

Secure Integer Computation with Applications in Economics

Ph.D. Progress Report by Tomas Toft

July 2005

Contents

1	Introduction and Motivation	5
1.1	Overview	6
1.2	Acknowledgements	7
2	Setting	9
2.1	The Parties	9
2.2	Overall Properties and Universal Composability	10
2.3	The Functionalities	10
2.4	Required Sub-Functionalities	11
3	General Computation	13
3.1	Complexities	13
3.2	Security Model	14
3.3	Sharings	15
3.3.1	Generating Sharings	15
3.4	Simple Computations in \mathbb{F}	17
3.4.1	Ideal Functionalities	17
3.4.2	Protocols for Simple Computations	17
3.5	Partial Single-TTP-Trust	18
4	Integer and Binary Computation	21
4.1	Sharing Integers	21
4.2	Sharing Bits and Binary Computation	22
4.2.1	Bits in $GF(2^8)$	23
4.2.2	Bits in \mathbb{Z}_p	23
5	Integer Comparison	27
5.1	The Protocol	27
5.2	Correctness	30
5.3	Complexities	31
5.4	Security	32

6	The Double Auction	35
6.1	Computation of the MCP	35
6.1.1	Bids and the MCP	36
6.2	The Double-Auction Implementation	38
6.2.1	Straight-Line Programs	38
6.2.2	Computing the MCP	38
6.3	Round vs. Communications Size Tradeoff	40
7	Conclusions and Future Work	43
7.1	Future Work	44
A	Choosing Shamir over Paillier	47
B	The Double-Auction in Pseudo-Code	49

Chapter 1

Introduction and Motivation

Consider the following problem. A number of parties wish to compute a function based on inputs they each have, however, they will not reveal these inputs to anyone. The solution to this is multi-party computation (MPC), which allows the parties to evaluate the function by working together. Concretely, the parties execute a protocol allowing them to determine the result without revealing their inputs.

Multi-party computation has been rigorously studied, and many theoretic results are known. However, though large-scale applications have been considered in theory, relatively few attempts of investigating their feasibility in practice have been made. The work presented here describes the theoretic background of the SCET project — Secure Computation, Economy, and Trust. The project is a collaboration between economists and cryptographers, the overall goal being to construct concrete MPC-solutions for real-world problems in economics. Thus the project is two-sided, focusing both on designing economic solutions as well as implementing MPC-systems.

The motivation for joining MPC and economics is that in economics, it is known that many forms of interaction can benefit from the existence of a trusted third party, known as a *mediator* or *social planner*. All players reveal all private information to this party who then determines the overall best solution. Such an independent party may be difficult to agree on or implement in practice, however. The private information, may be trade-secrets which the parties will be disinclined to reveal, fearing misuse. Numerous real-life experiences justify this fear.

MPC allows us to implement a mediator using multiple third parties rather than a single one. The third parties simply execute a secure computation determining the optimal solution, as the mediator would do. This requires less trust in the system, say that security is ensured if half of the third parties remain honest. As there is no longer a “single point of failure,” trust is distributed from one to many. Doing this in practice may result in a cheaper mediator, or obtaining one, where no single third party could be agreed on.

The goal, seen from the perspective of economics, is to design rules for interactions — known as mechanisms — which may benefit from having a mediator. There are numerous properties to consider in order to ensure theoretically sound solutions, however, the focus here is on the multi-party computation aspects and soundness will simply be assumed.

Auctions may be thought of as sets of trading rules, ie. mechanisms. There exists both theoretical work and a myriad of practical examples of such interactions. Moreover there are also countless examples of cheating auctioneers (mediators), thus lessening trust is desirable. In this work, we focus on a specific auction, namely the double-auction. This is essentially an exchange: Multiple parties wish to trade multiple units of some good and the goal is to determine a market price at which trade will occur. The SCET project has an industry partner with concrete settings where running such an auction using MPC would be advantageous. Trust in this setting could be distributed to eg. trade associations, government agencies, and independent accounting firms.

An additional auction — which also has immediate real-world applications — is a “triple double-auction.” This consists of three concurrent exchanges, where the bids on the markets may be interdependent. Work is currently being performed on the economic aspects of this. For a more in depth description of both auctions, see [2] or [3].

1.1 Overview

The work presented here focuses on providing the theoretical basis for secure integer arithmetic, including a comparison operation. This work is based partly on [1], as well as several new ideas. This functionality provided may be used to implement mediators of economic mechanisms. As an example, a secure double-auction is constructed based on our integer operations. As such, there are two concrete goals of this report. The first is to construct the actual integer computation functionality, most notably the comparison operator. The second is to implement a double-auction based on the functionality given. Though the nature of this report is theoretic, we also consider the SCET implementation, discussing the real-world complexities.

It should be noted, that this is a work in progress. The solutions presented have all been implemented, though not with all optimisations discussed in the text. In addition, security is to be shown through equivalence to ideal functionalities. These are presented loosely and have not undergone rigorous scrutiny, thus they may contain flaws. The proposed functionalities are a “first attempt” at their description, and better strategies with regard to the division of functionalities into sub-functionalities are very likely. The above also implies that formal simulator proofs have *not* been performed at the current date; we merely sketch why the protocols are expected to be secure. Up until now, the main focus of the

work has been the actual strategies — defining protocols as efficiently as possible. Strict proofs have therefore been postponed.

Considering the contents of this report, chapter 2 presents the setting (the players) and the overall functionalities desired, ie. the integer computation. In the following three chapters these functionalities are implemented. Chapter 3 provides an overview of the security and complexity models considered as well as a presentation of the fundamentals of our implementation. In chapter 4, the basics of our secure integer arithmetic are constructed on the results of the previous chapter. In addition to this, assisting functionality is presented. Chapter 5 presents the final part of the solution to our first goal, namely a secure protocol for integer comparison. Following this, in chapter 6 we present an implementation of a double-auction based on the functionality of chapter 2.

1.2 Acknowledgements

The author would like to thank Ivan Damgård for guidance and all members of the SCET project for cooperation and fruitful discussions. Dina Friis is thanked for proof reading and for support in general.

Chapter 2

Setting

This chapter presents the overall setting of our computations. First the different parties are presented and following this, there is a short description of the Universal Composability framework, the intended strategy for the proofs of security. Finally, we consider the overall desired functionality and the means for implementing it.

2.1 The Parties

In our settings, we split the participating parties up into two groups, the *inputters* and the *trusted third-parties* (TTP's). The former supply inputs for the computation — eg. they are bidders in an auction — while the latter perform the actual computation. This is the distribution of trust from one party to many. Concretely, we will require that the system remains secure if at most t TTP's collude. We say that t is the threshold of the system.

Naturally, there may be overlap between the two groups, it could even be the case that each inputter was also a TTP. However, actual security may not be increased by this. If the inputters are ordinary people, having few TTP's run on dedicated machines secured by professional administrators may be better than using multi-purpose home computers. In addition to this, performing the actual computation may be infeasible due to a large number of parties. There may be hundreds or even thousands of inputters, while the number of TTP's, n , is low. Typical values for (n, t) are expected to be $(3, 1)$ or $(5, 2)$.

In our implementation, there is an additional participant, namely the *coordinator*. This party initiates the computation, collects inputs, implements broadcast functionality, etc. However, this party has no responsibilities towards the computation beyond infrastructure. Thus, though it exists in the code, it will be ignored in the descriptions below. It is assumed that messages are delivered straight to the recipient(s) rather than being transferred through the coordinator.

2.2 Overall Properties and Universal Composability

Two main concerns of secure computation are the secrecy of the inputs and the correctness of the computed result. In addition to these two concerns, there may be other setting-specific properties, eg. that an auction bid is binding. These must be considered separately when constructing algorithms based on the functionality described here, which provides only secure computation.

The intended strategy for the security proofs is to perform them within the Universal Composability framework (UC framework), [5]. The idea in this framework is to show that a protocol is equivalent to a so-called *ideal functionality*, which cannot be corrupted. An ideal functionality consists of a truly honest, unhackable third party who receives inputs from all players, computes the output for each player, and returns these. All communication is secure by definition. Naturally, as we may specify any behaviour from the ideal functionality, this allows us to define secure interactions. Thus, anything shown equivalent to this — in a well-defined sense — is also secure. Clearly an ideal functionality may be viewed as an honest mediator.

The UC model has been chosen, as it allows protocols to be shown secure in any arbitrarily complex setting. In particular, it allows multiple instances of the protocols to be run in parallel. Doing this is very much necessary for efficiency reasons in our implementation. Experiments have shown dramatic speedup, when operations are performed “batched”, see [2]. In addition to this, the composability theorem of the UC framework allows sub-functionalities to be “plugged in”, ie. simple operations may be combined to more complex ones easily. This facilitates security proofs, moreover, when combining functionalities, the actual implementation is unimportant, only the ideal sub-functionality matters. Thus, sub-protocols may be interchanged freely, allowing us to “plug-and-play” different implementations of the same functionality if better solutions are discovered.

The ideal functionalities described in this report are the result of preliminary work. As noted, they are presented somewhat loosely and have not been rigorously verified. It is stressed that the current division into sub-functionalities is simply one possibility, and others may prove to be better. Concretising the functionalities and performing proper simulator proofs has high priority in the ongoing work.

2.3 The Functionalities

We now describe the intended functionality of our implementation. Intuitively it may be viewed as a secure computer capable of receiving non-negative integer inputs and performing a number of operations on them. This functionality may then be replaced by a network of TTP’s. Invocations proceed in the following

three steps:

1. The inputters supply their inputs, as well as a description of the computation to be performed.
2. Based on the inputs, the output(s) are computed by executing a number of straight-line programs consisting of operations taken from a fixed set described below.
3. The outputs are sent to the relevant inputters.

As noted, this can be viewed as a fully trusted mediator. The programs may be viewed as function calls which store the result in a variable, The straight-line programs are publicly known and assumed to be part of the specification. Straight-line simply denotes that the programs are non-branching. The overall description of the computation may simply be seen as a program using relevant function calls to implement the computation on secure variables. For practical reasons, the integer values are considered to be of some fixed bit-length, l , though this may be chosen arbitrarily.

Branching may occur *between* straight-line programs, ie. the programs to run and their inputs may be determined at run-time. However, branching may not be performed based on variables of the system immediately — this would imply, that the TTP’s would have knowledge of the branching condition, which they may not. Thus we consider two distinct types of variables, *secret* and *published*. Only published values may be used in selecting which program to run as well as on which inputs. Publishing a value simply refers to giving the TTP’s explicit knowledge of the variable. Thus in proofs of security, it must be verified that “leaking” this information is acceptable.

The set of operations on secret integer variables is small. In addition to publishing a variable, we may add or multiply two variables. We also implement comparison of two values, \geq . The result is an integer representing a boolean value, 0 for FALSE and 1 for TRUE. These operations are sufficient in order to implement a number of useful algorithms, in particular the secure double-auction described in detail in chapter 6.

2.4 Required Sub-Functionalities

In this section, our sub-functionalities are introduced. These are primarily ideal functionalities implementing the operators of the computation-step. This results in a list of functionalities to be implemented, which is done bottom-up in the following chapters.

Before presenting the operations, the distribution of trust is introduced. The idea is to use a threshold secret sharing scheme: Each of a number of parties have some part of a secret. No information on the secret may be obtained from

t shares, however, the secret is uniquely defined by any $t + 1$ shares. The value, t , is known as the threshold of the system. Sharing schemes are well known, and several possible choices exist. This allows the distribution of knowledge of variables — inputs and intermediate values — from one party to many, intuitively distributing trust.

As noted, our goal is operations on l -bit integers, namely addition, multiplication, comparison (\geq), and publishing. The latter is simply reconstruction in the secret sharing scheme. The rest follow a specific structure: The TTP's deliver their shares as input. Based on these, the ideal functionality reconstructs the values, computes the desired result, and returns shares of this to the TTP's. Intuitively this releases no information, assuming that the sharing scheme is secure.

In addition to the above, we will require computation on individual, shared bits. Often, working on the individual bits of a value may be required or advantageous, when performing more complex operations. This is the case for our comparison. Thus, to ensure the efficiency of this “bit-fiddling,” explicit ideal functionalities are considered, allowing us to tailor solutions rather than using integers to store the binary values. These functionalities follow the same structure as the ones for integer computation. The binary operations required are all basic: **and**, **or**, **not** and **xor**. These operations are sufficient to compute any function and detailed enough to provide an efficient implementation of a comparison — actually, **and** and **xor** are sufficient for our purposes, however, the rest are included for completeness.

A final ideal functionality to consider is the actual sharing of a secret, ie. sending shares to the TTP's. This is required to submit inputs to the system as well as to implement most of the operations. As we consider a large number of inputters and large computations, naturally, an efficient implementation is desired.

Chapter 3

General Computation

In this chapter, we first introduce and discuss the relevant complexity and security models. Following this, we present the choice of secret sharing scheme as well as (non-standard) protocols for sharing values. Based on the choice of sharing scheme, ideal functionalities are provided for computation, protocols implementing these are then briefly sketched. Finally, the concept of *partial single-TTP-trust* is introduced, which potentially allows a temporary increase of the security threshold at the beginning of a computation.

3.1 Complexities

When considering complexity of our operations, we focus on two aspects: Round complexity and communication complexity. These cover the number of “steps” in the protocol and the number of bits communicated during the entire execution. Both are concerned with the communication taking place during a computation, as this is our primary resource bottleneck. However, their focuses are very different, even resulting in possibilities of tradeoffs of one versus the other.

The sending of data from point A to point B , as well as waiting for it to arrive, are vastly more time consuming than local computations. Raw computing power is cheap, so while efficient algorithms for local computations are preferred, their exact details and therefore also analysis, are not relevant.

As noted, round complexity refers to the number of steps a protocol takes. A round simply consists of all parties sending a message to all others, no-matter how much data is sent. The reason for concentrating on this, is that such steps have shown to be very expensive, easily taking milliseconds where the CPUs are idle. The reason for this idle time is latency in the network, thus, parallelising multiple operations — which naturally may not depend on the output of each other — is therefore a logical step. Additional operations come for free, as the amount of data is irrelevant in this model.

In addition to round complexity, we also consider the communication com-

plexity, ie. the number of bits communicated. This is due to the fact that overall bandwidth is a limited resource. As we consider multiple TTP's run by different organisations, it is logical to assume that the actual computers used to execute a computation are not necessarily physically close — eg. they are connected through the internet. Though it is possible to invest in fast network connectivity over large distances, keeping the size of the communication low is advantageous, as requirements will be cheaper to implement. In addition, more efficient operations allow us to perform larger computations.

Note, that though we may parallelise multiple operations to gain round efficiency, the exact same bits required for serial computation must be communicated in the parallel version — the gain comes through a reduction of idle-time. We may also encounter round/communication size tradeoffs: It may be possible to reduce the round requirements by performing additional work which parallelises better. Through this, the overall running time may be reduced, though naturally this depends on the underlying network.

3.2 Security Model

The security we consider is against a passive attack (also known as the “honest but curious”-model) by a static adversary. The adversary specifies corrupt participants before protocol execution begins, and corrupted parties still follow the protocol — they only pool their knowledge to learn as much as possible. This is a standard definition of security. We may alter the protocols below in order to ensure security against adaptive adversaries as well, however, this comes at an efficiency price, as we cannot use the techniques of pseudo-random secret sharing in this security model.

The background for using passive rather than active security is primarily one of simplicity. An implementation guarding against an active attacker is (most likely) less efficient than one where only privacy is considered, as all parties follow the protocols. As we are concerned with the practicality of MPC, this is a valid point. However, the primary reason for not using a stricter security model is simply one of simplicity of implementation. Constructing a working implementation without active security is simpler than constructing one with. Extending our implementation to also consider security against an active attacker has therefore initially been postponed.

When considering security against an active adversary, there are two possibilities, namely ensuring that the computation terminates with the correct result or detecting that an attempt at cheating has occurred. There are standard techniques for implementing both of these possibilities for our low-level protocols. Though the former is a desirable property it is also more expensive in terms of complexities, and there are real-world settings, where the latter may suffice. If we consider eg. a double-auction as described in chapter 1, the TTP's are trade

associations, government agencies, independent accounting firms, etc. All parties have an interest in the computation running smoothly — public embarrassment will be the deterrent against cheating. Note though, that our high-level protocols will most likely have to be “tweaked” to provide security in this setting.

3.3 Sharings

Concerning secret sharing scheme, we have chosen shamir sharings as the basis of our system, see [9] for a full description. Other possibilities were considered, most notably a threshold variant of Paillier encryption. However, this strategy was discarded in favour of shamir sharings, due to size considerations. For a full discussion, see appendix A. Concerning notation, let \mathbb{F} be a field. A sharing of a value, $v \in \mathbb{F}$, is then written as $[v]_{\mathbb{F}}$. The field is written explicitly in the notation to avoid ambiguities. This notation will be used throughout the following chapters.

Recall that in the shamir secret sharing scheme, arbitrary threshold values may be chosen, ie. for a sharing between n parties, we may specify a threshold t , such that any $t + 1$ may reconstruct the secret fully, while any t have no information at all. When computing on shamir shares, a threshold of $\lfloor \frac{n-1}{2} \rfloor$ is standard (when considering a passive adversary), this will also be the threshold used here. Note that this implies security if any majority remains honest. Below we introduce Partial single-TTP-trust, which to some extent counters this “low” threshold value.

Note that having chosen shamir sharings, the reconstruction protocol simply consists of each party broadcasting its share. This implements the publishing functionality of chapter 2.

3.3.1 Generating Sharings

We now consider the generation of sharings. The most important being the possibility of having a party share a value. The ideal functionality for this is simple, the value is sent in and sharings are constructed and distributed correctly. Before implementing this, however, we introduce functionality for sharing a uniformly random value unknown to all. This is then implemented through pseudo-random secret sharing (PRSS). An overview of the technique is given below, see [6] for a full description of the technique. A variation of this is then used in a protocol implementing the sharing functionality. In the rest of this section, let n equal the number of TTP’s and t the corruption threshold.

As noted. the basic ideal functionality desired is the ability to create a sharing of a uniformly random variable unknown to all. Ie. the ideal functionality picks a uniformly random value, creates shares, and distributes these to the TTP’s. A “protocol” implementing this is PRSS. The basic idea of this is to consider

all subsets of the TTP's of size $n - t$. Each subset is assigned a pseudo-random generator, and these are distributed to all members of the subsets. For each subset we construct a sharing of a uniformly pseudo-random elements known to the parties of that subset. Adding these result in a sharing of a uniformly pseudo-random element unknown by all. Below it will be argued that performing these additions can be done without any communication at all. The whole operation may therefore be performed without communication providing.

A variation of the PRSS technique may be used for shamir sharing a value, $v \in \mathbb{F}$, using only a broadcast. This rests on the assumption that generators have been distributed in advance. The idea is to not only distribute generators as above, but also provide all pseudo-random generators to the party who will be performing the sharing. Thus, if $[r]_{\mathbb{F}}$ is created using PRSS as described above, the sharing party knows r , and may compute $v' = v - r$. This is then broadcast and all parties compute $[v]_{\mathbb{F}} = [v' + r]_{\mathbb{F}}$. As r is uniformly random, it masks v , thus seeing v' provides no info on the value shared. Functionality for computing such an addition without communication will be provided below. Doing this provides us with ideal functionality for sharing values. As the implementation only requires the broadcast of a single value, the complexity is f bits broadcast in one round, where f is the number of bits required to encode an element of \mathbb{F} . In practice, generators must be distributed, however, this must only be done once. As such, this may be considered a preprocessing step.

Note that this construction may be used to share multiple values efficiently. First compute generators for all subset and distribute these (securely) to the relevant parties. Following this, broadcast a set of values, $\{v_1, v_2, \dots, v_k\}$, such that each value refers to a value being shared as described above. Note that the distribution of generators and the broadcast may be performed in parallel, ie. in one round all in all.

The reason that this strategy is advantageous is that we may now broadcast a single element per element to be shared, rather than sending one element (securely) per TTP per element to be shared. This advantage comes at the price of distributing the generators securely to the TTP's. However, given a fixed number of TTP's, this requires a fixed number of bits. This implements the efficient delivery of inputs as described in chapter 2. As inputs may consist of many values — thousands, perhaps even more — using this variation of PRSS provides a significant reduction in the amount of bits to send in.

The distribution of generators in the above protocol requires some form of secure channels. These may be implemented in any way desired, eg. using public key cryptography. However, only the generators need to be encrypted, the v_i 's may be broadcast as they contain no information on the actual inputs, thus there is no need to keep them secret.

A final ideal functionality required which we also obtain through pseudo-random secret sharing is the generation of a random sharing of 0 with a threshold of $t' = 2 \cdot \lfloor \frac{n-1}{2} \rfloor$, ie. $2 \cdot t$. As with generating uniformly random elements, this

may be done without communication. Again, details of this are available in [6].

3.4 Simple Computations in \mathbb{F}

We now consider ideal functionalities as well as their implementations for computing on (shamir) shared values over a field \mathbb{F} . All operations considered in this section are field-operations, ie. multiplication and addition in \mathbb{F} . It should be noted that the protocols mentioned are all well known. They are described briefly for completeness of the presentation of our implementation, especially in order to give an intuition of the complexities involved.

3.4.1 Ideal Functionalities

Let $v \in \mathbb{F}$ be shamir shared, and let $c \in \mathbb{F} \setminus \{0\}$ be publicly known. We wish to be able to compute sharings of $v + c$ and $v \cdot c$, ie. having protocols equivalent to ideal functionalities being sent c and the shares of v , reconstructing v , and outputting shares of $c + v$ or $c \cdot v$ to the TTP's.

In addition to these, we also wish to be able to compute on multiple shared values. To elaborate, let $v_1, v_2 \in \mathbb{F}$ be shared between the TTP's. Consider an ideal functionality for computing the sum (product) of v_1 and v_2 , ie. all shares of both values are sent as inputs, the values are reconstructed, and the sum (product) is computed. Finally, shares of the result are returned to the TTP's.

Having described the ideal functionalities desired, we now present the (well-known) protocols implementing these.

3.4.2 Protocols for Simple Computations

Let the TTP's be numbered $1, \dots, n$, and recall the details of shamir sharings: Let TTP i be denoted by an element $e_i \in \mathbb{F}$. A value v is now shamir shared using a polynomial $p(\cdot)$ by having $p(0) = v$ and giving every party, i , the share $p(e_i)$.

Consider the two first functionalities — computation with a known value, c . By the definition of shamir sharings, it is easy to compute sharings of a polynomial sharing $c + v$ (or $c \cdot v$) given c and sharings of v . Simply add c to each share (multiply each share by c). This may be done by the TTP's without even communicating, thus clearly this is equivalent to the ideal functionality.

Concerning addition of two unknowns, the “protocol” follows in the exact same manner as the previous. Adding two shamir shares results in a share of the sum. This follows from the fact that each party i is given a unique point, $e_i \in \mathbb{F}$, which is used as the basis of all shares of i .

Multiplication is the complex of our basic operations, and the only one which requires communication. The idea behind the operation is the same as for addi-

tion, simply multiply the shares. However, contrary to addition, the degree of the polynomial used for sharing increases by this. If the initial degree is sufficiently low — ie. the threshold of the sharing scheme is less than half the number of the TTP's — we may still reconstruct the product. Unfortunately, we cannot perform any more multiplications, which is not acceptable. Moreover, contrary to the additions and the multiplication by a known constant, the resulting polynomial is not secure. If the product is reconstructed, knowledge of all the shares may release information on the factors, if one or more shares of these are known.

The solution to these problems is to reshare the value with a polynomial of lower degree. This may be done by sharing the shares of the product, and then *secretly* “reconstructing” the sharing using MPC. The latter is possible, as reconstruction is a linear function of the shares, and thus computable using only the above functionalities. As we have protocols equivalent with ideal functionalities for sharing values (from section 3.3.1 above) and for computing additions and multiplications by known constants, this is easily implementable.

As the computation needed for reconstruction does not require communication, the multiplication functionality is equivalent in complexity to having each party share a value. Using the PRSS solution, this requires a single broadcast of a value from each TTP. These may be performed in parallel, thus a single round is used, and the communication complexity is n times the number of bits required to encode an element of \mathbb{F} .

Note, that if the intention is to open a product rather than continue the computation, then we do not have to reduce the degree of the polynomial, however, as described above, we do need to reshare it. This can be done by adding a random sharing of 0 (shared with threshold $2 \cdot t$). Functionality for generating such a sharing was given in section 3.3.1, and it was noted, that this could be implemented without communication at all. Thus, this implements ideal functionality for multiplication followed by reconstruction (publishing in the terminology of chapter 2). The complexity is that of an ordinary reconstruction, ie. a broadcast of a single value from all parties in a single round.

3.5 Partial Single-TTP-Trust

A desirable property is the notion of self-trust of [4] — ie. each party of a computation must only trust itself. In our setting, with inputters and trusted third parties, this is equivalent to trusting that a single TTP will remain honest/unhacked, rather than trusting in a lesser majority. This is simply another way of stating that values are shared with a threshold of $n - 1$. No information is leaked unless *all* computing parties participate. This is in general not possible, when considering computation on shamir shared values, as a threshold of $\lfloor \frac{n-1}{2} \rfloor$ is required in order to perform multiplications.

When considering our setting, this means that the inputters must have faith

that at least half of the parties will not attempt to recover their inputs — depending on how valued the security of the data is, this may be a deterrent for taking part or providing truthful inputs. Thus, reducing the trust requirement is worth doing.

We present a strategy, which to some extent counters this problem. It does not provide full single-TTP-trust, however, it does provide an added security, which in some cases may be sufficient, hence the name *partial single-TTP-trust*. Concerning the strategy, note that rather submitting inputs by sharing them with threshold $\lfloor \frac{n-1}{2} \rfloor$, it may be shared with threshold $n - 1$. This satisfies the single TTP trust requirement, unfortunately we cannot perform multiplications on this. We may, however, compute any linear combination of the the inputs, as the degree of the polynomial used for sharing does not increase with these operations. Thus, we may boost the threshold at the beginning of the computation until a multiplication is required. The degree of the polynomial must then be reduced, eg. as done after a multiplication. If there are many parties, and the linear combinations depend on inputs of multiple of these, they will contain less information on the individual inputs, intuitively this increases security.

An application where partial single-TTP-trust is useful is the double auction. Concrete details will be provided in chapter 6. There, sums of inputs of all bidders are computed. As large number of bidders are expected, little information will be obtainable on the individual bids, as long as a single TTP remains honest.

Chapter 4

Integer and Binary Computation

The goals of this chapter is to construct secure computation on integers (except comparison) and bits, ie. most of the functionalities requested in section 2.4. A combination of the integer protocols of this chapter may be seen as a functionality for secure integer computation — similar to the overall goal, though without comparison.

We first consider our solution for integer computation. This is followed by a general solution for the binary functionalities. Finally the overall choices are presented along with two specialised protocols for operations on bits “stored as integers.” This will be used in the comparison protocol below.

4.1 Sharing Integers

We implement our primary goal of arithmetic on positive integers through computation on elements of \mathbb{Z}_p , where p is prime. Constructing shamir sharings over this field is standard.

We encode an l -bit integer value as an element of \mathbb{Z}_p in the natural, ie. as that integer modulo p , where p is chosen larger than 2^l . This implies that addition and multiplication in the field immediately translate to integer addition and multiplication, as long as no “overflow” occurs in the group — ie. when the result is less than p . To prevent overflow, p must be chosen sufficiently larger than the inputs, however, the bit-length of p may be viewed as a parameter of the system, thus it may be chosen arbitrarily large. Computing in this manner allows us to perform a *limited* number of integer operations.

In our implementation, we have chosen to work on 32-bit integers and a 65 bit long p . These choices allow us to compute on sufficiently large values for our desired applications, as well as give us “head-room” in the actual field for providing statistical security for our comparison operation. This is the only operation explicitly limited to 32 bits — addition and multiplication simply require a no-overflow guarantee in order to work. It should be noted, that for large values —

say 32 bits — only a single multiplication is allowed. As our main application, the double-auction, does not require multiplications this is currently not a problem.

The above discussion immediately allows us to implement a secure functionality for integer computation, though without comparison. This will be remedied in chapter 5, using the assisting functionality for binary computation presented below.

Though not required for the overall framework, we also provide functionality for performing subtraction on shared integers *under the assumption that the result is non-negative*. Ie. given sharings of a and b , we may obtain sharings of $a-b$. Such a subtraction may be implemented as a multiplication by -1 and an addition, ie. without communication. This will be required in the comparison below.

4.2 Sharing Bits and Binary Computation

The second goal of this chapter is to implement computation on shared bits. This can be constructed based on computation on shamir sharings over any field \mathbb{F} . Encode the binary values as the elements $0, 1 \in \mathbb{F}$, ie. as the additive and multiplicative identities. Now, **and** of two bits can be computed as a multiplication, while **not** consists of subtracting the bit from 1 (subtraction may be implemented as above). **or** can be implemented by subtracting the product of two bits (the **and**) from the sum:

$$b_1 \vee b_2 = b_1 + b_2 - b_1 \cdot b_2.$$

The final operator considered is **xor**. If 1 is its own additive inverse (say, in a field is of characteristic two), then this is simply addition in the field, which is free in our complexity models. If this is not the case, then we may convert our 0/1-represented bit to a 1/-1-represented bit, storing 0 as 1 and 1 as -1 . This conversion is a linear combination and therefore free. Using this alternative representation, **xor** is a multiplication. After performing one or more **xor**'s, we may return to a 0/1-representation. This solution is less desirable, as it requires a multiplication and therefore communication.

The above discussion of binary values in sharings immediately supply us with protocols equivalent to ideal functionalities for computing on shared bits. These protocols are based directly on the ideal functionalities for computing on shared field elements, and as such are immediately secure. Thus if we consider ideal functionalities for computing on shared bits — send in the shares, reconstruct the inputs to the computation, compute the result, and distribute shares to the TTP's — we immediately obtain implementations of **and**, **or**, **xor**, and **not**, as requested in chapter 2. We now introduce two concrete strategies for sharing bits, ie. two concrete fields.

4.2.1 Bits in $GF(2^8)$

Though it is possible to work on bits using \mathbb{Z}_p where p is a prime, computing in a field of characteristic two was shown to be more efficient. In addition, recall that we chose p large in order to allow integer operations to be performed modulo this number without field overflow. This implies that using the same p leads to wasting much space for every bit, furthermore, computation modulo a large value is also more CPU intensive. Choosing a small value for p , ie. choosing two primes, p and p' , and using one for integer computation and the other for binary computations can counter this problem. However, if we are willing to change strategy, $GF(2^8)$ is a better option.

$GF(2^8)$ is a field of characteristic two, so `xor` becomes an addition as noted above. Though $GF(2^m)$ for any value m may be used, $GF(2^8)$ is the most sensible choice. In order to allow for an acceptable number of TTP's, each of which must be assigned a unique element in the field, m must be chosen sufficiently large. However, we still wish to pack bits tightly in the field, and using $GF(2^8)$ allows a bit to be stored in a single byte, the minimal unit on a typical computer, while still permitting a few hundred TTP's.

Another benefit of choosing m low, is that multiplication may be performed much more efficiently. Recall that the elements of $GF(2^m)$ are polynomials over the field $GF(2)$, and that multiplication in this field is not implemented in hardware on an ordinary CPU. However by packing the coefficients tightly in bytes, we may simply perform a lookup in a table of multiplication results held in memory. Naturally, the size of the table is exponential in m , so m must be chosen small to ensure that the table has an acceptable size — even using $GF(2^{16})$ is infeasible in practice. Thus as $GF(2^8)$ is sufficient and only results in a table of around 65kB, this has been chosen.

4.2.2 Bits in \mathbb{Z}_p

Though we primarily compute on bits in $GF(2^8)$, we also require some binary computation in \mathbb{Z}_p in our comparison operation. We therefore consider ideal functionalities for two complex binary operations required. This is done explicitly in order to provide more efficient implementations than the naive ones based directly on binary computation in \mathbb{Z}_p . In the following, bits are stored as 0/1-values as described above, unless stated otherwise.

Generating a Uniformly Random Bit in \mathbb{Z}_p

We first consider an ideal functionality for generating a random bit unknown to all, ie. the ideal functionality flips a coin and outputs shares of the result to the TTP's. A protocol implementing this runs as follows:

First create a sharing of a uniformly random element in \mathbb{Z}_p , $[a]_{\mathbb{Z}_p}$. Square

and open $[a]_{\mathbb{Z}_p}$, this results in all parties knowing the value a^2 . If $a^2 = 0$ the parties start over, this happens with probability negligible in the bit-length of p . Each party now locally computes the same square-root of a^2 , a' , which equals $\pm a$. From this, a sharing of a uniformly random 1/-1-encoded bit, b , is computed as $[a/a']_{\mathbb{Z}_p}$. This may then be converted to a 0/1-encoding using the formula $(1 - b)/2$. Correctness of this protocol follow immediately from the observations interleaved with the description. This protocol may be implemented based on our ideal functionalities for generating a uniformly random sharing (section 3.3.1) and for opening a product of two shared values (section 3.4.2).

Concerning the round complexity of the actual protocol, using pseudo-random secret sharing implies that generating a random element is free. Moreover, the multiplication-publishing functionality requires only a broadcast of a field element from every TTP. Thus, basing the implementation on these, results in a protocol taking a single round, where $O(n \cdot \log_2(p))$ bits are broadcast, except with probability negligible in the bit-length of p . For simplicity, we will assume that the protocol takes a single round in the following. As p is chosen large, this is not an unrealistic assumption.

With regard to security, we only use ideal sub-functionalities, which are secure by definition. Therefore, as seeing a^2 does not reveal information on a (except that it is $\pm a$), the resulting bit is unknown to all, and the protocol generates a uniformly pseudo-random bit in \mathbb{Z}_p .

Efficient xor

The second desired ideal functionality is simply the computation of multiple `xor`'s followed by an opening of the result. Ie. the TTP's send in shares of several bits, the ideal functionality reconstructs the bits, computes their `xor`, and sends the result to all parties. This is clearly possible to do using our basic operations, however, as `xor` consists of a multiplication, this is expensive. A more efficient strategy than the naive approach is presented here.

The basic idea of the protocol implementing this functionality is to simply add the values. If there is no overflow within the group, then the least significant bit of the sum contains the correct result. However, the rest of the value contains additional information on the values `xor`'ed together, which may not be leaked.

This can be prevented by masking the top bits allowing the value to be opened. Let s be the (integer) sum of the bits and assume that it can be stored in an l -bit value, ie. we have `xor`'ed at most $2^l - 1$ bits. If we now generate a uniformly random, even m -bit integer, r , such that $2^m \gg 2^l$ and $2^m + 2^l < p$, then opening $s + r$ leaks no knowledge on s apart from the least significant bit except with probability negligible in $m - l$. As $2^m + 2^l < p$, there is no overflow in \mathbb{Z}_p when adding, thus the least significant bit of $s + r$ is the desired result, as r was chosen even.

Rather than generating a uniformly random r , we may let each party, i , share

a uniformly random, even m -bit value, r_i . We may then mask with the value $r' = \sum_{i=1}^n r_i$ rather than r , assuming that no overflow occurs in the generation of r' or when computing $s + r'$. Clearly this also yields the correct result and releases information with probability negligible in $m - l$ as above.

Regarding the complexity, this protocol requires a number of field additions (free), n secret sharings, and a publishing. Performing the secret sharings and basing the generation of shares on PRSS, we may implement the functionality in two rounds, with broadcasts of $O(n \cdot \log_2(p))$ bits.

An immediate use of this functionality is that given a sharing of a bit b , every party, i , may supply a secret bit, b_i , which may then be **xor**'ed onto the value and opened. This immediately results in an additive sharing modulo 2 of b — an **xor** sharing. Note that the b_i 's may be stored in the least significant bit of the r_i 's implying that constructing an additive sharing is no less efficient than the computation and opening functionality above.

Chapter 5

Integer Comparison

The most important integer operation of our implementation is the comparison primitive, ie. computing secretly answers to questions of the form $v_1 \geq v_2$, where v_1 and v_2 are positive l -bit integer values. Described as an ideal functionality, we have sharings of v_1 and v_2 and wish to obtain a sharing of a bit, $[r]_B$, such that r is 1 if $v_1 \geq v_2$, and 0 otherwise. The notation $[\cdot]_B$ simply denotes a shared bit. Similarly, the notation $[\cdot]_I$ will be used to denote a shared integer value. This notation is used to stress the fact that the shared values are integers and bits, and that the ideal sub-functionalities are indeed computations on these. Naturally these refer to elements of \mathbb{Z}_p and $GF(2^8)$, thus we may use all functionalities from the previous chapters replacing integer with \mathbb{Z}_p and bit with $GF(2^8)$. However, by considering integer and binary operations explicitly, any other implementation of the sub-functionalities may be “plugged in.”

The protocol presented implements the final ideal functionality on the “list” in section 2.4. Following this chapter, we may simply join all these to form the generic integer computation functionality presented in chapter 2.

The details — and sometimes strategy — of our comparison operation have changed multiple times. The operation described here is the current, theoretical version. Most of it has been implemented at the time of writing, however, some optimisations are missing, mainly due to recently thought up sub-functionalities, which have not been implemented. As the concrete details of this operation have been so prone to change, the results presented in this section can be seen as a snapshot of the current state of an ongoing work rather than as a “final product,” though the current solution is not expected to be altered greatly.

5.1 The Protocol

In this description of the implementation, v_1 and v_2 designate the l -bit integer inputs to the protocol. They are given as sharings, $[v_1]_I$ and $[v_2]_I$. In the following, assume that we have values m and t such that $2^t > 2^m \gg 2^l$. The

protocol consists of the following four steps: Preprocessing, reduction, lookup, and concluding computation.

Preprocessing The initial step of the protocol consists of creating a sharing of a uniformly random m -bit integer, $b \in \{0, 1, \dots, 2^m - 1\}$, $[b]_I$, as well as sharings of the individual bits of b , $[b_1]_B, [b_2]_B, \dots, [b_m]_B \in \{0, 1\}$. As suggested by the name, this step is independent of the values v_1 and v_2 , and may be executed in advance.

A sharing of such a b along with sharings of its bits can be created by sharing m uniformly random “bits” (0/1-values) as integers. Ie. constructing sharings $[b_i]_I$, for $i = 1 \dots m$. Given these, we may define $[b]_I$ as

$$[b]_I = \sum_{i=1}^m 2^{i-1} \cdot [b_i]_I.$$

As we wish to compute on the individual bits of b , we convert the sharings from $[b_i]_I$ to $[b_i]_B$. This is done as follows: For each b_i , each party j shares a random bit $b_i^j \in \{0, 1\}$ both as an integer and a bit. This results in sharings $[b_i^j]_I$ and $[b_i^j]_B$ for $j = 1, \dots, n$ and $i = 1, \dots, m$. We then compute and open

$$[\tilde{b}_i]_I = [b_i]_I \oplus \bigoplus_{j=1}^n [b_i^j]_I$$

for all values i . From this, sharings $[b_i]_B$ are constructed as

$$[b_i]_B = \tilde{b}_i \oplus \bigoplus_{j=1}^n [b_i^j]_B$$

Note that the actual implementation of the \oplus -functionality used above is unimportant. Moreover, it will definitely vary depending on the type of sharing it is used on. We specify these choices in the analysis of the protocol below.

Reduction The second step of the protocol consists of converting the problem of comparison to a problem of performing a lookup in a vector of pairs of bits. This is done in several sub-steps, each consisting of some alteration performed in order to transform the problem to a more manageable form.

First off, secretly compute a sharing, $[a]_I$, where $a = v_1 - v_2 + 2^l$ is an $l + 1$ -bit integer. Clearly $a \geq 2^l \Leftrightarrow v_1 \geq v_2$, so the problem has been transformed to finding the $l + 1$ 'th bit of a , which is 1 iff $v_1 \geq v_2$. Note, that a is positive, as v_2 is only l -bits long. Now, compute and publish $T = 2^l - b + a \in \mathbb{Z}_p$, and let T_i denote the i 'th bit of T . Note that

$$\begin{aligned} 2^l + a &= T + b \\ \Downarrow \\ a &= (T \bmod 2^{l+1} + b \bmod 2^{l+1}) \bmod 2^{l+1} \end{aligned}$$

This implies that the $l + 1$ 'th bit of a , a_{l+1} , can be computed as $T_{l+1} \oplus b_{l+1} \oplus C_l$, where C_l is the l 'th carry-bit when adding the binary representations of $T \bmod 2^{l+1}$ and $b \bmod 2^{l+1}$, ie. $\text{majority}(T_l, b_l, C_{l-1})$.

In the final part of this step, we convert the problem of computing a carry-bit to a lookup in a vector. The bits of b are shared as $[b_i]_B$'s and T is publicly known, so getting the l least significant bits of T and sharings of the l least significant bits of b are trivial. These bits are the low bits of the values modulo 2^{l+1} and determine the desired carry-bit fully.

We now compute sharings of bits, $[c_1]_B, [c_2]_B, \dots, [c_l]_B$, defined as $c_i = b_i \oplus T_i$. Moreover, for technical reasons described below, we let $c_0 = T_0 = 0$ and add this pair to the vector. The result is a vector of pairs of bits on which the lookup is performed:

$$\left[\begin{array}{c} \left(\begin{array}{c} c_0 = 0 \\ T_0 = 0 \end{array} \right), \left(\begin{array}{c} c_1 \\ T_1 \end{array} \right), \left(\begin{array}{c} c_2 \\ T_2 \end{array} \right), \dots, \left(\begin{array}{c} c_l \\ T_l \end{array} \right) \end{array} \right]$$

With the exception of the leftmost pair, the top elements are given as bit-sharings, $[c_i]_B$, while all the bottom elements are all publicly known. Note, that all computation in this step was simple computation on the shared values, immediately implementable based on the ideal sub-functionalities.

Lookup Recall that we wish to compute the l 'th carry-bit of the addition, $b+T$. Note that this equals T_i , where $c_i = 0$ and $i \in \{0, 1, 2, \dots, l\}$ is maximal. The intuition behind this is that if $c_i = 1$, the carry-bit from below propagates on up. If, however, $c_i = 0$ then $T_i = b_i$ and the carry-bit is either explicitly set or cleared. Computing a sharing of this T_i is the goal of this step. Note that we are ensured the existence of such a T_i , as we defined $c_0 = 0$.

Finding the correct T_i is done using the operator \diamond on pairs of bits, $\begin{pmatrix} x \\ X \end{pmatrix}$, defined as follows:

$$\begin{aligned} \begin{pmatrix} x \\ X \end{pmatrix} \diamond \begin{pmatrix} 0 \\ Y \end{pmatrix} &= \begin{pmatrix} 0 \\ Y \end{pmatrix} \\ \begin{pmatrix} x \\ X \end{pmatrix} \diamond \begin{pmatrix} 1 \\ Y \end{pmatrix} &= \begin{pmatrix} x \\ X \end{pmatrix} \end{aligned}$$

It is easily seen that this operator is associative and if repeatedly applied to the pairs of the vector, the resulting pair will contain the i 'th bit of T where $c_i = 0$ and i is maximal, ie. the desired result. The operator simply discards the right pair if the top-value of it is 1 (ie. it propagates the bit from below), otherwise it discards the left pair (the carry-bit is set or cleared explicitly). Implementing \diamond using binary operations can be done efficiently as follows:

$$\begin{pmatrix} x \\ X \end{pmatrix} \diamond \begin{pmatrix} y \\ Y \end{pmatrix} = \begin{pmatrix} x \wedge y \\ y \wedge (X \oplus Y) \oplus Y \end{pmatrix}$$

That the desired top bit is an **and** is clearly seen. Moreover, it is equally easy noted that the bottom bit is correctly computed — if y is not set, then Y is selected, otherwise X is selected. Thus, computing

$$\bigwedge_{i=0}^l \begin{pmatrix} c_i \\ T_i \end{pmatrix}$$

will result in sharings of the bits of the correct pair being computed.

Concluding Computation Having found the l 'th carry-bit, C_l , we may compute the final result, $[a_{l+1}]_B = T_{l+1} \oplus [b_{l+1}]_B \oplus [C_l]_B$, using the basic ideal functionalities. This is then taken as the result.

Note that the desired ideal functionality for secure integer computation of chapter 2 technically required the resulting bit to be an integer equal to zero or one. This requires a conversion, however, doing this efficiently based solely on addition and multiplication of positive integers seems infeasible. Using the fact that our integers are actually sharings of values in \mathbb{Z}_p , we may use the ideal functionality for computing on bits stored in \mathbb{Z}_p to transfer it there, before viewing it as an integer. This conversion will not be required in this work, as all results of comparisons will be published immediately. Phrasing a proper solution in terms of functionalities is part of the ongoing work.

5.2 Correctness

The preprocessing step is easily seen to work as intended. Clearly the $[b_i]_B$'s are sharings of the bits of b , as b is constructed directly from them. Moreover, as they were selected uniformly at random, b is a uniformly random m -bit number.

Concerning the movement of the b_i 's from integer to binary sharings, note that as the b_i^j 's have been shared in both fields, the only operations performed are to **xor** these random bits onto b_i twice, implying that the value does not change. Thus, the bits of b are all transferred correctly.

Correctness of the reduction, lookup, and concluding computations phases has been argued during the presentation. It was seen that

$$v_1 \geq v_2 \Leftrightarrow 1 = a_{l+1} = b_{l+1} \oplus T_{l+1} \oplus C_l$$

where a_{l+1} is the $l + 1$ 'th bit of a and C_l is the l 'th carry-bit of the addition of b and T (both modulo 2^{l+1}). It was argued that the lookup phase simply selected the top-most bit, where T_i and b_i were equal, ie. the desired carry-bit. This implies that a_{l+1} is correctly computed and is the desired result.

5.3 Complexities

In order to analyse the complexities of our concrete implementation of the protocol, we must specify the ideal functionalities involved explicitly. We therefore consider the integer and bit sharings as sharings over \mathbb{Z}_p and $GF(2^8)$ respectively. Thus the ideal functionalities are simply the ones presented in the previous chapters. Below, we analyse each step of the protocol individually, combining these in the end.

Recall that the preprocessing phase consisted of generating m uniformly random bits in \mathbb{Z}_p , their conversion to shares in $GF(2^8)$, and a linear combination of them. Note that we have ideal functionalities for generating the bits and computing the opening of each masked bit. The linear combination is free, as is the “reconstruction” of the bits in $GF(2^8)$, as these consist of only additions and multiplications by known constants.

As we may process the m bits in parallel, the complexity is independent of m . Generating a random bit may be done in one round, while computing the `xor` and opening takes two. However, the first round of the `xor` may be run in parallel with the generation of the bits, thus all in all, only two rounds are required.

Concerning the size of the communication, recall that the complexities of the sub-protocols both required the broadcasts of a constant number of elements per TTP. Thus, the overall complexity is $O(m \cdot n \cdot \log_2(\max(p, 2^8)))$ bits being communicated in the preprocessing phase.

However, there are a few optimisations that may be performed in order to reduce the size of the communication. First, we do not actually need to convert all bits from \mathbb{Z}_p to $GF(2^8)$, only the $l + 1$ least significant ones. Continuing, we may avoid constructing the $m - (l + 1)$ top bits all together. The idea is as follows: Generate the $l + 1$ least significant bits of b as above, and let each party j share a uniformly random $m' - (l + 1)$ -bit integer, r_j . Now define b as

$$b = \left(\sum_{j=1}^n 2^{l+1} \cdot r_j \right) + \left(\sum_{i=1}^{l+1} 2^{i-1} \cdot b_i \right)$$

where m' is chosen such that no overflow may occur when computing b . Clearly b is not uniformly random, however the reason for choosing b m bits long was to mask away the $l + 1$ least significant bits of another number, and for this purpose, the r_j 's and b_i 's work. Using both optimisations reduces the communication complexity to $O(l \cdot n \cdot \log_2(\max(p, 2^8)))$.

Focusing on the reduction phase, it is noted that the round complexity of this is one. The only operations present are additions in \mathbb{Z}_p , a single opening, and `xor` on bits shared in $GF(2^8)$, ie. more additions. The opening may be performed in one round, where $O(n \cdot \log_2(p))$ bits are broadcast.

The lookup phase is the most expensive with regard to round complexity. First note that to perform a single \diamond -operation, the $GF(2^8)$ -multiplication must be

applied twice, once for each of the `and`'s. As we are working in $GF(2^8)$, `xor` is free, therefore an \diamond -operation can be performed in one round, as the multiplications may be performed in parallel.

We start off with l bit-pairs (technically $l + 1$, however the leftmost pair is fully known implying one free \diamond). As the operator is associative, we may apply it to the operands in any order, (and thus also in parallel). This allows us to halve the number of operands every iteration resulting in $\lceil \log_2(l) \rceil$ rounds.

Concerning the communications complexity, a linear number of \diamond -operations are performed. Each of these require at most two multiplications, so $O(n \cdot l \cdot \log_2(2^8))$ bits are broadcast during the computation. We say “at most”, as in the first iteration, the T_i 's are known. Such open values may even propagate down through the calculation, however, this is not guaranteed, nor does this improve the big- O analysis.

All in all this leaves us with a comparison operation taking $\lceil \log_2(l) \rceil + 3$ rounds, as the concluding computations are additions and therefore free. Two of these rounds are used for preprocessing, and as such may be performed in advance. The overall number of bits communicated is $O(l \cdot n \cdot \log_2(\max(p, 2^8)))$ assuming that the optimisations are used.

In our implementation, we compute on 32-bit integers, thus implying eight rounds in total. As we have analysed the communications complexity using big- O , it is not possible to provide the exact number of bits communicated. We may, however, provide a quick “back of the envelope” calculation to give an overview, noting that the formulas provided below are only estimates.

Generating all $l + 1$ least significant bits of b and converting them to $GF(2^8)$ is done using (roughly) $3 \cdot n \cdot l$ broadcasts of elements in \mathbb{Z}_p and $n \cdot l$ broadcasts of elements in $GF(2^8)$. The reduction consists of a single opening, ie. n broadcasts of an element in \mathbb{Z}_p , while the lookup requires (at most) $1.5 \cdot n \cdot l$ multiplications of elements in $GF(2^8)$. Using five TTP's and 32-bit numbers — $n = 5$, $l = 32$ — this adds up to around 500 elements of \mathbb{Z}_p and 400 elements of $GF(2^8)$. An element of the former type takes up roughly eight bytes, while one of the latter may be contained in a single byte. All in all this results in around 5kB being broadcast. Clearly not an excessive amount.

5.4 Security

Security follows almost immediately from the use of the ideal sub-functionalities.

Concerning the preprocessing step, the only part to argue is that we do not reveal information by opening the b_i 's. However, this is not the case, as they are fully masked by the random bits, b_i^j . If a single party is honest, at least one b_i^j will be unknown to the adversary, and as it is uniformly random, \tilde{b}_i will be uniformly random, releasing no information on b_i .

For the reduction stage, as the sub-protocols are secure, all that may leak

information is the opening of the value $T = 2^t - b + a$. However, as b is an m -bit number and a is only an $l+1$ -bit number, where $2^m \gg 2^l$, clearly the probability that this happens is negligible in the difference in bit lengths, $m - (l + 1)$. As p is 65-bits long in our implementation, we may choose $t = 63$ which ensures that we cannot have overflow in the field when computing T . Choosing b as a random 62-bit number results in a probability of leakage of around 2^{-30} .

Finally, the lookup and concluding computations phases simply consists of applying our basic computation functionalities on shared values. As there are no openings, this is secure by definition. Thus, as all stages of the operation are secure, the whole operation is secure, implying that no information is leaked, though only with a statistical guarantee, due to the reduction phase.

Chapter 6

The Double Auction

Having constructed the sub-functionalities for our ideal integer computation functionality, we now focus on our second goal, implementing a secure double auction based on secure integer computation. In such an auction, several buyers and sellers meet to exchange multiple units of the same good. The overall goal is to compute a market clearing price (MCP), ie. a price at which all trade occurs. This value is based on “bids” submitted by the traders. All parties must then buy/sell according to their bids.

Naturally, for an auction there will be other requirements in addition to secrecy of the inputs and correctness of the result, eg. it should not be possible to deny a bid, as trade is required to take place. The focus here is on the actual distribution of trust — ie. the MPC — and as such, additional properties are outside the immediate scope, though they must be handled in real-world scenarios, eg. signing bids to ensure non-repudiation. From an economic perspective, there are also numerous properties to take into account regarding the mechanism, however, that the design is economically sound is not important from a cryptographic perspective, and soundness will simply be assumed. For additional details and discussion, see [3] or [2].

In this chapter, we first introduce the computation of the market clearing price as an ideal functionality, first specifying the concepts of bids and the MCP. Following this, we implement the computation of the MCP based on the ideal functionality for secure computation of chapter 2. Finally, a possibility for round vs communication size tradeoff is presented. This consists of a variation of the main algorithm presented.

6.1 Computation of the MCP

An ideal functionality specifying the computation of the market clearing price is presented in this section. This may then be used to implement a secure double-auction. The functionality progresses in three steps:

1. The bidders (inputters) send in their bids.
2. The functionality computes the MCP based on all bids.
3. The MCP is sent as output to all bidders.

Note the similarities between this and the ideal functionality for general integer computation presented in chapter 2, which provides the basis for the computation. The goal of this chapter will be to provide a number of straight-line programs consisting of additions, multiplications and comparisons over l -bit integers which may then be used to implement the computation of the MCP.

6.1.1 Bids and the MCP

In the theoretic version of the double-auction, each buyer (seller) submits a function, $\mathbb{R} \mapsto \mathbb{R}$ — mapping prices to amounts. This denotes that party's demand (supply) at any given price, ie. the amount the party is willing to buy (sell) if this is the MCP. These functions are monotone, decreasing for buyers and increasing for sellers. The intuition behind this is that if a buyer is willing to buy an amount of goods given some price, that deal does not become less attractive if the price drops. For sellers the intuition is reversed.

The market clearing price of the mechanism is found based on the aggregated supply and demand of the system, ie. the sums of all seller and buyer bids. Adding all seller- or buyer-bids result in monotone functions representing the whole supply or demand of the system. The MCP is defined as the price-value of the intersection between these functions. This definition results in an economically sound system.

In practice, supplying or storing arbitrary (monotone) functions over \mathbb{R} is not possible. We therefore consider these functions represented by points in a grid — ie. we fix a set of prices and amounts, and supply the points the functions passes through. Though this is a restriction, doing so is acceptable in practice.

More precisely, bids consist of the amount to be bought/sold for all designated prices. The points of a bid are represented as an array of amounts; the amount in the i 'th entry is simply the desired amount at price i . The associated price of i may be independent of the index used to represent it, except that price j is larger than price i for all $j > i$. For simplicity, let price i equal i in the following. It is stressed that this does not affect the algorithm, however, the description is simpler when the desired result is an index. In practice, the actual prices may be arbitrary. We may even vary the fine-grainedness, having more values around the expected result than at the extremes. This latter strategy is not possible with regard to the amounts; these must be given explicitly, though the actual *unit* of goods that the amount represents may be chosen freely.

In the following, assume that there are N bidders numbered from 1 to N , and for simplicity assume that they all provide both a buyer- and a seller-bid — one

bid may equal zero, thus this can be considered equivalent to submitting a single bid. Let b_j be the buyer-bid of party j and let s_j be the seller-bid. Both of these are arrays indexable by the prices, ie. $b_j[i]$ and $s_j[i]$ are the desired trade-amounts if the MCP is i .

We now consider the concept of MCP in this discrete world. Note that the aggregated supply and demand are functions exactly as the bids. They may therefore be represented in the exact same manner, defining the entries in the natural way. If s and b are the arrays representing the aggregated supply and demand functions, for price i , we let the aggregated supply at price i be:

$$s[i] = \sum_{j=1}^N s_j[i]$$

and the aggregated demand at price i :

$$b[i] = \sum_{j=1}^N b_j[i].$$

Note that as the bids represent monotone functions we have $b[i] \geq b[i + 1]$ and $s[i] \leq s[i + 1]$. However, as we are restricted to a discrete setting, the existence of a valid (exact) MCP is not guaranteed. Most likely there will not exist i such that $s[i] = b[i]$. Thus in the discrete setting, we define the market clearing price to be i_{mcp} where

$$b[i_{mcp}] \geq s[i_{mcp}] \wedge b[i_{mcp} + 1] < s[i_{mcp} + 1],$$

ie. the valid price immediately below the market clearing price.

Naturally, this may result in surplus demand, which is undesirable. However, auction design is simply the construction of trade-rules, and we therefore define our way out of the problem. The actual choice is of little importance, when considering the cryptographic aspects; finding the “intersection” is the main goal of any solution, so the presented choice is fine. Having defined the MCP, the computation may simply be viewed as determining the maximal i for which $b[i] \geq s[i]$.

There is one problem with the above definition, namely that there may not be a MCP at all. It may be the case that the MCP lies outside the price range, ie. it is larger than the maximal price or smaller than the minimal. In these cases, we cannot hope to find an “intersection” between the supply and demand curves. In the description of the algorithm we simply assume that a MCP exists. This is acceptable, as we may easily detect if this is not the case *after* the execution. Thus, this issue may safely be ignored.

6.2 The Double-Auction Implementation

We now describe the actual double-auction implementation based on our ideal functionality of chapter 2. Concerning notation, we let $[\cdot]$ denote a secret integer variable (or value) of the secure computation functionality. Variables and values *not* in brackets are considered published. Though the secure values are sharings in our implementation, it is stressed that $[\cdot]$ does *not* denote a sharing. Any solutions implementing the ideal functionality may be used. The notation has been chosen similar to that of sharings, to preserve the connection to the lower layers. In the description below, let s , b , N , s_j , and b_j be defined as above. Moreover, let k denote the number of prices taken from the set $\{1, 2, \dots, k\}$.

6.2.1 Straight-Line Programs

We now present the straight-line programs required for the implementation of the MCP-computation. Only two simple programs are required in order to compute the MCP.

addition: The `addition` program implements addition of multiple values. It takes a number of secret integer values as inputs, $[v_1], [v_2], \dots, [v_m]$, for any value m . The output of the program is a secret variable containing the sum, $[sum] = [\sum_{i=1}^m v_i]$.

compareAndPublish: The `compareAndPublish` program takes two secret integer values, $[v_1]$ and $[v_2]$ as input. The output of the program is the result of comparing v_1 and v_2 , ie. 1 if $v_1 \geq v_2$ and 0 otherwise. This result is *published* implying that it may be used in subsequent branching.

6.2.2 Computing the MCP

We now implement the ideal MCP-computation functionality based on the integer computation functionality of chapter 2 and the above straight-line programs. Recall that our goal is the value i_{mcp} , where $b[i_{mcp}] \geq s[i_{mcp}]$ and $b[i_{mcp} + 1] < s[i_{mcp} + 1]$. The computation is split up into two separate steps:

- Compute the aggregations, b and s .
- Compute the index, i_{mcp} .

The first step is easily performed with the `addition`-program. For all prices, i , we run the program twice, in order to compute $b[i]$ and $s[i]$. This computes the aggregations — ie. the supply and demand “curves” — from which the MCP will be found.

Now, consider an array t , where $t[i] = s[i] - b[i]$, and note that t is sorted, as the supply and demand are monotone, s increasing and b decreasing. Note also that the goal is the index of the largest entry less than or equal to zero. Thus we may compute the desired index simply by performing a binary search for zero. Branching may be performed based on the sign of the $t[i]$'s, which is simply the result of comparing $b[i]$ and $s[i]$. This may be done using the `compareAndPublish`-program. As this program publishes its output it may be used in the branching (the selection of subsequent indexes to test). A description of the entire algorithm in pseudo-code is available in appendix B. Correctness of the computation is easily seen. Clearly the aggregations are computed according to the definition, and concerning the correctness of the determined MCP, this follows immediately from the correctness of binary searching and the discussion above.

Security and Trust

We now argue that the double-auction computation is secure, ie. that it is equivalent to the ideal functionality described. The use of the computation functionality of chapter 2 immediately implies that the straight-line programs are executed correctly and securely. However, the computation publishes values underway, which may release information. Fortunately, this is not the case.

The only values published are the results of performing the comparisons, ie. knowledge of which of $b[i]$ and $s[i]$ is biggest for certain prices, i . However, by definition this information is trivially released for all prices by the MCP, i_{mcp} . As it is intuitively acceptable to release any information trivially computable from the public result, the computation is secure.

Concerning the trust required in the system when replacing the functionality with a set of n TTP's, we only guarantee security if $\lfloor \frac{n-1}{2} \rfloor$ remain honest. However, as the aggregations are linear combinations of the inputs, we may use the partial single-TTP-trust of section 3.5. Following the computation of the aggregations, the threshold is reduced, however, as there are expected to be many bidders, each bidder will be marginal and the sums will contain little information on individual bids. Thus concerning the secrecy of the inputs, they may be considered safe with a threshold of $n - 1$.

Analysis

In this analysis, the focus will exclusively be on the actual computation performed. The input phase is simply viewed as part of the functionality. Complexity is first considered in terms of the straight-line programs. These are then analysed, and based on this the actual round and communication complexity of the overall algorithm is specified.

The `addition`-program is only invoked in the computation of the aggregations. The number of invocations is $2k$. Concerning the `compareAndPublish`-program, it is invoked once per iteration of the binary search over an array of size k . This implies $\lceil \log_2(k) \rceil$ invocations. Note that we may reduce the number of `additions` to a logarithmic number as well simply by computing the aggregations lazily rather than eagerly; the eager solution was chosen for simplicity, as the choice does not affect the analysis below.

We now consider the complexity of the double-auction in terms of rounds and communication size — ie. we consider the actual sub-protocols. Specifying the complexities of the straight-line programs, we note that the `addition`-program performs a linear number of additions in the size of the input, while `compareAndPublish` executes a comparison and an opening. Note that in practice, the final round of the comparison does not have to be performed, as we open the result. The final multiplication may be replaced by the ideal functionality for multiplication followed by reconstruction described in section 3.4.2. Though these are not explicitly possible to specify within the integer computation functionality, we may simply assume, that the underlying protocols are optimised automatically. Thus, this simple optimisation is assumed to be used.

Recall that additions required no communication (and thus no rounds) implying that execution of the `addition`-program is free in both models. A comparison, on the other hand, could be performed in $\lceil \log_2(l) \rceil + 3$ rounds with a communication complexity of $O(l \cdot n \cdot \log_2(\max(p, 2^8)))$, where l was the bit-length, n was the number of TTP's, and p was the prime defining the field for the integers. Because of the simple optimisation described above, this is also the complexities of the `compareAndPublish`-program. Thus the entire protocol requires $\lceil \log_2(k) \rceil \cdot (\lceil \log_2(l) \rceil + 3)$ rounds, in which $O(\lceil \log_2(k) \rceil \cdot (l \cdot n \cdot \log_2(\max(p, 2^8))))$ bits are broadcast. However, the round complexity of the computation may be reduced further. The rounds for preprocessing are independent of the actual values, so they may be performed in parallel in the beginning. This results in a reduction of the round complexity to $2 + \lceil \log_2(k) \rceil \cdot (\lceil \log_2(l) \rceil + 1)$.

For a more practical perspective on the complexities, we return to our “back of the envelope” calculation. Recall that with five TTP's, 32-bit numbers and a 65-bit prime, all in all around 5kB were broadcast in a comparison. This required two rounds of preprocessing and six rounds of online computation. Considering a double-auction with 1024-prices, this may be performed using 10 `compareAndPublish`'s. Thus, executing this auction requires 62 rounds, in which 50kB of data is broadcast. Clearly this is feasible in practice.

6.3 Round vs. Communications Size Tradeoff

As described in chapter 3, we may encounter round vs. communication size tradeoffs when designing MPC-algorithms. This is the case for the double-

auction above. We may reduce the round complexity by executing multiple `compareAndPublish`'s in parallel. The extreme case of this is to process all prices in parallel. This determines the MCP in a single comparison-round rather than logarithmically many, though at the cost of a linear number of comparisons performed. Thus we reduce the round complexity of the auction to that of a comparison, while increasing the communication complexity by a factor of k .

This idea may be generalised to splitting the search-space into h segments of equal size (rather than two or k). We may then determine the segment containing the intersection by performing $h - 1$ `compareAndPublish`'s in parallel. By recursing on the determined interval, we may compute the MCP in $\log_h(k)$ iterations using $(h - 1) \log_h(k)$ comparisons. Depending on the bandwidth and latency of the network, this compromise may speed up the overall computation time.

For a concrete example, consider the case $h = 4$. The number of rounds is halved, while the added work consists of three comparisons per iteration rather than one. All in all, this implies that we perform $3/2$ times as many comparisons. In our example case with five TTP's, 32-bit integers, a 65-bit prime, and 1024 prices in the auction, this results in spending 32 rounds rather than 62, at the cost of broadcasting 75kB rather than 50. As rounds are extremely expensive and the added communication is small, it seems very likely, that this will indeed speed up the computation in practice, however, the strategy has not been implemented, so this is currently unverified. The gain from using larger values h is expected to be smaller, as a growing number of comparisons are required for a halving of the number of rounds. The optimal solution naturally depends on the network.

Chapter 7

Conclusions and Future Work

In this report the theoretical background for the SCET implementation of secure multi-party integer computation was presented. Efficient protocols were constructed and loosely argued secure. Moreover, the framework was used to construct a solution for a real-world problem, namely the double-auction. The protocols presented have all been implemented and shown to be practically realisable.

First the setting was introduced, and players were divided into two sets: inputters and trusted third parties. The first group consisted of the parties of the original economic setting, while the second were used as “less trusted” mediators to implement a more trustworthy third party, thus distributing trust.

Following this, general MPC was presented. In addition, the concept of *partial single-TTP-trust* was introduced, providing additional security almost for free. Basic integer computation — addition and multiplication — was implemented based on Shamir sharings over \mathbb{Z}_p . Functionality for binary computation was introduced as an aid when constructing more complex protocols. Based upon these, secure, l -bit integer comparison was constructed. Moreover, the protocol was shown efficient, requiring a number of rounds logarithmic in l . Further, the overall communication complexity was low, $O(n \cdot l \cdot \log_2(p))$, where n is the number of TTP’s and p is the prime used for defining the group. Combining the above protocols resulted in an implementation of the full integer computation functionality.

Rounding off, the double-auction was presented and shown to be securely constructible from our computation functionality. Moreover, it was noted that use of partial single-TTP-trust could result in essentially a threshold of $n - 1$. Finally, the protocol computing the market clearing price was argued efficiently implementable using the SCET MPC-framework.

7.1 Future Work

The work described in this report is very much “in progress.” As noted, functionalities have not been formalised fully — and current strategies may prove less than perfect under closer scrutiny. In conjunction with this, strict simulator proofs in the UC model should also be performed, properly demonstrating the security of the protocols and thereby also the implementation. These are the immediate, high-priority concerns.

On a larger time-scale, extending the functionality of our integer computation provides additional possibilities for future work — both theoretically and with regard to the implementation. Desirable extensions are integer division and modulo computation. Both of these seem to be difficult to implement efficiently — exactly how they compare to eg. integer comparison in efficiency is an open problem. Note that these operations actually consist of multiple possibilities — both operands may be secret, or either may be known. It is possible that the latter may allow more efficient solutions.

Extending the functionality can also be done through additional properties, eg. allowing for negative values. As noted above, we may perform subtractions, however, without negative values, such an operation may result in an illegal value, unless verified with a comparison. Though negative values may immediately be implemented, storing the values as -1 , -2 , etc, other possibilities may allow for other desirable properties.

Work on additional “high-level” algorithms is a final possibility. The triple double-auction is an obvious candidate, once the details are in place. Alternatively, tasks such as secure card-shuffling (permutation) or sorting could be considered. However, these plans are not in the immediate pipeline.

Bibliography

- [1] J. Algesheimer, J. Camenisch, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In M. Yung, editor, *Advances in Cryptology - CRYPTO 2002*, pages 417–432, Berlin, 2002. Springer-Verlag.
- [2] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications. *BRICS Report Series RS-05-18*, 2005.
- [3] Peter Bogetoft and Kurt Nielsen. Work in progress. 2005.
- [4] F. Brandt and T. Sandholm. Efficient privacy-preserving protocols for multi-unit auctions. In A. Patrick and M. Yung, editors, *Proceedings of the 9th International Conference on Financial Cryptography and Data Security (FC)*, Lecture Notes in Computer Science (LNCS). Springer, 2005.
- [5] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. <http://eprint.iacr.org/2000/067>, 2005.
- [6] Ronald Cramer, Ivan Damgrd, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proceedings of the Second Theory of Cryptography Conference*, pages 342–362, 2005.
- [7] I. Damgård and M. Jurik. A generalization, a simplification and some applications of Paillier’s probabilistic public-key system. In *Public Key Cryptography*, pages 110–136, Berlin, 2001. Springer-Verlag. Lecture Notes in Computer Science Volume 1992.
- [8] I. Damgård, M. Jurik, and J. Nielsen. A generalization, a simplification and some applications of Paillier’s probabilistic public-key system. Manuscript obtainable from authors. To appear in journal, preliminary version occurs in [7].
- [9] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

Appendix A

Choosing Shamir over Paillier

In this section, we discuss the reasons for choosing shamir sharings ([9]) over sharings based on a threshold version of generalised Paillier encryption, ie. a public key cryptography based solution ([8]). As noted in the text, the reasons for selecting shamir sharings rather than sharings based on Paillier encryption are due to considerations on the sizes of the sharings. While the Paillier based solution has desirable properties such as the possibility of single-TTP-trust, the downsides with regard to size were too great compared to the gain — even without the concept of partial single-TTP-trust.

To explain the problems of data size, consider a double-auction. It is expected that the auction is large, say 1000 bidding parties — our real-world examples have around 5,000 potentially interested parties, so this number is not excessive. Each bid consists of some large, fixed number of shared values, eg. 1,000. This yields 1,000,000 shared values representing more than one half GB of data. This is independent on the bit-length of the values, though may be varied through the security parameter of the encryption scheme. For shamir sharings the sharing size may be chosen “tightly.” In our solution roughly eight bytes are used per share, resulting in about 8MB of data per computing party. Note, however, that in the Paillier-based approach, all computing parties must also have access to all data. Naturally, for a larger computation such as a triple double-auction, the problems of size increase, as the sheer size of the dataset — even unshared — may become large and unwieldy.

A second problem with larger bid sizes, concerns the bidders themselves. If these are private individuals, without high-bandwidth access to the MPC-system (eg. through the internet), it is important to keep the size of a bid low. Using Paillier-encryption will result in a bid-size for a double-auction as above of hundreds of kB, whereas a shamir sharing based approach uses around 10kB per TTP — this may be reduced to 10kB in all plus an encryption for each of the TTP’s by using pseudo-random secret sharing. None of the numbers are alarmingly large, however, for larger computations, the problem will grow. Increasing the input size by a factor of 1,000 is possible — ie. not terribly inefficient — with shamir

sharings. This is not the case with the Paillier-based solution.

Thus as the SCET project is concerned with the feasibility of (relatively large) computations, the choice of sharing strategy has been the one which “wastes” the fewest bits, ie. shamir sharing over \mathbb{Z}_p .

Appendix B

The Double-Auction in Pseudo-Code

In this appendix, a pseudo-code version of the overall double-auction algorithm is presented. It is noted, that details vary slightly between this and the actual implementation, however, the alterations are primarily concerned with detection of anomalies ignored in this description.

```
Receive bids  $[s_j[i]]_{\mathbb{Z}_p}$  and  $[b_j[i]]_{\mathbb{Z}_p}$ ,  $j = 1 \dots N$  and  $i = 1 \dots k$ 
for  $i = 1$  to  $k$  do
     $[s[i]] \leftarrow \text{addition}([s_1[i]], [s_2[i]], \dots, [s_N[i]])$ 
     $[b[i]] \leftarrow \text{addition}([b_1[i]], [b_2[i]], \dots, [b_N[i]])$ 
od
low  $\leftarrow 1$ 
high  $\leftarrow k + 1$ 
while ( $low + 1 < high$ ) do
    mid  $\leftarrow \lfloor (low + high + 1) / 2 \rfloor$ 
     $r \leftarrow \text{compareAndPublish}([b[mid]], [s[mid]])$ 
    if ( $r == 1$ ) then
        low  $\leftarrow mid$ 
    else
        high  $\leftarrow mid$ 
od
mcp  $\leftarrow low$ 
```