Dissertation Type: Research - GG1K

UNIVERSITY OF BRISTOL

DEPARTMENT OF COMPUTER SCIENCE

Secure Two Party Computation

A practical comparison of recent protocols

Nicholas Tutte

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.

Tuesday 28th April, 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, Tuesday 28th April, 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 modifying .
- 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 [28].

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.

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 [14] 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], [11], [12], [13]) and to improve the efficiency of the original protocol itself ([9], [10]).

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 it's inputs as the function of other parties inputs.

In this sense we define the goal of an adversary to compromise any one 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*. Informally this means if one party gets their output then all parties get their output. However, generally this is omitted 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. Mostly Semi-Honest security can be used 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 unecessarily 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 [16].

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 then a sufficiently high deterrent probability 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. 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, but if we can split the seed across multiple servers and then get the servers to perform the computation as a SMC problem (where each server's input is their share of the secret, the output the value to compare to the token's input) then we can increase the cost to an attacker, as they will now have to compromise multiple servers. Such a service is in

development by Dyadic Security (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

Chapter 3

Technical Background

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(called the receiver) to get exactly one out-of two (and can be extended to k-out-of-n for $k < n$) values from another party (called the Sender). The receiver is oblivious to the other value(s), and the Sender is oblivious to which value the receiver received.

Oblivious Transfers were first suggested by Rabin in [17]. We define the functionality of a 1-out-of-2 OT protocol in Figure 3.1. Oblivious Transfers are vital to Yao Garbled Circuits, used to give the circuit Executor data it needs to evaluate the circuit under their input without revealing to the circuit Builder what those inputs were.

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 [20] 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 [21] 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 including proofs of security see [21].

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.

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 $\mathbb{G}$. 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.

Sender	Group $\mathbb{G} = (G, g, q)$	**Receiver**
$x_0, x_1 \in \{0,1\}^l$		$b \in \{0,1\}$
$C \xleftarrow{\$} G$		
	$\xrightarrow{C}$	
		$k \xleftarrow{\$} \mathbb{Z}_q$
		$h_0 \leftarrow g^k$
		$h_1 \leftarrow C/g^k$
	$\xleftarrow{h = h_b}$	
$r \xleftarrow{\$} \mathbb{Z}_q$		
$a \leftarrow g^r$		
$c_0 \leftarrow H(h^r) \oplus x_0$		
$c_1 \leftarrow H((C/h)^r) \oplus x_1$		
	$\xrightarrow{a, c_0, c_1}$	
		$y \leftarrow a^k (= g^{r \cdot k})$
		$x_b \leftarrow H(y) \oplus c_b$
		Output x_b

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 [20], 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

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.

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

- **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 the define a Dual-mode cryptosystem.

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 [20], 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.

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

- **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

- **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.

Trusted Setup - Peikert et al. state that the OT they provide requires a trusted setup. They claim this is a reasonable assumption and can be achieved using shared randomness, they suggest sunspots. To understand what the problem and why trusted setup is required consider the following case.

Suppose the Receiving party performs the setup, recall however, that the Sender will not be able to tell if the crs they are given by the Receiver is a Messy crs or a Decryptable crs. Therefore it is then childs play for the Receiver to produce a Decryptable crs and to keep the trapdoor, allowing the Receiver to decrypt on both branches giving access to all of the Senders inputs.

One might hope letting the Sending party perform setup might be better, in fact it is much worse. Clearly the clear logical reverse of the Receiver case is possible, where the Sender performs a Messy setup and as such can then, using the resulting trapdoor values, uncover what values the Receiver is requesting.

In short, both parties could potentially craft a malicious crs during setup allowing them to subvert the properties of the Dual-mode system, so the solution is to divide the work between the two parties so each provide some of the crs in such a way that neither has enough information to create the trapdoor that is worth anything to them .

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 [14]. 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. Further P_1 also generates a random permutation $\pi_i \in \{0,1\}$ for each w_i. 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.

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

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. 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 it's inverse D_K) as follows,

$$E_K(m) = AES_{k_1}(AES_{k_2}(...AES_{k_n}(m)...)), \text{ where } K = \{k_1, ..., k_n\}$$
$$D_K(m) = AES^{-1}_{k_n}(AES^{-1}_{k_{n-1}}(...AES^{-1}_{k_1}(m)...)), \text{ where } K = \{k_1, ..., k_n\}$$

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 (the builder of the circuit) sends this garbled version of the circuit to P_2 (the executor of the circuit). P_1 should send the garbling key for it's input bit ($k_1^{b_1}$), the full encrypted truth table and $c_1 = \pi(b_1)$. 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 $k_3^{G(b_1,b_2)} \| c_3$. Then by using π_3^{-1}, P_2 can extract a value for $G(b_1, b_2)$.

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 it's 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.

This is done 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 then whilst k_0 is generated randomly as usual we take $k_1 := k_0 \oplus R$. Then take any .

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 [15], 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 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. Mostly depending 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, one party cuts the cake in two, then the other party chooses a slice. 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.

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

This Cut and Choose seems very simple conceptually, but 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, each revealing something they should not have been able to discover.

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 which circuits have been chosen.

Having obtained its input keys the Executor then prove to the Builder what subset of the circuits it opened as check circuits so that the Builder can provide both keys for each of its input wires allowing for the opening of the whole circuit.

Thirdly, given all keys for the inputs wires how do we 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.

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 Executors input fulfils some condition (e.g. if the first bit is 1). Then the Executor aborting due to differing outputs from Evaluation Circuits leaks information, namely whether the Executor's input satisfies the condition or not.

In the classic protocol the Executor returns the majority output on each output wire. Suppose that we have s many circuits, t of which are selected as check circuits. Then clearly the output will only be corrupted if half of the Evaluation circuits are corrupted. This means that the Builder needs 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.

Chapter 4

Protocols

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.

4.1 Lindell and Pinkas 2011

Overview

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

Firstly 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 $\epsilon = 1 - 2^{-0.32s}$. Secondly it removes the need for the very large number of commitments entailed in [1] and thirdly it 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.

Cut and Choose Oblivious Transfer

The main new idea in this protocol is a modification of the PVW-OT from [20]. 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 represents a subset of the s circuits to be opened.

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 circuit it received all inputs, in doing so it commits to a set of check circuits. The Builder can then reveal all information required to fully decrypt the check circuit, allowing the Executor to test the correctness of each check circuit. 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 only $\frac{s}{2}$ 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.

Consistency of building party'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 $\{g^{r_j}\}_{j=1}^s$ and uses $H(g^{a_i^0 r_j})$

4.2 Lindell 2013

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 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.

Secure Computation to detect cheating

The builder constructs all the circuits so that the keys for output wires are consistent across all circuits, call these consistent output keys $\{b_i^0, b_i^1\}_{i=1}^s$. 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 (it's original input to the main compu-

tation) and the Executor inputs b. The Secure Computation returns x to the Executor if it's input b indicates it knows both b_i^0 and b_i^1 for some i, otherwise it returns garbage.

Clearly we need to be sure that 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.l

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 shall take these optimisations in one bound, 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, given only one of the pair the Executor gains no knowledge of δ. The executor's input to the sub-computation is then 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.

Thirdly, we can eliminate completely the comparison between δ and δ' with an elegant use of the OTs we were already going to have to perform. Suppose that we reduce the Executor's input to a single bit, indicating knowledge of δ.

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 the same occurs for our partner. This situation is only possible if both parties cheated, in which case we care little for their 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 $3/2$ less circuits for the same security level as [3].

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 the P_1 (resp. P_2) needs to know that P_2 (P_1) gave consistent inputs

to both the circuits P_2 (P_1) created and 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

$$k^0_{i,j} = g^{a^i_0}$$
$$k^1_{i,j} = \tilde{C}/g^{a^i_1}$$

Now consider the value of $A = h_i/k^b_{i,j}$ 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 $\tilde{C}$ 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^0_i, S^1_i) where i is the output wire. In the case of P_2 label them (T^0_i, T^1_i).

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^b_i with S many shares and a threshold such that $\frac{S}{2} + 1$ many shares are needed to reconstruct the secret. Label these shares $W^b_{i,j}$, 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 $\{W^0_{i,j}, W^0_{i,j}\}, \forall j \in J, \forall i$. This means P_1 is now in possession of $\frac{S}{2}$ many shares for S^b_i, 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 less circuits to achieve the same level of statistical security as Lindell-Pinkas, can we alter the sub-computation to use HKE?

This is a very simple concept, though we must be careful of a few subtle problems that do not present themselves on first glance, 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.

   We have not implemented a solution to this problem as such it is left open. Our instinct tells us that one could extend the circuit to take a second string from the Executor of the same size as the output and to XOR this additional input with the 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.

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.

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 it's 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'.

Chapter 5

Implementation Details

We do not have the space to go into depth about our implementation, however we shall touch on some of the high points.

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.

5.1 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 [22].

5.1.1 Tillich-Smart Circuit Files

We are using the circuits provided by [22], these circuits have been crafted with Yao Garbled Circuits in mind, applying some of the optimisations suggested in [9] 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 outputs. 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.

5.1.2 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. 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.

5.2 Elliptic Curve implementation

Throughout unless otherwise stated we have worked in Elliptic Curve groups, in particular on the curve *brainpoolP256r1* specified in [25]. This is a 256-bit curve and as such provides 128-bits of security. I suggest [23] as a high-level primer on ECC and [24] for a more technically detailed introduction.

For a quick reference on some of the algorithms we use for operations I warily suggest [27], 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. This is particularly desirable as it reduces the size of the number we are dealing with making computations quicker without sacrificing security. This point is illustrated in the Figure 5.1.

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 5.1: 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 [26].

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 then need to define the group by the set of elements and the group operation. The set of elements is simply defined 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 mostly exit as their operations reduce the number of modular inversions required for each group operation. Due to the speed of modular inversions in GMP we saw barely any performance improvement when trying Homomorphic coordinates and so have opted to stick with the simpler Affine representation.

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 5.1: 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.

5.2.1 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. This method pre-computes a number of multiplications of P, namely $\{0 \cdot P, 1 \cdot P, ..., 2^{w-1} \cdot P\}$ where w is the size of the windows in bits.

the exponent is then divided into windows.

5.3 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.

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.

5.3.1 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. With functions complete we were ready to attempt Shamir Secret Sharing.

5.4 Peikert, Vaikuntanathan and Waters OT

.

5.4.1 Other - AES, SHA-2

.

Chapter 6

Experiments

We shall be using the circuits provided in [22] for our experiments with varying randomised inputs, in particular we shall consider

- AES,
- 32-bit Addition,
- 32-bit Multiplication,
- SHA-256 hashing.

6.1 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 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.

6.2 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.

6.3 Results

We now give results of using each of the Protocol on the test circuits, we have generated a large set of random inputs for each circuit using a simple python script. This allows use to use the same inputs for each Protocol whilst also trying a range of random test inputs.

We give a short précis to each circuit detailing the expected results and a small amount of analysis of the results, further analysis of results comes later. All tests are configured to given a deterrent probability of $1 - 2^{40}$, or put in other terms, a statistical security parameter of 40.

6.3.1 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 32-bit integers.

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. Addition and Multiplication are both fairly silly examples for Secure Two Party computation as the functions completely leak the other party's input to each party.

6.3.2 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. 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 import the 'depth' (size of circuit discounting inputs) of the circuit has with regards to performance.

6.3.3 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 [22] for this computation has $\sim 39,000$ gates.

6.3.4 SHA-256 Hashing

The SHA-256 Hashing circuit is the largest we shall be testing upon with $236,112$ gates (around 6 times larger than AES. The circuit takes a 512 bit input from one of the parties and outputs the SHA-256 hash of this input.

This test should heavily favour Lindell's 2013 protocol over the Lindell-Pinkas 2010 protocol due to the size of the circuit. Of particular interest is the comparison of the HKE protocol versus the both variants of Lindell's 2013 protocol.

Chapter 7

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

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

[10] 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.

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

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

[13] 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.

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

[15] 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).

[16] 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.

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

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

[19] 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)

[20] 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.

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

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

[23] 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/`.

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

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

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

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

[28] 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 guide

This chapter deals with how to build and use the implementation provided. Furthermore it gives a short summary on each source code file and what its purpose is. 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

.

B.3 Source Code Documentation

.