputs 0 on that wire the P_2 can recover the secret $\tilde{S}_i^0$, similarly for $\tilde{S}_i^1$. If there is no circuit that outputs b on output wire i then P_2 sets $\tilde{S}_i^b$ to be random. Symmetrically P_1 obtains $\tilde{T}_i^0$ and $\tilde{T}_i^1$.

Finally the parties run *weak* secure equality tests (weak in the sense the inputs are revealed at the end) for each natural pair of secrets. P_1 inputs $X_i^b = S_i^b \oplus \tilde{T}_i^b$ whilst P_2 inputs $Y_i^b = \tilde{S}_i^b \oplus T_i^b$. If equality holds then the parties know that each party evaluated output wire i to b in at least one evaluation circuit.

If for some output wire i neither $X_i^0 = Y_i^0$ nor $X_i^1 = Y_i^1$ then both parties abort as they have no valid output for the i^{th} output wire. Huang et al. suggest that by convention the parties should test the 0-value secrets first, and if equality holds there skip the equality test on the 1-value secrets.

4.3.3 Advantages of symmetrical cut-and-choose

As the protocol is symmetrical, both parties will be working symmetrically reducing wall clock delays caused by one party having more work to do leaving the other party idle, so depending on how this works out in practise this could mean an improvement in wall clock time of around 3 times quicker.

Once again it is difficult to estimate how much of an improvement this will provide when implemented, but given two parties of similar capabilities we would expect high CPU/Wall time ratio, due to the lack of idling.

4.4 Merging Lindell 2013 and HKE 2013

In the Lindell 2013 protocol the Sub-computation is carried out using the Lindell-Pinkas 2010 protocol. This raises the question, given the HKE protocol requires fewer circuits to achieve the same level of statistical security as Lindell-Pinkas, can we alter the sub-computation to use HKE?

This is very simple conceptually, but we must be careful of a few subtle problems, indeed the solutions to these require us to modify the behaviour of the protocol outside

the sub-computation.

At this point it should be noted that while I will argue *informally* that my merging of Lindell 2013 and HKE I give no formal proof, as such this should not be used seriously till such a formal proof exists.

4.4.1 Problems to address

At first this seems like a trivial matter, merely change the sub-computation implementation to call HKE instead of Lindell-Pinkas. However, consider the following questions, several of which arise due to the symmetrical nature of HKE.

1. The consistency of the Builder's input to the main computation and the sub-computation must be assured, but now it must also be assured in the sub-computation circuit built by the Executor.

2. In the final optimisation of the sub-computation the Executor's input to the sub-computation circuit is a single bit, indicating if it knows δ. When the Executor is building some of the circuits what happens if the Executor giving its input bit as 1?

   We shall in fact give an attack that could be used here that would leak the value of one bit of the Builder's input. As such we are forced to 'roll back' to the previous level of optimisation.

3. The output of the sub-computation must be concealed from the Builder, else the Builder might be able to tell whether the Executor received inconsistent results from the main computation circuits. This would open the door to what is effectively a selective failure attack.

4.4.2 Consistency of Builder's inputs

Recall in the Lindell-Pinkas / Lindell protocols consistency of the builder's inputs can be assured by the use of a 'base key' for each bit value on each input wire. By using a common starting point and then adding in some randomness for each circuit the Builder creates keys that are still indistinguishable to the Executor. They can then run a Zero Knowledge proof that the keys used by the Builder are an Extended Diffie-Hellman Tuple based on the same 'base key'.

Clearly we could extend this to be used for the Builder's inputs to the Builder's sub-computation circuits. However, I see no way for this to be extended to the builder's inputs to the Executor's sub-computation circuits. Fundamentally the Zero Knowledge Proof proves that,

$$\forall j(g, g^{r_j}, g^{a_i^0}, k_{i,j}) \in DH \text{ OR } \forall j(g, g^{r_j}, g^{a_i^1}, k_{i,j}) \in DH$$

This causes serious problems because both parties need to know $k_{i,j}$, and as the Executor generated $k_{i,j}$ if the Builder tells the Executor which one to use for the proof then the Executor learns the Builder's input bit. Whilst there are probably ways to alter the Zero Knowledge Proof to account for this I propose a simpler and more efficient solution by using the HKE approach to consistency.

Suppose the Builder's inputs to the main circuit are produced so to be in the same form as given in Subsection 4.3.1. Furthermore the Builder's inputs to the sub-computation circuits he builds are also of this form. Both sets of inputs use the same $\tilde{C}$.

The Builder obtains his inputs for the sub-computation circuits sent by the Executor by Naor-Pinkas Oblivious Transfer as is usual in the HKE protocol. Then the Builder can prove the consistency of his input to all three sets of circuits using knowledge of logarithm trick used in the HKE protocol.

Whilst changing the consistency checks is a side-effect of the other changes it also gives significant performance improvements. Two main reasons for these appear to be the HKE approach using many more fixed-base point multiplications which are significantly faster, and the removal of the expensive Zero Knowledge Proofs.

4.4.3 Ensuring consistency of Executor's inputs

An attack on the OT optimisation

In the final optimisation suggested in [5] the sub-circuit is reduced so that the Executor only inputs a single bit, indicating knowledge of δ. The 0-value key for this wire is given freely to the Executor. The Executor then obtains the 1-value in a series of cut-and-choose OTs where the Executor only learns the value by.

This approach cannot be used when we are using a symmetric paradigm for the sub-computation because the Builder cannot verify that the consistency of the Executor's inputs beyond each circuit set. Suppose then that the Builder is honest the Executor therefore has not obtained δ. Then his input to the circuits created by the Builder will have to be 0. However, no such restriction exists on his input to the circuit he created.

Therefore the circuits evaluated by the Builder will output X (where X is his input to the main computation). The circuits evaluated by the Executor will output 00...0. So when it comes to the Secure Equality testing the parties will abort on the first bit where X is 1, leaking information about the Builder's input to the computation.

Rolling Back

The reason the aforementioned attack exists is the fact that the Executor can simply set its input to 1 and the Builder has no way to tell if this is consistent with the Executor's input to the other circuits.

We therefore take a step back and return to the Builder inputting the first S many bits of δ and the Executor inputting the first S bits of δ'. This does not cost us as much as one might think, while it increases the size of the circuit it does so only a little and by a factor that is unaffected by the size of the main computation inputs. This current round of modification to the sub-computation leaves us requiring $|X| \cdot S + 2 \cdot S^2$ many OTs with a circuit size of about $S + 2 \cdot |X|$

4.4.4 Hiding output from Builder

We need to ensure that the Builder gains no knowledge from the sub-computation about whether the Executor input $\delta' = \delta$. If the Builder can tell if the sub-computation output all zeroes or X then he knows if the Executor received inconsistent outputs from the main computation circuits. This gives rise to an indirect selective failure attack whereby the Builder crafts the main computation circuits so the Executor will get inconsistent outputs if some condition is met by the Executor's inputs. Thus if the Builder can discern that the Sub-computation output X he knows the Executor's input satisfies this condition.

This is not an issue in the original Lindell 2013 protocol as the Builder does not evaluate any circuits relating to the sub-computation. However, when we perform the sub-computation with the Huang-Katz-Evans protocol the Builder will be evaluating circuits and takes part in the output determination. To ensure the Builder does not learn anything about the output we further modify the sub-computation circuits to take an extra input from the Executor. This input should be same length as the output and is XORed with the old output.

This would mean the Builder only learns the XORed result of the circuit but the Executor could use his auxiliary input to recover the true output. As in the scenario where we care about the output being hidden from the Builder we can assume the Executor is honest. In this case the outputs from the Executor's circuits will be consistent and the Builder will only see one output and so gains no depth for the key.

Chapter 5

Experiments

We shall be using the circuits provided in [24] for our experiments with varying randomised inputs, in particular we shall consider

- 32-bit Addition,
- 32-bit Multiplication,
- AES encryption.

5.1 Measurement metrics

We shall be focusing on three main metrics for measuring performance of the protocols for both parties, namely CPU time used, Wall clock time (both in seconds) used and data sent (in terms of bytes).

We shall break these metrics down further so that we can see measure the performance of each part of the protocol for the purpose of identifying the bottlenecks for each protocol.

5.2 Testing Environment

All tests were carried out between two test machines each with an i7-3770S CPU clocked at 3.10 GHz with 8096 KB of cache and 32 GB of RAM. These machines both possess dedicated network cards for communications with the other member of the pair. Compilation was performed with g++ version 4.4.7.

5.3 Notes on the form of Experiments

All tests are configured to given a deterrent probability of $1-2^{40}$, or put in other terms, a statistical security parameter of 40.

For each protocol we ran a 100 evaluations on each test circuit, the figures given below are taken from the average of these experiments. As already noted we measure performance in terms or Wall/CPU time (in seconds) and in terms of data sent/received (in Bytes).

We further took sub-measurements for the important parts of the protocol, allowing us to identify which part of a protocol is the most expensive.

The HKE protocol is symmetric therefore there is no purpose is giving measurements for both parties when they are running in identical environments. As such for each measurement we give the average from both parties.

5.4 Expectations

Before giving the results of our experiments we shall first give some expectations we had for each circuit and some thoughts on the circuits themselves.

5.4.1 32-bit addition

The 32-bit addition circuit is the smallest circuit we consider, consisting of only 349 gates. Each party inputs 32-bits and the circuit outputs the addition of the inputs considered as a 33-bit integers (allowing for overflow).

We expected to see a poor showing from the Lindell 2013 protocol due to the circuits small size increasing the relative cost of the sub-computation. Due to the relatively high input wire to gate ratio we expect the cost of Oblivious Transfers (and other input size dependant part of the protocol) to be fairly high relative to the rest.

5.4.2 32-bit multiplication

The 32-bit multiplication circuit is significantly larger than addition yet smaller than AES, providing a good mid-way stepping stone to the AES circuit in terms of number of gates. Additionally, as the number of outputs is larger and this should affect the time taken to run output determination for the HKE protocol.

Furthermore, as the inputs sizes are the same for both parties the number of OTs is the same as in the Addition circuit, meaning we get so see how much importance the 'depth' (size of circuit discounting inputs) of the circuit has with regards to performance. We expect to see the costs of OTs to remain the same and for the circuit building/checking times to increase.

However, whilst the multiplication circuit has around 30 times more gates than the addition circuit we do not expect to see an increase in run time of 30 times, rather much less. As the circuits are bigger building circuits in parallel will yield more fruit as the overhead of parallelism lessen compared to actual work being done.

5.4.3 AES encryption

AES encryption is a classic benchmark for Secure two part computations. We will be considering the version without The RTL circuit provided from [24] for this computation has $\sim 39,000$ gates, 128 inputs for each party and 128 outputs. One party inputs a message, the other inputs a key.

We considered experimenting with the alternative circuit provided by [24] which takes in an expanded key schedule reducing the circuit size in exchange for an increase in the number of inputs by a factor of 10 for the Executor. However, while the reduction in circuit size would reduce building, checking and evaluation time this would be more than nullified by the increase in the number of inputs and the increase in the number of OTs involved.

5.5 Results

We now give results of using each of the Protocol on the test circuits with some comments on the results for each table, these comments will not focus overly much on the comparison between the protocols. That will be saved for later.

5.5.1 32-bit addition

Lindell-Pinkas 2010

	Builder			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Input Generation	0.18	0.02	0	0
Building Circuits	20.39	2.69	0	0
OT- Sender	82.75	11.80	1,214,877	655,111
Sending Circuits and Commitments	0.63	2.43	6,125,691	0
Open Check Circuits	3.19	3.20	289,396	2,214
Prove Input Consistency	6.82	7.27	18,110	79,784
Total	113.96	27.41	7,648,074	737,109

Figure 5.1: The performance of the Builder in the Lindell-Pinkas 2010 protocol evaluating the 32-bit addition circuit averaged over 100 trials.

	Executor			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT Prep Receiver	17.60	2.60	0	0
OT Transfer Receiver	18.93	14.18	655,111	1,214,877
Receiving Circuits and Commitments	0.40	0.05	0	6,125,691
Checking correctness	11.57	3.20	2,214	289,396
Verify Input Consistency	6.87	7.28	79,784	18,110
Evaluate Circuits	0.03	0.03	0	0
Total	55.90	27.45	737,109	7,648,074

Figure 5.2: The performance of the Executor in the Lindell-Pinkas 2010 protocol evaluating the 32-bit addition circuit averaged over 100 trials.

Lindell 2013

	Builder			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Input generation	0.09	0.01	0	0
Building Circuits and Commits	6.49	0.81	0	0
OT- Sender	37.83	8.98	284,040	135,781
Sending Circuits and Commits	0.00	0.02	1,889,281	0
Partially Open Check Circuits	0.99	0.99	88,982	692
Sub-computation to detect cheating	117.53	21.85	2,412,286	746,955
Send B-Lists (Fully Open Check circuits)	0.00	0.00	1,060	0
Prove Input Consistency	8.27	9.37	18,112	96,764
Total	171.21	42.03	4,693,761	980,193

Figure 5.3: The performance of the Builder in the Lindell 2013 protocol evaluating the 32-bit addition averaged over 100 trials.

	Executor			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT - Receiver	35.46	9.79	135,781	284,040
Receiving Circuits and commitments	0.01	0.03	0	1,889,281
Receive partial circuit openings	0.03	0.99	692	88,982
Evaluate Circuits	0.01	0.01	0	0
Sub-computation to detect cheating	53.37	21.85	746,955	2,412,286
Verify B-List	0.00	0.00	0	1,060
Checking correctness	3.63	0.58	0	0
Verify input consistency	9.16	8.79	96,764	18,112
Total	101.69	42.05	980,193	4,693,761

Figure 5.4: The performance of the Executor in the Lindell 2013 protocol evaluating the 32-bit addition averaged over 100 trials.

Huang-Katz-Evans 2013

Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Circuits prep.	2.34	0.31	0	0
Building circuits	0.12	0.02	0	0
Sending Circuits and Commitments	0.27	0.04	1,880,477	1,880,477
Exchange Secret Sharing Schemes	0.02	0.00	32,737	32,775
OT	10.80	1.35	136,744	136,744
Make and send commitments	10.93	1.37	374,062	374,062
Coin flip for J-set	0.08	0.01	2,532	2,532
Initial J-set checks	7.83	5.56	231,848	231,848
Logarithm Checks	0.95	0.13	23,044	23,044
Output Determination	1.15	0.47	13,630	13,630
Total	34.60	9.26	2,695,074	2,695,112

Figure 5.5: The performance of the Huang-Katz-Evans 2013 protocol evaluating the 32-bit addition circuit averaged over 100 trials.

L-HKE 2015	**Builder**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Building Circuits and Commits	1.57	1.48	77	77
OT- Sender	37.65	8.95	284,040	135,782
Sending Circuits and Hash List	0.02	0.03	1,883,800	0
Make and Send Commitments	10.79	1.39	374,062	0
Partially Open Check Circuits	0.05	0.01	132,979	660
Sub-computation to detect cheating	73.02	26.12	2,867,854	2,753,334
Send B-Lists (Fully Open Check circuits)	0.00	0.00	1,060	0
Prove Input Consistency	0.00	0.00	23,873	0
Total	123.10	37.98	5,567,746	2,889,855

Figure 5.6: The performance of the Builder in the L-HKE 2015 protocol evaluating the 32-bit addition circuit averaged over 100 trials.

L-HKE 2015	**Executor**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT - Receiver	35.25	10.42	135,859	284,117
Receiving circuits and Hashed List	0.01	0.05	0	1,882,740
Receiving Commitments	0.00	1.39	0	375,122
Receive partial circuit openings	0.01	0.01	660	132,979
Evaluate Circuits	0.01	0.01	660	132,979
Sub-computation to detect cheating	72.47	26.11	2,753,334	2,867,854
Verify B-List	0.00	0.00	0	1,060
Checking correctness	0.78	0.72	0	0
Verify input consistency	0.94	0.12	0	23,873
Total	109.46	38.82	2,889,855	5,567,746

Figure 5.7: The performance of the Executor in the L-HKE 2015 protocol evaluating the 32-bit addition circuit averaged over 100 trials.

5.5.2 32-bit multiplication

Lindell-Pinkas 2010	**Builder**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Input Generation	0.18	0.02	0	0
Building Circuits	31.53	4.21	0	0
OT- Sender	82.70	11.82	1,214,877	655,111
Sending Circuits and Commitments	1.01	4.10	202,012,291	0
Open Check Circuits	3.19	3.22	289,396	2,214
Prove Input Consistency	6.82	7.27	18,109	79,784
Total	125.43	30.65	203,534,673	737,109

Figure 5.8: The performance of the Builder in the Lindell-Pinkas 2010 protocol evaluating the 32-bit multiplication averaged over 100 trials.

Lindell-Pinkas 2010	**Executor**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT Prep Receiver	17.59	2.60	0	0
OT Transfer Receiver	18.69	14.20	655,111	1,214,877
Receiving Circuits and Commitments	1.65	1.75	0	202,012,291
Checking correctness	17.14	3.19	2,214	289,396
Verify Input Consistency	6.86	7.28	79,784	18,109
Evaluate Circuits	0.84	0.84	0	0
Total	63.53	31.50	737,109	203,534,673

Figure 5.9: The performance of the Executor in the Lindell-Pinkas 2010 protocol evaluating the 32-bit multiplication averaged over 100 trials.

Lindell 2013	**Builder**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Input generation	0.09	0.01	0	0
Building Circuits and Commits	9.76	1.24	0	0
OT- Sender	37.80	8.99	284,040	135,780
Sending Circuits and Commits	0.12	0.52	62,163,073	0
Partially Open Check Circuits	1.01	1.04	89,081	681
Sub-computation to detect cheating	117.91	22.38	2,412,286	746,955
Send B-Lists (Fully Open Check circuits)	0.00	0.00	2,052	0
Prove Input Consistency	8.31	9.67	18,111	97,167
Total	175.00	43.86	64,968,644	980,583

Figure 5.10: The performance of the Builder in the Lindell 2013 protocol evaluating the 32-bit multiplication circuit averaged over 100 trials.

Lindell 2013	**Executor**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT - Receiver	35.46	10.23	135,780	284,040
Receiving circuits and commitments	0.28	0.56	0	62,163,073
Receive partial circuit openings	0.03	1.01	681	89,081
Evaluate Circuits	0.26	0.26	0	0
Sub-computation to detect cheating	53.44	22.13	746,955	2,412,286
Verify B-List	0.00	0.00	0	2,052
Checking correctness	5.58	0.85	0	0
Verify input consistency	9.19	8.82	97,167	18,111
Total	104.28	43.88	980,583	64,968,644

Figure 5.11: The performance of the Executor in the Lindell 2013 protocol evaluating the 32-bit multiplication circuit averaged over 100 trials.

Huang, Katz and Evans 2013

Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Circuits prep.	2.36	0.32	0	0
Building circuits	3.53	0.46	0	0
Sending Circuits and Commitments	1.15	1.09	62,153,277	62,153,277
Exchange Secret Sharing Schemes	0.00	0.03	63,456	63,455
OT	10.92	1.42	136,744	136,744
Make and send commitments	11.08	1.40	374,062	374,062
Coin flip for J-set	0.09	0.01	2,532	2,532
Initial J-set checks	8.90	6.65	256,648	256,648
Logarithm Checks	0.95	0.13	23,044	23,044
Output Determination	0.80	0.82	26,154	26,154
Total	40.89	12.64	63,035,917	63,035,916

Figure 5.12: The performance of the Huang-Katz-Evans 2013 protocol evaluating the 32-bit multiplication circuit averaged over 100 trials.

L-HKE 2015	**Builder**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Building Circuits and Commits	1.57	1.48	77	77
OT- Sender	37.65	8.95	284,040	135,782
Sending Circuits and Hash List	0.02	0.03	1,883,800	0
Make and Send Commitments	10.79	1.39	374,062	0
Partially Open Check Circuits	0.05	0.01	132,979	660
Sub-computation to detect cheating	73.02	26.12	2,867,854	2,753,334
Send B-Lists (Fully Open Check circuits)	0.00	0.00	1,060	0
Prove Input Consistency	0.00	0.00	23,873	0
Total	123.10	37.98	5,567,746	2,889,855

Figure 5.13: The performance of the Builder in the L-HKE 2015 protocol evaluating the 32-bit addition circuit averaged over 100 trials.

L-HKE 2015	**Executor**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT - Receiver	35.25	10.42	135,859	284,117
Receiving circuits and Hashed List	0.01	0.05	0	1,882,740
Receiving Commitments	0.00	1.39	0	375,122
Receive partial circuit openings	0.01	0.01	660	132,979
Evaluate Circuits	0.01	0.01	660	132,979
Sub-computation to detect cheating	72.47	26.11	2,753,334	2,867,854
Verify B-List	0.00	0.00	0	1,060
Checking correctness	0.78	0.72	0	0
Verify input consistency	0.94	0.12	0	23,873
Total	109.46	38.82	2,889,855	5,567,746

Figure 5.14: The performance of the Executor in the L-HKE 2015 protocol evaluating the 32-bit multiplication circuit averaged over 100 trials.

5.5.3 AES encryption

Lindell-Pinkas 2010	**Builder**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Input generation	0.36	0.05	0	0
Building Circuits	114.32	15.39	0	0
OT- Sender	323.83	42.42	4,859,037	2,477,191
Sending Circuits and Commitments	1.37	14.81	663,253,047	0
Open Check Circuits	13.12	13.17	751,156	2,214
Prove Input Consistency	27.82	29.15	72,444	319,112
Total	480.82	114.98	668,935,684	2,798,517

Figure 5.15: The performance of the Builder in the Lindell-Pinkas 2010 protocol evaluating the AES encryption circuit averaged over 100 trials.

Lindell-Pinkas 2010	**Executor**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT Prep Receiver	67.53	9.03	0	0
OT Transfer Receiver	70.98	51.55	2,477,191	4,859,037
Receiving Circuits and Commitments	3.16	5.72	0	663,253,047
Checking correctness	56.90	13.15	2,214	751,156
Verify Input Consistency	27.45	29.14	319,112	72,444
Evaluate Circuits	1.15	1.15	0	0
Total	227.91	116.15	2,798,517	668,935,684

Figure 5.16: The performance of the Executor in the Lindell-Pinkas 2010 protocol evaluating the AES encryption circuit averaged over 100 trials.

Lindell 2013

	Builder			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Input generation	0.27	0.03	0	0
Building Circuits	36.24	4.58	0	0
OT- Sender	142.05	33.20	1,120,488	476,292
Sending Circuits and Commits	0.33	1.75	204,095,057	0
Partially Open Check Circuits	4.01	4.05	227,146	701
Sub-computation to detect cheating	183.19	37.52	5,018,302	746,955
Send B-Lists (Fully Open Check circuits)	0.00	0.00	4,100	0
Prove Input Consistency	33.18	38.11	72,444	385,743
Total	399.27	119.25	210,537,538	1,609,692

Figure 5.17: The performance of the Builder in the Lindell 2013 protocol evaluating the AES encryption circuit averaged over 100 trials.

	Executor			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT - Receiver	138.36	37.80	476,292	1,120,488
Receiving circuits and commitments	0.77	1.80	0	204,095,057
Receive partial circuit openings	0.04	4.02	701	227,146
Evaluate Circuits	0.35	0.35	0	0
Sub-computation to detect cheating	78.95	37.20	746,955	5,018,302
Verify B-List	0.00	0.00	0	4,100
Checking correctness	18.36	3.13	0	0
Verify input consistency	34.05	34.96	385,743	72,444
Total	270.99	119.27	1,609,692	210,537,538

Figure 5.18: The performance of the Executor in the Lindell 2013 protocol evaluating the AES encryption circuit averaged over 100 trials.

Huang, Katz and Evans 2013

Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Circuits prep.	9.42	1.22	0	0
Building circuits	11.15	1.44	0	0
Sending Circuits and Commitments	1.87	3.59	204,069,197	204,069,197
Exchange Secret Sharing Schemes	0.00	0.05	126,864	126,846
OT	43.04	5.55	546,952	546,952
Make and send commitments	44.27	5.60	1,495,342	1,495,342
Coin flip for J-set	0.10	0.01	2,532	2,532
Initial J-set checks	30.21	25.29	801,288	801,288
Logarithm Checks	2.85	0.47	92,164	92,164
Output Determination	1.60	1.64	52,010	52,010
Total	145.70	45.26	207,186,349	207,186,331

Figure 5.19: The performance of the Huang-Katz-Evans 2013 protocol evaluating the AES encryption circuit averaged over 100 trials.

L-HKE 2015	**Builder**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
Building Circuits and Commits	1.57	1.48	77	77
OT- Sender	37.65	8.95	284,040	135,782
Sending Circuits and Hash List	0.02	0.03	1,883,800	0
Make and Send Commitments	10.79	1.39	374,062	0
Partially Open Check Circuits	0.05	0.01	132,979	660
Sub-computation to detect cheating	73.02	26.12	2,867,854	2,753,334
Send B-Lists (Fully Open Check circuits)	0.00	0.00	1,060	0
Prove Input Consistency	0.00	0.00	23,873	0
Total	123.10	37.98	5,567,746	2,889,855

Figure 5.20: The performance of the Builder in the L-HKE 2015 protocol evaluating the AES encryption circuit averaged over 100 trials.

L-HKE 2015	**Executor**			
Step	**CPU Time**	**Wall Time**	**Bytes Sent**	**Bytes Recv**
OT - Receiver	35.25	10.42	135,859	284,117
Receiving circuits and Hashed List	0.01	0.05	0	1,882,740
Receiving Commitments	0.00	1.39	0	375,122
Receive partial circuit openings	0.01	0.01	660	132,979
Evaluate Circuits	0.01	0.01	660	132,979
Sub-computation to detect cheating	72.47	26.11	2,753,334	2,867,854
Verify B-List	0.00	0.00	0	1,060
Checking correctness	0.78	0.72	0	0
Verify input consistency	0.94	0.12	0	23,873
Total	109.46	38.82	2,889,855	5,567,746

Figure 5.21: The performance of the Executor in the L-HKE 2015 protocol evaluating the AES encryption circuit averaged over 100 trials.

Chapter 6

Conclusions

Bibliography

[1] Y. Lindell and B. Pinkas. *An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries*. To appear in the Journal of Cryptology. (Extended abstract appeared in EUROCRYPT 2007, Springer (LNCS 4515), pages 52–78, 2007.)

[2] Y. Lindell, B. Pinkas and N. P. Smart. *Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries*. Proceedings of the Sixth Conference on Security and Cryptography for Networks (SCN), 2008.

[3] Y. Lindell and B. Pinkas. *Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer*. In TCC 2011, Springer (LNCS 6597), pages 329–346, 2011

[4] A. Shelat, C.H. Shen. *Two-Output Secure Computation with Malicious Adversaries*, In EUROCRYPT 2011, Springer (LNCS 6632), pages 386–405, 2011.

[5] Y. Lindell. *Fast cut-and-choose based protocols for malicious and covert adversaries*, R. Canetti, J.A. Garay, (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pages 1–17. Springer, Heidelberg (2013).

[6] Y. Huang, J. Katz, D. Evans. *Efficient Secure Two-Party Computation Using Symmetric Cut-and-Choose*, In 33rd International Cryptology Conference (CRYPTO 2013), 2013.

[7] P. Bogetoft, D. Christensen, I. Damgård et al. *Secure Multiparty Computation Goes Live*, In Financial Cryptography and Data Security 2009, Springer LNCS 5628, pages 325-343, 2009.

[8] DYADIC, MPC Technical Primer, `https://www.dyadicsec.com/media/1093/mpc-primer.pdf`

[9] DARPA. *PROCEED Program webpage*. `http://www.darpa.mil/Our_Work/I2O/Programs/PROgramming_Computation_on_EncryptEd_Data_%28PROCEED%29.aspx`

[10] B. Pinkas, T. Schneider, N. P. Smart and S. C. Williams. *Secure Two-Party Computation is Practical*, ASIACRYPT 2009, 2009.

[11] V. Kolesnikov and T. Schneider. *Improved garbled circuit: Free XOR gates and applications*. In Automata, Languages and Programming – ICALP 2008, Springer-Verlag (LNCS 5126), pages 486 - 498, 2008.

[12] S. Jarecki and V. Shmatikov. *Efficient Two-Party Secure Computation on Committed Inputs.* In EUROCRYPT 2007, Springer (LNCS 4515), pages 97 - 114, 2007.

[13] J. Nielsen and C. Orlandi. *LEGO for Two-Party Secure Computation*. In TCC 2009, Springer (LNCS 5444), pages 368 - 386, 2009.

[14] T. Frederiksen, T. Jakobsen, J. Nielsen, et al. *MiniLEGO: Efficient Secure Two-Party Computation from General Assumptions*, In Advances in Cryptology - EUROCRYPT 2013, Springer (LNCS 7881), pages 537 - 556, 2013.

[15] A. Yao. *How to Generate and Exchange Secrets.* In 27th FOCS, pages 162–167, 1986.

[16] Y. Lindell, B. Pinkas. *A proof of security of Yao's protocol for two-party computation.* Journal of Cryptology 22(2), pages 161 - 188 (2009).

[17] I. Abraham, D. Dolev, R. Gonen and J. Halpern. *Distributed Computing Meets Game Theory: Robust Mechanisms for Rational Secret Sharing and Multiparty Computation*, Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, pages 53 - 62, 2006.

[18] M. Rabin. *How to exchange secrets with oblivious transfer.* Technical Report, TR-81, Aiken Computation Lab, Harvard University, 1981.

[19] B. Pinkas. *Secure Computation Lecture Series*, Lecture 5 - Oblivious Transfer, 2014.

[20] S. Even, O. Goldreich and A. Lempel. *A randomized protocol for signing contracts*, In Communications of the ACM, Vol. 28 Iss. 6, pages 637 - 647 (1985)

[21] C. Peikert, V. Vaikuntanathan and B. Waters. *A framework for efficient and composable oblivious transfer.* In: Wagner, D. (ed.) CRYPTO 2008, Springer (LNCS 5157), pages 554–571, 2008.

[22] Naor and B. Pinkas, *Efficient Oblivious Transfer Protocols*, Proceedings of SODA 2001 (SIAM Symposium on Discrete Algorithms), 2001.

[23] M. Naor and O. Reingold. *Synthesizers and Their Application to the Parallel Construction of Psuedo-Random Functions.* In the 36th FOCS, pages 170–181, 1995.

[24] Bristol Cryptography Group, *Circuits of Basic Functions Suitable For MPC and FHE.* `http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`.

[25] N. Sullivan, *A (relatively easy to understand) primer on elliptic curve cryptography*, October 2013, `http://arstechnica.com/security/2013/10/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/`.

[26] D. McGrew, K. Igoe and M. Salter, *Fundamental Elliptic Curve Cryptography Algorithms*, RFC 6090, February 2011.

[27] ECC Brainpool, *ECC Brainpool Standard Curves and Curve Generation*, October 2005, `http://www.ecc-brainpool.org/download/Domain-parameters.pdf`.

[28] NSA, *The Case for Elliptic Curve Cryptography*, January 2009, `https://www.nsa.gov/business/programs/elliptic_curve.shtml`.

[29] Wikipedia (various authors), *Elliptic curve point multiplication*, `http://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication`

[30] N. P. Smart, Cryptography, An Introduction : Third Edition, `https://www.cs.bris.ac.uk/~nigel/Crypto_Book/`

[31] A. Shamir, *How to Share a Secret.* In the Communications of the ACM, 22(11):612–613, 1979.

[32] Bob Jenkins. *ISAAC: a fast cryptographic random number generator*, `http://burtleburtle.net/bob/rand/isaacafa.html`.

Appendix A

Benchmarking components

Here I give some benchmarks of key components in my implementation such as communication, ECC encryption and circuit evaluation. I include these measurements so that others intending to implement these protocols with more efficient (e.g. library supplied) components can get a rough idea of what performance improvement they can expect.

A.1 Communications

We benchmark our communications between Diffie and Hellman. We focus on sending elements of the 256-bit ECC group and sending raw bytes in varying sizes and numbers of blocks.

Communication benchmarks will probably will elicit the most interest from readers intending on implementing these protocols themselves as the nature of our test environment de-emphasise the communication costs due to the close proximity of the two test machines.

A.2 Elliptic Curve Group Operations

We benchmark point addition, point doubling, point multiplication and fixed point multiplication. The fixed point multiplication includes the pre-computation of the relevant windows.

A.3 Oblivious Transfer

We benchmark all the Oblivious transfers we use, in each case we include the setup of the OTs in the measurements and we also state the communication costs (number of bytes exchanged). We vary the inputs relating to the number of input pairs and the number circuits.

A.3.1 Cut and Choose Oblivious Transfer

A.3.2 Modified Cut and Choose Oblivious Transfer

A.3.3 Naor Pinkas Oblivious Transfer

.

A.4 Circuit Building

Circuit building can be an expensive operation, furthermore as we take the re-building approach to circuit correctness checking it is carried out for each check circuit. We do

not include preliminary operations (e.g. generating consistent inputs for circuits).

A.5 Circuit Evaluation

Once a party has the inputs for a Yao Garbled Circuit the circuit must be evaluated. We show benchmarks for each binary circuit we shall be testing. Additionally we demonstrate the difference that AES-NI makes and the Free-XOR optimisations.

Appendix B

Implementation usage guide

This chapter deals with how to build and use the implementation provided. If you are the Bristol markers the implementation source code was submitted on SAFE in a zip file. Else you can download the source code from github. The project can be found at `https://github.com/nt1124/FourthYearProject`.

Unless otherwise stated I assume you are in the root directory of the source code (FourthYearProject). I have tested the implementation on Ubuntu (both 14.04 and 12.04), I give no guarantees for other operating systems.

B.1 Building

B.1.1 Dependencies

You will require the following to compile and run our code.

- `g++`, used to compile the code.
- GNU Multi-Precision Arithmetic Library, can be installed using the command `'sudo apt-get install libgmp-dev'`
- rt-library, used for wall clock timings.
- OpenMP, this is optional but its absence will have a serious impact on performance.
- AES-NI, again this is optional but preferred for performance.

B.1.2 Compilation

Compilation can be performed with the command

```
g++ circuitEvaluator.c -O3 -fopenmp -ffast-math -maes -lgmp -lrt
```

This will produce an executable called `a.out`, the output file can be changed inthe usual manner. If you do not have OpenMP or AES-NI you can still compile by removing the `-fopen-mp` or `-maes` flags respectively.

B.2 Running

.

Appendix C

Implementation Details

The final implementation is quite large and as such it would be thoroughly to go through all of it in any great detail. Furthermore a blow by blow account of the implementation of how we implement such primitives as Yao Garbled Circuits would detract from the purpose of this Dissertation, namely a comparison of several recent protocols, not a rehash of how to implement primitives.

However, this said we shall touch on some of the high points and some of the more interesting primitives. We also comment on the purpose of this implementation and suggest a few places where we feel the implementation could be improved.

Purpose of Implementation

It should be made abundantly clear that the implementation provided is *not* intended for real world use with actual confidentiality on the line, instead it is for the purposes of comparing the performance of the protocols under consideration.

Whilst the protocols have been implemented faithfully some of the lower level details not relevant to a comparison of the protocols are ignored, for example we do not established a secure connection between the two parties.

Where possible we have implemented everything myself and reused the same code across protocols, rather than using available libraries. This maintains a consistent quality of implementation, using libraries where appropriate would improve the quality of the implementation it would do so in an uneven manner as many areas cannot be done using a library. This could potentially give one protocol an unfair advantage over another leading to skewed results.

C.1 Yao Garbled Circuits implementation

Clearly we need to implement Yao garbled circuits, but before even that we have an ordinary binary circuit implementation and we need to understand the format of the circuit definition files given by [24].

C.1.1 Tillich-Smart Circuit Files

We are using the circuits provided by [24], these circuits have been crafted with Yao Garbled Circuits in mind, applying some of the optimisations suggested in [10] and trying to minimise the number of AND gates in order to take maximal advantage of the Free-XOR optimisation.

Throughout we shall refer to the format of the files as RTL. The first line of each RTL file saying how many gates and how many wires are in the circuit, the second line

tells us how many inputs party 1 and Party 2 give to the circuit and how many outputs there are. Note that without modification we can only provide output to either only the Executor or both parties.

From then on each line refers to a single gate of the binary circuit. The first number (call this number m) of a gate definition says how many inputs wires go into the gate, the second number (call this n) how many output wire come from the gate. Then the next m numbers are the input wire IDs, then the last n number are the IDs of the output wires. Finally the gate type is indicated, either AND, XOR, or INV.

So for example, '2 1 0 32 406 XOR' represents an XOR gate with ID 406 that takes two input wires which have IDs 0 and 32.

C.1.2 Creating binary circuits

By creating a binary circuit from the RTL files and then using this binary circuit (here on in the Raw input circuit) as a template for the creation of Yao Circuits we gain three advantages over reading from the RTL file to create a Yao Garbled Circuit directly.

Firstly this reducing the amount of file I/O, we only need read the file once. Secondly this means makes it easier for us to perform further optimisations on the circuits, for example wire switching the inversion gates to reduce the size of the circuits (note we have not actually done this). Thirdly we need to be able to execute the normal binary circuit in the course of the Lindell 2013 protocol.

We then create a Garbled circuit in the usual way using the raw input circuit to define the relations between gate rather than the RTL file.

C.2 Elliptic Curve implementation

Throughout unless otherwise stated we have worked in Elliptic Curve groups, in particular on the curve *brainpoolP256r1* specified in [27]. This is a 256-bit curve and as such provides 128-bits of security. I suggest [25] as a high-level primer on ECC and [26] for a more technically detailed introduction.

For a quick reference on some of the algorithms we use for operations I warily suggest [29], primarily for the virtue of clear pseudo-code. For obvious reasons do not rely to much on this source.

Elliptic curves groups are preferable over Schnorr groups for cryptographic purposes. They require smaller keys for the same level of security reducing the required size of the group. reducing the size of the numbers we are dealing with making computations quicker without sacrificing security. This point is illustrated in the Figure C.1.

Symmetric key size (bits)	**RSA/Diffie-Hellman key size** (bits)	**Elliptic Curve key size** (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Figure C.1: A table showing the key sizes needed to achieve levels of security in both the traditional RSA/Diffie-Hellman groups and in Elliptic Curve groups. Taken from [28].

Elliptic Curve Groups are usually represented in Additive notation, differing from the usual Multiplicative notation used for groups in cryptography. This means when we add points together where we would usually multiple elements and apply scalar multiplication to a point where we would raise an element to the power of a scalar.

We define a curve of the form $y^2 = x^3 + a \cdot x + b$ modulo some prime q, call this curve C. Say n is the number of bits required to represent q, then we say this is an n-bit curve.

We define the group by the set of elements and the group operation. The set of elements we give as,

$$\{(x, y) \in \mathbb{Z}_p^2 : \text{ where } y^2 = x^3 + a \cdot x + b\}$$

We will use the most intuitive representation of points on elliptic curves, namely just the (x, y) coordinates. We denote the identity in the group to be $(@, @)$ and the inverse of an element (x, y) is simply $x, -y)$.

This representation is sometimes called the *Affine* representation, other representations exist and are used when modular inversion are expensive as their operations reduce the number of inversions required for each group operation. Due to the speed of modular inversions in GMP we have opted to use Affine representation reducing bandwidth needed to send an element.

Take $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, then $(x_3, y_3) = P + Q$. Then,
if *($x_1 = x_2$* AND *$y_1 \neq y_2$) OR ($P = Q$* AND *$y_1 = 0$)* **then**
  $(x_3, y_3) = (@, @)$
else
  if *($P \neq Q$* AND *$x_1 \neq x_2$)* **then**
    $x_3 = (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2$
    $y_3 = (x_1 - x_3) * \frac{y_2 - y_1}{x_2 - x_1}$
  else
    $x_3 = (\frac{3 \cdot x_1^2 + a}{2 \cdot y_1})^2 - 2 \cdot x_1$
    $y_3 = (x_1 - x_3) * \frac{3 \cdot x_1^2 + a}{2 \cdot y_1} - y_1$
  end
end
Algorithm C.1: The group operation of the group of point on an Elliptic Curve defined by $y^2 = x^3 + a \cdot x + b$ in Affine Representation.

C.2.1 Elliptic Curve point scalar multiplication

As we noted above scalar multiplication of points is equivalent to Diffie-Hellman group exponentiations. As such we use scalar multiplication very often.

Take a point P and an integer n, consider $n \cdot P$, whilst we could compute this by $\sum_{i=1}^{n} P$ this would require n many additions. Where n can be very big (say 256-bits as in our group) this will require a stupendous number of group operations.

Many of the same tricks that can be applied to integer exponentiation also work here. For example the square-multiply trick (though here it is double-add). Many of these tricks depend on taking advantage of thinking of the binary form of the exponent and using doubling.

For standard point multiplication we have implemented the Windowed approach. Take a point P and a scalar n. We pre-compute a set of multiplications of P, namely

$\{w_i : w_i = i \cdot P\}_{i=0}^{2^w-1}$ where w is the size of the windows in bits. We then consider the exponent in the form of w-sized windows, call the integer representation of the window d_i.

$Q = 0$;

for $i = m$ *to* 0 **do**
  $Q := 2^w \cdot Q$ (using repeated point doubling);
  if *(* $d_i > 0$ *)* **then**
    Q := Q + $d_i \cdot P$;
    // Compute $d_i \cdot P$ using pre-computed values.
  end
end
Return Q;

Algorithm C.2: Windowed Scalar Elliptic Point Multiplication.

Fixed Point Scalar Multiplication

What if there is some point which we shall be scalar multiplying very often? For example many cryptographic protocols repeatedly take scalar multiplications of the generator of the group.

For such a point P we can pre-compute the values $\{w_i := 2^i \cdot P\}_{i=0}^{l}$ where l is the size in bits of the order of the group. When computing some $k \cdot P$ we can output $k \cdot P = \sum_{i=0}^{w}(k_i \cdot w_i)$ where k_i is the value of the i^{th} bit in k.

This method of fixed point scalar multiplication is regularly cited as being three times faster than standard Windowed Scalar multiplications (see [2] for example).

C.3 Verifiable Secret Sharing and Multi-precision Polynomials

For the Zero Knowledge Proof of Knowledge specified in [3] we need a Secret Sharing Scheme. For [6] we need to go one step further and have a Verifiable Secret Sharing Scheme.

A Secret Sharing Scheme is a way of obscuring a secret whilst distributing shares to a set of parties such that only certain combinations of shares will be able to reconstruct the secret. So consider perhaps a bank vault which requires at least 3 out of 10 keys. Here the secret is the vault opening, the shares are the keys and the parties are the bank employees holding the keys. In general we speak of a t-out-of-n scheme, where there are n shares and t of them are required to reveal the secret.

For a fairly comprehensive overview of Secret Sharing I suggest pages 349-360 of [30].

We have implemented Shamir's Secret Scheme (Shamir's) and its extension the Feldman's Scheme. Shamir's scheme is based on how many points are needed to uniquely define a polynomial curve.

Consider a polynomial K of degree n over the finite field $\mathbb{F}_q$. Then we can denote this polynomial as $K = \sum_{i=0}^{n} a_i \cdot x^i$. Any such polynomial of degree n can be uniquely defined given $n + 1$ (or more) points on the curve, given n or fewer points we gain no information about the polynomial.

C.3.1 Multi-precision polynomials

In order to use Shamir Secret Sharing we need an implementation of polynomials, furthermore in order to deal with the secrets of the the size we shall need to be dealing with we shall need Multi-precision polynomials.

While several libraries exist with support for Multi-precision polynomials these are not commonly installed and given the ease of using GMP it was much simpler to implement Multi-Precision polynomials ourselves.

We will not dwell on the details of this as this was quite trivial and is tangential. Suffice to say we have a structure for polynomials in a field, this structure contains a degree and a set of coefficients. We then coded functions to perform addition, multiplication and evaluation.

C.3.2 Shamir Secret Sharing

Shamir secret sharing was first proposed in [31] and gives a way to implement a Secret sharing scheme. The scheme consists of two algorithms, `Share` and `Recover`. We assume that each algorithm takes the field over which we work as an implicit input ($\mathbb{F}$).

`Share` takes a secret $a \in \mathbb{F}$ and a pair of integers t and n such that $t \leq n$. It returns a set of n shares (also elements in the field) such that $t+1$ many are needed to recover the secret a. Note that the shares are indexed in the order they are output by `Share`.

More concretely, to share a secret a we generate a polynomial $F(X) = a + f_1 \cdot X + f_2 \cdot X^2 + ... + + f_t \cdot X^t$. We then output $\{c_i = F(i)\}$ as the shares, note then that $F(0) = a$. The polynomial is *not* known to the parties.

`Recover` takes a set of $\tilde{m}$ shares $\{c_i\}$ where i indicates the index of the share. `Recover` returns a $\tilde{a} \in \mathbb{F}$. If $t \leq \tilde{m}$ and the c_i are all valid shares then $\tilde{a} = a$.

If we have $t+1$ or more valid shares of the secret then we can reconstruct the polynomial F by Lagrange interpolation. We will not dwell on the details of Lagrange interpolation.

C.3.3 Verifiable Secret Sharing

A Verifiable Secret Sharing (VSS) scheme is an extension to 'vanilla' secret sharing schemes where any party can check whether an input is a valid share to a secret. This additional property is very important for the protocol described in [6].

We have implemented the Feldman VSS scheme which is an extension of the Shamir scheme. The basic concept is that the sharing step also publishes a public commitment to the shares to all parties. Then using the candidate share, the index of the candidate share and the commitments any party can verify whether the candidate share is a valid share of the secret.

Recall the polynomial used for the scheme is of the form $F(X) = a + f_1 \cdot X + f_2 \cdot X^2 + ... + + f_t \cdot X^t$, and take g to be a generator of the field. Then the public commitments to the shares of this polynomial are,

$$p_0 = g^a, p_1 = g^{f_1}, p_2 = g^{f_2}, ..., p_t = g^{f_t}.$$

As the name suggests these public commitments are sent to all parties. Then given a share c and an index for the share i any party can verify that c is the valid i^{th} share by computing $\prod_{j=0}^{t} p_0^{ij}$ and testing this equals g^c. If it does it is a valid share, else it is not.