Primitives and Applications for Multi-party Computation

Tomas Toft

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Primitives and Applications for Multi-party Computation

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Tomas Toft
23rd March 2007

Abstract

The basic problem of multi-party computation was phrased well over twenty years ago. Since then, numerous advances in that area have been made. This ranges from general completeness results to efficient solutions for specific problems. The present work leans towards the second of those two.

This thesis considers protocol construction based on simple – but abstract – primitives, allowing secure arithmetic computation. These are modelled using the UC-framework, and it is noted that efficient realisations exist. This provides a formal model of secure computation, which allows easy construction of protocols based on the provided arithmetic primitives. The model may be viewed as an extension of secret sharing, though it may also be realised through other techniques.

This work sets out with two distinct tasks in mind. Based on what is essentially a linear secret sharing scheme with a multiplication protocol, the goals are:

1. to create efficient, constant-rounds protocols for small, but useful, tasks.

2. to use these sub-protocols in the construction of MPC-protocols focusing on high-level applications with real-world motivations.

The sub-tasks considered are those of comparison of secret values (both with regard to equality and inequality) as well as bit-decomposition. The latter consists of securely determining the binary representation of a secret shared value, making it available for further computation. An efficient, constant rounds solution for this task facilitates the construction of further efficient protocols, notably it allows computation to be phrased as Boolean circuits over the bits of the binary representation.

Regarding applications, scenarios in the realm of closed-bid auctions are considered. MPC removes the need to trust a single party, the auctioneer, who determines the sales price based on the bids. The parties themselves may take on that role, performing the desired computation without leaking information on their bids. Further, the task of securely solving linear programming problems is considered. Many real-world problems – particularly in economics – may be phrased as linear programs, thus, this provides a powerful tool with immediate applications, though it may equally well serve as a primitive in further secure computations.

Acknowledgements

To Bente and Bjarne Toft
And to Dina Friis, for always being there

With special thanks to...

Jesper Nielsen and Matthias Fitzi – for discussions and answering questions, I've learnt a lot.

All of the crypto group at Daimi, in particular the SCET and SIMAP people – for three good years.

Joan Feigenbaum and everyone else at Yale University– for my time there.

The other PhD-students at Daimi, particularly:

- Troels Sørensen – for sharing T019 and for usually knowing the algorithm solving my problems. Your sense of humour was punishment though.
- Michael Pedersen – for sharing T213, and for all the chats.
- Tord Reistad – for chats, humour, and enthusiasm.

Frokostklubben (both the old guys who left and all the newcomers) – for lunch.

The people who helped with this thesis, proof-readers and the LaTeXnician: Kåre Christiansen, Dina Friis, Lone Laursen, Jesper Nielsen, Tord Reistad, and Bjarne Toft.

All the nice people I've meet in crypto.

All the people who have answered my various questions and pointed me in the right direction for the past three years – research related and otherwise.

And with gratitude to my adviser, Ivan Damgård. For the questions and answers, and for advice, support, guidance, and for always having time. And encouragement when things looked dark. Without you this work would not have been.

Tomas Toft,
Århus, 23rd March 2007.

Contents

Introduction

Chapter 1

Introduction

Consider the motivating problem of this thesis: A number of parties – these may represent people, organisations, companies, ... – all have some amount of information. They would like to gain some knowledge based on their *combined* information, however, they may be mutually distrusting and are disinclined to reveal their secrets. Alternatively the information may be confidential, such that they are not even allowed to provide it to other entities. Such settings are highly common in the world, a few examples include:

- Elections – The overall winner of the election should be determined without revealing "political affiliation" of the voters.
- Auctions – Bidders may be disinclined to reveal their belief of the true value of an item; such knowledge may be (mis)used by others setting them at a disadvantage at a later time.
- Benchmarking – Companies may gain from measuring their performance and comparing it to that of their competitors. However, none are willing to share e.g. production costs or trade secrets with the others.
- Data-mining multiple databases – Performing data-mining on e.g. patient records from several hospitals may provide benefits to both medical research and to society in general. However, patient confidentiality prevents such an aggregation from taking place.

Handling such issues in real life is costly and often difficult, or even impossible. If some impartial third party can be agreed on – one that all parties trust – then having the parties provide their data to that party allows the desired result to be obtained. However, this requires trust in an additional party. Moreover, that party should be provided with incentives not to misuse the trust – the real-world solution is typically to pay that party sufficiently such that it cannot be bribed.

In general, the field of economics provides many such settings which may benefit from access to a trusted third party, e.g. many forms of auctions and trading mechanisms. The trusted party – also known as a mediator – simply collects all information and computes the overall best course of action: The coordination or distribution of resources which provide the greatest benefit to

the entire system. However, as noted above agreeing on such a party and ensuring trustworthiness is not a simple task.

The solution to the above problem considered in this work is obtained through cryptography, specifically multi-party computation (MPC). Multi-party computation (and its sibling secure function evaluation (SFE)) allows parties to interact – run a protocol – thereby obtaining outputs dependent on their inputs. This may be done without disclosing more information on inputs than what is implied by the outputs – while still ensuring their correctness. For an election, the output would be the winner – possibly including the margin of victory, i.e. how many voted for each of the candidates. For an auction, this would designate the price as well as who gets to buy and/or sell. The parties running the protocol could be the parties themselves, but also a *set* of trusted third parties. Inputs are distributed to these in a manner ensuring that none of them know the actual inputs, though the output may still be computed from the distributed ones. The latter still requires trust to be placed in other entities, however, distributing trust ensures that there is no longer a single point of failure – inputs are only compromised if *several* of the trusted parties are malicious.

The area of multi-party computation and secure function evaluation has a long history. Since Yao's initial paper [Yao82], many positive results on what may be accomplished in different settings have been demonstrated – classic results include those of Yao [Yao86], Chaum *et al.* [CCD88], Ben-Or *et al.* [BGW88] and Goldreich *et al.* [GMW87]. MPC may be based on a number of different techniques, ranging from circuit scrambling over secret sharing to (threshold) public-key crypto-systems. The common theme in all of these is that they essentially provide evaluation of circuits – either Boolean or arithmetic over some field or ring.

The good news is that – theoretically – any computation that can be performed may be performed securely – i.e. using MPC. Any polynomial-size (Boolean) circuit may be evaluated on arbitrarily distributed inputs. The bad news is that theory and practice do not always agree, and what is theoretically acceptable may not be feasible in practice. Though any computation may be phrased as a Boolean circuit, complex tasks never are – imagine a circuit for performing data-mining on databases with a size of multiple gigabytes. Therefore, specialised protocols are usually introduced when considering large-scale applications of MPC.

Most real-world computation has to do with numbers, not Boolean circuits. Integer arithmetic – addition and multiplication – may be simulated using arithmetic over a finite ring or field – this is no different from computing on values of some fixed bit-length. Thus, MPC-solutions providing arithmetic circuits may be used to perform integer arithmetic. However, many basic integer operations are not provided by such arithmetic[1]: Integer division is not immediate from field-arithmetic, nor is comparison of values – testing equality and inequality.

[1]Finite field arithmetic may of course be used to compute any function over that field, phrasing it as a polynomial. However, this approach is generally not feasible as the degree of the polynomial will depend on the size of the underlying field.

Particularly the latter is a must have. Without being able to determine the larger of two values, then e.g. determining the winning bid of an auction, or comparing costs and profits of companies when benchmarking is not possible.

The above provides the motivation for the goal of this work: The construction and use of what is essentially a *secure integer computer* based on standard multi-party computation protocols. This must provide not only efficient integer arithmetic (addition, subtraction, and multiplication), but generally all primitives that would be expected to be available on a "standard" computer. The most important of these are the comparison of values (testing equality and inequality), however, this is not all. Other operations such as modulo reduction and integer division should also be possible. (These will not be utilised in the present work, thus they will only be explored marginally, however, it is noted that they are possible.) In order to handle these, bit-decomposition is considered – essentially this is a representation change with field-elements being mapped to their binary representation. Though this representation is undesirable with regard to arithmetic, most other tasks benefit from it – thus representation change is desirable as it opens up the gates for Boolean circuits based on the binary representation of values.

The secure computer may be viewed as the trusted mediator from above. However, there are a few differences compared to a "real" computer, notably that it cannot run arbitrary "programs." Essentially only straight-line programs may be run, i.e. programs that do not branch or loop. Conditional behaviour requires knowledge of the values on which to branch if they are to occur. I.e. if they occur, it must be argued that the information obtained is acceptable. This is not to say that conditional behaviour cannot occur securely, however, in that case it must phrased obliviously. The construction of application below employs various tricks to ensure this, thereby preserving privacy.

1.1 An Overview

The contributions of the author fall into two distinct categories: First, the construction of efficient primitives for integer computation. Second, the use of these primitives in the construction of complex protocols solving high-level tasks. This thesis is similarly split up into two distinct parts covering these, though the model of secure computation is introduced before they are considered.

Part I sets the stage for the work to be performed. It provides a formal model of secure (arithmetic) computation – including the definition of what is meant by the word "secure." The model constructed is essentially a simple generalisation of previous work, and as such should not be seen as a contribution of the author. It is based on properties of primitives providing MPC rather than specific details. The model is constructed with adaptability in mind, i.e. it may easily be altered to consider e.g. alternative adversarial powers. This provides a quite "clean" platform for protocol construction in the subsequent parts. Moreover, by considering abstract primitives, *any* concrete protocols with the required properties may be used with the results of this work – the choices are

made with concrete candidates in mind, thus the model is well motivated.

Part II extends the model of computation through additional primitives. These are constructed in the basic model provided, i.e. based solely on arithmetic operations. Initially simple constructs are introduced which are needed for the efficient construction of the overall primitives. Some of these are previously known and have been included for completeness. Others have been taken from the paper [DFK+06] by Damgård, Fitzi, Kiltz, Nielsen, and the author. The remainder of part II consists of the core contributions, construction of actual primitives – constant-rounds equality testing, inequality testing, and bit-decomposition.

Part III then contains the second contribution: The construction of high-level MPC applications using the primitives introduced in the preceding part. All of these are heavily motivated by economics. The results include sealed bid auctions, notably the double auction of the SCET project – Secure Computing, Economy and Trust – of which the author was a part. The project included an industry partner with a real-world setting motivating this auction type. Additionally, means for solving linear program problems, while leaking a minimal amount of information, is provided. Many optimisation problems of economics may be phrased as linear programs, thus this provides a powerful tool. Also, the protocol may be used as a sub-protocol in even more complex computations – indeed this is the case for many of the applications considered. As such, part III may be viewed as a "library of functions" for even further use.

The thesis concludes with some final remarks in part 15.5. The initial goals are revisited, and commented on in the light of the work performed. Further, future directions of research are explored – both with regard to the protocols considered in details as well as additional primitives and applications.

Part I

The Model of Computation

Chapter 2

On the Model of Computation

Part I sets the stage for this thesis by introducing a formal model for secure multi-party computation. The goal is to present a well-defined, common model which provides a solid foundation for the chapters to come. I.e. it should provide basic operations and describe allowed adversarial behaviour, while still being flexible enough such that these may be altered, e.g. for capturing alternative notions of security. Moreover, the model should allow composition – i.e. it should allow construction to progress in stages in an iterative fashion, with more and more complex protocols constructed from simpler ones.

The overall goal is of course secure computation on integer values, however, basic MPC is performed over some finite ring or field. Naturally, this may simulate integer computation, but for the basic model, the underlying structure will be considered explicitly. Access to the exact nature of the underlying elements is required in the construction of primitives in part II; "pure integer computation" is not considered until part III.

Security is considered within the UC-framework of Canetti [Can00b]. This provides a basis for protocol construction as well as a strong concept of security. Chapter 3 provides an overview of the framework; moreover, it includes comments on why this is seen as a solid base for the present work. Chapter 4 then introduces the model of computation as an ideal functionality of the UC-framework. The abstract model is inspired by concrete protocols, examples are considered in chapter 5. These motivate the model through (references to) means of realising the basic, abstract primitives of the work. Part I concludes with additional comments on the model and protocol construction in chapter 6. This includes choices and assumptions for the following parts.

Chapter 3

Security of Protocols and the UC-framework

This chapter introduces a formal notion of protocols as well as their security. The definitions used are those of universal composability (UC), and a light overview of that framework is provided. The chapter is intended as a reminder and to give an intuitive level of understanding of security and the UC-framework, as well as explain the background of this choice. Knowledge of standard concepts from cryptography is assumed.

The approach often taken when considering security of protocols is the definitional one. The intended behaviour of the protocol in question is specified, this includes the inputs and outputs, as well as the amount of influence allowed and knowledge gained by malicious users; these are coordinated by an entity known as the *adversary*. A protocol is said to be secure if the output of the parties running the protocol and that of any adversary interacting with its execution are "equivalent" – in a well-defined sense – to output generated by the parties and a second adversary, interacting with a trusted third party "implementing" the intended behaviour.

In the UC-framework such a party is called an *ideal functionality*, and the associated protocol in which the parties interact with the ideal functionality, the *ideal protocol*. This second adversary is often known as a *simulator*, as it emulates the original using only knowledge and influence deemed acceptable. Equivalence therefore implies that any malicious result obtainable in a run of the real protocol is also obtainable when interacting with the ideal functionality. This implies both privacy (that no information is disclosed, except that which is deemed acceptable) and correctness (that the desired outcome occurs).

The UC-framework introduced by Canetti in [Can00b] uses this approach. It extends the notion of simulation similar to previous work by Canetti [Can00a], by introducing an additional entity, the *environment*. This represents everything external to the protocol execution, additional protocols and their adversaries, human users providing input, etc. The adversary remains and represents the concrete attack on the protocol. Security is defined by requiring that no environment can distinguish between an execution of the protocol and a simulation based on the ideal functionality. This ensures security even for protocols running on related inputs in arbitrarily complex settings, whereas standard

simulator arguments provide only security in a stand-alone setting.

This improved security guarantee is a major benefit of using the UC-framework, though it is not the only one. Working within the UC-framework allows modular construction of protocols, implying that proofs of security can be "split up" with focus placed on the task at hand based on abstract primitives guaranteed to be secure – rather than trying to analyse the combined protocol as a whole.

An additional benefit is that the framework is highly customisable. It encompasses multiple adversarial powers through simple variations. Arbitrary corruption structures – i.e. the sets of parties that the adversary is allowed to corrupt – can be specified independently of the framework and the functionalities.

Moreover, while computational powers are bounded in the basic definition of UC-security, this may be altered to consider e.g. unconditional security. Further, the effects of corruptions are not specified by the framework, instead they are stated as part of the functionality. This allows the framework to encompass adversaries which are either active or passive and adaptive or static with no difference except in the specification of functionalities.

Section 3.1 of this chapter describes the UC-framework. It provides an overview of what is (formally) meant by a protocol execution as well as an execution of the ideal protocol based on the ideal functionality. Then security within the UC-framework is defined and the concept of *hybrid protocols* introduced. The latter allows composition of protocols and leads up to the composition theorem of the UC-framework. Finally, following the presentation of the UC-framework, the background for considering UC-security in this work is discussed in section 3.2.

3.1 The UC-framework

The following description of the UC-framework is intentionally provided on a high level. The focus has been on providing an overview of relevant details sufficing for the following chapters. Broadly speaking, this consists of explaining what is meant by "protocol," "ideal functionality," and "equivalent." For an overview focusing more on the UC-framework itself the reader is referred to [Can01]; [Can00b] provides the full, detailed presentation. Additional discussion in particular on unconditional security may be found in [Can00a], in which a predecessor of UC with lesser security guarantees is presented. Many of the discussions in that work also apply to the UC-framework.

3.1.1 Protocol Execution Within the UC-framework

There are three types of entities in a protocol execution, two of which are unique: The environment, $\mathcal{Z}$, and the adversary, $\mathcal{A}$. The third type refers to the parties taking part in the protocol, $P_1, \ldots, P_n$. All three are modelled as interactive Turing machines (ITM). The specific details will not be covered here, except their communications tapes, which are required in the descriptions of allowed communication below, see [Can00b] for a full description. The *input tape* is

where the input of an ITM is provided by its initiating ITM. The *sub-routine output tape* is where sub-routines (which are also ITMs) of this ITM write their output. Finally, the *incoming communication tape* is intended for messages exchanged in a protocol-run. The ITMs are allowed to communicate with each other by writing to each others tapes as described below. This constitutes an execution of the protocol in question. Termination occurs when $\mathcal{Z}$ decides on an output-value based on its entire view, this value is considered the output of the protocol execution.

The environment, $\mathcal{Z}$, and adversary, $\mathcal{A}$, represent the attack on the protocol. As noted above, the environment represents everything external to the protocol, while the adversary represents the concrete attack. $\mathcal{Z}$ provides inputs to all parties and interacts arbitrarily with the adversary during the execution of the protocol. It also simulates any additional protocol executions (including any adversarial behaviour) influencing the current one, e.g. providing inputs. $\mathcal{A}$, representing the concrete interaction with the protocol, focuses on protocol-specific details of the attack; altering the inputs of corrupt parties, modifying or delaying messages, etc. Splitting the attack into two is crucial to UC-security. $\mathcal{Z}$ has full knowledge of what is expected to happen, however, when $\mathcal{A}$ must be simulated, the simulator may be denied important knowledge such as the inputs of honest parties.

$P_1, \ldots, P_n$ represent the parties taking part in the protocol. They are required to be instances of the same interactive Turing machine $\pi-$ viewed as a program rather than a machine – representing the protocol. Variations in behaviours, i.e. different roles in the protocol, can be obtained by specifying the role as part of the input. For feasibility of the protocols in question, the parties are assumed to be computationally bounded in their inputs and the security-parameter, probabilistic polynomial-time (PPT). Depending on the notion of security, this may also be the case for $\mathcal{Z}$ and $\mathcal{A}$, though in that case the latter is also allowed running time polynomial in the messages of the protocol. One point which should be stressed is that with regard to protocols within the UC-framework, the participating parties need not be known a priori. They may be added on-the-fly in an adversarially coordinated manner. However, such a feature is not a concern of this work. While it may be possible to construct MPC-protocols which *do* allow this, for simplicity it will be assumed that the number of parties and their identities are known when computation begins.

In addition to the entities above, a control function is required in the UC-framework. This function specifies what tapes of other parties a given entity may write to. Given a sender, a receiver, and a tape of that receiver, returns either allow or disallow depending on whether the operation is allowed. The model aims at providing only the most basic communication; it is unauthenticated, insecure, and unreliable. The communication allowed is specified in the following paragraph; any other attempt at communication (writing to the tapes of other parties) is prohibited and will have no effect.

The environment may interact with all parties, $P_1, \ldots, P_n$ by providing inputs on their input tapes, further, any output is sent back to $\mathcal{Z}$ by writing it on $\mathcal{Z}$'s sub-routine output tape. When receiving information in this manner, the identity of the sending party is added to the string written on the tape of $\mathcal{Z}$. In

the same manner $\mathcal{Z}$ and $\mathcal{A}$ may interact freely with each other at any point in the execution, implying that arbitrary information may flow between the two at any time. Concerning inter-party communication, no direct communication between different parties P_i and P_j is allowed. In order to provide the adversary with power over the communication, parties may only write to the incoming communications tape of $\mathcal{A}$ requesting that a provided message is delivered to some other party or parties. The adversary may then choose to write the message to the incoming communications tape of the intended receiver, however, there is no obligation to do so. Messages delivered need not be related to those sent by parties and arbitrary behaviour is allowed of $\mathcal{A}$; the message may be discarded, altered, delivered to other parties, or simply held back until additional messages are seen. Finally, the parties may invoke sub-routines in the form of additional ITMs. Except for input and output – which occur between invoker and invokee via the sub-routine output tape of the former and the input tape of the latter – these are equivalent to parties of the protocol. All other communication is disallowed and prevented by the control function.

Having introduced the different entities it is time to present the notion of protocol execution. An execution of a protocol π (representing the ITM of the parties) with regard to an environment, $\mathcal{Z}$ and an adversary, $\mathcal{A}$, (and control function) is initiated by providing input z to $\mathcal{Z}$ along with the security-parameter, κ and randomness $r_{\mathcal{Z}}$. After this it is activated. At any given time only one ITM is running, thus $\mathcal{Z}$ must hand over control to other entities allowing them to run. This is handled in the following way. When an ITM, S, writes a message to a tape of another ITM, R, S enters a waiting state, while R is activated; naturally this assumes that S was allowed to write to the desired tape of R. If an entity halts or enters the waiting state without writing to the tape of another entity, the environment is activated next.

Based on its inputs $\mathcal{Z}$ initialises $\mathcal{A}$ by providing it with its initial input, however, in addition to this the adversary is provided with its own source of randomness, $r_{\mathcal{A}}$ as well as κ. Once the environment is run again it may provide inputs for the parties, $P_1 \ldots, P_n$ in any order it likes (and at any pace) potentially dependent on $\mathcal{Z}$'s view of the execution. When party P_i is activated for the first time, either because it receives an input or because $\mathcal{A}$ delivered a message, it is initialised and provided with fresh randomness r_{P_i} unknown to $\mathcal{Z}$ and $\mathcal{A}$ as well as a copy of the security parameter κ. The entities then run in this manner, interleaved and one at a time, with the parties sending messages to each other through the adversary. Any output generated by honest parties is sent to $\mathcal{Z}$ as noted above, while corrupt parties – described further below – output $\perp$; their "true" output may be included in that of the adversary. The protocol execution terminates when $\mathcal{Z}$ decides to output some value based on its view – the inputs and outputs of the parties as well as that of the adversary.

While the protocol is running, $\mathcal{A}$ may *corrupt* a party, P_i, by sending a special (**corrupt**) message to P_i. This message may only be sent if the environment approves though. Formally, $\mathcal{Z}$ must have sent a message to $\mathcal{A}$ stating that this corruption is acceptable. Protocols are demonstrated secure with regard to some corruption structure $\mathcal{C}$, which specifies allowed corruptions i.e. the sets of parties which may become corrupt. This is viewed as part of $\mathcal{Z}$, with pro-

tocol security argued by quantifying over all environments with some specific corruption structure.

The exact nature of corruption is unspecified in the UC-framework, but various possibilities are allowed and can be "plugged in." As a concrete example, the approach taken in this work – as well as in [Can00b] – is that of an active adversary, This is easily changed to a passive one. When P_i becomes corrupt, its current state is sent to $\mathcal{A}$. This may include that party's entire history if the parties are not guaranteed to erase past data. In addition to this, all future inputs are forwarded to $\mathcal{A}$, who is also given full control over the future actions of P_i. Finally any future output sent to $\mathcal{Z}$ by P_i is $\perp$; $\mathcal{A}$ provides $\mathcal{Z}$ with output on behalf of P_i, potentially containing the full history of that party if available and desired.

The output of a run of a protocol – i.e. the output of the environment – gives rise to a distribution. Let $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)$ denote a random variable distributed as the output of a run of protocol π with environment $\mathcal{Z}$ and adversary $\mathcal{A}$ on security-parameter κ and input z for $\mathcal{Z}$ (the control function is left implicit here). Further, let $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the probability ensemble $\{\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$, the infinite set of distributions obtained by varying κ and z. This ensemble will be used in the definition of security below.

3.1.2 The Ideal Functionality and Associated Protocol

Describing the intended behaviour of a protocol is done by constructing an ideal functionality describing that behaviour. Based on this, an *ideal protocol* for $\mathcal{F}$ is constructed. Such a protocol is very similar to a real protocol as described above. Specifically, the whole setup echoes that of a protocol execution as above. The environment $\mathcal{Z}$ is the initial ITM run on input z. It initiates the execution of the adversary, $\mathcal{A}$, and provides inputs to the parties in exactly the same manner. The key difference is that in this setting, an additional entity exists, namely the ideal functionality $\mathcal{F}$. This entity – modelled as an additional PPT ITM – behaves as a trusted third party. All parties are connected to it by direct, secure (secret and authenticated) communications channels, and the parties simply hand over their inputs to $\mathcal{F}$. Based on all inputs $\mathcal{F}$ computes the outputs of all parties – which may depend on the internal randomness of $\mathcal{F}$ – and sends these back. Formally the control function must be updated, however, the intuition is that $\mathcal{F}$ is a common sub-routine of all parties. Thus, in this "ideal world" $\mathcal{Z}$ interacts with dummy parties in the sense that they act as simple communications channels between the environment (providing the inputs) and the functionality (determining the outputs).

Concerning the adversary, no information is leaked through the communication with $\mathcal{F}$, as – in contrast to the real protocol execution described above – the communication bypasses $\mathcal{A}$ completely. However, protocols cannot always avoid leaking information, e.g. an encryption may hide the contents of a message, but still provides a bound on its size. This is modelled by allowing $\mathcal{F}$ and $\mathcal{A}$ to communicate directly with each other using their communications tapes. Upon receiving a message from P_i, $\mathcal{F}$ provides $\mathcal{A}$ with any relevant information, while $\mathcal{A}$ is allowed to send well-defined orders to $\mathcal{F}$ which are then executed

– these describe adversarial influence deemed acceptable. An example is an active adversary altering the input of a corrupt party.

In a run of the ideal protocol of functionality $\mathcal{F}$, corruptions are handled by $\mathcal{F}$, i.e. the corruption structure $\mathcal{C}$ is "included in" $\mathcal{F}$. When $\mathcal{A}$ wishes to corrupt party P_i, it sends a message to $\mathcal{F}$ stating this. If it is accepted, i.e. if performing the desired corruption does not violate $\mathcal{C}$, the environment is notified as in the real world. Moreover, $\mathcal{F}$ provides $\mathcal{A}$ with any information currently known by P_i, this may include all previous inputs and outputs. All future inputs and outputs of that party are then forwarded to $\mathcal{A}$ when they become available to $\mathcal{F}$, while P_i is handed $\perp$ as output from then on. Finally, as an active adversary is considered it gains full power over P_i. At the very minimum this allows inputs to be altered. Alternative adversarial models are easily constructed through minor alterations. E.g. a passive adversary may be considered by requiring that corrupt parties must follow their specification, i.e. provide the given inputs.

As in an execution of a real protocol this gives rise to a probability distribution. Similar to above, let $\mathrm{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}}(\kappa, z)$ denote a random variable associated with the distribution occurring when the ideal protocol of $\mathcal{F}$ is run with environment $\mathcal{Z}$ and adversary $\mathcal{A}$ on security parameter κ and input z for the environment. Also, let $\mathrm{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}}$ denote the associated distribution ensemble $\{\mathrm{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}}(\kappa, z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^*}$, which will be used below in the definition of security.

3.1.3 UC-security and the UC-theorem

A protocol π is considered *secure*, if it is "equivalent" to executing the ideal protocol of an ideal functionality $\mathcal{F}$ describing its intended behaviour. Any malicious behaviour achieved by an adversary $\mathcal{A}$ in an execution of π should also be realisable by another adversary, the simulator $\mathcal{S}$, in the ideal protocol where the parties interact directly with $\mathcal{F}$. $\mathcal{S}$ then emulates the view of $\mathcal{A}$ in the ideal setting, while $\mathcal{Z}$ works as an interactive distinguisher; thus $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ and $\mathrm{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{Z}}$ should be the same in a well-defined sense. Without loss of generality, it will be assumed that the output of $\mathcal{Z}$ is a single bit – a guess on whether it is interacting with $\mathcal{A}$ and π or with $\mathcal{S}$, the dummy parties, and $\mathcal{F}$.

Formalising the above, a binary distribution ensemble $\{X(\kappa, a)\}_{\kappa\in\mathbb{N},a\in\{0,1\}^*}$, is an infinite set of binary distributions parametrised by a positive integer, κ, and some arbitrary input, $a \in \{0,1\}^*$, where each pair (κ, a) gives rise to some binary distribution. Equivalence is then defined through indistinguishability:

Definition 3.1 (Indistinguishability) *Let $X = \{X(\kappa, a)\}_{\kappa\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(\kappa, a)\}_{\kappa\in\mathbb{N},a\in\{0,1\}^*}$ be two binary distribution ensembles. X is said to be indistinguishable from Y if for any $c, d \in \mathbb{N}$ there exists k_0 such that for all $\kappa > k_0$ and $a \in \cup_{k\leq\kappa^d}\{0,1\}^k$*

$$\left|\Pr[X(\kappa, a) = 1] - \Pr[Y(\kappa, a) = 1]\right| < \kappa^{-c}.$$

Indistinguishability of two ensembles, X and Y, is written $X \overset{p}{\approx} Y$ with p denoting that equivalence only quantifies over all a of polynomial length.

Ordinarily when considering indistinguishability, there is no requirement on the part of a to be polynomial in κ. This is a technical requirement of the UC-framework, when computationally bounded adversaries and environments are considered. The running time of $\mathcal{Z}$ depends on both the security parameter and its input, however, if the latter is "too large," it may allow computation which is super-polynomial in κ. Thus, only polynomial size inputs are considered in the definition of indistinguishability. Based on this the definition of UC-security can be defined.

Definition 3.2 (UC-security) *A protocol π is said to UC-realise a functionality $\mathcal{F}$ if for any PPT adversary $\mathcal{A}$, there exists a PPT adversary $\mathcal{S}$, such that for any PPT environment $\mathcal{Z}$:*

$$\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \overset{p}{\approx} \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}.$$

A protocol π UC-realising a functionality $\mathcal{F}$ is said to be UC-secure.

Unconditional UC-security Definitions 3.1 and 3.2 provide UC-security in the cryptographic setting where computation is limited to being probabilistic polynomial-time. However, while this is the basic framework, the definitions can be extended to cover alternative computational powers of adversaries. Providing unconditional security in the UC-framework is a simple matter of providing $\mathcal{A}$ and $\mathcal{Z}$ with unbounded computational power, i.e. dropping the requirement that they are probabilistic polynomial-time. Moreover, as there is no bound on the running time, the definition of indistinguishability should reflect this, requiring that *all* distributions are negligibly far apart, not just those with inputs of polynomial length – this is written $X \approx Y$. A final point is that simulation of an adversary $\mathcal{A}$ should require a comparable amount of computational resources as running the adversary itself. Thus, it is required that $\mathcal{S}$ is polynomial in the complexity of $\mathcal{A}$. Giving the adversary and environment unbounded computational powers in this manner is the well-known concept of *statistical* rather than *computational* security. Requiring the distributions to be identical provides *perfect* security.

Definition 3.3 (Statistical/Perfect UC-security) *A protocol π is said to statistically UC-realise a functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$, there exists an adversary $\mathcal{S}$ with running time polynomial in that of $\mathcal{A}$, such that for any environment $\mathcal{Z}$:*

$$\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}.$$

A protocol π statistically UC-realising a functionality $\mathcal{F}$ is said to be statistically UC-secure. Similarly, if the ensembles are equal, i.e. $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\kappa, z)$ and $\mathrm{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa, z)$ are identical for all $\kappa \in \mathbb{N}$ and $z \in \{0,1\}^*$, π is said to perfectly UC-realise $\mathcal{F}$ and be perfectly UC-secure.

Universal Composition The final part of this section consists of the composition theorem of the UC-framework, which is applicable with regard to all

notions of realisations of functionalities presented. However, before it is presented, the hybrid model is introduced along with the UC composition operation. For simplicity a slightly reduced composition operation and theorem are presented here. The operation of [Can00b] considers replacement of arbitrary sub-protocols, while the one here considers only replacement of the ideal protocol of a functionality by a concrete protocol realising it. The difference is minor and a result of the focus of this work.

Protocols in the hybrid model – also known as *hybrid protocols* – are defined as standard, real-world protocols with regard to communication. The difference is that the real-world model is extended in the sense that in addition to ordinary communication, the parties are allowed to interact with one or more ideal functionalities; for simplicity of this description it will be assumed that there is only one, $\mathcal{F}$. This model is denoted the $\mathcal{F}$-hybrid model, where in addition to being able to communicate with the adversary, the parties may invoke an unbounded number of instances of $\mathcal{F}$ in the natural way: The parties are allowed to invoke dummy-parties of $\mathcal{F}$ in the same way as the environment interacting with the ideal protocol of $\mathcal{F}$. These sub-parties forward inputs to the functionality, and return output directly to their initiators' sub-routine output tape. In the $\mathcal{F}$-hybrid model the adversary is allowed to interact with the protocol as in the real world, as well as with the instances of $\mathcal{F}$ (in the manner that $\mathcal{F}$ defines), however, the dummy-parties are considered incorruptible.

The composition operation is defined in the obvious way. Let protocol π be defined in the $\mathcal{F}$-hybrid model, while protocol ρ UC-realises (perhaps statistically or perfectly) $\mathcal{F}$. It is now possible to consider the composed protocol, written $\pi^{\rho/\mathcal{F}}$, where the invocations of $\mathcal{F}$ performed by the parties of π are replaced by invocation of ρ. I.e. whenever a party of π is instructed to invoke a dummy party of $\mathcal{F}$, a party of ρ is run with the same input instead. This party then participates in an execution of protocol ρ – interleaved with the rest of π and possibly other instances of ρ – returning output to the sub-routine output tape of the invoking party, exactly as the dummy parties in the hybrid model. It is stressed that in difference to the dummy parties, the parties of ρ are *not* incorruptible, however, by the UC-theorem below, *no* adversary gains *any* (useful) advantage against the composed protocol that does not already exist within the hybrid model. A final remark on the description of the composition operation is that, as both π and ρ are polynomial-time so is the composed protocol $\pi^{\rho/\mathcal{F}}$.

Based on the hybrid model and the composition operation, the UC-composition theorem is stated, see [Can00b] for the proof.

Theorem 3.1 (The Universal Composition Theorem) *Let $\mathcal{F}$ and $\mathcal{G}$ be two ideal functionalities, and let π UC-realise $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Moreover, let ρ UC-realise $\mathcal{G}$. Then the composed protocol, $\pi^{\rho/\mathcal{G}}$, UC-realises $\mathcal{F}$. Similarly, if π and ρ are statistical UC-realisations, $\pi^{\rho/\mathcal{G}}$ statistically UC-realises $\mathcal{F}$, and if they are perfect UC-realisations, $\pi^{\rho/\mathcal{G}}$ perfectly UC-realises $\mathcal{F}$.*

This allows protocols to be constructed and proven secure in a hybrid model, easing the job considerably and providing independence of the underlying prim-

itives. Note that combining different notions of security is possible and results in the "minimum" of the two; perfect UC-realisation implies statistical UC-realisation which in turn implies UC-realisation.

One final point to note with regard to theorem 3.1 is that it may be applied multiple times, i.e. complex protocols can be constructed and proven secure in an iterative fashion. However, for technical reasons the theorem allows only a *constant* number of applications – the complexity of the simulator may no longer be polynomial. This is of no consequence in this work.

3.2 On Using the UC-framework

There are several reasons why the UC-framework has been chosen as the basis for arguing security of the multi-party computation protocols of this work. First off, in its nature multi-party computation is highly reactive as opposed to secure function evaluation. It consists of repeated inputs and calculations, which may depend arbitrarily on the previous steps. The UC-framework is designed to be reactive, preserving state and potentially with additional inputs generated at run-time. Thus, it is imminently suited, while ensuring security of protocols in arbitrary, complex settings. This is not only a strong guarantee, but may be a requirement in many practical settings, e.g. protocols running over the Internet.

A second benefit is that multiple adversarial behaviours and powers are possible. The same functionality may encompass several possibilities with little alteration. Moreover, realising primitives can then be chosen based on their properties and the (security) requirements of the high-level application. Essentially, the protocols obtained through the composition theorem inherit the properties of the primitives when no "real" communication occurs. Thus, working within a hybrid model allows the primitives to be abstracted away. This not only allows the details of the primitives to be ignored, it ensures that different primitives can be used for the same application in differing settings while still requiring only one proof of security. An added benefit of this is that if new (improved) primitives realising the ideal functionality for multi-party computation are discovered, *any* previously constructed high-level protocol within the hybrid-model *immediately* has a new (improved) realisation.

The final, major selling point is that of composability of protocols. Though this is not the key selling point of the UC-framework, it is noted that modular constructions of protocols are a benefit; it allows easier analysis as simpler protocols are analysed. This idea was used above in order to abstract away the details of the primitives, however, there are further benefits. If all interaction between parties take place through the ideal MPC-functionality, the only leakage of information can occur when information is intentionally revealed. Moreover, the iterative approach to protocol construction allows the initial basis (functionality) to be of limited complexity and thus easily realisable. It can then be extended with with more complex "primitive operations," which must only be argued (UC-)secure in the hybrid model.

Thus, new capabilities are added with simpler proofs of security, and as a bonus they are *independent* of the most basic primitives. The extended

functionality in turn provides a better hybrid model for constructing protocols solving even more complex tasks, i.e. for constructing protocols for concrete applications.

Chapter 4

A UC-functionality for Secure Computation

Having given a brief intuition of the overall goal of secure, multi-party computation in the introduction, it is time to provide a more formal description of it within the UC-framework considered in chapter 3. I.e. it is time to define a UC-functionality, $\mathcal{F}_{\mathbf{MPC}}$, for performing computation securely.

The goal is to provide a general functionality for arithmetic multi-party computation allowing simple construction of protocols performing computations in the associated hybrid model. In the chapters to come this then serves as the basis for constructing additional primitives as well as protocols solving complex problems based on the original arithmetic and the added primitives. This implies that the functionality is described in broad terms. The focus has been on creating a clean base providing the core primitives needed while stripping away irrelevant details of possible realisations. The work is heavily inspired by the arithmetic black-box (ABB) of Damgård and Nielsen [DN03], however, their work considered security of specific protocols, whereas the current aims at formalising a general, well-defined model for secure computation.

The chapter is divided into five parts. First, in section 4.1 the main ideas behind the functionality as well as its overall structure are considered. Following this, section 4.2 introduces the notion of rounds. In difference to secure function evaluation, multi-party computation does not provide a result "in one go" rather it is an iterative process which must be reflected by the functionality. Sections 4.3 and 4.4 provide the details of $\mathcal{F}_{\mathbf{MPC}}$, first considering initialisation, and then the computation itself. The chapter concludes in section 4.5 with a number of variations of the specified functionality, considering alternative notions of adversarial powers.

4.1 A High-Level View of $\mathcal{F}_{\mathbf{MPC}}$

The structure of the presented functionality is very similar to the arithmetic black-box of [DN03]. Both start by initialising an "empty state." Working in *rounds* $\mathcal{F}_{\mathbf{MPC}}$ repeatedly receives commands for inputting values from the parties or performing computation which updates the state. Additionally, the values assigned to the variables in the state can be *output* to the parties. The main difference between the two is that for the present no specific realising

protocols are considered, thus there are no specific details referring to these.

The primitives provided here are as simple and general as possible, while leaving the possibilities for alteration – e.g. with regard to alternative adversarial models – as open as possible.

The adversary is considered to be both active and adaptive. From this, it is simple to alter the functionality to fit other models such as passive or static adversaries. This requires only removal of options for the simulator. Regarding corruption structure $\mathcal{C}$ and computational powers, these do not influence $\mathcal{F}_{\mathbf{MPC}}$, thus they remain unspecified until a protocol is to be shown realising.

Overall, $\mathcal{F}_{\mathbf{MPC}}$ proceeds in two distinct phases: First an initialisation phase occurs in which all requirements of the computation are set up. The second phase then consists of the actual computation. This phase is further split up into *rounds* in which commands are executed, this is explained further below. Any attempt at computation prior to initialisation as well as any subsequent attempts at initialisation is ignored by $\mathcal{F}_{\mathbf{MPC}}$ ensuring that initialisation occurs at the very beginning and only once.

The initialisation phase consists of $\mathcal{F}_{\mathbf{MPC}}$ selecting the ring, $\mathbb{R}$, in which computation will occur. The exact details are left unspecified as they are tied to the concrete protocol in question. The parties are allowed to provide arbitrary data for $\mathcal{F}_{\mathbf{MPC}}$, which then selects $\mathbb{R}$ based on protocol specific rules. The phase is singled out explicitly, as it ensures that $\mathcal{F}_{\mathbf{MPC}}$ knows $\mathbb{R}$ and that the parties agree. Moreover, it also allows an adaptive choice fulfilling arbitrary requirements. The ring may be chosen based on the number of parties or based on requirements of the computation to be performed. Additionally, it may depend on the security-parameter κ. The approach also provides the possibility of using sub-functionalities for choosing $\mathbb{R}$, e.g. as was done in [DN03], where the ring was specified by the public key of a threshold-crypto-system generated before computation.

The computation phase consists of the parties repeatedly providing *commands* to the functionality, which are then executed. This occurs in rounds; in each round the state of $\mathcal{F}_{\mathbf{MPC}}$ is updated based on the inputs provided. All the functionality does is store a list of defined (set) variables along with the value currently assigned to each of them. Initially all variables are undefined, and when a command sets variable x for the first time it is defined taking a value $v \in \mathbb{R}$ specified by the command, i.e. (x, v) is stored within the functionality. The value assigned to x will be written $\mathrm{val}(x)$.

There are two basic types of commands. Assignments are used to set variables, i.e. updating or inserting pairs (x, v) in the list of variables. This covers both inputs of the computation from individual parties as well as arithmetic expressions computed over already defined variables, such as $x \leftarrow x_1 + x_2$. The second type allows outputting the values assigned to variables, i.e. for the parties to learn $\mathrm{val}(x)$ for some variable x. Until a value is output, all that a party knows about it is what is implied by the inputs provided by that party.

When $\mathcal{F}_{\mathbf{MPC}}$ considers the next command to execute, it requires that all honest parties provide the *same* command. In general the inputs of the corrupt parties are ignored unless explicit activity on the part of corrupt parties is required. The only exception to the above is the command specifying that

some party, P_i, should input a value, (**input** $P_i : \ x \leftarrow v$) with $v \in \mathbb{R}$. This differs as the value may be known only to P_i; indeed if all inputs were known to all there would be no need to use multi-party computation. Thus, rather than requiring that the honest parties specify the value, they must simply *authorise* the assignment by inputting the command (**input** $P_i : \ x \leftarrow ?$). Note that this is an example of a command where input from corrupt parties cannot be ignored: Even if P_i is corrupt, it should still be allowed to follow the protocol and input a value. However, if any command other than (**input** $P_i : \ x \leftarrow v$) for $v \in \mathbb{R}$ is given, the only effect will be that all parties are notified that P_i refused to cooperate.

If the honest parties do not agree, but input different commands, the functionality becomes corrupted in the sense that it forwards its entire history of inputs (commands) to the adversary thus ensuring that all values input into the system become known. Additionally, the adversary is given full control, all future inputs from all parties are forwarded to the adversary who then specifies their outputs and hands these back through $\mathcal{F}_{\mathbf{MPC}}$. The idea behind the "corruption of $\mathcal{F}_{\mathbf{MPC}}$" is that it ensures that *if* the players ever disagree on the computation taking place – i.e. if environment provides different commands for the honest parties – the simulator will be able to simulate the interaction perfectly, as it has full information. For any realising protocols this means that absolutely no guarantees of security or correctness are given, if the honest parties can be tricked into performing differing actions, either executing different protocols or the same protocols using mismatched data.

4.2 Rounds of $\mathcal{F}_{\mathbf{MPC}}$

The computation phase is split up into rounds thus allowing updating the state in a iterative fashion. This is a key point of multi-party computation opposed to secure function evaluation. It allows computation to be adaptive, depending on values output. Moreover, it allows improvements to complexity, as values may be output, modified and re-input into the computation, thus performing costly computation "outside of $\mathcal{F}_{\mathbf{MPC}}$" – though at the cost of leaking information.

Concerning complexity, when protocols are constructed *round complexity* is generally considered. This refers to an overall count of messages sent during an execution. In MPC, primitive operations may typically be parallelised thus resulting in better round complexities – indeed, round complexity is often a prime focus both in theory and practice. Thus, if any arguments of complexity in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model are to make sense, the concept must be present within $\mathcal{F}_{\mathbf{MPC}}$ and must be related to the complexity of the realising protocols. As these are unspecified, ensuring that relation is of course not possible in general, however, primitives (in the form of commands) may be counted. These translate to a concrete count if specific, realising protocols are considered.

Arbitrary parallelisation cannot occur though. First off, commands may depend on each other. If one protocol requires input which is output by another, then it cannot be run until the other provides that output. There is also a second more subtle reason. Security may not be guaranteed under ar-

bitrary parallelisation. This may be the case even when the basic protocols are UC-secure in a "stand-alone" setting. Though composition is at the heart of the UC-framework, proofs are not immediately applicable when protocols have non-disjoint state; and states here may be highly related, with the same variable taking part in numerous expressions. Though it may be possible to employ universal composition with joint state (JUC) [CR03] in order to argue security, most likely the whole structure of the arithmetic functionality must be simulated. Thus introducing a notion of *parallelism* – a description of restrictions on the commands which may occur in parallel, i.e. in a single round – and arguing security with regard to this is preferable as it eases the description of $\mathcal{F}_{\mathbf{MPC}}$. The parallelism is in a sense an add-on to $\mathcal{F}_{\mathbf{MPC}}$ and different realising protocols may allow different amounts of parallelising. Thus the problem is postponed in the sense that the parallelism must be specified when proving realisation. This ensures a well-defined concept of round complexity, while avoiding limitations on the expressive power of the basic protocols.

Formally, a round of $\mathcal{F}_{\mathbf{MPC}}$ consists of all parties sending an ordered list of commands to $\mathcal{F}_{\mathbf{MPC}}$ which are then processed in parallel, one command from all parties at a time. During this it is verified that the notion of parallelism is not violated by any commands – offending commands are simply ignored. Once all commands have been processed a bulk message based on the results of all commands is then sent to the adversary. This consists of the entire effect of the individual commands, i.e. the concatenation of the information the adversary would normally obtain. The adversary then corrupts parties and interacts with $\mathcal{F}_{\mathbf{MPC}}$ as if the commands were individual ones. When allowed the functionality sends the results of all (possibly failed) commands to all parties, i.e. the concatenation of the output they would receive normally.

Note that failures may cascade within a bulk-message – the failure of one command may provoke additional ones. The immediate example is when the input of one depends on the output of another; as the input is unavailable the command cannot be executed. $\mathcal{F}_{\mathbf{MPC}}$ handles this by "rolling back" the effect of such commands before sending failure messages, not only for the initial failing operations but also any associated ones.

For completeness, a simple "dummy" parallelism is considered. Assume that only computational commands for basic arithmetic are provided, i.e. the addition or multiplication of two variables (or one variable and a constant). First, let all commands of $\mathcal{F}_{\mathbf{MPC}}$ require one round – this includes input, output and the evaluation of the binary operators. Further, only one command per round will be allowed. This provides a simple notion by allowing no parallelisation. However, while better alternatives exist for most realisations, it allows parallelisation to be ignored in the description of $\mathcal{F}_{\mathbf{MPC}}$. In chapter 6 below, a more expressive possibility – used often in literature – is introduced; this is then used in parts II and III.

4.3 The Initialisation Phase in Detail

The initialisation phase consists of all parties providing arbitrary data to $\mathcal{F}_{\mathbf{MPC}}$ who then decides on a ring $\mathbb{R}$. This is then provided to the adversary, $\mathcal{A}$, and all parties, $P_1, \ldots, P_n$. It is merely a setup phase, ensuring that $\mathbb{R}$ is well-defined and known by $\mathcal{F}_{\mathbf{MPC}}$ as well as the parties and the adversary. Also, it allows $\mathbb{R}$ to be chosen adaptively based on inputs; potentially through sub-protocols (or -functionalities). For example, if $\mathbb{Z}_m$ is to be used for simulating integer-computation, it must be ensured that m is chosen sufficiently large such that no overflow – no reductions modulo m – occurs during the course of the computation. The phase is simple, described in broad terms as details are very much dependent on realising protocols.

The phase progresses as follows. All parties send an initialisation message to $\mathcal{F}_{\mathbf{MPC}}$, with P_i sending $(\mathbf{init} : d_i)$ for $d_i \in \{0,1\}^*$. $\mathcal{F}_{\mathbf{MPC}}$ then decides on a ring $\mathbb{R}$ based on well-defined, protocol-specific rules. The choice may depend on the d_i – e.g. that all (honest) parties provide a description of the same ring, which is then selected – however, this is not a requirement. The only restriction made is that the required computation is polynomial time. The description of $\mathbb{R}$ is sent to the adversary, possibly along with auxiliary data, e.g. required for simulation of the realising protocol. When accepted by the adversary, $\mathcal{F}_{\mathbf{MPC}}$ initialises the list of variables and sends the description of $\mathbb{R}$ to all parties. This concludes the init phase.

However, before accepting $\mathbb{R}$, $\mathcal{A}$ may corrupt parties and update their inputs, potentially resulting in a new ring being chosen and sent to $\mathcal{A}$. Again the adversary is allowed to perform corruptions and alter inputs, indeed the behaviour is repeated again and again until $\mathcal{A}$ accepts a choice of ring. A simpler solution would be to require $\mathcal{A}$ to make *all* corruptions and update *all* d_i in one go. However, this is less powerful; e.g. it does not allow the adversary to choose parties to corrupt based on the inputs of others, thus an iterative approach has been taken.

In the initialisation phase, any other messages than $(\mathbf{init} : d_i)$'s are ignored by $\mathcal{F}_{\mathbf{MPC}}$. Similarly, so are initialisation messages received after the phase has ended. This ensures that the phase occurs only once, before attempting any computation.

4.4 The Computation Phase in Detail

The computation phase deals solely with secure computation over $\mathbb{R}$. The phase is protocol independent in the sense that the details specific to a realising protocol are left general. Examples include the ring $\mathbb{R}$ and the notion of parallelism. Thus, this part of $\mathcal{F}_{\mathbf{MPC}}$ may be viewed as containing the *core* properties of multi-party computation. Regarding this section, it is assumed that initialisation has occurred, i.e. that $\mathbb{R}$ has been generated and distributed to the parties. Moreover, $\mathcal{F}_{\mathbf{MPC}}$ has initialised the list of defined variables.

There are essentially three types of commands which $\mathcal{F}_{\mathbf{MPC}}$ accepts: *input*, *computation*, and *output*. Each command will be considered individually, being

viewed as atomic and non-parallelisable by the dummy notion of parallelism. The structure of all command executions follow the same template. Upon receiving a command from all honest parties, $\mathcal{F}_{\mathbf{MPC}}$ verifies that it is the same, otherwise the functionality becomes corrupt as described in section 4.1 above. If the same command is given, it is forwarded to the adversary ensuring that the adversary is aware of what is occurring. $\mathcal{A}$ may then interact with $\mathcal{F}_{\mathbf{MPC}}$ on behalf of any of the corrupt parties and also corrupt further if desired. This may in turn lead to changes in the effect of the command, including its failure. Finally, $\mathcal{A}$ authorises $\mathcal{F}_{\mathbf{MPC}}$ to execute the command and send its effect – e.g. "failure" – as output to the parties.

Keeping the ideal functionality general and adaptable, $\mathcal{F}_{\mathbf{MPC}}$ allows the adversary to prevent *any* commands from executing. This provides $\mathcal{A}$ with the greatest amount of power; further, it is easily reducible if less powerful adversaries are to be considered. Removing adversarial powers is a simpler task than adding them to an existing description. Essentially the motivation behind letting the adversary fail commands is that performing an action may require activity by all parties – including the corrupt. Though protocols exists which guarantee termination (when an "input phase" has occurred), this does not cover all possibilities for realisation, thus the more general approach is taken in the description of $\mathcal{F}_{\mathbf{MPC}}$.

input On message (**input** $P_i : \ x \leftarrow ?$) from all honest P_j, $j \neq i$, and message (**input** $P_i : \ x \leftarrow v$) with $v \in \mathbb{R}$ from P_i, $\mathcal{F}_{\mathbf{MPC}}$ stores that the command to be executed sets $\mathrm{val}(x)$ to v.[1] (**input** $P_i : \ x \leftarrow ?$) is then forwarded to the adversary, who may choose to corrupt parties. Note that if P_i is corrupt, then the adversary knows v, as it has access to P_i's internal state. Moreover, in this case $\mathcal{A}$ may choose to input a different value $v' \in \mathbb{R}$ from P_i by sending an (**update** $P_i : \ x \leftarrow v'$) message. $\mathcal{F}_{\mathbf{MPC}}$ updates the command to be executed to setting $\mathrm{val}(x)$ to v' instead. Also, when P_i is corrupt $\mathcal{A}$ may cancel the assignment by sending the message (**cancel** $P_i : \ x \leftarrow v$); in this case $\mathcal{F}_{\mathbf{MPC}}$ notes that the command fails.[2] Finally, when $\mathcal{A}$ allows it, $\mathcal{F}_{\mathbf{MPC}}$ executes the command, i.e. sets $\mathrm{val}(x)$ to v (or v' if $\mathcal{A}$ updated it), and outputs the result of the command to all parties, either (**failure**), if P_i was corrupt and $\mathcal{A}$ cancelled the command, or (**success**) if x was set to some value.

Addition of constant For $c \in \mathbb{R}$ and variables x and x', where x' is defined, on message (**let** $x \leftarrow x' + c$) from all honest parties, $\mathcal{F}_{\mathbf{MPC}}$ forwards the command to $\mathcal{A}$. $\mathcal{A}$ may then corrupt parties, however, as the inputs of the corrupt parties are ignored by $\mathcal{F}_{\mathbf{MPC}}$, updating them makes no differences, and it is therefore not allowed. The adversary may authorise the execution of the command, after which $\mathcal{F}_{\mathbf{MPC}}$ sets $\mathrm{val}(x)$ to $\mathrm{val}(x') + c$ and sends (**success**) to all parties indicating that the command was execute. Alternatively, $\mathcal{A}$ may

[1] If P_i is corrupt, and $\mathcal{F}_{\mathbf{MPC}}$ has not received an input from it, it considers the command to have failed. In this case, a message, (**failure** : $P_i : \ x \leftarrow ?$) is sent to $\mathcal{A}$.

[2] Cancelling inputs of *honest* parties could equally well be considered; this provides the adversary no additional power.

disallow the command, in this case $\mathrm{val}(x)$ is left as it were, and the parties receive a message indicating failure, (**failure**).

Multiplication by constant This is similar to addition of a constant. For $c \in \mathbb{R}$ and variables x and x' where the latter is required to be defined, on message (**let** $x \leftarrow c \cdot x'$) from all honest parties, $\mathcal{F}_{\mathbf{MPC}}$ forwards (**let** $x \leftarrow c \cdot x'$) to the adversary who may then either allow its execution – $\mathrm{val}(x)$ is set to $c \cdot \mathrm{val}(x')$ and (**success**) is sent to all parties – or it may be denied – the command has no effect, and (**failure**) is sent instead.

Addition Again, this is similar to the commands described above. For variable x and defined variables x_1 and x_2, on message (**let** $x \leftarrow x_1 + x_2$) from all honest parties, the command is forwarded to the adversary, who either allows it, in which case $\mathcal{F}_{\mathbf{MPC}}$ sets $\mathrm{val}(x)$ to $\mathrm{val}(x_1) + \mathrm{val}(x_2)$ and sends (**success**) to all parties, or denies it, in which case (**failure**) is sent.

Multiplication As above, for variable x and defined variables x_1 and x_2, on message (**let** $x \leftarrow x_1 \cdot x_2$) from all honest parties, $\mathcal{F}_{\mathbf{MPC}}$ forwards the command to $\mathcal{A}$. The command is either allowed or denied by $\mathcal{A}$, in the first case $\mathrm{val}(x)$ is set to $\mathrm{val}(x_1) \cdot \mathrm{val}(x_2)$, and a message indicating either success or failure is sent to the parties.

Output On message (**output** x) from all honest parties, where x is a defined variable, $\mathcal{F}_{\mathbf{MPC}}$ sends (**output** $: x = \mathrm{val}(x)$) to the adversary. As with the other commands, arbitrary corruptions as specified by $\mathcal{C}$ may then occur, however, again altering the inputs of the corrupt parties has no effect and is ignored. Having received $\mathrm{val}(x)$, $\mathcal{A}$ may now choose to publish it to the parties, or to fail it. If $\mathcal{F}_{\mathbf{MPC}}$ is allowed to provide the value to the parties, the message (**output** $: x = \mathrm{val}(x)$) is sent to all of them. Otherwise, (**failure**) is sent.

4.5 Alternative Adversarial Powers

As specified above, the scenario considered consists of an active and adaptive adversary, corrupting parties in sets of some corruption structure $\mathcal{C}$. Many alternatives exist, however, the presented model was chosen for its strength. *Removing* an adversarial power from a functionality is an easier task than *adding* one. Popular alternatives are considered here. With regard to the unspecified properties of the adversary – the corruption structure and the computational powers – these remain *outside* the functionality, and must be specified in conjunction with concrete, realising protocols.

Restricting the adversarial powers can be done as discussed in [Can00b]. Passive adversaries are considered by not allowing $\mathcal{A}$ to neither alter the inputs of the corrupt parties, nor fail any commands. In a similar manner, static (or non-adaptive) parties can be handled by stating that all corruptions of parties must occur *before* the init phase, i.e. the first message of all parties must be a corruption. Erasing parties – i.e. a setting in which the honest parties may

be trusted to erase unwanted data – can be handled by assuming that it can be specified whether inputs and outputs must be simulated. Moreover, $\mathcal{F}_{\mathbf{MPC}}$ must be extended by an "undefine-variable" command.

A final point to consider is termination. When the adversary is active, termination is not ensured in general, as the corrupted players are allowed to behave arbitrarily, including doing nothing. This is modelled in $\mathcal{F}_{\mathbf{MPC}}$ as the adversary failing commands. There are interesting scenarios where security requirements prevent ensuring termination. Guaranteeing privacy of inputs even if all other parties collude – the notion of self-trust of Brandt and Sandholm [BS05] – is desirable and may be required in certain applications. However, if privacy cannot be compromised, it implies that the party must take active part in the protocol, thus it may be sabotaged.

However, other applications may require guaranteed termination. There exist protocols which ensure termination, and $\mathcal{F}_{\mathbf{MPC}}$ is easily altered to reflect this property. By simply removing the ability of the adversary to prevent commands their execution is guaranteed. The exception to this rule is the input command. Its success is tied to a single party, and it cannot be guaranteed that parties provide input. Thus, it must be accepted that the command may fail. However, as the honest parties are notified, they are free to specify a "default value" for that input for the remainder of the computation.

Chapter 5

Motivating $\mathcal{F}_{\mathrm{MPC}}$ – On Possible Realisations

Having constructed a model for performing multi-party computation in the form of an ideal functionality within the UC-framework, the motivation for choices made will be considered. I.e. it will be argued that $\mathcal{F}_{\mathbf{MPC}}$ provides a sensible basis for constructing secure computation as it is realisable. In a sense, this chapter may be seen as brief comments on issues of a full (real-world) implementation of the protocols discussed in parts II and III. Further, the candidates considered provide motivation for the following chapter, particularly with regard to the notion of complexity to be used in the remainder of this work.

The chapter approaches the realisation of $\mathcal{F}_{\mathbf{MPC}}$ in a top-down manner. Section 5.1 considers the concept of *secret sharing* which provides a basis for much MPC. Further, one popular construct which realises $\mathcal{F}_{\mathbf{MPC}}$ is noted. However, these protocols require access to secure communication, i.e. the protocols are constructed in a simpler hybrid model. Primitives realising this model are then explored further in 5.2 along with an alternative realisation in a simpler hybrid model.

5.1 Popular Techniques for MPC

Motivating the model of computation provided by $\mathcal{F}_{\mathbf{MPC}}$ is done by considering possibilities for realisations. The main focus here will be on solutions based on secret sharing; indeed many forms of MPC are based directly on such schemes. However, in order to demonstrate that this is not the only possibility, the section concludes with an alternative.

$\mathcal{F}_{\mathbf{MPC}}$ described in the previous chapter is essentially (the functionality for) a secret sharing scheme with some extra features allowing computation. Such schemes allows a party D – known as the *dealer* – to *share* a value, v, among a number of other parties. Each party is given some information – a bit-string representing that party's *share* – and based on this information, v can be *reconstructed*; if sufficiently many agree (where "sufficiently" may depend on the scheme and/or the application), the parties pool their shares and compute v. Until reconstruction occurs, *no information* on v is obtainable by any set of "insufficiently few" parties. Thus, multi-party computation can be seen as an extension to secret sharing. For that reason, terminology from secret sharing

will be used throughout in the discussion below.

A secret sharing scheme is *linear* if given multiple sharings of values over some ring $\mathbb{R}$, a sharing of any linear combination over that ring can be constructed by the parties, such that a party's shares of the latter, depend only on its shares of the former. Based on any such scheme, most of $\mathcal{F}_{\mathbf{MPC}}$ may be realised. Input is simply sharing, output reconstruction, and addition and multiplication by a constant follows from the linearity of the scheme. Multiplication of two shared values is not immediate, thus in order to perform this, a protocol must be constructed. As demonstrated by Cramer *et al.* [CDM00], this can be done for any linear secret sharing scheme – high-level MPC protocols are often considered based on arbitrary schemes with a multiplication protocol, i.e. the underlying details are stripped away. When specific schemes are considered, improved protocols may often be constructed, e.g. with regard to complexity.

The protocols constructed in parts II and III communicate solely through $\mathcal{F}_{\mathbf{MPC}}$. Thus, all the added tasks considered are statistically or even perfectly UC-realised in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model. Thus, primitives secure in the *information theoretic setting* (also known as the secure channels model) imply unconditional or perfect security of the protocols constructed below. Secure channels may be modelled in the UC-framework through sub-functionalities. Two specific functionalities are required, namely $\mathcal{F}_{\mathbf{SSC}}$ providing synchronous and perfectly secure communication, and $\mathcal{F}_{\mathbf{Broadcast}}$ allowing all parties to broadcast authenticated messages to all others. The former may be formulated explicitly similarly to the functionalities of [Can00b] for secure communication, $\mathcal{F}^{l}_{\mathbf{SMT}}$ (secure message transmission), and for synchronous communication, $\mathcal{F}_{\mathbf{Syn}}$. The desired functionality is essentially a merger of the two. (Synchronous) broadcasts may be constructed as a hub, letting the functionality gather (possibly empty) messages from all parties and then forwarding everything to everyone.

Shamir's secret sharing scheme [Sha79] combined with the protocols of Ben-Or *et al.* [BGW88] may be used to perfectly UC-realise $\mathcal{F}_{\mathbf{MPC}}$ in the $\mathcal{F}_{\mathbf{SSC}}$-hybrid model. This allows computation over any field $\mathbb{F}$ – the details of its selection are irrelevant here. Improved variations of those protocols are available due to Gennaro *et al.* [GRR98]. These protocols can be shown secure against active and adaptive adversaries corrupting at most $\tau = \lfloor \frac{n-1}{3} \rfloor$ out of the n parties. Thus, perfectly secure multi-party computation (as defined by $\mathcal{F}_{\mathbf{MPC}}$) is possible in the hybrid model representing the information theoretic scenario. The protocols ensure termination, even in the face of active adversaries, assuming that all inputs are initially shared (or that default values are taken when corrupt parties refuse to share). Alternatively, by considering the $\mathcal{F}_{\mathbf{Broadcast}}, \mathcal{F}_{\mathbf{SSC}}$-hybrid model statistical security against both active and adaptive adversaries corrupting up to $\tau = \lfloor \frac{n-1}{2} \rfloor$ can be ensured, e.g. through the protocols of Rabin and Ben-Or [RBO89].

An alternative, motivating example for $\mathcal{F}_{\mathbf{MPC}}$ comes based on any *semantically secure homomorphic encryption* scheme. One possibility for constructing secure computation arises, when a threshold variation of such a scheme exists – i.e. where the private key may be "shared" among the parties, implying that decryption can only occur when sufficiently many take part in the decryption process. Sharing is then equivalent to broadcasting an encryption

of the desired value, while reconstruction is decryption. When the homomorphism allows addition of encrypted values, computing linear combinations of the "shared" values are immediate. Moreover, based on any such scheme a protocol for multiplication is provided by Damgård *et al.* [CDN01]. One example of such a scheme is Damgård and Jurik's extension of Paillier's crypto-system [Pai99, DJ01]. Ensuring UC-security against adaptive adversaries is not immediate, however, the difficulties may be overcome as demonstrated in the construction of the arithmetic black-box of Damgård and Nielsen [DN03]. This allows secure computation modulo a power of an RSA-modulus representing a public key, i.e. $\mathbb{R} = \mathbb{Z}_m$, where m is a power of the product of two odd primes. Naturally, using public-key cryptography reduces the security of the protocols to computational only.

5.2 Beyond the Assumption of Secure Channels

Based on the protocols noted above $\mathcal{F}_{\mathbf{MPC}}$ may be statistically or even perfectly UC-realised in the $\mathcal{F}_{\mathbf{Broadcast}}, \mathcal{F}_{\mathbf{SSC}}$-hybrid model. This provides security in the information theoretic setting, however, an authenticated broadcast channel as well as unconditionally (or perfectly) secure communication channels between all pairs of parties are *strong* assumptions. It provides a useful model, though such primitives are rarely – if ever – present in realistic settings. Thus, these sub-functionalities should be realised using cryptographic techniques.

Nearly all work on multi-party computation assume authenticated communication between the parties; this approach is also taken here, i.e. an ideal functionality for authenticated communication $\mathcal{F}_{\mathbf{AUTH}}$ is assumed. Based solely on this, any UC-secure encryption scheme may be used to provide UC-security for protocols constructed in the information theoretic setting. Further, even when a broadcast channel does not exist, this too may be realised using only point-to-point communication.

Beginning with the latter, broadcasts in the true sense of the word are difficult to perform in practice. The desired effect is that one party provides a distinct value that all others will agree on. However, broadcasts are typically not provided by the underlying communication layer; indeed, the model of protocols in the UC-framework provides the adversary with full control over all inter-party communication. Thus, for UC-security $\mathcal{F}_{\mathbf{Broadcast}}$ must be realised in a setting with access to only $\mathcal{F}_{\mathbf{AUTH}}$.

Passive adversaries are not a problem: All parties follow the protocol to the letter, and can be specified to send the desired value to all others using the authenticated channels. With regard to active adversaries, however, it must be ensured that the same value is sent to all, i.e. that all (honest) parties agree on the value received. Such a verification can be implemented at the cost of more interaction using protocols for solving *Byzantine agreement*. Essentially all pairs of parties ensure that they have the same value, such protocols may be argued UC-secure. Though such protocols may provide perfect security, it is only possible when at most $\tau \leq \lfloor \frac{n-1}{3} \rfloor$ parties may be corrupted. See [Nie03] for a full discussion.

Secure communication channels may be viewed as setup assumptions, however, it cannot be UC-realised by an arbitrary encryption scheme. UC-secure communication is not provided by all schemes, the difficulty in obtaining UC-security lies in the cryptographic setting. The simulator must provide the environment with the messages – encryptions – sent by all parties, even though it has no information on what these should contain.

Static adversaries may be handled using any semantically secure public-key crypto-system. When asked to provide an encryption of an unknown value, an encryption of a random value is provided instead. No computationally bounded environment can distinguish this from an encryption of what should have been sent. The drawback is that every time a message is to be sent, a new key-pair must be generated. This may be avoided by using a scheme secure against *chosen cipher text attacks* instead.

Adaptive corruptions cannot be handled though, as e.g. subsequent corruption of the receiver would result in the adversary (and therefore the environment) learning both the value and the private key, which may be used to distinguish. This difficulty can be overcome by employing a *non-committing encryption scheme* [CFGN96]. Both solutions reduce the security guarantees to computationally bounded adversaries only.

An alternative solution avoids secret communication altogether. *Pseudo-random secret sharing* (PRSS) [CDI05] allows sharing random values without communication between the parties after a (one-time) setup phase has been completed. This may then be used to remove the need for secrecy. For technical reasons security is reduced to consider static adversaries only.

The main idea consists of distributing a number of (cryptographically secure) *pseudo-random number generators* (PRNG) among the parties. Based on these, the parties may construct a sharing of a pseudo-random value unknown to all. Essentially all complements of unqualified sets of parties are assigned a generator, which provides randomness hiding the outcome.

This technique can then be used to allow parties to share values using broadcasts. In addition to distributing the PRNG's as described above, all are given to the dealer D as well; this implies that D will know the generated value. When D then wishes to share a value v, it broadcasts the difference between v and a freshly generated random value. The parties may then add this value to the random one to obtain a sharing of v. This eliminates the need for secret communication, essentially the random value is a one-time-pad masking v. Note that in order to allow *all* parties to take the role of dealer, PRNG's must be distributed in this manner for *every* party.

Chapter 6

Using the $\mathcal{F}_{\mathrm{MPC}}$-Hybrid Model

Most of the model of computation has already been specified through the description of $\mathcal{F}_{\mathbf{MPC}}$. However, the ideal functionality has been constructed in broad terms for adaptability, and while this allows a large number of possible realisations, it also makes the model vague. An often used model of secure multi-party computation consists of assuming a linear secret sharing scheme along with a multiplication protocol. This loose setting – motivated by the fact that much MPC bases itself on exactly this – provides the foundation of much work. By specifying details of $\mathcal{F}_{\mathbf{MPC}}$ – e.g. providing an improved description of allowed parallelisation – this may be used to construct what is essentially the same model; in other words, the loose setting may be formalised through $\mathcal{F}_{\mathbf{MPC}}$.

This chapter considers the final details of the model used in future chapters based on the general functionality for secure computation $\mathcal{F}_{\mathbf{MPC}}$. In addition to this, notation easing the description of secure computation is introduced. Rather than formulating secure computation as protocols, an algorithmic approach is taken. This is preferable as it is more in line with the overall goal – executing a protocol securely performing some computation.

The chapter is divided into three sections. Initially in section 6.1 details are specified for the chapters to come; this includes assumptions made in order to simplify the work. Section 6.2 considers protocol construction in the hybrid model. The chapter concludes with a discussion of complexity of the protocols constructed in section 6.3.

6.1 Specifying the Model of Computation

$\mathcal{F}_{\mathbf{MPC}}$ provides the backbone of the model, and few details need to be considered. However, the ideal functionality was constructed in order to maximise adaptability, thus allowing a multitude of realisations, whereas the protocols considered in future chapters consider specific purposes. Therefore, certain details of the general model will be specified, this includes removing "inconsequential" adversarial powers. $\mathcal{F}_{\mathbf{MPC}}$ was constructed with as strong adversarial powers as possible to allow as many realisations as possible – as well as simple adaption of the functionality e.g. for alternative notions of security. Though

constructing protocols in a more general model is preferable – e.g. with regard to a more powerful adversary – if this does not provide *interesting* consequences, the main effect will simply be more tedious protocol construction.

As in the description of $\mathcal{F}_{\mathbf{MPC}}$ the adversary considered will be both active and adaptive and in addition to this, the parties will be non-erasure, i.e. all previous values will be seen. However, these choices are of little consequence; all interaction between parties will occur through $\mathcal{F}_{\mathbf{MPC}}$, thus the adversarial behaviour possible is extremely limited. Concerning the corruption structure $\mathcal{C}$, the exact details will remain unspecified. Only a single, *reasonable* assumption is made, namely that at least one party remains honest.

Regarding $\mathcal{F}_{\mathbf{MPC}}$, one adversarial power is removed, namely the possibility of disallowing commands. Many protocols for multi-party computation – including the ones of [GRR98] – provide guarantees of termination. Moreover, the possibility of non-termination is of little consequence. The protocol in question may abort leaving some or all of the parties without a final result, but no more. While this is undesirable, it does not violate correctness or privacy of the protocol in question (assuming that the protocol is not rerun on altered inputs), thus for simplicity the possibility of cancellation of commands – except for inputs by corrupt parties – is not considered. Termination is either provided by the primitives or not salvageable by the applications considered, the problem lies outside the scope of this work. Therefore, the only adversarial influence considered lies in the inputs of corrupt parties which may be altered or cancelled.

A large motivating factor of this work is to be able to solve concrete, real-world problems. However, most real-world computational problems are not formulated over some arbitrary, finite ring $\mathbb{R}$; they occur over the integers (which may represent fractions with a denominator of limited size). Arithmetic on non-negative integers may be simulated using $\mathcal{F}_{\mathbf{MPC}}$ by performing computation modulo some value $m \in \mathbb{N}$, i.e. in the ring $\mathbb{Z}_m$. As long as the final result is positive and less than m (i.e. is not reduced modulo m), the computation performed is equivalent to its integer counterpart.

For the main motivating example of Shamir's secret sharing scheme and the protocols of Gennaro *et al.*, m must define a field – choosing m prime ensures this. The alternative motivation of MPC based on Paillier encryption – e.g. as used by the arithmetic black-box of Damgård and Nielsen – implies that m is a power of an RSA-modulus. In order to distinguish the two, $\mathbb{Z}_m$ will only be used to refer to the latter. The main focus of this thesis will be computation in the prime field $\mathbb{Z}_q$. The bit-length of q will be written ℓ, i.e. $\ell = \lceil \log(q) \rceil$ – logarithms are implicitly base two unless explicitly not so. The exact nature of initialisation phase will not be considered. It may be assumed that initially all parties send the prime q to $\mathcal{F}_{\mathbf{MPC}}$. (For Paillier encryption, the modulus is chosen by the ideal functionality based on the security parameter. The modulus is provided to the parties, while the simulator is given its factorisation.)

Computing on integers of a bounded size is not out of the ordinary, indeed this is typically the case. At the lowest level computers work with values of some specified bit-length and reductions – *overflow* from here on – may occur and may cause problems. Dealing with potential overflow is much more difficult

when considering multi-party computation though. Only arithmetic over $\mathbb{Z}_m$ is provided by $\mathcal{F}_{\mathbf{MPC}}$, thus detection is not immediately possible. Based on protocols of the subsequent chapters solutions may be constructed, however, the computation required make these intractable for most if not all applications.

Rather than attempting to deal with this problem through protocol construction, the issues is sidestepped. When considering the execution of some application, it is reasonable to assume that both the size of the inputs and the required arithmetic computation will be known. Based on these, an upper bound on the possible values of the computation may be found; choosing m larger ensures that overflow does not occur. Note that this is stricter than the above as intermediate results must not overflow. This is required if any "non-arithmetic" computation is considered – this includes many of the primitives introduced in part II; as an example, consider comparison: $5 + 2 = 0$ is clearly not true over the integers, however, if it was simulated using $\mathbb{Z}_7$ then it is.

It is stressed that avoiding overflow refers to applications only. Though the motivation for the primitives constructed in part II are integer operations, properties of the underlying ring – including using overflow – are used to construct these efficiently.

6.2 Notation for Protocol Construction

Formal construction of multi-party computation protocols in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model consists of describing interactive Turing machines preparing messages containing commands and interacting with the ideal functionality. However, protocol construction will be formulated in an algorithmic manner with focus on *computation* rather than *protocol execution*. This allows the prime focus to be placed on secure computation and intuition behind the protocols constructed; moreover, computation translates readily to commands of $\mathcal{F}_{\mathbf{MPC}}$ implying that a formal ITM may be obtained easily.

6.2.1 Variables

In order to discuss the algorithmic specification of protocols, the notion of variables must be considered. Though all $\mathcal{F}_{\mathbf{MPC}}$ does is essentially to provide variables, these are not sufficient for describing the behaviour of the parties. The variables of $\mathcal{F}_{\mathbf{MPC}}$ allow information to be stored *secretly*, however, variables for values known by the parties are also required, e.g. for counters as well as publicly known values, e.g. output by the functionality. Thus, two types of variables will be considered, *public* and *secret*. The former are variables of the parties initiating the computation through $\mathcal{F}_{\mathbf{MPC}}$, while the latter are variables of the ideal functionality. Naturally, their existence is echoed in the parties, i.e. the parties maintain a list of defined variables, though they do not know the values assigned to them.

In order to distinguish between public and secret variables, differing notation for the two is provided. An often used notation for secret sharing schemes consists of placing square brackets around a value, thereby signifying that it is shared. The same approach is taken here regarding variables of $\mathcal{F}_{\mathbf{MPC}}$, as

shared values may be seen as the means to realise the functionality. Writing $[x]_{\mathbb{R}}$ is used to denote a variable of $\mathcal{F}_{\mathbf{MPC}}$ containing a (secret) value of the ring $\mathbb{R}$. Public variables on the other hand will be written *without* square brackets and association to the underlying ring, x. When constructing commands based on public variables, the value represented is viewed as a constant, $\mathcal{F}_{\mathbf{MPC}}$ is oblivious to even the existence of public variables.

The notation explicitly states the ring over which computation occurs. The subscript simply denotes that it is a variable over $\mathbb{R}$ and may be used in arithmetic operations. This is done in order to allow secret variables to have different characteristics. In the protocols below, computation on secret, binary values will be required. *Bits* or *binary values* will be used to refer to values that are either the additive or multiplicative identity, i.e. values of $\{0, 1\} \subseteq \mathbb{R}$. A variable is *bit-wise stored*, written $[x]_B$, if the individual bits of the value are stored in secret variables. I.e. it is shorthand for $[x_{k-1}]_{\mathbb{R}}, \ldots, [x_0]_{\mathbb{R}}$, for some k where the $[x_i]_{\mathbb{R}}$ contain the binary representation of the k-bit value of $[x]_B$. Note that the notation removes any reference to $\mathbb{R}$, it is essentially a bit-string encoded using elements of $\mathbb{R}$ used to represent a value. This value may be (and is often) an element of the ring, this is stressed by the notation: It is written in human-readable form with the most significant bit first. The same "variable name" may be used for multiple variables – when done the intention is to stress a relationship between them. The typical example will be $[x]_{\mathbb{R}}$ and $[x]_B$, where the latter represents the binary encoding of the former.

In order to keep notation uncluttered it is simplified and thereby slightly abused. Secret variables over $\mathbb{Z}_m$ will be written $[x]_m$ rather than $[x]_{\mathbb{Z}_m}$, indirectly specifying the ring. Further, the explicit reference to the ring may be omitted – writing $[x]$ – when it is either unspecified or implicit from context. Finally, though $[x]_B$ does not refer to one variable of $\mathcal{F}_{\mathbf{MPC}}$ but many, reference to the value is still written $\mathrm{val}([x]_B)$. There are two possible interpretations of this, both of which will be used below – the desired one is implied by the context. It will either be used to refer to a tuple of bits,

$$\mathrm{val}([x]_B) = \big(\mathrm{val}([x_{k-1}]_m), \ldots, \mathrm{val}([x_0]_m)\big),$$

or to refer to the value those bits represent

$$\mathrm{val}([x]_B) = \sum_{i=0}^{k} 2^i [x_i]_m.$$

This computation is performed over the integers to avoid ambiguity, though it may be considered a value of $\mathbb{Z}_m$ if $\mathrm{val}([x]_B) < m$.

6.2.2 Computation

The goal of performing any computation is to end up with the final result stored in one or more variables. These may be public or secret depending on the application; the latter implies that the former is obtainable – the desired value may simply be output. As specified above, secure computation using $\mathcal{F}_{\mathbf{MPC}}$ will be described in an algorithmic fashion. This may then be translated to a formal

description of an ITM interacting with the ideal functionality as described in chapter 4.

Using an algorithmic approach to describe multi-party computation allows cleaner descriptions of the solutions to the tasks at hand. Bundling arithmetic operations in more complex expressions ensures less clutter and allows intuition to be reflected by the descriptions. The only drawback of the approach is a more involved complexity analysis, as at this point the actual commands and messages must be considered. Using pseudo-code also allows the control-flow (as well as local computation) of the parties taking part to be interleaved with the secure computation performed. Loops allow simple formulation of repetitive behaviour while branching may specify adaptive behaviour, e.g. computation performed may depend on computed values.

Branching and loops should be mostly self explanatory. `If`-statements allow branching, while loops repeatedly iterate until some condition is satisfied. However, one important point must be noted. *Control-flow cannot be based on secret variables.* The values of such variables are stored within $\mathcal{F}_{\mathbf{MPC}}$ and as such are not known to the parties, thus clearly they cannot be used to determine protocol execution.

All other operations can be handled as assignments, both public computation as well as computation representing commands for $\mathcal{F}_{\mathbf{MPC}}$. Public computation is ordinary pseudo-code, thus only computation of $\mathcal{F}_{\mathbf{MPC}}$ is explored in detail. The functionality provides commands for input, output, and binary arithmetic operators. The notation allows the two latter to be joined in arbitrary expression, while the former remains a distinct operation. The notation of inputs does not lend itself to being inserted in complex expressions, however, this is minor as these are rarely used.

Beginning with input, for party P_i, variable $[x]$, and value $v \in \mathbb{R}$

$$P_i : \ [x] \leftarrow v$$

will be used to denote that P_i inputs a value. For P_i this translates to the command (**input** $P_i : \ [x] \leftarrow v$), while all other parties send (**input** $P_i : \ [x] \leftarrow ?$) to $\mathcal{F}_{\mathbf{MPC}}$.

The output command (**output** $[y]$) gives rise to a public value, thus for x and $[y]$,

$$x \leftarrow \texttt{output}([y])$$

sets x to $\mathrm{val}([y])$. Rather than simply outputting values of secret variables, arbitrary expressions may be output – the intended meaning is to perform the computation and then output the result. Note that using the output command is the *only* means for public variables to be set based on secret ones.

The remaining statements of protocol construction consists of assignments based on arbitrary computation. Presently this consists of *arbitrary* arithmetic expressions as well as renaming (joining multiple variables to a bit-wise storing), however, additional expressions will be considered in chapters to come. Arithmetic expressions will be written using an infix notation for readability; multiplications will be denoted using '$\cdot$', while '$+$' will represent addition, e.g.

as in the assignment

$$[x] \leftarrow [x_1] \cdot [x_2] \cdot [x_3] + v \cdot [x_4]$$

where v is a value. In addition to this sums and products over multiple terms may be specified using $\sum$ and $\prod$, e.g. as

$$\sum_{i=1}^{k} (\alpha_i \cdot [x_i]).$$

All such expressions are easily translated to a number of commands of $\mathcal{F}_{\mathbf{MPC}}$.

6.2.3 Additional Primitives – Syntactic Sugar

The above provides the required notation for describing arithmetic computation using $\mathcal{F}_{\mathbf{MPC}}$. However, in order to ease protocol construction and improve readability, additional simple operations will be considered and implemented. This may be seen as syntactic sugar, which is translated first to arithmetic and then to commands. As an initial example, subtraction is added in the natural way. $[x] - [y]$ is simply shorthand for $[x] + (-1) \cdot [y]$, where (-1) is the additive inverse of the multiplicative identity.

Though not immediate, some computation requires control-flow to be based on secret variables. To achieve this at the minimal level the following solution is introduced. Based on a secret bit $[b]$ – i.e. with $\mathrm{val}([b]) \in \{0, 1\}$ – conditional selection, written

$$[b] \ ? \ [a] : [a'],$$

can be constructed. This describes the selection of either $[a]$ (if $[b]$ is 1) or $[a']$ (if $[b]$ is 0) and can be implemented using arithmetic as

$$[b] \cdot ([a] - [a']) + [a'].$$

Finally, Boolean computation on secret bits will be required throughout. Simple binary, Boolean expressions may be implemented based on field arithmetic as follows: Encoding `true` as 1 and `false` as 0, $\wedge$ is simply the product of the two, $\vee$ is the sum minus the product, and $\neg$ is 1 minus the value. The final operation considered is exclusive or, $\oplus$, which may be implemented as the difference squared. However, computing the $\oplus$ of multiple bits can be done by converting them to a ± 1 encoding – with -1 denoting `true` and 1 denoting `false`. The product is then the desired result which may be converted back to a 0/1 encoding. Boolean expressions ranging over multiple terms will be expressed compactly similarly to the $\sum$-notation of sums.

6.3 Complexity in the $\mathcal{F}_{\mathbf{MPC}}$-Hybrid Model

When constructing protocols complexity must be considered, and the primary concern here will be communication. Though keeping the amount of computation low is desirable, this is secondary as communication is considered a

more costly resource. For this work it will suffice that the computation required is feasible, i.e. that the ITM's representing the parties are probabilistic polynomial-time.

When considering communications, there are two distinct measures, both of which will be considered. *Communication complexity* which refers to the overall amount of communication – i.e. the number of bits sent between the parties – and *round complexity* which counts the overall number of times messages of arbitrary size are transmitted (between all parties). The former is self explanatory, however, the latter is often viewed as the more important of the two, with much work – including the present – concerning itself with constructing protocols with low (preferably constant) round complexity. The motivation for this focus is that the overhead associated with data transmission is high and independent of the amount sent. Thus keeping the number of messages low is of high priority.

As the protocols of future chapters will be constructed in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model, complexity cannot be considered directly. The messages transmitted between the parties and $\mathcal{F}_{\mathbf{MPC}}$ consider all operations equal, however, this is simply an abstraction; $\mathcal{F}_{\mathbf{MPC}}$ provides a framework, and the complexity of the primitive operations may vary. Thus, though complexity must be considered with regard to the commands of $\mathcal{F}_{\mathbf{MPC}}$ as these represent protocol execution, it must also take the underlying primitives into account. Regarding rounds, the functionality was created explicitly with such a concept to allow complexity analysis. However, again the underlying primitives must be considered as the notion does not suffice yet. When $\mathcal{F}_{\mathbf{MPC}}$ was introduced, the specified parallelism explicitly disallowed parallelisation of commands for simplicity. A better notion must be introduced in order to provide proper complexity analyses.

The model of complexity considered in this work bases itself on linear primitives; it is typically formulated as an assumption of a linear secret sharing scheme with a multiplication protocol. The candidates for realising $\mathcal{F}_{\mathbf{MPC}}$ noted in chapter 5 provide the motivation for this. Thus, though other realisations of the ideal functionality may be possible, complexity will be considered with regard to linear primitives. The exact formalisation of complexity is contained in the following sections.

6.3.1 Communication Complexity

In the model of complexity obtained by assuming that the primitives realising $\mathcal{F}_{\mathbf{MPC}}$ are linear, different commands have different complexities. Note first that only input, output, and binary multiplication of secret variables must be considered. All linear (secure) computation requires only local computation, and is therefore costless in the present model where only communication is considered – these refer to sharing, reconstruction, and the assumed multiplication protocol, when using the terminology of secret sharing.

Naturally, all these primitives could be counted, however, rather than doing so, focus is put on multiplications of two secret variables. For most realisations, the multiplication protocol will be the most complex of the three; it is typically constructed based on the others. Moreover, it is typically also the one which

finds most use. The two remaining commands only supply input and extract output; as the goal is multi-party computation it is reasonable to assume that some computation requiring multiplications occur between the two. Thus, multiplication of two secret variables is considered the basic unit of communication complexity. The measure will be denoted *the number of secure multiplications*, however, the word "secure" – implying that two secret variables are multiplied – will often be left implicit. It is stressed that even when this is done, multiplications of linear combinations are not included in the above count as they do not require interaction. Regarding inputs and outputs, an input from all parties will be considered equivalent to a multiplication. Outputs on the other hand will be disregarded.

6.3.2 Round Complexity

$\mathcal{F}_{\mathbf{MPC}}$ already provides a measure of round complexity. However, rounds of $\mathcal{F}_{\mathbf{MPC}}$ must be related to messages of the underlying protocols, i.e. a notion of parallelism must be considered. Sequential command execution allowed formulation of $\mathcal{F}_{\mathbf{MPC}}$, however, it does not provide a realistic model of parallelisation of protocols. Linear primitives are considered, and it will be assumed – which is the case for all candidates considered – that they may be parallelised arbitrarily, *as long as any required result from a preceding command is available.*

Recall that in each round all parties provide a series of commands (in bulk) to $\mathcal{F}_{\mathbf{MPC}}$ which are then executed, updating variables and outputting associated values. Each round is initiated by the environment providing inputs (commands) to all parties. These are forwarded to $\mathcal{F}_{\mathbf{MPC}}$, who in turn sends any data specified by the individual commands to the adversary $\mathcal{A}$. The adversary reacts, corrupting parties and altering commands as allowed by $\mathcal{F}_{\mathbf{MPC}}$ as if the commands were individually executed. Finally, all parties receive their output, which is passed on back to the environment.

For simplicity it will be assumed that input, output, and multiplication of two secret variables require exactly one round each. Thus, as these are assumed to parallelise arbitrarily, an unbounded number may be performed with the sole restrictions: Variables set by an ($\mathbf{input}\ P_i :\ x \leftarrow ?$) message or a ($\mathbf{let}\ x \leftarrow x_1 \cdot x_2$) message are not available for use in subsequent multiplication or output commands until the beginning of the following round. Naturally, as outputs are not received by the parties until the end of the round, the values contained cannot be used in any commands of the same round. Linear combinations of secret variables require no interaction, and are therefore costless as was the case with communication complexity above – essentially this means that an arbitrary number may be performed at the beginning and end of each round.

Though rounds of $\mathcal{F}_{\mathbf{MPC}}$ provides a measure, it is stressed that rounds within the functionality are *not* directly equivalent to rounds of communication. A round in the functionality describes what may be performed concurrently, but says nothing on the actual messages sent in an execution of a realising protocol. When analysing protocols in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model, interactions with the ideal functionality are counted, and each of these may be replaced

by some number of concrete messages when a realising protocol is considered. Naturally, constant round primitives are required if any protocols constructed are to be constant rounds themselves. This is the case for all the candidates for realisation considered, thus under big-$\mathcal{O}$ round complexity will be equivalent.

Round complexity will be further divided into two categories, *preprocessing* and *online computation*. Many desirable operations cannot be phrased as simple arithmetic, thus (secret) random values are employed to boost efficiency and sidestep difficulties. Naturally, these random values are independent of the values (and even the variables) used in the secure computation. Thus, their generation (and any computation depending solely on them) may be performed in advance. This computation is referred to as preprocessing, while online computation denotes the computation which depends on the actual inputs.

Overall round complexity may be improved by considering preprocessing. As this is independent of the actual inputs of the computation, all of it may be parallelised. This is of consequence when protocols are constructed in an iterative fashion, repeatedly joining simpler primitives to solutions for more complex tasks. As an example, consider multiple, sequential invocations of some complex task. A further motivation for preprocessing is that it is beneficial in a setting where there is idle-time, perhaps combined with a requirement for swift online reaction time. This may be the case if computation is to occur at some specified time, e.g. after a deadline for submitting data. The computation itself cannot be performed until all data is present, however, any *preprocessing* can.

Part II

Complex Primitives

Chapter 7

Extending the Basic $\mathcal{F}_{\mathrm{MPC}}$

$\mathcal{F}_{\mathbf{MPC}}$ provides a basic model for the construction of multi-party computation protocols, however, it is not sufficient. Though arithmetic in a ring or field may be used to simulate integer computation, most interesting applications need more than simply addition, subtraction, and multiplication. Securely and efficiently determining the greater of two values is required, as are other primitives. Thus, this part focuses on the construction of efficient, *constant-rounds* solutions for several such tasks.

Throughout the following chapters, the words "secure computation" and "protocol" will be used interchangeably. A secure computation is essentially a protocol, even though it is constructed in a hybrid model with no (direct) interaction between the parties. Commands of the functionality represent concrete messages exchanged.

All of the primitives considered will be introduced as extensions of $\mathcal{F}_{\mathbf{MPC}}$, realised in the original hybrid model. They will be implemented with regard to the underlying field or ring. In this part, computation is performed in the prime field $\mathbb{Z}_q$ for some odd prime q unless specified otherwise. However, most constructs generalise to computation modulo RSA-moduli and powers of such, i.e. to Paillier based MPC.

New commands are introduced and realised one by one, thereby adding more and more power to the model of computation, one step at a time. The final outcome is essentially a model of secure computation, which readily allows applications. And, just as important: The model is well-founded, as demonstrated through its realisation in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model.

The computation required in order to realise an added command can generally not be performed in a single round of $\mathcal{F}_{\mathbf{MPC}}$. Thus, differently from the initial commands, the ones introduced may be multi-round – the result is *not* ready for use at the beginning of the round following the one in which the commands was provided, but at some later point. Therefore, when specifying the intended behaviour of such a command, the number of rounds required before a result is ready must be presented. This is secondary and heavily tied to the realising computation, thus an alternative realisation would have the same effect but most likely a different round complexity. It may therefore be left implicit, though this is generally not done.

It is stressed that though the required number of rounds must be specified

in the description of the command this need not be constant. Naturally, it may not depend on the actual inputs – this would leak information – however, it may depend on publicly known values, e.g. q. Though it is possible to specify non-constant-rounds computation, the overall focus is on providing constant-rounds primitives. Exchanging messages carries an overhead – ensuring that this is independent of the computation performed is a key goal.

Though $\mathcal{F}_{\mathbf{MPC}}$ is extended through added command, these are not considered directly in subsequent protocol construction. Rather, the algorithmic notation introduced in chapter 6 will be extended along with $\mathcal{F}_{\mathbf{MPC}}$. For "simple" (binary) operators, an infix notation will (typically) be used, as this eases readability, whereas more general computation, not readily described in such a manner, will be represented by function calls. Both are easily translated to commands, moreover the notation used in those commands will mirror that of the algorithmic notation.

7.1 Randomness and Masking

$\mathcal{F}_{\mathbf{MPC}}$ provides only arithmetic over some finite field or ring. As arithmetic is not practical for the tasks considered, additional standard techniques must be utilised to overcome its shortcomings. *Masking* secret variables of $\mathcal{F}_{\mathbf{MPC}}$ – generating a uniformly random value unknown to all and outputting another based on this and the variable of $\mathcal{F}_{\mathbf{MPC}}$ – will be used repeatedly. If the value output is uniformly random (or at least indistinguishable from that), then no information is leaked. However, it still *depends* on the variable in ways related to the random value generated.

Generating a uniformly random group element (of the additive or multiplicative group) and adding (multiplying) that and the variable in question results in a uniformly random value – at least when the value of the variable was in that group originally. Seeing $r + \mathrm{val}([a]_q)$ reveals nothing about $\mathrm{val}([a])$. An alternative method is to assume that the value of $[a]$ is of a bounded size – e.g. k bits. If the modulus q defining the field is sufficiently large, $\log(q)) \gg k + \kappa$ for security parameter κ, then letting r be a uniformly random $(k + \kappa)$-bit value, $r + \mathrm{val}([a])$ is statistically indistinguishable from a uniformly random $(k+\kappa)$-bit value – i.e. information is leaked with probability less than $2^{-\kappa}$.

7.2 On Extending $\mathcal{F}_{\mathbf{MPC}}$

Though extending $\mathcal{F}_{\mathbf{MPC}}$ provides an excellent means for describing the creation of a more powerful basis of secure computation, it is not formally correct. Commands are not added to $\mathcal{F}_{\mathbf{MPC}}$, instead a whole new ideal functionality is introduced. This is equivalent to the original except for the new command and realised in the hybrid model of that. However all common commands are trivially realisable, thus only the one added will be considered. Therefore, it will generally be said that *the command* is realised in the simpler hybrid model, though this is technically incorrect. What is meant is that *the ideal functionality containing that command* is realised.

In order to argue UC-security, a simulator must be constructed. However, this is not done explicitly in order to keep things simple. Secure computation having the same effect as the desired command is introduced. No messages are exchanged between the parties, thus only the messages received from the ideal functionality (which specify the command in question) must be constructed and given to the environment at the appropriate time – doing this is trivial. In addition to this, inputs and outputs of $\mathcal{F}_{\mathbf{MPC}}$ must be considered. For the former it must be argued that incorrect inputs from corrupt parties do not affect the computation. Regarding the latter, the values output will be either leaked by the command (in which case the simulator is notified of what to choose) or it will be a masked value, and generating a uniformly random value (over the appropriate set) suffices.

As simulation is essentially free through the description of the computation, realisation will be argued through correctness and privacy (and round complexity). It will be demonstrated that the computation provides the desired result, and that no outputs leak information – i.e. that values are masked as described above, such that uniformly random values may be used instead. Round complexity is simply required in order to demonstrate that the correct number of rounds is used, i.e. it is a small technical requirement.

There are a few remaining issues with regard to realisation, all related to the secret variables of $\mathcal{F}_{\mathbf{MPC}}$. First off, *inputs* will be used to refer to the input-variables of a computation, while *output* refers to the variable assigned the answer. Though the words used are the same, these should *not* be confused with inputs by parties or output of values by $\mathcal{F}_{\mathbf{MPC}}$. It is clear from the context which of the two meanings apply, the former refers to input/output of a *sub-computation* (which may be viewed as a function call) while the latter refers to input/output of the system.

The second issue is related to input variables for a computation – i.e. inputs in the former sense. Some commands introduced have restrictions on acceptable inputs, as the computation realising it has so. This is modelled implicitly by letting the ideal functionality provide the simulator with all input-values when one or more are illegal – in this case no security guarantees are provided, and this approach allows easy simulation.

The final issue occurs as multi-round commands are considered. It must be ensured that inputs are *only* updated as described by the computation in question, i.e. that they cannot be manipulated through malicious behaviour in mid-computation. In order to do so formally, input variables must be copied to temporary variables and computation should be viewed as taking place in a sandbox. This implies that the parties must "filter out" commands provided by the environment, which attempt to access any of the variables of the sandbox. The issue is noted, but for simplicity ignored. It is merely book-keeping, thus computation will simply be viewed as "standalone," i.e. no other computation occurs simultaneously.

7.3 An Overview of Part II

Part II is divided into chapters, each relating to a particular primitive. Each chapter is essentially based on one article (or idea), however, common sub-primitives have been extracted and placed separately in chapter 8. There a more powerful ideal functionality $\mathcal{F}_{\mathbf{MPC-R}}$ is introduced – $\mathcal{F}_{\mathbf{MPC}}$ extended with secret, random element generation. In addition to the construction of the ideal functionality, that chapter also considers efficient constructs for several often used tasks.

The remaining chapters focus on the construction of complex primitives. Equality testing is the topic of chapter 9, while inequality is treated in chapter 10. Part II concludes with a powerful primitive, namely that of bit-decomposition – converting values stored in ordinary variables of $\mathcal{F}_{\mathbf{MPC}}$ to bit-wise stored ones. Originally, this provided the first constant rounds solutions to both equality and inequality testing. Though these have later been improved, the result is still important, bridging the gap between arithmetic and binary circuits. In addition to the main contributions, the chapters contain minor, direct applications of the results.

Chapter 8

Basic Primitives

This chapter introduces a number of primitives in the form of sub-protocols for the subsequent chapters. Roughly speaking, they may be split into two categories: Secret random value generation and non-trivial constructs for improving efficiency. Basing itself on random value generation, the latter provides constant round solutions to several useful primitives needed below.

Most of the protocols presented are well-known and included mainly for completeness and to allow detailed complexity analyses below. They are present with regard to computation in a prime-field $\mathbb{Z}_q$, however, the protocols may be realised when computing within the ring $\mathbb{Z}_m$ where m is (a power of) an RSA-modulus – though sometimes at a greater cost.

All protocols presented provide security against active adversaries. Typically this follows immediately, as only computation using $\mathcal{F}_{\mathbf{MPC}}$ occurs, thus the statement is left implicit. The only adversarial influence allowed is the alteration of inputs of corrupt parties. When this occurs, security against active adversaries is argued explicitly.

The structure of the chapter follows the division of the protocols. Initially, section 8.1 considers an extension of $\mathcal{F}_{\mathbf{MPC}}$ which provides commands for the generation of uniformly random values unknown to all parties. The hybrid model constructed there is then used in section 8.2 as the basis for constructing constant round solutions for powerful primitives. The chapter concludes with a discussion on translating the protocols to computation over $\mathbb{Z}_m$ in section 8.3.

8.1 Random Value Generation

The protocols of the coming chapters, require the ability to generate random values unknown to all parties and storing these in variables of $\mathcal{F}_{\mathbf{MPC}}$. Thus an extended ideal functionality $\mathcal{F}_{\mathbf{MPC-R}}$ is introduced and realised in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model. It is equivalent to the original except that commands resulting in the generation and storing of uniformly random values in the functionality are provided as well – apart from these $\mathcal{F}_{\mathbf{MPC-R}}$ is trivially realised through $\mathcal{F}_{\mathbf{MPC}}$. Each new command specifies a distinct subset of $\mathbb{Z}_q$ – for prime q – from which a random value must be selected. Most protocols may be extended to rings $\mathbb{Z}_m$, where m is (a power of) an RSA-modulus, this is discussed further

in section 8.3 below.

The commands introduced are written as "function-calls," the intuition being that $\mathcal{F}_{\mathbf{MPC-R}}$ is instructed to invoke a *local* function generating the value. This also translates well to the algorithmic view of protocol construction, with functions written directly within expressions. These are readily translated back to commands of $\mathcal{F}_{\mathbf{MPC-R}}$.

All commands introduced here follow the same structure: For function `Rand`() the associated command will be written

$$(\textbf{let } [x]_q \leftarrow \texttt{Rand}()).$$

Upon receiving this from all honest parties in round R, the functionality generates a fresh random value v from the appropriate subset and notifies the adversary that $[x]_q \leftarrow \texttt{Rand}()$ is run. Letting r denote the number of rounds required to execute the command, $\mathcal{F}_{\mathbf{MPC-R}}$ sets $\mathrm{val}([x]_q)$ to v in round $R+(r-1)$ and notifies the adversary and all parties that the command has terminated and $[x]_q$ is ready for use.

One final point to note before introducing the desired commands is that most of the protocols realising them have non-zero probabilities of failure. Thus, they cannot perfectly UC-realise the commands, since failure trivially implies a real-world execution. Though the possibility of failure will be observed during the presentations, it will be disregarded when considering security. The full discussion of failures is postponed to section 8.1.5, until then detection suffices.

8.1.1 (let $[r]_q \leftarrow \texttt{Rand}()$)

`Rand`() – used as an example above – denotes the selection of a uniformly random value over all of $\mathbb{Z}_q$. This requires a single round, i.e. the termination-message is returned in the same round as the command is given.

Consider the following protocol in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model: Each party, P_i selects a uniformly random value, $v_i \in \mathbb{Z}_q$. All these values are then input; for $i \in \{1, 2, \ldots, n\}$ the parties execute

$$P_i: \ [r_i]_q \leftarrow v_i.$$

If any corrupt party refuses to input a value, then the input of that party will be assumed to be set to some dummy-value, e.g. 0. Following this, $[r]_q$ is computed as:

$$[r]_q \leftarrow \sum_{i=1}^{n} [r_i]_q$$

and the parties output a termination-message equivalent to the one received from the functionality.

As at least one party is honest, at least one v_i is uniformly random, thereby implying that the value assigned to $[r]_q$, $\sum_{i=1}^{n} v_i$, is uniformly random – thus the value is distributed correctly. Further, neither the adversary nor any of the

parties know the value assigned to $[r]_q$ as it depends on inputs from *all* parties.[1] For complexity, each party inputs a single value in the same round, which was assumed equivalent to one multiplication. Therefore, as the final linear combination is costless, generation of a random value is considered equivalent to one multiplication in a single round. Note that in difference to most of the protocols considered here, the protocol actually perfectly UC-realises the extension in the original $\mathcal{F}_{\mathbf{MPC}}$-hybrid model.

Formal simulation of the protocol is easily obtained: The simulator must construct commands for the protocol exactly as the adversary in question would do, i.e. providing the values to be input by corrupt parties (or no command if this is refused) as well as commands allowing honest parties to input values (if desired). Further, the replies from the functionality for these commands are also generated; this consists of the entire view of the corrupt parties and is indistinguishable from a real view. Subsequent corruptions are handled in the same manner except that the commands are generated as specified by the protocol.

Note that if $\mathcal{F}_{\mathbf{MPC}}$ is realised using pseudo-random secret sharing, this may be utilised here as an alternative, immediate realisation of $(\mathbf{let}\ [r]_q \leftarrow \texttt{Rand}())$. The main advantage is the fact that once setup has been completed, PRSS does not require interaction, thus like linear combinations it is costless. This improves the complexity of all protocols using the command – and any other random generation, as all protocols for generating random values below utilise this.

8.1.2 $(\mathbf{let}\ [r]_q \leftarrow \texttt{Rand}^*())$

The second set considered is $\mathbb{Z}_q^*$, i.e. the generation of non-zero values as q is prime. This is specified using the command

$$(\mathbf{let}\ [r]_q \leftarrow \texttt{Rand}^*()),$$

which requires 3 rounds in order to execute. This is realised in the original $\mathcal{F}_{\mathbf{MPC}}$-hybrid model with access to the $(\mathbf{let}\ [r']_q \leftarrow \texttt{Rand}())$ above.

The protocol realising this extension consists of generating of two random values of $\mathbb{Z}_q$, the product of which is output. If this is 0, the protocol fails, while if not, the former of the two random values is assigned to $[r]_q$. Correctness is easily verified: A non-zero product implies that both factors are non-zero, thus the value assigned to $[r]_q$ is uniformly random and non-zero, i.e. uniformly random over $\mathbb{Z}_q^*$. Moreover, the protocol is successful exactly when both of the random values are non-zero, thus privacy – that the value generated is unknown to all parties and the adversary – is ensured. The second random factor masks the value assigned to $[r]_q$ implying that the value output is independent of it.

The protocol consists of the generation of two random values – which may be done in parallel – after which a multiplication and an output occurs. Thus, the

[1] The sole exception occurs when only one party is honest, and none of the corrupt parties provide input. However, though the honest party will have added information, this concerns only the random value. Moreover, the situation may trivially be detected, ensuring that privacy will not be violated.

protocol requires three rounds, and is equivalent to three multiplications (and one output). As the overall result is equivalent to that of the ideal behaviour, the protocol perfectly UC-realises the extension except with regard to failure – formal simulation simply consists of constructing commands as above as well as simulating the output, i.e. generating a uniformly random, non-zero value. All subsequent simulations are more or less trivial variations of this, thus they will be left out from here on.

Note that the multiplicative inverse of $[r]_q$ may be computed securely for free following this protocol. After revealing the masked value, the inverse may be computed by multiplying the second random value by the inverse of the product output. It is assumed that variable $[r^{-1}]_q$ is implicitly assigned this value, this will be used below.

8.1.3 (**let** $[r]_q \leftarrow$ `RandSq`())

The command (**let** $r \leftarrow$ `RandSq`()) will be used to instruct $\mathcal{F}_{\mathbf{MPC-R}}$ to generate a uniformly random (non-zero) square of $\mathbb{Z}_q$, i.e. a non-zero quadratic residue modulo q. This is done in 4 rounds of $\mathcal{F}_{\mathbf{MPC-R}}$ extended with the above commands.

This functionality is easily realised: Generating a uniformly random element of $\mathbb{Z}_q^*$ and squaring it clearly provides the desired result. Moreover, privacy is immediate as no values are output. Thus, as failure occurs only if (**let** $[r']_q \leftarrow$ `Rand`*()) fails, the functionality is perfectly UC-realised, though not in the original $\mathcal{F}_{\mathbf{MPC}}$-hybrid model. The complexity is that of `Rand`*() plus the multiplication needed for squaring the random value, i.e. four rounds containing four multiplications.

While the above protocol is presented for clarity, it is noted that if `RandSq`() is constructed from scratch a round can be saved. The squaring required may be performed in parallel with the multiplication of the `Rand`*() protocol. Thus, it will be assumed that the protocol terminates in round $R + 2$ rather than $R + 3$, i.e. that only three rounds are required until the variable in question is set.

8.1.4 (**let** $[b]_q \leftarrow$ `RandBit`())

The final subset considered is $\{0, 1\} \subset \mathbb{Z}_q$ – i.e. random bits. The command (**let** $b \leftarrow$ `RandBit`()) instructs $\mathcal{F}_{\mathbf{MPC-R}}$ to generate a random bit and store it in the variable $[b]_q$. This requires three rounds.

A protocol realising this was considered in [DFK+06]. $[r]_q \leftarrow$ `Rand`() is used to generate a uniformly random value which is then squared and output. If this is zero, the protocol fails, however, if not then its unique root $r' \in \{1, 2, \ldots, (q-1)/2\}$ is computed locally by all parties. $[b]_q$ is then set through the following secure computation

$$[b]_q \leftarrow 2^{-1} \cdot ([r]_q \cdot (r'^{-1}) + 1).$$

The value assigned to $[r]_q$ must be either r' or $-r'$, thus multiplying $[r]_q$ by the inverse of r' results in ± 1, which is then mapped to $\{0, 1\}$. Seeing the

square of $[r]_q$ does not disclose *which* of the two roots it contains. Moreover, the two possibilities occur with probability 1/2 each, thus the final bit is uniformly random and unknown to all parties. Complexity is easily considered: one random value is generated, squared and output, and this is followed by a linear combination mapping it to $\{0, 1\}$. Thus, the complexity is equivalent to two multiplications in three rounds. Disregarding the possibility of failure – occurring when the output value is 0, i.e. when $\mathrm{val}([r]_q) = 0$ – this perfectly UC-realises (**let** $[b]_q \leftarrow \texttt{RandBit}()$).

8.1.5 Avoiding Aborts

Several of the protocols described above for generating random values may abort without result. As noted this implies that perfect UC-realisation is not possible, indeed unless the probability of aborting is negligible, the protocols do not UC-realise the ideal functionality; the environment may distinguish between the protocol and the ideal functionality based on failure alone.

However, no privacy of inputs is compromised by aborts. Failure only affects the generation of a random value and is detected before the values are used further. Thus, even though $\mathcal{F}_{\mathbf{MPC-R}}$ may not even be UC-realised in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model, *perfect privacy* of all inputs is still ensured. Further, there are techniques for reducing the probability of aborting and several possibilities for dealing with them if they do occur. Thus, it is always possible to statistically UC-realise $\mathcal{F}_{\mathbf{MPC-R}}$ in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model though at an increased cost. The possibilities for dealing with failure are explored first, for this reason assume that $\mathbb{Z}_q$ is of large characteristic, i.e. that $q > 2^\kappa$ for security parameter κ. This implies that failures occur with negligible probability. The removal of this assumption is discussed below.

In the event of a failure, multiple avenues of action are possible – the relevant choice may depend on the overall application. The protocol in the hybrid model may abort. This provides a perfectly secure protocol, which may terminate without result with negligible probability.[2] Alternatively, as all probabilities of success are large, rerunning the random generation protocol upon failure results in a value in an expected constant number of attempts. In this case, the complexity argued of the overall protocol (under big-$\mathcal{O}$) will be *expected* rather than *guaranteed*. A third alternative consists of providing a guess at the final result. This provides imperfect correctness, with incorrect results occurring with negligible probability. Finally, it is possible to simply *output* all inputs of all parties. Obviously this leaks information, however, it does so with negligible probability, thus unconditional rather than perfect security is provided.

If q is small – i.e. when its bit-length is not linear in the security parameter – the probability of aborting is non-negligible. In this case, the probability of failure may be reduced by executing multiple, concurrent instances of the protocol in question; any successful one may be used as the "real" execution. When κ instances are run, the probability that all fail is negligible in κ. Even better, when *multiple* random values are required – which will typically be the

[2]Note that this is *not* due to any action on the part of the adversary. It is simply a non-zero chance of overall failure.

case – a Chernoff bound may be used to argue that – except with probability negligible in κ – more than κ successful invocations can be ensured with only a (small) constant factor overhead in the form of additional attempts.

8.2 Basic Constructs for Round Efficiency

This section considers protocols for specific, computational tasks in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model. The tasks – which may already be performed using the existing notation – are useful in many of the complex constructs considered later on. However, the complexity of the immediate, naïve solutions is not as efficient as desired. Thus, alternative means providing the same overall results are therefore considered; round complexity is reduced at the cost of performing additional multiplications of secret variables, though only a constant factor more.

8.2.1 Unbounded Fan-in Multiplication

The first construct concerns itself with unbounded fan-in multiplication of non-zero values. Naturally, computing $\prod_{i=1}^{k} [a_i]_q$ may already be performed, however, $\lceil \log(k) \rceil$ rounds are required for the "naïve" solution. However, the full product may also be computed in a constant number of rounds using the technique of Bar-Ilan and Beaver [BB89] seen as protocol 1.

Protocol 1 The Bar-Ilan and Beaver multiplication protocol

Input: Secret variables $[a_1]_q, \ldots, [a_k]_q$ with $\mathrm{val}([a_i]_q) \in \mathbb{Z}_q^*$ for all $i \in \{1, 2, \ldots, k\}$.

Output: $[p]_q$, such that $\mathrm{val}([p]_q)$ is set to $\prod_{i=1}^{k} \mathrm{val}([a_i]_q)$.

```
    for i = 1,...,k do
        [r_i]_q ← Rand*()
    end for
    m_1 ← output([a_1]_q · [r_1]_q)
5:  for i = 2,...,k do
        m_i ← output([r_{i-1}^{-1}]_q · [a_i]_q · [r_i]_q)
    end for
    [p]_q ← (∏_{i=1}^{k} m_i) · [r_k^{-1}]_q
```

The intuition behind the protocol is that the random $[r_i]_q$ mask the non-zero $[a_i]_q$[3] and the inverse of the previous mask, thus ensuring privacy as seeing the m_i reveals no information on the $[a_i]_q$. Correctness is then verified as the $[r_i]_q$ and $[r_i^{-1}]_q$ cancel out:

$$[p]_q = \Big(\prod_{i=1}^{k} m_i\Big) \cdot [r_k^{-1}]_q = \Big(\prod_{i=1}^{k} ([r_{i-1}^{-1}]_q \cdot [a_i]_q \cdot [r_i]_q)\Big) \cdot [r_k^{-1}]_q = \prod_{i=1}^{k} [a_i]_q.$$

However, it is with regard to complexity that the protocol is interesting. The masked product – i.e. the product of the m_i – may be computed *locally* by all

[3] If any $\mathrm{val}([a_i]_q) = 0$ then this fact is leaked.

parties as these are public values. The masked result is then securely unmasked resulting in a *secret* variable containing the desired product.

It is noted that the iterations of each loop are independent of the others, i.e. they may be performed in parallel. The random element generation requires $3k$ multiplications in three rounds, while the computation of the m_i require $2k-1$ multiplications in another three rounds (noting that m_1 may be computed in parallel with the other m_i). For simplicity, the overall requirement will be considered $5k$ secure multiplications in six rounds. Viewing the random element generation and multiplication of $[r_i]_q$ with the inverse of their predecessor as preprocessing (noting that the latter may be done as $\mathbb{Z}_q^*$ is commutative), the computation requires only k multiplications in two rounds online.

A simple observation allows the computation of prefix-products, $\prod_{i=1}^{i_0} [a_i]_q$ for any $i_0 \in \{1, 2, \ldots, k\}$, for free in the current model of complexity once the full product has been computed. The key observation is that it may be computed exactly as $[p]_q$:

$$[p_{(1,i_0)}]_q \leftarrow (\prod_{i=1}^{i_0} m_i) \cdot [r_{i_0}^{-1}]_q.$$

Indeed, the original product is simply a special case.

8.2.2 Symmetric Boolean Functions

Computing on binary values is advantageous at times, thus an efficient means for evaluating symmetric Boolean functions is introduced. Assume that variables $[b_1]_q, \ldots, [b_k]_q$ are set to binary values and that $k < q-1$. Moreover, a symmetric Boolean function $f : \{0,1\}^k \mapsto \{0,1\}$ is to be evaluated, taking the $[b_i]_q$ as inputs. The output of a symmetric Boolean function depends only on the number of 1's in the input, not on their location. Thus, as the bits are actually elements of $\mathbb{Z}_q$ the function may be written as

$$f([x_1]_q, \ldots, [x_k]_q) = \phi(1 + \sum_{i=1}^{k} [x_i]_q)$$

for some function ϕ mapping the number of 1's (plus one for technical reasons) to the desired result. By Lagrange interpolation, a polynomial over $\mathbb{Z}_q$ with coefficients $\alpha_0, \ldots, \alpha_k$ may be constructed such that $\phi(x) = \sum_{i=0}^{k} \alpha_i x^i$ for $x \in \{1, 2, \ldots, k+1\}$.

The above observation allows the following efficient secure computation. First $[a]_q \leftarrow 1 + \sum_{i=1}^{k} [b_i]_q$ is computed in order to store the relevant number of 1's as $[a]_q$. Following this, all powers of $[a]_q$ from 1 to k are computed, these are denoted $[a^1]_q, \ldots, [a^k]_q$. This may be done in a constant number of rounds using the Bar-Ilan and Beaver multiplication protocol above as $\mathrm{val}([a]_q) \in \{1, 2, \ldots, k+1\}$ is non-zero, by setting all k factors to $[a]_q$. The $[a^i]_q$ may be computed as a prefix-product of the $[a]_q$ for $i \in \{1, 2, \ldots, k\}$. Finally,

based on all k powers, $\phi([a]_q)$ may be evaluated as

$$\sum_{i=0}^{k} \alpha_i [a^i]_q.$$

By the discussion above and the definition of ϕ and the α_i, the desired result is computed. Moreover, privacy is ensured as all computation is performed over secret variables, and none of these are output except in sub-computation already argued secure. Complexity is equivalent to that of the Bar-Ilan and Beaver multiplication protocol – computing $[a]_q$ and evaluating ϕ are both linear. Thus, the final complexity is $5k$ multiplications in six rounds. When preprocessing is applicable only k multiplications in two rounds are required online, similar to the binary values will be written $\bigwedge_{i=1}^{k} [b_i]_q$ and $\bigvee_{i=1}^{k} [b_i]_q$ when performed over multiple (secret) variables. The associated computations may be performed in a constant number of rounds using $\mathcal{O}(k)$ multiplications, as both are obviously symmetric Boolean functions – `AND` is 1 exactly if all input bits are 1, while `OR` is 0 only if all inputs are 0.

8.2.3 Prefix-`OR`

The final extension considered is the computation of prefix-`OR` of a bit-vector $[a]_B = ([a_1]_q, \ldots, [a_k]_q)$. I.e. computing $[b]_B = ([b_1]_q, \ldots, [b_k]_q)$ such that

$$\mathrm{val}([b_i]_q) = \bigvee_{j=1}^{i} \mathrm{val}([a_i]_q).$$

Due to its complexity – and the fact that no "short" notation currently exists – a command (**let** $[b]_B \leftarrow \mathtt{pre}_\vee([a]_B)$) is introduced for this. In the algorithmic notation it is written as $[b]_B \leftarrow \mathtt{pre}_\vee([a]_B)$. The extension of the ideal functionality is defined as expected: Upon receiving (**let** $[b]_B \leftarrow \mathtt{pre}_\vee([a]_B)$) from all honest parties in round R, $\mathcal{F}_{\mathbf{MPC-R}}$ computes the result and notifies the adversary that the command has been given. In round $R + 11$ the $[b_i]_q$ are set and the parties and adversary notified that the execution has terminated and the $[b_i]_q$ are ready.

The command may be realised in $\mathcal{F}_{\mathbf{MPC-R}}$ using the method by Chandra, Fortune and Lipton [CFL83a] as described by Damgård *et al.* in [DFK+06] on which this section is based. For notational convenience, assume that $k = \lambda^2$ for an integer λ. $[a]_B$ will be split into λ blocks of λ bits each. Due to this the bits are renamed, $[a_{(i-1)\lambda+j}]_q$ is rewritten as $[a_{i,j}]_q$ for $i, j \in \{1, 2, \ldots, \lambda\}$. Thus,

$$[a]_B = ([a_{1,1}]_q, [a_{1,2}]_q, \ldots, [a_{1,\lambda}]_q, [a_{2,1}]_q, \ldots, [a_{2,\lambda}]_q, \ldots, [a_{\lambda,\lambda}]_q),$$

and for $i \in \{1, 2, \ldots, \lambda\}$ $[a_{i,1}]_q, \ldots, [a_{i,\lambda}]_q$ is called a block of $[a]_B$. The desired output will be split in blocks using the same notation.

The full computation is seen as protocol 2. Privacy is immediate as all computation occurs within $\mathcal{F}_{\mathbf{MPC-R}}$ while all outputs occur in sub-computation already argued secure above. As for the correctness, the variables have the following interpretation. $\mathrm{val}([x_i]_q) = 1$ iff the i'th block contains a 1. Therefore,

Protocol 2 Securely computing prefix-`OR`

Input: $[a]_B = ([a_{1,1}]_q, \ldots, [a_{\lambda,\lambda}]_q)$.

Output: $[b]_B = ([b_{1,1}]_q, \ldots, [b_{\lambda,\lambda}]_q)$ such that $\mathrm{val}([b_{i,j}]_q)$ is the logical `OR` of the $(i-1)\lambda + j$ first $[a_{i,j}]_q$.

```
for i = 1,...,λ do
    [x_i]_q ← ⋁_{j=1}^{λ} [a_{i,j}]_q
end for
for i = 1,...,λ do
    [y_i]_q ← ⋁_{j=1}^{i} [x_j]_q
end for
[f_1]_q ← [y_1]_q
for i = 2,...,λ do
    [f_i]_q ← [y_i]_q − [y_{i−1}]_q
end for
for j = 1,...,λ do
    [c_j]_q ← Σ_{i=1}^{λ} [f_i]_q · [a_{i,j}]_q
end for
for j = 1,...,λ do
    [d_j]_q ← ⋁_{i=1}^{j} [c_i]_q
end for
for i = 1,...,λ do
    for j = 1,...,λ do
        [b_{i,j}]_q ← [y_i]_q − [f_i]_q + [f_i]_q · [d_j]_q
    end for
end for
[b]_B ← ([b_{1,1}]_q, ..., [b_{λ,λ}]_q)
```

$\mathrm{val}([y_i]_q) = 1$ iff there is a 1 in one of the i first blocks, and $[f_i]_q$ is 1 iff the i'th block is the first block to contain a 1. Hence the sequence of values of the $[f_i]_q$ has form $(0, ..., 0, 1, 0, ..., 0)$. Letting i_0 be the position of the single 1-bit, for $i < i_0$, the i'th block of the output should be all 0's. For $i > i_0$, the i'th block of the output should be all 1's. Finally, the i_0'th block of the output should the prefix-`OR` of the i_0'th input block.

The $[c_j]_q$ contain a copy of the i_0'th input block – it is easily verified that $\mathrm{val}([c_j]_q) = \mathrm{val}([a_{i_0,j}]_q)$. Thus the $[d_j]_q$ contain the prefix-`OR` of the i_0'th input block to be inserted as the i_0'th block of the output. Everything is joined in the final `For`-loops, $[y_i]_q - [f_i]_q$ is 1 when the initial 1 occurred in a *previous* block, while $[f_i]_q \cdot [d_j]_q$ inserts the prefix-`OR` of the i_0'th input block as the i_0'th output block.

Concerning complexity, computing the $[x_i]_q$ requires λ `OR`'s of size λ. In addition to this, two naïve prefix-`OR`'s are required for the $[y_i]_q$ and $[d_j]_q$ along with $2\lambda^2$ multiplications for the $[c_j]_q$ and $[b_{i,j}]_q$. A naïve prefix-`OR` of size λ may be realised in constant rounds using

$$\sum_{i=1}^{\lambda} p(i),$$

where $p(i)$ denotes the number of multiplications needed for a i-ary fan-in `OR`. As $p(i) = 5i$, this implies that the overall number of multiplications is

$$\lambda \cdot (5 \cdot \lambda) + 2 \cdot (\sum_{i=1}^{\lambda} 5 \cdot i) + 2\lambda^2 = 12\lambda^2 + 5\lambda.$$

For simplicity in the analyses below, the complexity is "rounded off" to $12\lambda^2 + \lambda^2 = 13k$. For $\lambda \geq 5$ clearly this is greater.

Regarding rounds, all bodies of loops parallelise, however, the loops themselves must be executed in sequence – the output of one is needed as input for the next. Things are not all bad though, as all preprocessing needed for the unbounded fan-in `OR`'s may be performed in parallel initially. Thus, two of the three batches of `OR`'s may be consider to be online only implying a round complexity of $6 + 2 \cdot 2 + 2 = 12$ rounds – stripping away the preprocessing this becomes eight rounds online.

8.3 Extending to $\mathbb{Z}_m$

Most of the protocols described above translate immediately to computation over $\mathbb{Z}_m$ where $m = (pq)^a$ is a power of an RSA-modulus with prime factors p and q – i.e. the motivating example of multi-party computation based on generalised Paillier encryption. By the motivation, it is assumed that the bit-length of p and q is linear in κ and that the adversary is computationally bounded.

Concerning the generation of uniformly random elements, `Rand`() may be implemented exactly as for $\mathbb{Z}_q$. This is also the case for `Rand`*() – "non-zero" is simply replaced with "invertible." As the bit-length of the prime factors of m are linear in κ, the probability of generating elements of $\mathbb{Z}_m \setminus \mathbb{Z}_m^*$ is negligible. Finally, squares of $\mathbb{Z}_m^*$ are simply obtained through squaring as above.

Generating bits poses a problem however, as every square of $\mathbb{Z}_m^*$ has *four* roots rather than two. Thus an alternative strategy must be employed; the following is a simple but less efficient solution. All n parties, $P_1, \ldots, P_n$ input a uniformly random bit encoded as an element of $\{\pm 1\}$. The product of these is also uniformly random in $\{\pm 1\}$ and may be mapped to $\{0, 1\}$. Naturally it must be verified that the inputs are indeed in $\{\pm 1\}$, this is done by squaring, outputting, and verifying that this is 1. The remaining two roots cannot be chosen by malicious parties, as knowledge of these implies a factorisation of m, i.e. breakage of the encryption scheme. This solution requires $\mathcal{O}(n)$ multiplications.

Regarding round complexity, this may remain constant as the multiplication protocol of Bar-Ilan and Beaver translates to unbounded fan-in multiplication over $\mathbb{Z}_m^*$. When only invertible elements are multiplied – which both "bits" are – and the masks are chosen at random from $\mathbb{Z}_m^*$ privacy is ensured.

As constant round multiplication extends to $\mathbb{Z}_m$, so does the constant rounds evaluation of symmetric Boolean functions. The sum of the binary inputs is guaranteed to be invertible when the number of inputs are less than the least factor of m. This is clearly the case as the bit-length of the latter depends

on κ. As symmetric Boolean functions are available, this in turn implies that prefix-`OR` may be computed in a constant number of rounds. None of the three final tasks require random bits, thus their complexities are equivalent to their $\mathbb{Z}_q$ counterparts, i.e. a number of multiplications linear in the number of inputs.

Chapter 9

Equality

The previous chapter introduced $\mathcal{F}_{\mathbf{MPC-R}}$ extending $\mathcal{F}_{\mathbf{MPC}}$ with commands for random generation. Based on these, alternative means for performing often used tasks more efficiently were introduced. This chapter uses these added primitives to construct the first "real" extension: Securely testing equality of two secret variables. Constructing the solution in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model, a command ($\mathbf{let}\ [b]_q \leftarrow [a]_q \stackrel{?}{=} [a']_q$) is added instructing the ideal functionality to compare defined variables $[a]_q$ and $[a']_q$. If the values of the two are equal, the secret variable $[b]_q$ is set to 1, otherwise it is set to 0. Naturally, neither the adversary nor any of the parties should obtain any information on $[a]_q$, $[a']_q$, or $[b]_q$. This chapter is based on an unpublished article by the author [Tofa], which presents a simple and efficient protocol solving the problem.

The problem of securely determining equality in multi-party computation is not a new one. Observing first that it suffices to test whether the difference of $[a]_q$ and $[a']_q$ is 0, the problem is equivalent to securely determining if a single secret variable equals 0. If the result is to be output, the solution is simple. A uniformly random mask from $\mathbb{Z}_q^*$ may be generated using ($\mathbf{let}\ [m]_q \leftarrow \texttt{Rand}^*()$), multiplied onto the value, and this product output. If the input is 0 so is the output, otherwise this will be uniformly random over $\mathbb{Z}_q^*$.

However, when the result is to be stored in a *secret* variable of $\mathcal{F}_{\mathbf{MPC-R}}$, the task is not as easy. An efficient, constant round solution was until recently an open problem. The task was a primitive in the constant round protocols for unconditionally secure linear algebra due to Cramer and Damgård [CD01], however, a solution was only provided for fields of low characteristic. For fields of high characteristic, access to the individual bits of the inputs were assumed. Through constant round bit-decomposition of secret shared values – considered in chapter 11 – Damgård *et al.* provided the first efficient constant round protocol solving the problem in [DFK+06]. The result allowed an equality test by bridging the gap between arithmetic and binary circuits. For $\mathbb{Z}_q$, where q is an ℓ-bit prime, the resulting equality test, though constant round, required $\mathcal{O}(\ell \log(\ell))$ secure multiplications.

Nishide and Ohta provided a protocol sidestepping bit-decomposition in [NO07]. The present author independently noted a similar construction as well, though at a later date. In [NO07] Nishide and Ohta also presented an equality

test similar to the one here, however, their discovery was made subsequently to the one by the author – an early version containing the core ideas was presented in a talk at Stevens Institute of Technology on Feb. 14th. 2006. The two results are similar, though the solutions exhibit slightly differing properties.

The protocol presented provides perfect security, but only imperfect correctness; with probability negligible in the security parameter κ differing inputs may be reported equal. In difference to all previous solutions, however, the number of secure multiplications required depends solely on κ and not on the underlying field. Moreover, by sacrificing perfect security (leaking information with negligible probability), perfect correctness can be ensured.

The remainder of this chapter is organised as follows. Section 9.1 describes the desired extension to $\mathcal{F}_{\mathbf{MPC-R}}$ formally, after which section 9.2 introduces secure computation with the desired properties realising the extension in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. This section contains the bulk of the work of this chapter. Section 9.3 concludes with final comments, variations of the protocol, and some immediate applications.

9.1 The Desired Extension to $\mathcal{F}_{\mathbf{MPC-R}}$

$\mathcal{F}_{\mathbf{MPC-R}}$ is easily extended with the possibility of doing equality tests. The desired functionality is equivalent to the original except that an additional command may be provided by the parties. Upon receiving message

$$(\mathbf{let}\ [b]_q \leftarrow [a]_q \stackrel{?}{=} [a']_q)$$

from all honest parties in round R, $\mathcal{F}_{\mathbf{MPC-R}}$ tests whether $[a]_q$ and $[a']_q$ are equal; the adversary is then notified that the command is being executed. In round $R+7$, $[b]_q$ is finally set and all parties as well as the adversary are notified of this. Comparing a variable and a publicly known value is immediate from this – a specific command for this is not considered.

As noted above, when the result of an equality test is to be output, this may be realised by simply masking non-zero inputs through a multiplication. As the effect is obtainable using the above extension plus an output command, this is not explored further, however, for efficiency, an explicit command:

$$(\mathbf{output}\ [a]_q \stackrel{?}{=} [a']_q)$$

could be introduced, providing better complexity, both regarding rounds and the number of multiplications required. The formal description is immediate from the secret test above and the output command. Focusing on equality with 0, generating a uniformly random non-zero value, multiplying it onto the input and outputting the product requires five rounds of $\mathcal{F}_{\mathbf{MPC-R}}$. It is noted that various minor improvements may be made when constructing the computation from scratch; such fine-tuning is left out as the gain is minimal.

9.2 Realising the Extension

When realising the equality command, the focus will be on providing equality with 0. A general equality test may be constructed from this by testing the difference of the two secret variables. The intuition behind the protocol realising the extension is extremely simple. Adding zero to a value with some property always results in a value with that property, however, this may not be the case when a non-zero value is added. The idea is explained further below. Based on this observation, a test is introduced which correctly determines non-zero values with probability essentially 1/2. This test is repeated in order to reduce the probability of failure to an acceptable level; combining the results of κ executions reduces the probability to a negligible amount.

In section 9.2.1 an algebraic prerequisite of the test is noted. Following this, section 9.2.2 introduces a sub-protocol. This is then used as (the main part of) the equality test with high probability of failure; section 9.2.3 expands and explains the brief intuition given above for the full protocol.

9.2.1 An Algebraic Prerequisite

A single algebraic prerequisite is required, namely the following lemma due to Perron [Per52]. It considers distribution of quadratic residues and non-quadratic residues (non-residues for short) of specific subsets of $\mathbb{Z}_q$.

Lemma 9.1 *For prime $q = 4m + 1$, with quadratic residues $r_1, \ldots, r_{2m+1}$ (including 0 as a residue) and for any non-residue $a \in \mathbb{Z}_q$ there will be exactly m quadratic residues and $m + 1$ non-residues in the set*

$$\{r_1 + a, \ldots, r_{2m+1} + a\} \subset \mathbb{Z}_q.$$

For prime $q = 4m - 1$ with quadratic residues $r_1, \ldots, r_{2m}$ (including 0 as a residue) and for any non-zero $a \in \mathbb{Z}_q$ there will be exactly m quadratic residues and m non-residues (possibly including 0) in the set

$$\{r_1 + a, \ldots, r_{2m} + a\} \subset \mathbb{Z}_q.$$

9.2.2 Testing Quadraticity

Before constructing the protocol for testing equality of values in $\mathbb{Z}_q$, a protocol for testing quadraticity of secret variables is considered. Quadratic residues have previously been considered in the context of MPC, e.g. by Feige *et al.* [FKN94]. Given a non-zero $[a]_q$, a computation resulting in a secret bit stating whether or not $[a]_q$ contains a square is desired. Protocol 3 does exactly this both securely and efficiently; where ω used in step 3 refers to an arbitrary, public non-square. Keeping things formal, a command (**let** $[s]_q \leftarrow \mathtt{sq}?([a]_q)$) is introduced. Upon receiving it from all honest parties in round R, $\mathcal{F}_{\mathbf{MPC-R}}$ notifies the adversary immediately – moreover, if $\mathrm{val}([a]_q) = 0$ the adversary is informed of this as well. In round $R + 5$ $[s]_q$ is set to 1 if $[a]_q$ was square and 0 otherwise; all parties are then notified that the command has been executed.

Protocol 3 `sq?()` – determining quadraticity securely

Input: $[a]_q$ with $\mathrm{val}([a]_q) \in \mathbb{Z}_q^*$.
Output: Binary $[s]_q$ such that $\mathrm{val}([s]_q) = 1$ iff $\mathrm{val}([a]_q)$ is a quadratic residue.

$[r]_q \leftarrow \texttt{RandSq}()$
$[\bar{s}_m]_q \leftarrow \texttt{RandBit}()$
$[m]_q \leftarrow [r]_q \cdot ([\bar{s}_m]_q \,?\, \omega : 1)$
$c \leftarrow \texttt{output}([a]_q \cdot [m]_q)$
5: $s_c \leftarrow \texttt{sq?}(c)$ ▷ Note that this is public computation
$[s]_q \leftarrow [\bar{s}_m]_q \oplus s_c$

Concerning correctness of the computation, note that $[m]_q$ is a square if and only if $\mathrm{val}([\bar{s}_m]_q) = 0$; $[\bar{s}_m]_q$ selects either a square (1) or a non-square (ω) which is multiplied onto a random square. Thus, if $\mathrm{val}([\bar{s}_m]_q) = 0$, then $\mathrm{val}([a]_q)$ is square exactly if c is. Conversely, if $\mathrm{val}([\bar{s}_m]_q) = 1$, then $[a]_q$ and c differ in their quadraticity. Computing the quadraticity of the public value c and flipping the result if $[\bar{s}_m]_q$ is 1 therefore yields the correct result.

As all computation occurs using $\mathcal{F}_{\mathbf{MPC-R}}$, the only possible information leak comes from seeing c output in line 4. However, $[m]_q$ is uniformly random over $\mathbb{Z}_q^*$ – if $[\bar{s}_m]_q$ is 0 it is a uniformly random quadratic residue by definition, while if $[\bar{s}_m]_q$ is 1 it is a random non-residue. Thus $[m]_q$ masks the non-zero $[a]_q$ perfectly implying that no information may be gathered from c.

Concerning the complexity of the protocol, note that steps 5 and 6 require no interaction being local computation on the publicly known c and a simple linear combination. The preceding steps consist of one invocation of `RandSq`() and one of `RandBit`(). Further, only two secure multiplications and a single output are required. The conditional selection – ordinarily equivalent to a multiplication – chooses between two *public* values implying that a costless linear combination suffices. As the random values may be generated concurrently, overall the computation requires eight secure multiplications in six rounds. The result is summarised in the following lemma:

Lemma 9.2 *$\mathcal{F}_{\mathbf{MPC-R}}$ may be extended with a command* (**let** $[s]_q \leftarrow \texttt{sq?}([a]_q)$) *for securely determining quadraticity of a non-zero variable $[a]_q$. This sets a secret binary variable to the answer, i.e. $[s]_q$ is set to 1 exactly when the* $\mathrm{val}([a]_q)$ *is a quadratic residue. The extended functionality may be realised in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model through the protocol* `sq?()`. *The computation may be performed using $\mathcal{O}(1)$ secure multiplications in $\mathcal{O}(1)$ rounds.*

9.2.3 Testing Equality

An equality test may now be constructed based on quadraticity testing. Though the full extension to $\mathcal{F}_{\mathbf{MPC-R}}$ considers equality of two secret variables, the focus here will be on equality with 0, the general case is immediately obtainable. The basic idea behind the protocol presented is that if $[a]_q$ is set to 0, then $\mathrm{val}([a]_q) + s$ is quadratic, for any square $s \in \mathbb{Z}_q^*$, however, if it is non-zero, by

lemma 9.1 this is only true for approximately half of the quadratic residues.[1] Thus, selecting a uniformly random s and testing if $[a]_q + s$ is square will identify non-zero values with probability essentially 1/2. This is repeated in order to obtain an arbitrarily small error probability, the overall result being the logical `AND` of the individual tests.

In order to ensure privacy, $[a]_q + s$ is required to be non-zero. Thus, for technical reasons, the details depend on whether q is congruent to 1 or 3 modulo 4, the differences are minor though. Protocol 4 describes the case of $q \equiv 3 \bmod 4$. The case of $q \equiv 1 \bmod 4$ is explained in the discussion of privacy below.

Protocol 4 Securely testing equality with 0 when $q \equiv 3 \bmod 4$

Input: $[a]_q$
Output: Binary $[b]_q$, such that – except with probability negligible in κ – $[b]_q$ is 1 exactly if $\mathrm{val}([a]_q) = 0$

```
    [a']_q ← [a]_q · [a]_q
    for i = 1,...,κ do
        [s_i]_q ← RandSq()
        [b_i]_q ← sq?([a']_q + [s_i]_q)
5:  end for
    [b]_q ← ⋀_{i=1}^{κ} [b_i]_q
```

Protocol 4 statistically UC-realises testing equality with 0 in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. Correctness is obtained as discussed above: $\mathrm{val}([a]_q) = 0$ implies that $[a']_q + [s_i]_q$ will be square, thus all $[b_i]_q$ – and therefore also $[b]_q$ – are set to 1. If, on the other hand, $\mathrm{val}([a]_q)$ is non-zero, then $[a']_q$ is a non-zero square. For $v \in \mathbb{Z}_q^*$, letting $S_v = \{v + s \mid s \in \mathbb{Z}_q^*\text{: square}\}$, testing $[a']_q + [s_i]_q$ for quadraticity in step 4 of the loop is therefore equivalent to testing a uniformly random element of $S_{\mathrm{val}([a']_q)}$. Thus, the overall protocol is equivalent to testing κ uniformly random samples of $S_{\mathrm{val}([a]_q)^2}$. By lemma 9.1 this set contains $\frac{q+1}{4} - 1$ quadratic residues and $\frac{q+1}{4}$ non-residues. The probability that one such $[a]_q{}^2 + [s_i]_q$ is a non-residue is

$$\frac{|\{s \mid s \in S_{\mathrm{val}([a]_q)^2}\text{: non-square}\}|}{|S_{\mathrm{val}([a]_q)^2}|} = \frac{\frac{p+1}{4}}{\frac{p-1}{2}} = \frac{m}{2m-1} > \frac{1}{2}.$$

As the $[s_i]_q$ are independent, $[b]_q$ is therefore set to 0 except with probability less than $2^{-\kappa}$, which is negligible in κ. Thus, except with probability negligible in the security parameter, the computation provides the correct result.

Privacy is easily seen to be preserved as no variables are output except in sub-protocols already argued secure. Thus, assuming that all sub-protocols are given proper inputs, the protocol releases no information. `sq?()` is given legal (i.e. non-zero) input as $p \equiv 3 \bmod 4$ implies that the additive inverse of a non-zero quadratic residue is a non-residue. Further, by their construction the $[b_i]_q$ are binary, implying that any κ-ary fan-in `AND` protocol is acceptable.

[1] Though not stated, the lemma also applies to quadratic a when $q \equiv 1 \mod 4$.

For prime $q \equiv 1 \bmod 4$ the tested expression must be altered to ensure privacy. Fixing some public non-residue, ω, step 4 is replaced by

$$[b_i]_q \leftarrow \texttt{sq}?(\omega \cdot [a']_q + [s_i]_q),$$

which ensures that only non-zero values are tested – for such q the additive inverse of a quadratic residue is also quadratic. Correctness is argued as above through the unused half of lemma 9.1.

The secure computation consists of a single secure multiplication for the initial squaring of the input plus the generation of κ unknown, random squares and κ invocations of $\texttt{sq}?(\cdot)$; this requires $4\kappa + 8\kappa = 12\kappa$ secure multiplications. As logical `AND` is a symmetric function, the κ-ary fan-in `AND`-gate may be evaluated as described in section 8.2.2 using 5κ multiplications. Overall, this implies $17\kappa + 1$ multiplications. Note that in order to employ the constant round solution for symmetric binary functions, it is required that $\kappa < q - 1$. Section 10.1 provides a more efficient alternative in this case.

Concerning round complexity, note that each iteration of the `For`-loop does not depend on any of the previous ones. They may therefore be executed in parallel, implying a constant round complexity overall. Taking a closer look – constructing the full computation in the most basic $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model– all preprocessing (e.g. random element generation) may be performed in parallel; this concerns not only the $[s_i]_q$ and the elements needed in the quadraticity tests, but also the preprocessing of the κ-ary `AND`. As the initial squaring of $[a]_q$ may be performed concurrently with this, round complexity is equivalent to that of $\texttt{sq}?()$ plus the two online rounds required for the `AND`-gate. The overall complexity is therefore extremely low: Only eight rounds of interaction with the ideal functionality are required. This is also the complexity of the general equality test comparing two variables, as computing the difference between the two is costless. From the above discussion the following theorem is obtained:

Theorem 9.1 (Equality) *The ideal functionality, $\mathcal{F}_{\mathbf{MPC-R}}$, may be extended with a command* (**let** $[b]_q \leftarrow [a]_q \stackrel{?}{=} [a']_q$) *for determining whether two secret variables are equal, storing the result in a third secret variable. The extension may be statistically UC-realised in the basic $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. The realisation provides the result in $\mathcal{O}(1)$ rounds using only $\mathcal{O}(\kappa)$ multiplications. It provides perfect privacy of the inputs and result, however, with negligible probability in the security parameter differing inputs may be considered equal. Thus only statistical security can be guaranteed within the UC-framework.*

9.3 Concluding Remarks

Having presented and realised an extension to $\mathcal{F}_{\mathbf{MPC-R}}$ providing an equality test, a few concluding remarks are made. As an algorithmic approach is utilised when describing computation rather than specifying explicit commands of $\mathcal{F}_{\mathbf{MPC-R}}$, notation must be introduced for comparison. Keeping notation similar to that of the command, expressions of the form

$$[a]_q \stackrel{?}{=} [a']_q$$

will be used to denote a comparison of $[a]_q$ and $[a']_q$. The result is a binary value, which may be inserted directly into larger expressions. The translation into actual commands of $\mathcal{F}_{\mathbf{MPC-R}}$ is immediate.

The remainder of the chapter consists of variations and applications of the equality test. Section 9.3.1 considers sacrificing perfect privacy for perfect correctness – regarding UC-security this is a pure gain. Following this, the ideas of the equality test are generalised in section 9.3.2. The computation is extended to rings $\mathbb{Z}_m$ encompassing the second motivating example, threshold Paillier encryption. The chapter is concluded with a collection of simple applications of the equality test in section 9.3.3. These are included in order to demonstrate uses of the test, but are not formalised as commands of $\mathcal{F}_{\mathbf{MPC-R}}$ as they are not required in the chapters to come.

9.3.1 Ensuring Perfect Correctness

Perfect correctness of the equality test is easily obtained through a small extension. The required task is simply to *publicly* verify that the correct result has been determined. Focusing on equality with zero, it can be noted that an incorrect result can occur only when the input is non-zero. Moreover, for such inputs the desired result *is* 0, thus either the input or the result should be 0. Success may therefore be ensured by verifying that the product of the two is 0 using a public-output equality test. Naturally, if an incorrect result has been determined this is leaked, implying that the input was non-zero. However, this occurs with negligible probability, thus this variation still statistically UC-realises the extension, though in a few more rounds; the additional verification of the result increases complexity marginally.

9.3.2 Extending to $\mathbb{Z}_m$

The equality test just described generalises to computation over the ring $\mathbb{Z}_m$ for applicable m. As the idea is essentially the same, the differences are only sketched, moreover for simplicity it will be assumed that m is (a power of) an RSA-modulus, $m = (pq)^a$ for primes $p, q \equiv 3 \mod 4$ and *odd* a. I.e. the result is applicable for MPC based on threshold Paillier encryption. The complexity, is worse than that of computing in a prime field due to less efficient primitives. Moreover, security against active adversaries is not immediate, but can be ensured in certain settings, e.g. when p and q are large and the adversary is computationally bounded. Alternatively a cut-and-choose solution may be employed. It is noted, that the techniques generalise further, the actual requirement is that $m = \prod_{i=1}^{t} p_i^{a_i}$, for odd a_i and distinct primes $p_i \equiv 3 \mod 4$[2], all greater than $\kappa + 1$.

The idea for $\mathbb{Z}_m$ considers Jacobi symbols rather than quadratic residues. Recalling that for $m = (pq)^a$ for odd primes p, q, the Jacobi symbol of $c \in \mathbb{Z}_m$, written $(\frac{c}{m})$, is defined based on the Legendre symbol – written $(\frac{c}{p})$ for prime p and defined to be ± 1 depending on whether $c \in \mathbb{Z}_p^*$ is a quadratic residue

[2]This requirement may be ignored if the bit-length of the primes are linear in the security parameter.

modulo p. The Jacobi symbol is defined as

$$\left(\frac{c}{m}\right) = \left(\frac{c}{p}\right)^a \cdot \left(\frac{c}{q}\right)^a.$$

This equals $(\frac{c}{p}) \cdot (\frac{c}{q})$, as only odd a are considered here.

The quadraticity test of section 9.2.2 can be generalised to computing the Jacobi symbol of an element in $\mathbb{Z}_m^*$ by generating the mask as a uniformly random element of $\mathbb{Z}_m^*$ along with a secret variable holding the Jacobi symbol of that element. The equality test is then constructed exactly as for $\mathbb{Z}_q$: If $[c]_m$ is 0, then squaring and adding a random square, $s \in \mathbb{Z}_m^*$, will always result in a Jacobi symbol of 1. However, if $[c]_m$ is non-zero, then each of the Jacobi symbols, $(\frac{\mathrm{val}([c]_m)^2+s}{p})$ and $(\frac{\mathrm{val}([c]_m)^2+s}{q})$, will be -1 with probability approximately one half.[3] This implies that

$$\left(\frac{\mathrm{val}([c]_m)^2 + s}{m}\right) = \left(\left(\frac{\mathrm{val}([c]_m)^2 + s}{p}\right)\left(\frac{\mathrm{val}([c]_m)^2 + s}{q}\right)\right)^a$$

is -1 with probability roughly one half. Concluding, the Jacobi symbols are mapped to bits their κ-ary `AND` is computed securely. As both primes are greater than $\kappa+1$, this may be done using the standard technique for symmetric Boolean functions.

It remains to show how to generate the mask needed for the Jacobi symbol-protocol. In general, letting each of the n parties P_i input a uniformly random value of $\mathbb{Z}_m^*$ to $\mathcal{F}_{\mathbf{MPC-R}}$, $[r_i]_m$ along with the Jacobi symbol of that value, $[j_i]_m$ and computing

$$[r]_m \leftarrow \prod_{i=1}^{n} [r_i]_m$$

and

$$[j]_m \leftarrow \prod_{i=1}^{n} [j_i]_m$$

provides the desired result when a passive adversary is considered. Active adversaries may be handled using e.g. cut-and-choose.

For Paillier based MPC, random bit generation is possible even in the face of an active adversary. Such a functionality allows a different approach: Letting ω denote an arbitrary, public value such that $(\frac{\omega}{(pq)^t}) = -1$, the parties provide $\mathcal{F}_{\mathbf{MPC-R}}$ with commands setting the following variables:

$$[r]_m \leftarrow \texttt{RandSq}(); \quad [b]_m \leftarrow \texttt{RandBit}(); \quad [b']_m \leftarrow \texttt{RandBit}()$$

Computing the negation of the bits, $[\bar{b}]_m \leftarrow 1 - [b]_m$ and $[\bar{b'}]_m \leftarrow 1 - [b']_m$, the mask may be constructed as

$$([b]_m \cdot [b']_m - [\bar{b}]_m \cdot [\bar{b'}]_m + [b]_m \cdot [\bar{b'}]_m \cdot \omega - [\bar{b}]_m \cdot [b']_m \cdot \omega) \cdot [r]_m.$$

[3] If either prime divides $\mathrm{val}([c]_m)$, one Jacobi symbol will always be one, however, this does not influence the analysis

Noting that the two bits select an element of $\{\pm 1, \pm\omega\}$, it is seen that the final value is uniformly random over $\mathbb{Z}_m^*$ as $p, q \equiv 3 \bmod 4$. Moreover, the Jacobi symbol is easily computable, it is 1 iff $\mathrm{val}([b]_m) = \mathrm{val}([b']_m)$. The complexity of generating a uniformly random bit, though constant rounds is linear in the number of parties, n. Thus the complexity of the Jacobi test is $\mathcal{O}(n)$ implying that the equality protocol requires $\mathcal{O}(n \cdot \kappa)$ multiplications. The result is summarised in the following theorem, noting that the result is not restricted to powers of RSA-moduli. Essentially the described test is a simulation of the standard $\mathbb{Z}_q$ test for every prime-factor p_i of m – if an odd number of these denote non-zero input then the full test does so too.

Theorem 9.2 *$\mathcal{F}_{\mathbf{MPC-R}}$ providing computation over the ring $\mathbb{Z}_m$, where $m = \prod_{i=1}^{t} p_i^{a_i}$ for distinct primes $p_i \equiv 3 \mod 4$ and odd a_i, may be extended with a constant rounds equality test, (**let** $[b]_m \leftarrow [a]_m \stackrel{?}{=} [a']_m$). This test may be statistically UC-realised against a passive adversaries in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model using $\mathcal{O}(n \cdot \kappa)$ secure multiplications, where n is the number of parties and the security parameter κ is less than $p_i - 1$ for all i. Further, if m is an odd power of an RSA-modulus (and random bit generation is provided), security against active adversaries can also be ensured. The computation provides perfect privacy even though it only statistically UC-realises the extension as with probability negligible in κ an incorrect result is determined.*

9.3.3 Immediate Applications

Equality testing provides a number of minor but immediate applications. They may be seen as sub-tasks of larger computations similarly to equality testing itself. All protocols will be considered over $\mathbb{Z}_q$.

Exception handling Consider the exception trick of [DFK+06]. The idea is that some secure computation requires input different from a single value, for simplicity 0. This input may be avoided by computing

$$[a']_q \leftarrow [a]_q + ([a]_q \stackrel{?}{=} 0)$$

and continuing with $[a']_q$ rather than $[a]_q$. Clearly $[a']_q$ is non-zero and equal to $[a]_q$ when this is non-zero. If desired the proper output for 0 may be obtained upon termination using a conditional selection. As the equality test here is much more efficient than previous ones, the exception trick is not only more efficient, but also applicable under more circumstances.

"Unbounded" fan-in `AND`- and `OR`-gates Another immediate application of the command for testing equality is logical `AND` and `OR` of multiple values. Similarly to the symmetric Boolean function technique of chapter 8, k bits $[b_1]_p, [b_2]_p, \ldots, [b_k]_p$ may be considered as elements of $\mathbb{Z}_q$. Assuming that $k < q$, adding them together allows us to efficiently compute the logical `AND` (`OR`) of all the bits, as all bits are 1 (0) exactly if the sum is k (0). Both can be handled using a single equality test, i.e. using only $\mathcal{O}(\kappa)$ multiplications rather than $\mathcal{O}(k)$.

Small set membership Assume that a set $S = \{s_1, s_2, \ldots, s_t\} \subset \mathbb{Z}_q$ is given (for some *small* t) along with a secret variable $[a]_q$. Securely determining a bit $[b]_q$, such that $[b]_q$ is 1 iff $\mathrm{val}([a]_q) \in S$ can be done by considering any t-degree polynomial, P, mapping s_i to 0 for all $1 \leq i \leq t$ and everything else to arbitrary, non-zero values. This may be constructed using Lagrange interpolation. Assuming that $[a]_q$ is non-zero, all powers up to t may be computed in constant rounds using $\mathcal{O}(t)$ multiplications using a prefix-product. Thus, $P([a]_q)$ may be evaluated securely and tested for equality with 0, which occurs exactly when $\mathrm{val}([a]_q) \in S$. The case of $\mathrm{val}([a]_p) = 0$ can be handled by the exception trick above. All in all, this requires $\mathcal{O}(t + \kappa)$ multiplications in $\mathcal{O}(1)$ rounds.

Using the above as a primitive, the complexity can be improved by splitting S up into $\sqrt{t}/\sqrt{\kappa}$ disjoint subsets, each of size $\sqrt{t} \cdot \sqrt{\kappa}$. (For simplicity it is assumed that κ divides t and both are square.) When determining set membership of multiple sets, the powers of $[a]_q$ need only be computed once. Thus, by reusing the powers of $[a]_q$, membership of *all* $\sqrt{t}/\sqrt{\kappa}$ subsets may be performed using

$$\mathcal{O}\Big(\sqrt{t} \cdot \sqrt{\kappa} + (\sqrt{t}/\sqrt{\kappa}) \cdot \kappa\Big) = \mathcal{O}\Big(\sqrt{t} \cdot \sqrt{\kappa}\Big)$$

multiplications. The overall result is then simply the sum of all the membership tests – the sets are disjoint, so at most one 1 will occur.

Testing set membership may be used to generalise exception handling: The polynomial P may be defined such that all illegal values are mapped to 0, the ideas may be combined by computing

$$[a']_q \leftarrow P([a]_q) \stackrel{?}{=} 0 \,?\, c : [a]_q.$$

This securely sets $[a']_q$ to c, when $[a]_q$ contains an illegal value, but allows the value to "pass through" to $[a']_q$ when it is legal. Thus, given a set of t illegal inputs, all can be avoided using $\mathcal{O}(\sqrt{t} \cdot \sqrt{\kappa})$ multiplications in a constant number of rounds.

Chapter 10

Inequality

The problem of inequality is very much similar to that of equality considered in the previous chapter. The sole difference is the desired outcome – a bit should be set to 1 if and only if the first input is less than the second. This is one of the oldest problems in secure computation – often referred to as Yao's millionaires' problem, [Yao82] – and many different solutions in many settings exist, though much of the literature refer to only to bit-wise stored inputs and a two-party case. Examples of related work include [BDJ+05, Fis01, ST04, DGK]. However, where much work, including the original paper, consider secure function evaluation – i.e. where the result is obtained by the parties – the outcome here should be a bit stored in a variable of $\mathcal{F}_{\mathbf{MPC-R}}$.

Though the original formulation of the problem is old, improvements have repeatedly been made. Focusing on the present setting – i.e. a realisation of $\mathcal{F}_{\mathbf{MPC}}$ using linear primitives – an efficient, constant-rounds protocol was until recently an open problem. Through bit-decomposition, Damgård *et al.* [DFK+06] provided the first constant-rounds solution, this required $\mathcal{O}(\ell \cdot \log(\ell))$ multiplications, where ℓ is the bit-length of the prime q. This was later improved by Nishide and Ohta [NO07] based on sub-protocols of [DFK+06] A similar construct was noted independently but subsequently by the present author (and included in the paper of Nishide and Ohta – their initial solution was based on interval-testing rather than determining the least significant bit), this version is presented below. All of the solutions in this paragraph build on core ideas from the bit-by-bit method (BBM) of [BDJ+05], described in detail in [Tof05].

The structure of the extension to $\mathcal{F}_{\mathbf{MPC-R}}$ for handling inequality testing is equivalent to the one for testing equality: ($\mathbf{let}\ [b]_q \leftarrow [a]_q \overset{?}{<} [a']_q$) is added to the set of commands, and upon receiving it from all honest parties in round R, $\mathcal{F}_{\mathbf{MPC-R}}$ compares $\mathrm{val}([a]_q)$ and $\mathrm{val}([a']_q)$. The adversary is then notified that the command is being executed. In round $R + 24$ variable $[b]_q$ is set to the result determined in round R and all parties and the adversary are notified that the command has terminated. As with equality, comparing a variable and a public value is immediate from this.

Concerning this chapter, rather than attempting to provide the full solution in one go, a simpler problem is handled initially. Section 10.1 considers comparison between a publicly known value and a bit-wise stored one. This is then

used as a primitive in section 10.2 to construct the general solution realising the extension. Often when comparing variables of $\mathcal{F}_{\mathbf{MPC-R}}$, it will be known that their values are of a limited size, i.e. they are bounded. A more efficient approach is presented in section 10.3 which is applicable only in such cases. This is an unpublished result due to the author, it is essentially a continuation of the work of [BDJ+05]. The chapter concludes with a few remarks in section 10.4.

10.1 Comparing Bit-wise Stored Values

Rather than attempting to compare secret variables directly, a simpler task is considered first, namely that of comparing a public value and a bit-wise stored variable, i.e. one where access to the individual bits is ensured. Recalling that $[a]_B$ is short-hand for an array of binary variables $([a_{k-1}]_q, \ldots, [a_0]_q)$, viewing this as a binary encoding of a value in $\mathbb{Z}_{2^k}$, the comparison of $[a]_B$ and constant $c \in \mathbb{Z}_{2^k}$ may be defined as follows

$$\sum_{i=0}^{k-1} 2^i \cdot \mathrm{val}([a_i]_q) \overset{?}{<} c.$$

Note that comparison of values over $\mathbb{Z}_q$ follows directly from this by viewing $\mathbb{Z}_q$ as a subset of $\mathbb{Z}_{2^k}$ for $k = \lceil \log(q) \rceil$. This task is *much* simpler than comparing arbitrary secret variables, moreover it is an important building block in the construction of the full comparison protocol below. The computation below is taken from [DFK+06].

A formal extension to $\mathcal{F}_{\mathbf{MPC-R}}$ is *not* introduced, rather writing $[a]_B \overset{?}{<} c$ for variable $[a]_B$ and constant c is seen as syntactic sugar for the computation described as protocol 5. For simplicity it is assumed that $[a]_B$ and c are of the same bit-length. Privacy is immediate as all computation occurs within

Protocol 5 Comparing a bit-wise stored variable and a public constant

Input: $[a]_B = ([a_{k-1}]_q, \ldots, [a_0]_q)$ and $c \in \mathbb{Z}_{2^k}$; c_i will be used to refer to the i'th bit of c.

Output: Binary $[b]_q$, such that $\mathrm{val}([b]_q) = 1$ iff $\sum_{i=0}^{k-1} 2^i \cdot \mathrm{val}([a_i]_q) < c$

```
    for i = 0,...,k-1 do
        [d_i]_q ← [a_i]_q ⊕ c_i
    end for
    [d]_B ← ([d_{k-1}]_q, ..., [d_0]_q)
 5: [e]_B ← pre_∨([d]_B)
    [f_{k-1}]_q ← [e_{k-1}]_q
    for i = 0,...,k-2 do
        [f_i]_q ← [e_i]_q − [e_{i+1}]_q
    end for
10: [b]_q ← Σ_{i=0}^{k-1} c_i · [f_i]_q
```

$\mathcal{F}_{\mathbf{MPC-R}}$ – using primitives already argued secure – and no variables are output.

For correctness, assume first that the compared values are not equal. The variables have the following intuition: $\mathrm{val}([d_i]_q)$ is 1 exactly when the i'th bit of $[a]_B$ and c differ. Thus, the binary $[e_i]_q$ is set to 1 iff for some j, $i \leq j < k$, $[d_j]_q$ is 1. Consequently all but one $\mathrm{val}([f_i]_q) = 0$ – the single 1 occurs at the most significant index where $[a]_B$ and c differ. The final computation simply selects the associated c_i, which is the desired result. It remains to argue correctness when $\mathrm{val}([a]_B) = c$. In this case $[d]_B$ contains all 0's and these propagate down. All $[e_i]_q$ and $[f_i]_q$ will be 0, thus $[b]_q$ is correctly set to 0.

Regarding complexity, only step 5 requires secure multiplications. It is easily verified that all other computation is costless. Thus, the complexity is equivalent to that of the (**let** $[b]_B \leftarrow \mathtt{pre}_\vee([a]_B)$) command, $13k$ multiplications in 12 rounds, four of which may be viewed as preprocessing.

Protocol 5 has some immediate applications. It is now possible to generate uniformly random values of $\mathbb{Z}_q$ along with their binary representation. Formally, $\mathcal{F}_{\mathbf{MPC-R}}$ is extended with a command (**let** $([r]_q, [r]_B) \leftarrow \mathtt{Rand}()$). This is equivalent to (**let** $[r]_q \leftarrow \mathtt{Rand}()$), except that not only is a uniformly random element generated and stored in variable $[r]_q$, the binary representation of that value is stored in variables $[r_{\ell-1}, \ldots, [r_0]_q]_q$, where $\ell = \lceil \log(q) \rceil$ is the bit-length of q. Moreover, 13 rounds are required, thus when receiving the command in round R, $\mathcal{F}_{\mathbf{MPC-R}}$ notifies the adversary and returns a termination-message in round R+12. When considering algorithmic descriptions of secure computation,

$$[r]_B \leftarrow \mathtt{Rand}_{\mathtt{bits}}()$$

will be used to denote the assignment of the bits of a fresh random value to $[r]_B$. No explicit mention of variable $[r]_q$ is made, its assignment is left implicit keeping notation cleaner though technically incorrect.

The command is realised in the obvious way: ℓ uniformly random bits are generated and it is verified that these do indeed represent a value of $\mathbb{Z}_q$. The value is then constructed from these. The details are seen in protocol 6. When

Protocol 6 Realising (**let** $([r]_q, [r]_B) \leftarrow \mathtt{Rand}()$) in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model

Input: None.

Output: $[r]_q$ and $[r]_B$ such that $[r]_q$ is assigned a uniformly random value of $\mathbb{Z}_q$, and $[r]_B$ contains the binary representation of that value.

for i = 0,...,ℓ-1 **do**
  $[r_i]_q \leftarrow \mathtt{RandBit}()$
end for
$[r]_B \leftarrow ([r_{\ell-1}]_q, \ldots, [r_0]_q)$
5: $c \leftarrow \mathtt{output}([r]_B \stackrel{?}{<} q)$
if $c \stackrel{?}{=} 0$ **then**
  Abort
end if
$[r]_q \leftarrow \sum_{i=0}^{\ell-1} 2^i \cdot [r_i]_q$

the computation does not abort, by this intuition it provides the correct result.

Privacy is immediate, as the protocol has no inputs only information may be leaked on the result $[r]_q$. However, only c is output and this leaks no other information than whether $\mathrm{val}([r]_q) \in \mathbb{Z}_q$ – if this is not the case, the generated values are discarded. As $\mathrm{val}([r]_q) \in \mathbb{Z}_q$ is a requirement of a successful run no information is disclosed by c. In the computation, ℓ random bits are generated, and a comparison between a bit-wise stored value and a public value is performed. Finally, the result of the comparison is output. This requires $2\ell + 13\ell = 15\ell$ multiplications. Noting that the generation of the bits parallelise with the preprocessing of the comparison, 13 rounds (noting the one required for the output) of interaction are required rather than the naïve 16.

Similar to most of the realisations of random value generation in the basic $\mathcal{F}_{\mathbf{MPC}}$-hybrid model, the present computation aborts with probability $\frac{2^\ell - q}{2^\ell}$ which may be as high as $1/2$. If q is large (i.e. ℓ linear in the security parameter) and can be selected freely, then the problem is minimal; q may be chosen as $2^\ell - k$ for some small k implying a negligible probability of termination. If this is not the case, a different strategy must be employed, all solutions presented in section 8.1.5 are applicable here as well.

The command $(\mathbf{let}\ ([r]_q, [r]_B) \leftarrow \mathtt{Rand}())$ may be used to construct an alternative equality test providing both perfect correctness *and* perfect privacy, i.e. equality may be perfectly UC-realised in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. The drawback of the solution is that the complexity is linear in the bit-length of q. This was originally noted by Nishide and Ohta and independently by the author. The idea is to generate a uniformly random $[r]_q$ along with its binary representation $[r]_B$. $[a]_q$ may then be tested for equality with 0 by outputting $[a]_q + [r]_q$ (which is uniformly random) and testing whether the binary representation of this is equivalent to that of $[r]_B$. I.e. perform a bit-wise `XOR` and logical `OR` the individual bits. Clearly this is correct, non-leaking, and requires $\mathcal{O}(\ell)$ multiplications in $\mathcal{O}(1)$ rounds.

10.2 Comparing Arbitrary Values of $\mathbb{Z}_q$

Having provided a means for comparing a bit-wise stored value to a public one, the extension of $\mathcal{F}_{\mathbf{MPC-R}}$ allowing the comparison of two secret variables,

$$(\mathbf{let}\ [b]_q \leftarrow [a]_q \overset{?}{<} [a']_q),$$

may be perfectly UC-realised in the basic $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. The idea consists of two parts: Solving a simpler problem where one term is fixed, and using this to obtain the overall result.

Comparing both inputs to $\frac{q+1}{2} \in \mathbb{Z}_q$ – written as a fraction for readability – allows the result to be determined immediately when one is smaller and the other greater. However, when the tests provide the same result, no information may be computed. It can be noted, though, that in this case

$$[a]_q < [a']_q \Leftrightarrow \left([a']_q - [a]_q\right) < \frac{q+1}{2}.$$

As the inputs differ by at most $\frac{q-1}{2}$, computing $[a']_q - [a]_q$ *within the field* results in a value less than $\frac{q+1}{2}$ exactly when $[a']_q$ is the larger; when $\mathrm{val}([a']_q) < \mathrm{val}([a]_q)$ an underflow occurs resulting in a greater value. Thus by performing three tests:

$$[b_a]_q \leftarrow [a]_q \overset{?}{<} \frac{q+1}{2}; \quad [b_{a'}]_q \leftarrow [a']_q \overset{?}{<} \frac{q+1}{2}; \quad [b_\delta]_q \leftarrow ([a']_q - [a]_q) \overset{?}{<} \frac{q+1}{2}$$

the final result may be computed as

$$[b]_q \leftarrow [b_a]_q \oplus [b_{a'}]_q \,?\, [b_a]_q : [b_\delta]_q. \tag{10.1}$$

It remains to be seen how to compare a secret variable $[a]_q$ to $\frac{q+1}{2}$. A computation doing this is seen as protocol 7. The intuition consists of two parts. First, the least significant bit of the value $2 \cdot \mathrm{val}([a]_q) \in \mathbb{Z}_q$ is the desired result. If the value of $[a]_q$ is less than $\frac{q+1}{2}$, doubling it makes it even; however, when it is larger, then a reduction modulo q occurs implying that this product is odd.

Letting b denote the desired bit, the second part of the intuition explains how this is extracted. Viewing the addition of step 2 as being performed over the integers rather than $\mathbb{Z}_q$, $b \oplus \mathrm{val}([r_0]_q) = c_0$, the least significant bit of c. This is also the case when no overflow occurred in the masking. If overflow did occur, however, then a reduction modulo q has occurred and c_0 has been inverted. Thus if overflow can be determined, b may be computed. This is done in step 3; if $\mathrm{val}([r]_B) \leq c \in \mathbb{Z}_q$ subtracting $\mathrm{val}([r]_q)$ from c does not result in an underflow, thus no overflow occurred and $[\bar{o}]_q$ is set to 1. It may be verified that $\leq$ is simulated correctly using $<$. Otherwise $[\bar{o}]_q$ will be 0; thus correctness of $[b]_q$ is ensured. Privacy is ensured as $[r]_q$ is a uniformly random mask, thus no information is gathered from c.

Protocol 7 Securely comparing variables to $\frac{q+1}{2}$

Input: $[a]_q$.

Output: Binary $[b]_q$ such that $\mathrm{val}([b]_q) = 1$ iff $\mathrm{val}([a]_q) < \frac{q+1}{2}$.

$[r]_B \leftarrow \texttt{Rand}_{\texttt{bits}}()$

$c \leftarrow \texttt{output}(2 \cdot [a]_q + [r]_q)$

$[\bar{o}]_q \leftarrow [r]_B \overset{?}{<} (c+1)$ ▷ The addition of 1 is performed over the integers

$[b]_q \leftarrow c_0 \oplus [r_0]_q \oplus (\neg [\bar{o}]_q)$ ▷ c_0 denotes the least significant bit of c

---

Returning to the realisation of the full command (**let** $[b]_q \leftarrow [a]_q \overset{?}{<} [a']_q$), three invocations of protocol 7 are needed in addition to the concluding computation. Correctness and privacy follow by the above discussion, thus the remaining point to consider is complexity. The three invocations of protocol 7 result in $3 \cdot (15\ell + 13\ell + 1) = 84\ell + 3$ multiplications. Noting that the preprocessing of the comparison may be performed in parallel with the generation of the mask, these require $13 + 8 + 2 = 23$ rounds (of which the 13 may be viewed as preprocessing). The concluding computation described in equation 10.1 requires two additional multiplications in two rounds. Thus, as the three comparisons

to $\frac{q+1}{2}$ may be performed in parallel, the overall complexity is $84\ell + 5$ secure multiplications in 25 rounds, 13 of which may be viewed as preprocessing.

As both correctness and privacy are perfect, clearly the computation described perfectly UC-realises the extension of $\mathcal{F}_{\mathbf{MPC-R}}$ with the command

$$(\mathbf{let}\ [b]_q \leftarrow [a]_q \overset{?}{<} [a']_q)$$

in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. This provides the main result of this chapter, stated as theorem 10.1.

Theorem 10.1 (Inequality) *$\mathcal{F}_{\mathbf{MPC-R}}$ extended with* $(\mathbf{let}\ [b]_q \leftarrow [a]_q \overset{?}{<} [a']_q)$ *– a command for securely determining inequality of secret variables in $\mathcal{O}(1)$ rounds – can be perfectly UC-realised in the original $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model using $\mathcal{O}(\ell)$ secure multiplications, where ℓ is the bit-length of the prime q defining the field in which computation occurs.*

10.3 An Efficient Test for Bounded Values

For multi-party computation, the overall goal will typically be to perform some *integer* computation securely. Thus computation over $\mathbb{Z}_q$ is merely performed to simulate this. Given bounds on the inputs of parties along with a description of the computation, q must be chosen sufficiently large to ensure that overflow *cannot* occur. Having bounds on the inputs implies bounds on all intermediate values, moreover these may be significantly smaller than q. In this case, a more efficient realisation of comparison is possible. Disregarding correctness and privacy for non-legal inputs, the complexity depends only on the bit-length of the actual values rather than that of q.

The solution presented sacrifices perfect security, however – at least when the core idea is based on binary representation – this seems to be required. Intuitively, if perfect security is to be realised, uniformly random masks over all of $\mathbb{Z}_q$ (or similar sets, e.g. $\mathbb{Z}_q^*$) must be generated such that certain bits are available. Extracting bits is difficult – the realisation above required exactly this – thus, generating the desired bits first and *then* constructing the mask seems a better strategy. However, the construction of uniformly random values becomes difficult; section 10.1 above considered such a task, which was done by generating *all* bits (and discarding incorrect candidates) resulting in a complexity dependant on q. Thus, it is reasonable to be content accept security which is "only" statistical.

Formalising the problem, let $[a]_q$ and $[a']_q$ be variables of $\mathcal{F}_{\mathbf{MPC-R}}$ such that it is known that $\mathrm{val}([a]_q), \mathrm{val}([a']_q) < 2^k$ for some k with $2^{k+\kappa} \ll q$. Thus, the variables are restricted to k bits while the bit-length of q is at least κ greater. The computation trivially generalises to differing bit-lengths of the variables. Defining the goal slightly different from above, the task will be to set a bit $[b]_q$ to $\mathrm{val}([a]_q) \overset{?}{\leq} \mathrm{val}([a']_q)$. The difference - using $\overset{?}{\leq}$ rather than $\overset{?}{<}$ is due to the intuition behind the protocol as will be seen. The two are equivalent, with one being the "negation" of the other.

The main idea behind the computation is to extract the k'th bit of $c = 2^k + \mathrm{val}([a']_q) - \mathrm{val}([a]_q)$, c_k. This is the desired result as

$$\begin{aligned} & c_k = 1 \\ \Updownarrow \\ & c = 2^k + \mathrm{val}([a']_q) - \mathrm{val}([a]_q) \geq 2^k \\ \Updownarrow \\ & \mathrm{val}([a']_q) - \mathrm{val}([a]_q) \geq 0. \end{aligned}$$

Protocol 8 computes this bit securely, denoting it $[b]_q$.

Protocol 8 Comparing bounded, secret variables

Input: $[a]_q$ and $[a']_q$ containing k-bit values where $2^{k+\kappa} \ll q$.
Output: Binary $[b]_q$, such that $\mathrm{val}([b]_q) = 1$ iff $\mathrm{val}([a]_q) \leq \mathrm{val}([a']_q)$.

$[c]_q \leftarrow 2^k + [a']_q - [a]_q$
for i = 0,. . . ,k+κ -1 **do**
  $[r_i]_q \leftarrow$ `RandBit`()
end for
5: $[r]_q \leftarrow \sum_{i=0}^{k+\kappa-1} 2^i \cdot [r_i]_q$
$[r \bmod 2^k]_q \leftarrow \sum_{i=0}^{k-1} 2^i \cdot [r_i]_q$
$[r \bmod 2^k]_B \leftarrow ([r_{k-1}]_q, \ldots, [r_0]_q)$
$m \leftarrow$ `output`$([c]_q + [r]_q)$
$[u]_q \leftarrow [r \bmod 2^k]_B \overset{?}{>} (m \bmod 2^k)$
10: $[c \bmod 2^k]_q \leftarrow (m \bmod 2^k) - [r \bmod 2^k]_q + 2^k \cdot [u]_q$
$[b]_q \leftarrow 2^{-k} \cdot ([c]_q - [c \bmod 2^k]_q)$

Correctness is easily verified by the above observation and through the intuition behind the bit-extraction. The idea is to compute $[c]_q \bmod 2^k$ and subtract this from c. The difference is either 0 or 2^k, which is then mapped to $\{0, 1\}$. Thus, correctness has been shown assuming that $\mathrm{val}([c \bmod 2^k]_q) = \mathrm{val}([c]_q) \bmod 2^k$. This is the case as all computation (except the final mapping to $\{0, 1\}$) may be seen as occurring over the integers – q was chosen greater than $2^{k+\kappa}$, thus no computation overflows the field.

Essentially $[c \bmod 2^k]_q$ is assigned the value

$$((m \bmod 2^k) - (\mathrm{val}([r \bmod 2^k]_q)) \bmod 2^k$$

which clearly equals

$$(\mathrm{val}([c]_q) + \mathrm{val}([r]_q) - \mathrm{val}([r]_q)) \bmod 2^k.$$

Reducing both m and $[r]_q$ modulo 2^k before the computation does not introduce errors when computing modulo 2^k, however, as the subtraction occurs over $\mathbb{Z}_q$ the possibility of underflow (modulo 2^k) must be considered. This occurs exactly when $\mathrm{val}([r \bmod 2^k]_q) > m \bmod 2^k$, therefore $[u]_q$ computed in step 9

is set to 1 exactly when an underflow occurs, allowing 2^k to be added. Note that for readability a greater-than test has been used; this is easily constructed from the bit-wise comparison in the same complexity.

Privacy is immediate, the only output being m occurring in step 8. As $[c]_q$ is only $(k+1)$-bit, the value seen is statistically indistinguishable from a uniformly random $(k+\kappa)$-bit value, thus no information is revealed except with negligible probability in the security parameter. Regarding complexity, the generation of $[r]_q$ (in all its forms) requires $k+\kappa$ random bits. Everything else is costless being either renaming or linear. In addition to this, a value is output and a bit-wise comparison performed. All in all, $(k+\kappa)\cdot 2 + k\cdot 13 = 15k + 2\kappa$ multiplications in $3+1+12 = 16$ rounds are needed. Round complexity may be improved by executing the preprocessing of the bit-wise comparison in parallel with the generation of the bits of $[r]_q$ and the output, this reduces the requirement to 12 rounds. Concluding, note that preprocessing makes a difference for complexity here: Only $\mathcal{O}(k)$ secure multiplications are needed online for performing the bit-wise comparison. The result is summarised in the following theorem.

Theorem 10.2 *$\mathcal{F}_{\mathbf{MPC-R}}$ extended with a command for comparing secret variables of* bounded *size (i.e. k-bit) may be statistically UC-realised in the basic $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model using $\mathcal{O}(k+\kappa)$ multiplications in $\mathcal{O}(1)$ rounds. Of these only $\mathcal{O}(k)$ of these are required online.*

The section is concluded with a small optimisation for passive adversaries. It may be observed that the top κ bits of $[r]_q$ are used solely in order to generate the mask. In other words it is *not* required that they are available for computation. Thus, a random value generated without these suffices. This can be accomplished by generating the k least significant bits of $[r]_q$ as above and letting each of the n parties P_i input a uniformly random κ-bit value $[r_t^i]_q$ – which may be done as the parties follow the protocol. If $[r]_q$ is set by

$$[r]_q \leftarrow 2^k \cdot \Big(\sum_{j=1}^{n} [r_t^j]_q\Big) + \Big(\sum_{i=0}^{k-1} 2^i \cdot [r_i]_q\Big)$$

it will not be uniformly random. However, its $k+\kappa$ least significant bits will be, thus $[c]_q$ is still masked. Intuitively the variation can be viewed as every party providing the top bits of $[r]_q$. This reduces the preprocessing complexity to $\mathcal{O}(k)$; the only drawback is a slightly stronger requirement of q, as the sum of the $[r_t^i]_q$ must not overflow: $q \gg 2^{k+\kappa+\lceil\log(n)\rceil}$.

10.4 Concluding Comments

Having extended $\mathcal{F}_{\mathbf{MPC-R}}$ with a command (**let** $[b]_q \leftarrow [a]_q \overset{?}{<} [a']_q$) for testing inequality, and realised that extension, a few concluding comments on testing inequality are made.

Two realisations have been presented – one general and the other requiring bounded inputs – and both have their merits. As noted, when simulating

integer computation the size of variables to be compared may be known to be bounded, thus the lower complexity of the second protocol is a definite advantage. However, it cannot stand alone. Not only does it not provide perfect security which may be desirable, restrictions are made on q which might not always hold, namely that its bit-length is linear in the security parameter. Further, when active adversaries are considered, it must be verified that inputs for the computation are legal, e.g. that k-bit inputs truly are k-bit. A protocol with restrictions on its input-variables cannot be used for this. The general comparison operator is therefore critical.

Having realised an ideal functionality storing the result in a secret variable, one could ask oneself if – similar to equality – realising a variation where the result is output is more efficient. This is not the case, as will be demonstrated – based on any comparison with public output, one storing the result in a variable may be constructed.

Assume that the functionality has been extended with (**output** $[a]_q \overset{?}{<} [a']_q$). Upon receiving it, $\mathcal{F}_{\mathbf{MPC-R}}$ immediately notifies the adversary of the command and the result; at some later point, the parties also receive the output. Constructing a comparison storing the result in a variable is simple when access to this command is provided. Assume first that $\mathrm{val}([a]_q) \neq \mathrm{val}([a']_q)$. The trick is then to initially flip a coin and conditionally swap the inputs based on this.

$$[\tilde{b}]_q \leftarrow \texttt{RandBit}(); \quad [\tilde{a}]_q \leftarrow [\tilde{b}]_q \,?\, [a']_q : [a]_q; \quad [\tilde{a'}]_q \leftarrow [\tilde{b}]_q \,?\, [a]_q : [a']_q$$

Comparing these new values results in the disclosure of a uniformly random bit – $[\tilde{b}]_q$ masks the output value. Based on this bit and $[\tilde{b}]_q$ the correct result may therefore be computed. Potentially equal inputs may be handled through the exception trick described in section 9.3.3. Alternatively, if the values are bounded $2 \cdot [a]_q$ may be compared to $2 \cdot [a']_q + 1$ instead. It is clear that these are not equal and that the result is not altered. Thus, based on any inequality test with public output an efficient one which does not disclose information is easily constructed.

It is noted that – assuming bit generation is available – all protocols translate directly to computing modulo odd m. Thus all protocols introduced apply to the second motivating example of threshold Paillier encryption, where computation occurs modulo a power of an RSA-modulus.

Chapter 11

Bit-extraction

When performing multi-party computation, the complexity of a task is often related to the representation of the inputs. This is particularly the case when focus is on constant round complexity. Comparison – as described in the previous chapter – is one example: When access to a bit-wise storing of the inputs was provided, the solution was relatively straight forward, while the general solution was not. The solution provided was essentially a transformation of the problem to one with a "nicer" (i.e. binary) representation. Indeed, if a primitive for converting a variable $[a]_q$ to its binary representation $[a]_B$ had been available, comparison of two values would have been almost immediate.

Though comparison was eased through a bit-wise representation, this does not imply that this representation is superior. Simple computation such as addition and multiplication – costless and the basic measure of complexity – are not immediate when using this representation. Depending on the task at hand, one representation may therefore be preferable to another, thus changing representation during a computation is desirable.

Though the previous chapter demonstrates that comparison does not require bit-extraction, many problems do not have (efficient) constant round solutions – at least with current knowledge – unless the inputs are bit-wise stored. Examples include exponentiation and modulo reduction. These problems are greatly simplified when access to the individual bits are provided. Thus, the goal of this chapter is to construct a *constant round* bit-extraction protocol in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. This allows representation change implying that many tasks may be computed efficiently in constant round. For completeness, it is noted that translation from binary representation to "ordinary" values of $\mathbb{Z}_q$ is trivial; it has implicitly been utilised repeatedly in the past chapters.

A considerable amount of previous work on (unconditionally secure) constant-rounds multi-party computation exist (e.g. [BB89, FKN94, IK97, CD01, Bea00, IK00, IK02]). This work has shown that some functions can be computed in constant rounds with unconditional security, but this has been limited to restricted classes of functions, such as NC_1 or non-deterministic log-space. In [ACS02] Algesheimer, Camenisch and Shoup presented a protocol for securely computing the bit-decomposition $[a]_q \mapsto [a]_B$. However, opposed to the secure computation below, correctness and privacy were only provided when $\mathrm{val}([a]_q)$ was guaranteed to be noticeably smaller than q. Moreover, their protocol was

not constant rounds, nor was it secure against active adversaries. Based on the protocol described, Nishide and Ohta provided an improved version in [NO07]. The gain, however, is only a constant factor – it is noted below.

This chapter is an adaptation of [DFK+06]. Some sections of that article have been omitted, in particular those relating to sub-tasks considered in chapter 8 above. Others have been inserted more or less directly, though all have received an overhaul in order to fit the style of this thesis. In particular, all complexity analyses have been updated to reflect complexity within the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model.

The remainder of this chapter is organised as follows. The problem of bit-extraction is formalised and realised in section 11.1. It is done so in a hybrid model which provides an additional command. The bulk of the work of this chapter then consists of realising this command in the basic $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model in section 11.2. The overall complexity of the realisation is then considered in section 11.3, which concludes with additional comments and applications.

11.1 Bit-extraction – The Overall Computation

As always, the extension to the ideal functionality comes in the form of an additional command, (**let** $[a]_B \leftarrow \texttt{bits}([a]_q)$). Upon receiving this from all honest parties in round R, $\mathcal{F}_{\mathbf{MPC-R}}$ notifies the adversary that the command is being executed. In round $R + 50$ $[a]_B$ is set to the result, and the parties and the adversary are notified that the command has terminated. This section considers a realising computation of the bit-extraction command in the simpler $\mathcal{F}_{\mathbf{MPC}}$-hybrid model.

In order to perform the computation, an additional primitive is required, namely that of bit-wise addition. Given variables $[a]_B = ([a_{k-1}]_q, \ldots, [a_0]_q)$ and $c \in \mathbb{N}$, the command (**let** $[s]_B \leftarrow \texttt{Add}([a]_B, c)$) (written $[s]_B \leftarrow \texttt{Add}([a]_B, c)$ in the algorithmic notation) will be used to denote the computation of the sum of the (binary encoded) $\mathrm{val}([a]_B)$ and c, storing the result (in binary) in $[s]_B$. The command is formalised and realised in section 11.2 below. Note that in contrast to most other computation performed, this addition is performed over the integers, not $\mathbb{Z}_q$, hence the notation. Writing addition as a function rather than in infix notation ensures that the task is not confused with costless additions over $\mathbb{Z}_q$.

Using the command for bit-wise addition, (**let** $[a]_B \leftarrow \texttt{bits}([a]_q)$) may now be realised in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. The computation is shown as protocol 9. As always, the prime q is assumed to be ℓ-bit. Correctness is easily verified based on the intuition behind the computation. The main idea is contained in steps 1 to 3: A uniformly random value is generated, subtracted from the input, and added again. Intuitively – as the concluding addition results in a bit-wise storing – the correct result is determined and stored in the desired manner. However, this is not entirely true. The bit-wise addition is performed over the integers, not $\mathbb{Z}_q$, thus an overflow may occur implying that $[d]_B$ only contains a value *congruent* to the desired result. The remaining computation

Protocol 9 Bit-extraction using access to bit-wise addition.

Input: $[a]_q$.
Output: $[a]_B$, the binary encoding of the value of $[a]_q$.

$[r]_B \leftarrow \texttt{Rand}_{\texttt{bits}}()$
$c \leftarrow \texttt{output}([a]_q - [r]_q)$
$[d]_B \leftarrow \texttt{Add}([r]_B, c)$
$[\bar{o}]_q \leftarrow [d]_B \overset{?}{<} q$
5: $[d']_B \leftarrow \texttt{Add}([d]_B, 2^\ell - q)$
for i = 0,...,ℓ-1 **do**
  $[a_i]_q \leftarrow [\bar{o}]_q \,?\, [d_i]_q : [d'_i]_q$
end for
$[a]_B \leftarrow ([a_{\ell-1}]_q, \ldots, [a_0]_q)$

simply reduces modulo q.

As both c and the value of $[r]_B$ are elements of $\mathbb{Z}_q$, their "integer" sum is less than $2q$. Thus, there are only two possibilities: Either $[d]_B$ contains the correct result or q must be subtracted. The computation concludes by determining both candidates – denoting the second $[d']_B$ – and securely selecting the correct one; the choice is based on $[\bar{o}]_q$, which is binary and set to 1 exactly when an overflow has *not* occurred. Thus, assuming that $[d'_{\ell-1}]_q, \ldots, [d'_0]_q$ is correct when an overflow does occur, clearly $[a]_B$ is correct. The subtraction needed to construct $[d']_B$ is performed by *adding* the positive $2^\ell - q$ and discarding the top bit of the result – i.e. reducing modulo 2^ℓ. As

$$(\mathrm{val}([a]_q) + q) + (2^\ell - q) = 2^\ell + \mathrm{val}([a]_q)$$

over the integers and $\mathrm{val}([a]_q) < q < 2^\ell$, the ℓ low bits of $[d']_B$ do indeed contain the correct result when $\mathrm{val}([d]_B) \geq q$, implying that the computation provides the correct result.

None of the sub-functionalities leak any information, thus privacy is immediate as the output, c, is uniformly random and independent of $\mathrm{val}([a]_q)$ due to the uniformly random mask $[r]_q$. Thus, bit-extraction is realised, though not from scratch in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. The missing piece – addition of bit-wise stored values – is the subject of the following section.

As one of the constructs is missing, complexity of the computation cannot be considered in full until this has been remedied. The discussion is therefore postponed to section 11.3. A preliminary analysis disregarding the two bit-wise additions is made though. The computation consists of the generation of the mask, comparing the "unmasked" value to q in step 4 (noting that this $[d]_B$ is $(\ell + 1)$-bit), and selecting the correct solution in the `For`-loop. All in all this requires $15\ell + 13(\ell + 1) + \ell = 29\ell + 13$ multiplications.

With regard to rounds, the conditional selections of the loop do not influence each other and may be performed in parallel in a single round. Thus assuming that bit-wise addition is possible in constant round, the overall computation may be realised in constant round. This is the case, as will be seen. The precise analysis is postponed to section 11.3.

It is noted that this bit-extraction is slightly different than the one presented in [DFK+06]. Notably, in that work one of the bit-wise additions had *both* inputs stored in binary in $\mathcal{F}_{\mathbf{MPC-R}}$. The variation of Nishide and Ohta requires only one bit-wise addition, though with both inputs secret. The key observation is that either c or $2^l - q + c$ is added to $[d]_B$, thus by detecting overflow based on c and $[r]_B$, the relevant addend may be selected securely. Both variations, however, are marginal improvements on the original.

11.2 Addition of Bit-wise Stored Values

Before realising the command for bit-wise addition the desired behaviour of the extension is specified formally. Let c be a public k-bit integer and let $[a]_B = ([a_{k-1}]_q, \ldots, [a_0]_q)$ be defined. For simplicity – and without loss of generality – it will be assumed that both values are of the same bit-length. Similar to other commands, upon receiving (**let** $[s]_B \leftarrow \texttt{Add}([a]_B, c)$) from all honest parties in round R the result is computed and the adversary notified that the command has been given. In round $R + 21$ $[s]_B$ is set and all parties and the adversary are notified of this.

The remainder of this section realises this extension in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. Initially a required sub-protocol is considered in section 11.2.1. Sections 11.2.2 to 11.2.4 then construct the addition top down, each section lacking a building block constructed in the subsequent one.

11.2.1 Generic Postfix Computation

Assume that some alphabet $\Sigma \subseteq \{0,1\}^v$ for some integer v is given along with bit-wise stored inputs $[a_k]_B, \ldots, [a_1]_B$ where $\mathrm{val}([a_i]_B) \in \Sigma$. Assume further that an associative binary operator $\circ : \Sigma \times \Sigma \mapsto \Sigma$ is given, the goal is now to compute postfix-$\circ$, i.e.

$$([b_k]_B, \ldots, [b_1]_B) = \texttt{post}_\circ([a_k]_B, \ldots, [a_1]_B),$$

such that for all $i \in \{1, 2, \ldots, k\}$

$$\mathrm{val}([b_i]_B) = \mathrm{val}([a_i]_B) \circ \mathrm{val}([a_{i-1}]_B) \circ \ldots \circ \mathrm{val}([a_1]_B).$$

Assume that it is possible to securely compute an i-ary fan-in $\circ$ in r rounds using $C(i) = c \cdot i$ multiplications for some constant c. For short $[a_{i_1}]_B \circ \ldots \circ [a_{i_0}]_B$ will be referred to as the "sum" of $[a_{i_1}]_B, \ldots, [a_{i_0}]_B$. For notational convenience this will be written $\circ_{i=i_1}^{i_0} [a_i]_B$, moreover without loss of generality it will be assumed that k is a power of two.

Using the method by Chandra, Fortune and Lipton [CFL83b], for each $i = 1, \ldots, \log(k)$ the sequence $[a_k]_B, \ldots, [a_1]_B$ is split up into consecutive blocks of size 2^i items each. Letting $[b_{i,j}]_B$ be the "sum" of the j'th such block, i.e. $[b_{i,j}]_B = \circ_{m=j\cdot 2^i+2^i}^{j\cdot 2^i+1} [a_m]_B$. There are $k-1$ of the "sums" $b_{i,j}$, namely one of length k, two of length $k/2$, up to $k/2$ of length two. The complexity for computing all of them in parallel is thus r rounds and $\sum_{i=1}^{\log(k)} 2^i C(k \cdot 2^{-i}) = \sum_{i=1}^{\log(k)} C(k) = \log(k) \cdot C(k)$ secure multiplications.

It is easy to see that each of the k $[b_i]_B$ can be computed as a "sum" of at most $\log(k)$ of the $[b_{i,j}]_B$'s. Doing this in parallel for all $[b_i]_B$'s costs another r rounds and at most $k \cdot C(\log(k))$ multiplications. Therefore the total complexity is upper bounded by $2r$ rounds and $\log(k) \cdot C(k) + k \cdot C(\log(k)) = 2c(k \log(k))$ secure multiplications. It is noted, that these bounds are conservative, e.g. any preprocessing of the two steps may be performed in parallel thus reducing round complexity.

11.2.2 Overall Bit-wise Addition Computation

The overall computation needed to add a bit-wise stored number and a public value is simple assuming that the carry bits may be computed. Writing $\texttt{Carries}([a]_B, a')$ for that computation resulting in $[c]_B = ([c_k]_q, \ldots, [c_1]_q)$, the computation seen as protocol 10 results in the sum of the inputs being bit-wise stored in $[d]_B$.

Protocol 10 Bit-wise addition based on $\texttt{Carries}([\cdot]_B, \cdot)$.

Input: $[a]_B = ([a_{k-1}]_q, \ldots, [a_0]_q)$ representing $a \in \mathbb{Z}_{2^k}$ and $a' \in \mathbb{Z}_{2^k}$ with binary representation $(a'_{k-1}, \ldots, a'_0)$.

Output: $(k+1)$-bit $[d]_B$ representing the sum of a and a'.

$[c]_B \leftarrow \texttt{Carries}([a]_B, a')$
$[d_0]_q \leftarrow [a_0]_q + a'_0 - 2 \cdot [c_1]_q$
for i = 1,. . . ,k-1 **do**
  $[d_i]_q \leftarrow [a_i]_q + a'_i + [c_i]_q - 2 \cdot [c_{i+1}]_q$
5: **end for**
$[d_k]_q \leftarrow [c_k]_q$
$[d]_B \leftarrow ([d_k]_q, \ldots, [d_0]_q)$

Correctness is immediate – $[d_i]_q$ computed in step 4 is clearly the i'th bit of the sum, while steps 2 and 6 are special cases of the same computation. Privacy is similarly trivial. If the computation of the carries is secure, privacy is ensured as no outputs are performed. The complexity is equivalent to that of the invocation of $\texttt{Carries}(\cdot, \cdot)$ presented below.

11.2.3 Computing Carry Bits

The well-known *carry set/propagate/kill* algorithm may be used to compute the carries. Let $\Sigma = \{S, P, K\}$. The algorithm uses an operator $\circ : \Sigma \times \Sigma \mapsto \Sigma$, defined by $S \circ x = S$ for all $x \in \Sigma$, $K \circ x = K$ for all $x \in \Sigma$, and $P \circ x = x$ for all $x \in \Sigma$. This is the carry-propagation operator, and it can be verified to be associative.

Describing first the intuition behind the algorithm, this is then translated to computation within $\mathcal{F}_{\mathbf{MPC-R}}$. Given two bit-wise-represented k-bit numbers $a = (a_{k-1}, \ldots, a_0)$ and $a' = (a'_{k-1}, \ldots, a'_0)$, for $i = 0, \ldots, k-1$, let $e_i = S$ iff a carry is set at position i (i.e. $a_i + a'_i = 2$); $e_i = P$ iff a carry is propagated at position i (i.e. $a_i + a'_i = 1$); and $e_i = K$ iff a carry is killed at position i, (i.e. $a_i + a'_i = 0$). It is straightforward to verify that $c_i = 1$ (the i'th carry bit is set)

if and only if

$$e_{i-1} \circ ... \circ e_0 \circ K = S.$$

The final K is included for clarity – if omitted the result could be P which is still not equal to S.

S, P, and K will be represented with bit vectors $(1,0,0),(0,1,0)$, and $(0,0,1) \in \{0,1\}^3$. The e_i's in this representation can easily be computed from the a_i's and a'_i's, hence, given a protocol for unbounded fan-in computation of the carry-propagation operator $\circ$ on this representation, the carries may be computed as described in protocol 11 using a $\texttt{post}_\circ(\cdots)$ operation. Privacy is

Protocol 11 Computing the carries of the addition of $[a]_B$ and a'.

Input: k-bit $[a]_B$ containing value $a \in \mathbb{Z}_{2^k}$ and $a' \in \mathbb{Z}_{2^k}$ with binary representation $(a'_{k-1}, \ldots, a'_0)$.

Output: $[c]_B = ([c_k]_q, \ldots, [c_1]_q)$ such that $[c_i]_q$ is the i'th carry bit in the addition of a and a'.

for i = 0,. . . ,k-1 **do**
  $[s_i]_q \leftarrow [a_i]_q \wedge a'_i$
  $[p_i]_q \leftarrow [a_i]_q \oplus a'_i$
  $[k_i]_q \leftarrow \neg([a_i]_q \vee a'_i)$
5: $[e_i]_B \leftarrow ([s_i]_q, [p_i]_q, [k_i]_q)$ ▷ $\text{val}([e_i]_B) \in \Sigma$
end for
$([f_{k-1}]_B, \ldots, [f_0]_B) \leftarrow \texttt{post}_\circ([e_{k-1}]_B, \ldots, [e_0]_B)$
for i = 0,. . . ,k-1 **do**
  $([s_i]_q, [p_i]_q, [k_i]_q) \leftarrow [f_i]_B$
10: $[c_{i+1}]_q \leftarrow [s_i]_q$
end for
$[c]_B \leftarrow ([c_k]_q, \ldots, [c_1]_q)$

immediate as no outputs are performed, while correctness follows readily from the above arguments.

Complexity is equivalent to that of step 7, i.e. $\texttt{post}_\circ(\cdots)$. All computation of the initial loop is linear as a' is public, while the remainder merely renames variables. Section 11.2.1 provided a solution to $\texttt{post}_\circ([e_{k-1}]_B, \ldots, [e_0]_B)$ requiring a constant number of rounds, assuming a constant round protocol for computing the $\circ$-operator with unbounded fan-in; the following section shows how to do this.

11.2.4 Unbounded Fan-in Carry Propagation

It remains to show how to compute $\circ_{i=1}^{k}[e_i]_B$ in a constant number of rounds, where $[e_i]_B$ is shorthand for a triple $([s_i]_q, [p_i]_q, [k_i]_q)$ of binary values, with $\text{val}([e_i]_B) \in \Sigma$ as defined above. Note that the kill-bits k_i should not be confused with k denoting the bit-length of the values. The computation – seen as protocol 12 – uses prefix-`AND` ($\texttt{pre}_\wedge(\cdot)$) in step 1. This is defined equivalently to prefix-`OR` of chapter 8, and implemented in the same complexity using De Morgan's Rule.

Protocol 12 Computing unbounded fan-in carry propagation

Input: $([e_k]_B, \ldots, [e_1]_B)$ such that $[e_i]_B = ([s_i]_q, [p_i]_q, [k_i]_q)$ and $\mathrm{val}([e_i]_B) \in \Sigma$.
Output: $[e]_B$ such that $\mathrm{val}([e]_B) = \circ_{i=1}^{k} \mathrm{val}([e_i]_B) \in \Sigma$.

$[\tilde{p}]_B \leftarrow \mathtt{pre}_\wedge([p_k]_q, \ldots, [p_1]_q)$
$[c]_q \leftarrow [k_k]_q + \sum_{i=1}^{k-1}([k_i]_q \wedge [\tilde{p}_{i+1}]_q)$
$[b]_q \leftarrow [\tilde{p}_1]_q$ $\quad \triangleright\ \mathrm{val}([\tilde{p}_1]_q) = \bigwedge_{i=1}^{k} \mathrm{val}([p_i]_q)$
$[a]_q \leftarrow 1 - [b]_q - [c]_q$
5: $[e]_B \leftarrow ([a]_q, [b]_q, [c]_q)$

The overall idea of the computation is to determine if propagation occurs – i.e. if all the $[e_i]_B$ equal P – and otherwise to determine the most significant non-propagating $[e_i]_B$ and extract the kill-bit from this. Based on these, the result is immediate. It should be clear that $\mathrm{val}([b]_q) = 1$ (a propagate) iff $p_i = 1$ for $i = 1, \ldots, k$, making $[b]_q$ the final propagate-bit. Furthermore, $\mathrm{val}([c]_q) = 1$ (a kill) iff there exists some i such that $\mathrm{val}([k_i]_q) = 1$ and $\mathrm{val}([p_k]_q) = 1, \ldots, \mathrm{val}([p_{i+1}]_q) = 1$. Step 2 is in fact

$$[c]_q \leftarrow [k_k]_q \vee \bigvee_{i=1}^{k-1}([k_i]_q \wedge [\tilde{p}_{i+1}]_q),$$

however, since k_i and p_i are never 1 simultaneously it can be seen that at most one of the expressions $[k_i]_q \wedge [\tilde{p}_{i+1}]_q$ equals 1. Thus, the `OR` may be computed through a sum. By the representation it follows that the set-bit is correct.

The computation is constructed entirely from existing commands and no variables are output, thus no information is leaked. Complexity is equivalent to one invocation of a k-ary prefix-`AND` plus one round of $k-1$ multiplications for the `AND`'s of step 2. The rest is simply costless additions or renaming of existing variables. Overall, the computation therefore requires $13k+k-1$ multiplications (rounded off to $14k$ for simplicity) in $12+1 = 13$ rounds, four of which may be viewed as preprocessing.

11.3 Concluding Remarks

Section 11.2 provides a constant round computation perfectly UC-realising bit-wise addition, (**let** $[s]_B \leftarrow \mathtt{Add}([a]_B, a')$), in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. By section 11.1 this implies a perfect UC-realisation of (**let** $[a]_B \leftarrow \mathtt{bits}([a]_q)$) in the same hybrid model.

Concerning the complexity of bit-wise addition, recall that this was equivalent to that of $\mathtt{post}_\circ$, i.e. $2\log(k) \cdot C(k)$ multiplications, where $C(k) = 14k$, i.e. $28k\log(k)$ multiplications all in all. Further, the two unbounded fan-in carry propagations needed, imply 26 rounds overall. As preprocessing parallelises this may be reduced to 22 rounds. Moreover, disregarding preprocessing altogether 18 rounds are required online.

Theorem 11.1 *An $\mathcal{O}(1)$-round command for addition of a bit-wise stored k-bit value and a public (k-bit) constant – providing a bit-wise storing of the result – may be perfectly UC-realised in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model using $\mathcal{O}(k\log(k))$ secure multiplications.*

Based on theorem 11.1, the computation of section 11.1 is easily seen to perfectly UC-realise ($\mathbf{let}\ [a]_B \leftarrow \mathtt{bits}([a]_q)$) thereby providing bit-extraction in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. Recalling that the complexity of the realising computation is equivalent to $29\ell + 13$ multiplications plus two bit-wise additions (one of size ℓ and one of size $\ell + 1$), the overall complexity is seen to be (approximately)

$$(29\ell + 13) + 28\ell\log(\ell) + 28(\ell + 1)\log(\ell + 1) \approx 56\ell\log(\ell) + 30\ell$$

multiplications. As for rounds, the analysis is more involved – though clearly only a constant number is required. Noting that the computation of $[\bar{o}]_q$ and $[d']_B$ parallelise (with the latter being the most expensive) and constructing the full computation from scratch, thus allowing preprocessing to be parallelised the requirement is reduced to 51 rounds. The generation of $[r]_B$ dominates preprocessing (13 rounds), while the two (now online) additions require 18 rounds each. Finally, two rounds are required for outputting the masked input and computing the final result. The result is summarised in theorem 11.2 – the main result of this chapter.

Theorem 11.2 (Bit-extraction) *$\mathcal{F}_{\mathbf{MPC-R}}$ extended with a constant round command ($\mathbf{let}\ [a]_B \leftarrow \mathtt{bits}([a]_q)$) for bit-extraction may be perfectly UC-realised in the basic $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model. The realising computation may be performed using $\mathcal{O}(\ell\log(\ell))$ secure multiplications.*

Apart from complexity the result generalises to computation over rings $\mathbb{Z}_m$ when the hybrid model provides the required primitives. Naturally, if the complexities of the primitives are worse – such as is the case for bit-generation when considering multi-party computation based on Paillier encryption – the complexity of the constructed protocol will be worse than stated above.

The solution provided is not optimal in the sense that $\mathcal{O}(\ell\log(\ell))$ multiplications are required for performing an addition. Ignoring rounds, it is simple to construct a linear solution (in terms of multiplications) by performing the carry propagation one step at a time. Indeed, as carry propagation is associative, round complexity may be reduced to logarithmic: $\mathtt{post}_\circ$ may be computed using divide an conquer. Pairing each bit-position with its neighbour, and applying $\circ$ in parallel reduces the problem to one of half size. Moreover, from a solution of this, a full solution is easily constructed. Though the solution is not obtained in constant round, the number of rounds is *logarithmic in the bit-length of the prime q*. For most practical purposes this should suffice.

It does not seem reasonable to expect a bit-extraction protocol with sub-linear complexity, as the output contains a linear number of secret variables. However, this still leaves a gap of a factor of $\log(\ell)$ multiplications to the solution provided. Overcoming this gap – preferably by presenting a constant rounds bit-extraction – is an open problem.

11.3.1 Applications

Concluding the chapter, multi-party computation for specific tasks utilising the bit-decomposition command is considered. All applications are constant-rounds within $\mathcal{F}_{\mathbf{MPC-R}}$ and may be statistically UC-realised, though this is not explicitly done. It is stressed that even though the number of secure multiplications is always polynomial in $\ell = \lceil \log(q) \rceil$ and the number of rounds is constant not much effort has gone into optimising the running time and round complexity. The computations have been included in order to demonstrate feasibility.

Private exponentiation

Consider the exponentiation function $\exp : \mathbb{Z}_q \times \mathbb{Z}_q \mapsto \mathbb{Z}_q$ is given by $\exp(x, a) = (x^a \bmod q) \in \mathbb{Z}_q$. Dealing first with the case where the exponent a is publicly known and $[x]_q$ is a variable of $\mathcal{F}_{\mathbf{MPC-R}}$, the goal is to set variable $[x^a]_q$ to $\mathrm{val}([x]_q)^a$.

Assume that the functionality may be asked to generate a uniformly random, non-zero value and store it as $[r]_q$ along with its ath power which is stored as $[r^a]_q$. It will later be demonstrated how to implement such a protocol in constant rounds. Securely computing the exponentiation function is then straightforward (using the Bar-Ilan and Beaver [BB89] inversion trick): First the parties generate a random $[r]_q$ along with $[r^a]_q$, then they output $[x]_q \cdot [r]_q$ obtaining $\mathrm{val}([x]_q) \cdot \mathrm{val}([r]_q) \in \mathbb{Z}_q$. The parties locally compute

$$y = (\mathrm{val}([x]_q) \cdot \mathrm{val}([r]_q))^a = \mathrm{val}([x]_q)^a \cdot \mathrm{val}([r]_q)^a$$

and assign $y \cdot \left([r^a]_q\right)^{-1}$ to $[x^a]_q$, where element inversion is realised through the Bar-Ilan and Beaver inversion protocol. It is easy to see that this protocol is private as long as $\mathrm{val}([x]_q) \neq 0$, which may be handled using the exception trick of chapter 9.

It remains to realise the generation of a random, non-zero $[r]_q$ together with its ath power $[r^a]_q$. In the honest-but-curious model this is done by letting each player P_i locally input a random non-zero value together with its ath power,

$$P_i : \ [r_i]_q \leftarrow ?; \quad P_i : \ [r_i^a]_q \leftarrow ?.$$

Now compute $[r]_q$ as the product of all $[r_i]_q$ and $[r^a]_q$ as the product of all $[r_i^a]_q$.

The computation may be made robust against active adversaries using a cut-and-choose technique: In addition to $[r_i]_q$, $[r_i^a]_q$, party P_i inputs $[s_i]_q$ and $[s_i^a]_q$. The players then generate a public, but uniformly random bit b and compute and output $([s_i]_q, [s_i^a]_q)$ or $([s_i]_q \cdot [r_i]_q, [s_i^a]_q \cdot [r_i^a]_q)$, according to the value of b and verify that the second number is the first raised to the public a. This can be repeated in parallel an appropriate number of times.

Consider now the case where the exponent $[a]_q$ is secret, i.e. the parties have $[x]_q$ and $[a]_q$ and want to compute $[x^a]_q$ such that its value is $\mathrm{val}([x]_q)^{\mathrm{val}([a]_q)}$. This case can be reduced to the previous one, run the bit decomposition protocol to obtain the bits of $[a]_q$, $[a]_B = ([a_{\ell-1}]_q, \ldots, [a_0]_q)$ of the exponent. Then, using

unbounded fan-in multiplication, $[x^a]_q$ may now be computed via the equation

$$\begin{aligned}\mathrm{val}([x]_q)^{\mathrm{val}([a]_q)} &= \mathrm{val}([x]_q)^{\sum_{i=0}^{\ell-1} 2^i \mathrm{val}([a_i]_q)} \\ &= \prod_{i=0}^{\ell-1} \mathrm{val}([x]_q)^{2^i \mathrm{val}([a_i]_q)} \\ &= \prod_{i=0}^{\ell-1} \left(\mathrm{val}([a_i]_q) \mathbin{?} \mathrm{val}([x_i]_q)^{2^i} : 1\right) \in \mathbb{Z}_q\end{aligned} \tag{11.1}$$

where the ℓ powers of $[x]_q$ may be computed in parallel using the above exponentiation protocol, again using the exception trick to avoid $\mathrm{val}([x]_q) = 0$.

Modulo reduction

Let $m \in \{2, 3, \ldots, q-1\}$ be a public integer, and consider the "modulo m" function, $\mathrm{mod}_m : \mathbb{Z}_q \mapsto \mathbb{Z}_q, x \mapsto (x \bmod m) \in \{0, 1, \ldots, m-1\}$. It will be demonstrated how to privately compute this in constant-rounds, i.e. given m and variable $[x]_q$, the goal is to compute $\mathrm{mod}_m(\mathrm{val}([x]_q))$ securely storing it as $[\mathrm{mod}_m(x)]_q$. This directly implies an integer division protocol, subtract and multiply by the inverse of m.

The players first extract $[x]_B = ([x_{\ell-1}]_q, \ldots, [x_0]_q)$. Note that if m is a power of 2, $m = 2^a$, then $[\mathrm{mod}_m(x)]_q$ is immediate:

$$[\mathrm{mod}_m(x)]_q \leftarrow \sum_{i=0}^{a-1} 2^i \cdot [x_i]_q.$$

If m is not a power of two, then assume that $\ell \cdot m < q$ and consider $y = \sum_{i=0}^{\ell-1} \mathrm{val}([x_i]_q) \cdot (2^i \bmod m)$ over the integers. Clearly

$$\mathrm{val}([x]_q) \bmod m = (\sum_{i=0}^{\ell-1} \mathrm{val}([x_i]_q) \cdot (2^i \bmod m)) \bmod m = y - tm$$

for some integer $t \in \{0, 1, \ldots, \ell-1\}$. Computing t securely (storing it as $[t]_q$) amounts to performing $\ell - 1$ comparisons

$$y \overset{?}{\geq} m, 2m, \ldots, (\ell-1)m$$

in parallel and setting $[t]_q$ to the sum of results. y may be computed as an element of $\mathbb{Z}_q$ due to the bound on $\ell \cdot m$. The result is then

$$\left(\sum_{i=0}^{\ell-1} [x_i]_q \cdot (2^i \bmod m)\right) - m \cdot [t]_q$$

computed over $\mathbb{Z}_q$.

If $\ell \cdot m \geq q$ the computation of y is not possible. However, in this case it can be noted that

$$\mathrm{val}([x]_q) \bmod m = \mathrm{val}([x]_q) - tm$$

for some $t \in \{0, 1, \ldots, \ell-1\}$. t may be determined securely in the same manner as for y above.

Private modulo reduction

Let $[x]_q$ and $[m]_q$ be given, where $\mathrm{val}([m]_q)$ is of a known, bounded bit-length $\ell_0 \ll \ell$. Again the goal is to compute $[x \bmod m]_q$. There already exists an efficient protocol to approximate $\mathrm{val}([x]_q) \bmod \mathrm{val}([m]_q)$ due to [ACS02] but it does not run in a constant number of rounds. Combining the techniques above with the results of [ACS02] and [KLM05] (the latter one approximates the fractional part of $1/m$ by a Taylor polynomial), an approximation of $\mathrm{val}([x]_q) \bmod \mathrm{val}([m]_q)$ may be obtained efficiently and in constant-rounds. The exact result may then be obtained by executing an appropriate number of comparisons ensuring that the result lies in the interval $[0, m-1]$.

Along with exponentiation above, this allows a protocol which given $[x]_q$, $[a]_q$, and $[m]_q$ computes $\mathrm{val}([x]_q)^{\mathrm{val}([a]_q)} \bmod \mathrm{val}([m]_q)$ securely. Restricting the focus to prime $[m]_q$, compute first

$$[x \bmod m]_q \leftarrow [x]_q \bmod [m]_q; \quad [a \bmod m-1]_q \leftarrow [a]_q \bmod ([m]_q - 1).$$

Assuming that q is sufficiently large (of bit-length $\ell > \ell_0^2$) no overflow occurs in the product of equation 11.1, thus the exponentiation may be viewed as having occurred over the integers. Reducing this modulo $[m]_q$ provides the desired result.

Part III

Applications and High-level Protocols

Chapter 12

On Applications

The third part of this thesis concerns itself with applications of multi-party computation. It consists of high-level tasks which may be performed – and may benefit from – using cryptographic protocols. This includes real-world motivation. Moreover, though the protocols considered have immediate uses, they may also be seen as "library functions" which may be applied in other even more complex settings.

One area which benefits in particular from MPC is economics. In general, modern economics focuses on providing the right incentives to ensure desired coordination of available resources. A number of parties wish to redistribute such resources – money, goods, information, ... – however, this should be done on the basis of private information of each of the entities – information that they will not disclose to each other. Through a mediator – a trusted third party – the optimal coordination may be obtained. All parties of the system provide their inputs (their preferences, e.g. their view of the value of an item) and the mediator computes the optimal redistribution of resources based on these (e.g. letting the one who values the item most purchase it – providing the seller with maximal profit). As the mediator is trustworthy, this does not release excess information.

In practice, however, such a mediator may not be readily available. The parties in question may not be able to agree on a trustworthy third party. Moreover, even if an acceptable third party can be found, the parties may still be disinclined to provide entirely truthful inputs, if the trust is not 100% – naturally this results in sub-optimal coordination. It may also be costly to obtain the services of such an entity, as the third party must be counter-bribed. MPC provides what is essentially a trusted party, implemented through cryptographic protocols. This allows the parties to dispose of the "third party." Alternatively, the techniques may be used to distribute trust between multiple third parties, thereby lessening the required trust – and the potential for misuse of data. This could be modelled by altering $\mathcal{F}_{\mathbf{MPC}}$ to contain two types of parties: *Inputters*, who only provide inputs, and *computers* who authorise inputs and provide computational commands to the functionality (e.g. performing the role of the auctioneer).

From the point of view of economics there are additional issues. If optimal coordination is to occur, it must be required that truthful inputs are provided –

this is *not* ensured automatically as individual parties may benefit from "lying." As an example consider a sealed-bid first price auction, i.e. an auction where the highest bidder wins, paying her bid. If it is expected that all others perceive the value of the item in question much lower than one self, then bidding a lower amount than the actual perceived value may result in a better deal. Though the design of economically sound mechanisms – providing incentives for truth-telling – is entirely out of the scope of this work, the point is noted in order to stress that sound applications are considered (apart from the first price auction).

12.1 The Model of Computation

The model of computation will be represented by an additional functionality $\mathcal{F}_{\mathbf{EMPC}}$. This represents an *extended* version of the basic $\mathcal{F}_{\mathbf{MPC-R}}$ – i.e. a copy of $\mathcal{F}_{\mathbf{MPC-R}}$, but extended with the commands UC-realised in part II. This consists of the comparisons (equality and inequality) as well as bit-extraction. Moreover, the constant round primitives of chapter 8 will be used as well. Similar to part II, new commands will be introduced and added to $\mathcal{F}_{\mathbf{EMPC}}$ thereby extending it underway. Doing this stresses the fact that though applications are considered, the protocols may also be viewed as library functions which may be used in other contexts than their original.

The main primitive used in the remaining chapters is the inequality test, although testing equality is also required. It is stressed that although candidates were considered in chapters 9 and 10, *any* realisations of the commands may be used. When considering complexity in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model, this is equivalent to the MPC functionalities of the previous chapters except that not only will multiplications be counted, so will comparisons.

For simplicity the complexity of the two comparisons will be considered equivalent. Under big-$\mathcal{O}$-analysis no advantage is lost in any of the applications considered. Thus, it will be assumed that c_r rounds and c_m multiplications are required for equality and inequality testing.[1] The former is assumed constant, while the latter may depend on e.g. the bit-length of the values in question. Additional motivation for equating the two is found when *perfect* realisations are considered; in this case *both* require $\mathcal{O}(\log(q))$ multiplications when computing within $\mathbb{Z}_q$.

The perfect UC-realisation of the applications constructed using $\mathcal{F}_{\mathbf{EMPC}}$ in the *basic* $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model provides a *strong* security guarantee on the protocols constructed below – perfect privacy (except as violated by the protocols themselves) is obtainable through only arithmetic and random element generation. This then provides a statistical UC-realisation in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model constructed in part I – this even ensures perfect privacy, as the environments possibility of distinguishing comes through failed random generation, at which point the parties may choose to abort.

In all applications, it is assumed that $\mathcal{F}_{\mathbf{EMPC}}$ essentially provides compu-

[1] Complex applications requiring multiple comparisons may of course benefit from parallelising preprocessing. c_r may be seen as implicitly excluding preprocessing, which must then manually be included in the final rounds-count.

tation over the integers. For this reason reference to the underlying ring will be removed from the notation – i.e. secret variables will be written $[x]$. However, in general, it will be assumed that these are simulated as elements of $\mathbb{Z}_q$ for some prime q chosen sufficiently large beforehand thereby ensuring that overflow does not occur. (In the descriptions of protocols the required size of q will always be considered based on the initial inputs.) This assumption is made for simplicity – non-prime moduli are generally possible, with difficulties being noted.

It will generally be assumed that inputs for the relevant computations are present and well-formed. Active adversaries providing junk-input are not ignored, however, input verification – if required – is generally simple thus it is not considered in detail. A typical restriction is that an input (of $\mathbb{Z}_q$) must be of some known, fixed bit-length. This may be verified through a comparison.

12.2 Added Notation for Arrays

All of the applications to be described consider related data – i.e. easy specification of multiple related values is required. Thus, notation for arrays of secret variables is introduced. The overall idea is similar to that of bit-wise stored values; the sole difference being that arbitrary values may be contained in the arrays. However, notation is written differently to distinguish it clearly from the earlier. Variables representing arrays - i.e. multiple related variables of $\mathcal{F}_{\mathbf{EMPC}}$ – will be written in boldface using capital letters,

$$[\mathbf{A}] = \big([a_1], [a_2], \ldots, [a_k]\big).$$

For $i \in \{1, 2, \ldots, k\}$ indexing is written $[\mathbf{A}](i)$ meaning $[a_i]$. This notation will also be used in assignments,

$$[\mathbf{A}](i) \leftarrow [x].$$

Finally, the length of a shared array – k above – will be written as $[\mathbf{A}].\mathrm{len}$.

It will also be required to index based on variables of $\mathcal{F}_{\mathbf{EMPC}}$, i.e. to evaluate expressions of the form

$$[\mathbf{A}]([i]).$$

To facilitate *secret indexing* yet another "type" is added. A secret index i will be stored as essentially unary counter; an array consisting of all 0's except for the i'th position, which contains a 1. As an example, consider the index "3" out of four possible:

$$\big([0], [0], [1], [0]\big).$$

Variables of such indexes will be written in boldface using lower case, denoting that in essence it is an array. Performing an indexing operation is now done by computing a sum of products

$$[\mathbf{A}]([\mathbf{i}]) = \sum_{j=1}^{[\mathbf{A}].\mathrm{len}} [\mathbf{A}](j) \cdot [\mathbf{i}](j).$$

This is clearly correct, preserves privacy, and requires $[\mathbf{A}].\mathrm{len}$ secure multiplications, all of which may be performed in parallel, i.e. in a single round.

Assignments using indexes stored in this manner are also possible. The statement $[\mathbf{A}]([\mathbf{i}]) \leftarrow [x]$ is equivalent to updating *every* entry of $[\mathbf{A}]$ as

$$[\mathbf{A}](j) \leftarrow [\mathbf{i}](j) \ ? \ [x] : [\mathbf{A}](j).$$

As indexing, this requires $[\mathbf{A}].\text{len}$ multiplications which may all be performed in a single round.

For notational convenience, indexes of length strictly less than the full length of the array may be used for indexing. It will be implicitly assumed that the index-array is extended with 0-entries in order to obtain the full length required. Conversely, using a *longer* index implies truncating the index (which may leave it all 0). The integer value of an index may be computed as $\sum_{j=1}^{[\mathbf{i}].\text{len}} j \cdot [\mathbf{i}](j)$. Continuing the notational abuse of $\text{val}(\cdot)$, $\text{val}([\mathbf{i}])$ will be used to denote the integer value of the index

$$\sum_{i=1}^{[\mathbf{i}].\text{len}} j \cdot \text{val}([\mathbf{i}](j)).$$

Moreover, $\text{val}([\mathbf{A}]([\mathbf{i}]))$ will be used to denote the value of the entry of $[\mathbf{A}]$ specified by $[\mathbf{i}]$, i.e. $\text{val}([\mathbf{A}](\text{val}([\mathbf{i}])))$.

Note that general, secure array indexing is possible, i.e. computing $[\mathbf{A}]([i])$ where $[i]$ is some arbitrary value, though at most $[\mathbf{A}].\text{len}$. The index representing the value of the variable must simply be computed.

Concluding, the notation introduced for arrays will also be used for secret matrices. Indexing is then done using two variables with $[\mathbf{A}](r, c)$ denoting the entry in the r'th row, c'th column. The r'th row of $[\mathbf{A}]$ will be written $[\mathbf{A}](r, \cdot)$, similarly $[\mathbf{A}](\cdot, c)$ will denote the c'th column. Both can be computed from secret indexes by viewing the columns or rows as separate arrays. General, secure indexing into $[\mathbf{A}]$ is immediately obtainable as well.

12.3 An Overview of Part III

Three high-level applications of multi-party computation are considered in this work,[2] two of which are auctions. MPC (and cryptography in general) is imminently suited as tools for constructing and providing security in auctions. Thus a large body of work on different auctions in various settings exists – a few examples include [FR95, NPS99, Bra01, BS05]. Auctions contain numerous privacy issues as well as issues of integrity and availability. As MPC is the focus here, privacy of inputs and correctness of the result will be the main concerns. Additional requirements e.g. non-repudiation of bids are ignored – for the purposes here it suffices to perform the required computation.

The auctions considered are the "standard" first price and second price auctions in which a single item is sold. These are realised in chapter 13. Chapter 14 then considers a double auction – an exchange in which multiple buyers and sellers compute a so called market clearing price, the current price at which goods are traded.

[2] Four if the GCD-computation of appendix A is considered as well.

The final application considered is the solving of linear programming problems. (Coordination) problems of economics may often be phrased as linear programs, thus an MPC tool for solving such problems is the gateway to a large host of applications. Some of these applications are immediate, whereas others may require initial computation on the inputs in order to construct the linear program to solve. The protocol handles the second setting, and thereby implicitly also the first. Though the motivation arises from economics, linear programs may be used elsewhere, thus naturally other applications may exist in other areas.

Chapter 13

Simple Auctions – Determining the Maximal Value

This chapter considers two very basic types of auctions – sealed-bid first- and second price auctions. Sealed-bid simply implies that all parties provide one *secret* bid. In both cases the highest bidder wins, the price to be paid differs though. As noted in chapter 12, much work on auctions and multi-party computation exists; references will not be repeated, however, it is noted that such auction types have been considered. In this work, they are included as immediate applications of a command for securely determining the maximal of multiple values, and to demonstrate UC-realisation in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model. To the authors knowledge, the non-trivial of the protocols for computing the maximal below have not previously been considered in a MPC-setting, though they are simply the direct application of results of parallel algorithms.

Ideal functionalities for the auctions are realised based on a computation determining the maximal of multiple variables of $\mathcal{F}_{\mathbf{EMPC}}$. This primitive is important not only for the current chapter, indeed the present description is an extension of an appendix of an unpublished paper by the author [Tofb] included as a later chapter. As such, the task is a powerful primitive with multiple uses in its own right, though motivation in the form of auctions has been added.

This chapter is split up into two parts, sections 13.1 and 13.2. The bulk of the work is contained in the former, in the form of the ideal functionality for determining maximal values along with multiple realisations of it (allowing round complexity/communication complexity trade-offs). The latter then uses this functionality in the construction of the first and second price auctions. The multiple realisations determining the maximal immediately imply multiple realisations of the auctions in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model.

13.1 Determining the Maximal Value

Computing the maximal of multiple values is best phrased through arrays of secret variables. Not only does this ease the description of the input, it also allows the "identity" of the maximal value to be determined rather than the value itself, i.e. its index. From this the maximal value is easily determined.

As in part II above, formalisation of the extension of $\mathcal{F}_{\mathbf{EMPC}}$ is done by

adding an additional command (**let** $[\mathbf{i}] \leftarrow \texttt{MAX}([\mathbf{A}])$). This is then realised in the basic $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model. The effect of the maximal-command is to set $[\mathbf{i}]$ to the index of the maximal, i.e. such that

$$\mathrm{val}([\mathbf{A}]([\mathbf{i}])) = \max_{1 \leq j \leq [\mathbf{A}].\mathrm{len}} \mathrm{val}([\mathbf{A}](j)).$$

In case of multiple candidates, the *minimal* one will be used. Upon receiving the command in round R from all honest parties, $\mathcal{F}_{\mathbf{EMPC}}$ determines the index of the maximal, and notifies the adversary that the command is being executed. Then in round $R + r([\mathbf{A}].\mathrm{len}) - 1$ $[\mathbf{i}]$ is set, and the adversary and parties are notified that the command has terminated.

Here r is a function mapping array-lengths to rounds; this approach is taken as multiple realisations with different round complexities will be considered. Initially the trivial logarithmic-rounds solution is noted in section 13.1.1. A constant-rounds solution is then constructed in section 13.1.2, though at the cost of communication complexity. Section 13.1.3 then describes a loglog-rounds realisation using only a linear number of comparisons based on the two previous solutions. All these protocols are *direct applications* of results from parallel algorithms, see any text-book e.g. [Jáj92].

13.1.1 The Obvious Solution – log-rounds

The obvious solution is to consider a binary tree over the array, comparing pairwise and discarding half of the values at every iteration. For simplicity assume that $k = [\mathbf{A}].\mathrm{len}$ is a power of two. From the solution to the sub-problem of half size and the result of the comparisons, the index is easily constructed. The full (recursive) computation is seen as protocol 13. Privacy is immediate as

Protocol 13 $\max_{\mathcal{O}(\log(k))}(\cdot)$ – Computing the maximal in $\mathcal{O}(\log(k))$ rounds.

Input: $[\mathbf{A}]$ such that $k = [\mathbf{A}].\mathrm{len}$ is a two-power.
Output: $[\mathbf{i}]$ such that $\mathrm{val}([\mathbf{A}]([\mathbf{i}]))$ is maximal.

```
if [A].len ?=1 then
    [i] ← (1)
else
    for j = 1,...,k/2 do
5:      [B](j) ← [A](2j − 1) ?< [A](2j)
        [A′](j) ← [B](j) ? [A](2j) : [A](2j − 1)
    end for
    [i′] ← max_{O(log(k))}([A′])
    for j = 1,...,k/2 do
10:     [i](2j) ← [B](j) · [i′](j)
        [i](2j − 1) ← [i′](j) − [i](2j)          ▷ (1 − [B](j)) · [i′](j)
    end for
end if
```

computation occurs in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model and no variables are output.

For complexity, it is easily verified that $k-1$ comparisons and $2k-2$ multiplications are required. This is optimal in the sense that the maximal cannot be determined using less than a linear number of comparisons. Noting that the iterations of the two loops parallelise perfectly, round complexity becomes equivalent to that of a comparison plus two. The number of recursive calls is $\log(k)$, implying an overall round complexity of $\log(k)(c_r+2)$.

13.1.2 The Sub-optimal Constant Rounds Solution

Though a logarithmic round complexity is not horrible it is possible to reduce it quite easily. Protocol 14 provides a constant round solution, the intuition of which is to compare all entries of $[\mathbf{A}]$ to all others. The results are then stored in a $k \times k$ matrix, naturally the desired index is the one which is associated with the (unique) column containing all 1's – i.e. the one which was greater than all others. The details are seen as protocol 14, by the above intuition, clearly $[\mathbf{i}]$ computed in step 11 is the desired index.

Protocol 14 $\max_{\mathcal{O}(1)}(\cdot)$ – Computing the maximal element in $\mathcal{O}(1)$ rounds.

Input: $[\mathbf{A}]$.
Output: $[\mathbf{i}]$ such that $\mathrm{val}([\mathbf{A}]([\mathbf{i}]))$ is maximal.

```
    for j = 1,...,k do
        [B](j,j) ← 1
    end for
    for j = 1,...,k do
 5:     for j' = j+1,...,k do
            [B](j,j') ← [A](j) ?> [A](j')
            [B](j',j) ← 1 − [B](j,j')
        end for
    end for
10: for j = 1,...,k do
        [i](j) ← ⋀_{j'=1}^{k} ([B](j',j))
    end for
```

As above, privacy is ensured as computation occurs in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model and nothing is output. $(k^2-k)/2$ comparisons are required as well as k k-ary fan-in AND-gates. Assuming that the prime q is greater than k – which is not an unreasonable assumption – the AND-gates may be realised using equality-tests, i.e. complexity is $(k^2+k)/2$ comparisons, this is rounded up to k^2.[1] All iterations of the loops parallelise, thus the comparisons of step 6 may all be performed in parallel. This is also the case for the AND-gates, i.e. $2c_r$ rounds are required all in all.

It is noted that through a small variation, the above protocol may be used to solve another problem also considered in literature, namely t'th rank. The objective is to compute the (index of the) t'th greatest element. This is triv-

[1] This solution is used for brevity, it is noted that complexity may be improved by using the symmetric Boolean function approach of chapter 8.

ially obtained by noting that the desired index is that of the unique column containing exactly $k+1-t$ 1's.

13.1.3 Optimal Communication, loglog Rounds

While the logarithmic round complexity of the obvious solution was not entirely acceptable, the quadratic number of comparisons required for the constant rounds solution above will be unacceptable in most circumstances. However, the constant rounds solution may be used to construct a *linear* solution running in only $\mathcal{O}(\text{loglog}(k))$ rounds. This is done in two steps, first an $\mathcal{O}(\text{loglog}(k))$ rounds solution using $\mathcal{O}(k \cdot \text{loglog}(k))$ comparisons and multiplications is constructed. This is then used in conjunction with protocol 13 to construct a linear one.

The initial solution is seen as protocol 15, its intuition is as follows: Assume that $[\mathbf{A}].\text{len} = k$ is square, $k = \lambda^2$, and divide $[\mathbf{A}]$ into λ sub-arrays of length λ each. Apply recursion, thereby obtaining a secret index for each sub-array denoting the entry of the maximal in those. Insert these λ candidates for the maximal into an array and apply the constant rounds solution to this. Based on all $\lambda+1$ indexes the full index is constructed – for $h, j \in \{1, 2, \ldots, \lambda+1\}$ entry $(h-1) \cdot \lambda + j$ contains the maximal value iff the overall maximal was in the h'th sub-array *and* the j'th entry of that sub-array was its maximal. As

Protocol 15 $\max_{\mathcal{O}(\text{loglog})}(\cdot)$ – Computing the maximal in $\mathcal{O}(\text{loglog}(k))$ rounds.

Input: $[\mathbf{A}]$ of square length, $[\mathbf{A}].\text{len} = k = \lambda^2$.
Output: $[\mathbf{i}]$ such that $\text{val}([\mathbf{A}]([\mathbf{i}]))$ is maximal.

```
    for h = 1,...,λ do
        for j = 1,...,λ do
            [A_h](j) ← [A]((h − 1) · λ + j)
        end for
 5: end for
    for h = 1,...,λ do
        [i_h] ← max_O(loglog)([A_h])
        [A′](h) ← [A_h]([i_h])
    end for
10: [i′] ← max_O(1)([A′])
    for h = 1,...,λ do
        for j = 1,...,λ do
            [i]((h − 1) · λ + j) ← [i′](h) · [i_h](j)
        end for
15: end for
```

always privacy is immediate.

Disregarding the recursive calls λ indexing-operations of length λ are performed in the second loop. In addition to this a constant rounds solution is applied to a problem of size λ, while λ^2 multiplications are used to construct the final solution. Overall complexity is k comparisons and $2k$ multiplications (i.e. it is linear) in two rounds more than the constant round solution requires

as all loops parallelise.

Including recursion, it may be verified that $\text{loglog}(k)$ iterations are required, thus overall $\text{loglog}(k) \cdot (2c_r + 2)$ rounds are required. Moreover, as communication complexity is linear except for recursion, overall $k \cdot \text{loglog}(k)$ comparisons and $2k \cdot \text{loglog}(k)$ multiplications are required in all.

The above is then used to construct a *linear* loglog-rounds solution through *accelerated cascading*: Performing $\lceil \text{logloglog}(k) \rceil$ iterations of the logarithmic solution, protocol 13 reduces the problem by a factor of $\text{loglog}(k)$. From there the $\mathcal{O}(k \cdot \text{loglog}(k))$ protocol is applied, and as the size of the problem has been reduced the overall complexity is linear in k. For more details see [Jáj92] sections 2.6.2 and 2.6.3.

Considering a more precise analysis, both halves require less than k comparisons and $2k$ multiplications, thus overall less than $2k$ comparisons and $4k$ multiplications are performed. All in all this requires

$$\text{logloglog}(k) \cdot (c_r + 2) + \text{loglog}(k/\text{loglog}(k)) \cdot (2c_r + 2)$$

rounds; for readability the bounding expression

$$\text{loglog}(k) \cdot (3c_r + 4)$$

will be used instead.

Lemma 13.1 *$\mathcal{F}_{\mathbf{EMPC}}$ extended with a command* (**let** [**i**] $\leftarrow$ `MAX`([**A**])) *for securely computing the index of a maximal entry of* [**A**] *may be perfectly UC-realised in the basic $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model. The realisation can be done using $\mathcal{O}([\mathbf{A}].\mathit{len})$ secure comparisons and multiplications in $\mathcal{O}(\text{loglog}([\mathbf{A}].\mathit{len}))$ rounds.*

A final observation is that all of the above protocols may be used with *any* comparison operator on *arbitrary* data, as long as there exists some notion of "maximal." Naturally, when an alternative comparison operator is considered, complexity will depend on this. An immediate variation possible is the computation of the entry of the *minimal*.

13.2 Realising the Simple Auctions

Having introduced computation securely determining the index of the maximal entry of an array, simple auctions may be constructed easily. A sealed-bid first price auction is one where a single item is sold. All bidders provide their bid – the amount they are willing to pay – and the highest bidding party wins. As the name suggests, bidder pays his given price. As noted in chapter 12 this type of auction does not provide incentives to give truthful bids. This is in contrast to sealed-bid second price auctions, where only the second highest bid is paid. Intuitively, nothing may be gained by not bidding the perceived value of the item – the consequences will be either that the auction is lost, but could have been won at an acceptable price, or that the auction is won, but an unacceptable price must be paid.

Ideal functionalities for both auctions are simple; due to their similarities they will be described together. All parties provide their bid – a single value. The adversary is allowed to corrupt parties and alter their inputs, but receive no information at all on the bids of the honest parties. When authorised by the adversary, the ideal functionality determines the winner (the highest bidder) and the price (the highest or second highest bid depending on the type of auction) and sends this first to the adversary and then to all parties. The two functionalities will be denoted $\mathcal{F}_{\mathbf{1st-price}}$ and $\mathcal{F}_{\mathbf{2nd-price}}$

Perfect UC-realisation of $\mathcal{F}_{\mathbf{1st-price}}$ is trivial in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model extended with a command for computing the index of the maximal of an array. All parties input their bids, which are stored in an array $[\mathbf{B}]$ – party P_j's bid at entry $[\mathbf{B}](j)$ The index of the maximal – w – is computed and output, this is the identity of the winner. Following this the winning bid, $[\mathbf{B}](w)$, is output.

The realisation of $\mathcal{F}_{\mathbf{2nd-price}}$ is marginally more complex. The parties provide bids and determine the identity of the winner in in the same manner. The winning bid is then removed from the array of bids and the maximal of the remaining ones determined and output. This may be done using a second application of the max-command. (It is noted that this is overkill – when a realisation is considered, the results of comparisons performed in the initial application of max may be reused in the second thereby resulting in a much lower complexity.) These results are summed up in the following theorem.

Theorem 13.1 *$\mathcal{F}_{\mathbf{1st-price}}$ and $\mathcal{F}_{\mathbf{2nd-price}}$ with n bidders may be perfectly UC-realised in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model extended with the* (**let** $[\mathbf{i}] \leftarrow \mathtt{MAX}([\mathbf{A}])$) *command. The complexity of the realisation is equivalent to that of maximum. This implies two possible perfect UC-realisations in the basic $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model – $\mathcal{O}(n)$ comparisons and multiplications in $\mathcal{O}(\log\log(n))$ rounds, or alternatively $\mathcal{O}(n^2)$ in $\mathcal{O}(1)$ rounds.*

Note that no action is required with regard to active adversaries. Inputs for the realising computation in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model must be verified, however, simply allowing all elements of the field $\mathbb{Z}_q$ to represent bids ensures that illegal bids cannot occur. As no integer computation is performed the modulus must not be chosen larger than the inputs to counteract overflow.

Chapter 14

The SCET Double Auction

The goal of any auction is to determine a price at which trade will occur between the participants. However, auctions are basically sets of trade-rules; participants provide bids – information describing their preferences – which determines the price through some computation. The auction considered in this chapter is the double auction of the SCET project – Secure Computing, Economy, and Trust of which the author was a part.

Double auctions have important real-world uses, e.g. the typical stock exchanges. However, though a general mechanism is constructed, the focus of SCET has been on distribution of agricultural production rights, motivated by its industry partner, Danisco. For a more in-depth discussion of the double auction, SCET, and the Danisco case see [BDJ+05, Tof05, BBNN07]. During the project, the protocol presented below was implemented, demonstrating its feasibility in practice (using early versions of the primitives of part II).

This chapter is divided into three parts. Section 14.1 formalises the problem and solution and provides an ideal functionality specifying the intended behaviour. This is then realised in section 14.2. The chapter concludes with two variations of the secure computation due to the author in section 14.3.

14.1 The Double Auction

A double auction is a large auction where multiple buyers and sellers meet to exchange units of the same good. The goal is to determine a market clearing price (MCP), the price at which all trade will take place. Letting n denote the number of parties – both buyers or sellers – all of these provide bids (explained further below) committing them to potential deals. Based on these the MCP is computed and through this the actual deals to take place determined. The goal of this chapter is to construct this computation in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model, outputting the market clearing price while leaking no other information on the bids.

An ideal functionality $\mathcal{F}_{\mathbf{DoubleAuction}}$ securely determining the MCP is easily constructed. Assume that there are n parties who wish to take part. These bidders provide their inputs (bids) to $\mathcal{F}_{\mathbf{DoubleAuction}}$ after which the adversary takes action, corrupting parties and potentially updating their inputs. Finally,

when the adversary allows, $\mathcal{F}_{\mathbf{DoubleAuction}}$ computes the MCP based on the inputs and sends this to the adversary and the parties. It remains to specify the exact details of bids and the required computation, however, it should be clear that realising this in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model amounts to demonstrating how the desired computation may be performed.

A bid in a double auction consists of a function over the positive real values, $\mathbb{R}_+ \mapsto \mathbb{R}_{0+}$, mapping prices to amounts of the good in question. This denotes the amount to be traded given that that price becomes the market clearing price. For sellers this is denoted supply, while for buyers it is demand. These functions must be monotone, decreasing for buyers and increasing for sellers. The intuition is that buying a given amount does not become a less attractive deal if the price drops, for sellers that intuition is reversed.

The market clearing price to be computed is found based on the aggregated supply and demand of the entire system, i.e. 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 then defined as the price-value at the intersection of these functions. This definition is economically sound, the MCP is the price where supply and demand meet – all bidders must then buy or sell whatever amount they specified at that price.

However, $\mathcal{F}_{\mathbf{EMPC}}$ does not provide computation over real values. Moreover, storing arbitrary (though monotone) functions with an infinite domain is *entirely* out of the question. Thus, a discrete reformulation of the problem using bounded integer values must be considered, thereby ensuring that the computation may be performed using $\mathcal{F}_{\mathbf{EMPC}}$.

Restricting the set of prices to $P = \{1, 2, \ldots, p_{max}\} \subset \mathbb{N}$ and the possible amounts of the bids to $A = \{0, 1, \ldots, a_{max}\} \subset \mathbb{N}$ restricts the supply and demand functions to monotone functions mapping P to A. This may be seen as points on the original function "rounded off" to the grid defined by P and A. Thus, this representation provides a simplified view of the supply and demand functions, which *can* be represented within $\mathcal{F}_{\mathbf{EMPC}}$. P and A must simply be chosen large enough for providing a sufficiently fine-grained representation of supply and demand. Note that by varying the actual *units* used for measuring price and amount, an arbitrary fine-grainedness may be obtained.

The concept of market clearing price transfers quite readily to this discrete world. The supply and demand functions are still monotone, mapping prices of P to amounts, though these now belong to the set $\{0, 1, \ldots, n \cdot a_{max}\}$ rather than A as there are n parties. The sole problem is that as a discrete setting is considered, most likely there will be no price where supply and demand are equal – i.e. no economically sound intersection. However, simply rounding off the result provides an adequate definition as discussed below. The discrete MCP will be the maximal price $i_{MCP} \in P$ where the aggregated demand is greater than or equal to the aggregated supply, i.e. the valid price immediately below the real MCP (in both senses of the word).

The solution may result in surplus demand, however, from a cryptographic perspective this issue is irrelevant. Auctions are merely trade-rules, therefore specifying a rule for handling surplus demand eliminates the problem – though naturally soundness (with regard to economics) must be ensured. The

actual rule may depend on the application, but from the point of view of $\mathcal{F}_{\mathbf{DoubleAuction}}$, the key point is to determine the market clearing price and nothing more.

A more serious problem may occur when a discrete problem is considered. The supply and demand curves may not intersect in the interval considered, i.e. between 1 and p_{max}. Similar to the above, this problem is also ignored. p_{max} may be chosen freely, thus basing it on an educated guess for i_{MCP} means non-intersection is unlikely. Moreover, even if no MCP is found this may be detected. No additional information is leaked, and bidders may simply provide additional inputs.

14.2 Realising the Double Auction

Realising $\mathcal{F}_{\mathbf{DoubleAuction}}$ in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model consists of specifying the inputs, the computation, and the outputs. Providing inputs are immediate from the input-command of $\mathcal{F}_{\mathbf{EMPC}}$. All of the n parties provide two arrays, with P_j inputting (points on) its demand curve $[\mathbf{D_j}]$ and supply curve $[\mathbf{S_j}]$. Providing both curves ensures that buyers and sellers cannot be distinguished – one of them may simply be a (constantly 0) dummy curve.

Performing the actual computation consists of two steps, computing the aggregated curves for the entire system and finding their intersection. This must then be output, however, the computation considered "leaks" the desired information, thus an explicit output is not required at the end. The aggregated supply and demand curves – represented by arrays $[\mathbf{S}]$ and $[\mathbf{D}]$ – are simply sums over the curves of all the participants, i.e. they are computed as

$$[\mathbf{S}](i) \leftarrow \sum_{j=1}^{n} [\mathbf{S_j}](i) \tag{14.1}$$

and

$$[\mathbf{D}](i) \leftarrow \sum_{j=1}^{n} [\mathbf{D_j}](i). \tag{14.2}$$

As defined above the market clearing price is the maximal price i_{MCP} such that $\mathrm{val}([\mathbf{D}](i_{MCP})) \geq \mathrm{val}([\mathbf{S}](i_{MCP}))$. Viewing the arrays as points on monotone curves, with $[\mathbf{S}]$ increasing and $[\mathbf{D}]$ decreasing, the solution to the problem is immediate: The problem may be viewed as a single array of decreasing (i.e. sorted) values $(\mathrm{val}([\mathbf{D}](i)) - \mathrm{val}([\mathbf{S}](i)))$ over the integers. The desired index is the maximal containing a non-negative value, which may be determined through a binary search, repeatedly comparing $[\mathbf{D}](i)$'s to $[\mathbf{S}](i)$'s. Leaving out the computation of the aggregated curves above, the secure computation is seen as protocol 16.

Correctness follows by the above discussion, while privacy is ensured as the computation is performed in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model. The only information leakage occurs through the outputs in step 5. However, though information is disclosed this is acceptable; the desired outcome of the computation is to *output* the unique intersection i_{MCP} between the curves represented by $[\mathbf{D}]$ and

Protocol 16 Computing the market clearing price for a double auction

Input: The aggregated supply and demand curves $[\mathbf{S}]$ and $[\mathbf{D}]$ with $[\mathbf{S}].\text{len} = [\mathbf{D}].\text{len} = p_{max}$.

Output: The market clearing price i_{MCP}, i.e. the maximal i_{MCP} such that $\text{val}([\mathbf{D}](i_{MCP})) \geq \text{val}([\mathbf{S}](i_{MCP}))$.

```
    l ← 1                                          ▷ i_MCP ≥ l
    h ← p_max + 1                                  ▷ i_MCP < h
    while l + 1 < h do
        m ← ⌊(l + h + 1)/2⌋                        ▷ Note: Integer division
5:      c ← output([D](m) ?< [S](m))
        if c ?= 1 then
            h ← m
        else
            l ← m
10:     end if
    end while
    i_MCP ← l
```

$[\mathbf{S}]$. Knowledge of i_{MCP} combined with the fact that the curves are monotone implies the results of comparisons of *all* $[\mathbf{D}](i)$ and $[\mathbf{S}](i)$ including, but not limited to, the ones output. Thus revealing this information – though slightly prematurely – does not violate privacy of the inputs.

Ignoring inputs, complexity of the whole auction is equivalent to the complexity of protocol 16; computing the aggregated curves is costless. This requires $\lceil \log(p_{max}) \rceil$ comparisons (and outputs) performed one by one. This implies $\mathcal{O}(\log(p_{max}))$ rounds. It is noted that computation complexity may be reduced slightly: The full aggregated curves need not be computed, it suffices to compute the points to compare lazily, i.e. performing the relevant computations 14.1 and 14.2 immediately before step 5 where they are used. This does not reduce communication or round complexity though.

The final points to consider consists of the choice of modulus defining the field in which computation occurs and a discussion of active adversaries. The ring must be chosen by the ideal functionality before computation (including inputs) ensues. Assuming that inputs are to be of some specified bit-length ℓ, no values of the computation will be greater than $n \cdot 2^\ell$. Thus computing modulo $q > n \cdot 2^\ell$ ensures that no overflow occurs.

Regarding active adversaries it must be verified that inputs are proper, i.e. that all values are ℓ-bit and that they represent monotone functions. Verifying this is trivial, however the cost is disregarded. This may be viewed as an assumption that the participants provide proper inputs – they take part as it is in their best interest to do so, and the design of the auction implies that truthful bids are optimal.[1] Moreover, as the (real-world) goal is the exchange of goods, some additional information on the inputs of parties is revealed to

[1] In difference to (most) cryptography, economics considers rational entities attempting to maximise their own outcome.

others, namely the amount traded. If this consists of an illegal value, then that party has cheated. Alternatively, chances are that the surplus demand will be large or surplus *supply* will be encountered. The former will raise suspicion, while the latter is clearly incorrect; both may prompt further investigation. The realisation in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model is stated as theorem 14.1, a perfect UC-realisation in the $\mathcal{F}_{\mathbf{MPC-R}}$-hybrid model is immediate as well.

Theorem 14.1 *The ideal functionality $\mathcal{F}_{\mathbf{DoubleAuction}}$ may be perfectly UC-realised in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model through $\log(p_{max})$ individual comparisons, i.e. in $\mathcal{O}(\log(p_{max}))$ rounds.*

14.3 Variations of the Double Auction

Two variations of the double auction will be described. The first considers a trade-off – reducing the number of rounds at the cost of performing additional secure computation. The second alters the ideal functionality slightly, the MCP should be computed, but not output.

Reducing round complexity The round complexity of the double auction computation may be reduced by running essentially the same algorithm, but performing multiple comparisons in parallel in each iteration of the loop. The cost of the reduction comes in the form of the added computation, i.e. in an increased communication complexity. The extreme case is to process all prices in parallel, testing all p_{max} candidates in one go. This determines the market clearing price in a single comparison-round rather than logarithmically many, though at the cost of a linear number of comparisons performed. In other words, round complexity of the auction may be reduced to that of a comparison.

The idea generalises to splitting the search-space into h segments of equal size (rather than two or p_{max}). The segment containing the intersection may be determined through $h-1$ parallel comparisons, testing

$$[\mathbf{D_i}](j \cdot \lfloor p_{max}/h \rfloor) \overset{?}{<} [\mathbf{S_i}](j \cdot \lfloor p_{max}/h \rfloor)$$

for $j \in \{1, 2, \ldots, h-1\}$. Recursing on the determined interval allows the MCP to be found in $\log_h(p_{max})$ comparison rounds though at the cost of $(h-1)\log_h(p_{max})$ comparisons all in all. Depending on the underlying primitives and the communication network, this may be preferable.

Secret Binary Searching The second variation considers the case where the market clearing price should be determined but not output, i.e. a secret index should be determined. This can be seen as a general, secret binary search command of $\mathcal{F}_{\mathbf{EMPC}}$ rather than as a variation of the realisation of $\mathcal{F}_{\mathbf{DoubleAuction}}$. When executing a double auction the result – the market clearing price – *will* become known; if not sooner then when money changes hands.

Performing a binary search consists of repeatedly performing a test at the current index, thereby obtaining the following one. However, if no information is

to be leaked, then not only must the result of the test not be output, it must not be known *which* index is tested. The initial index is public and therefore easily computable, moreover given the (secret) index at one level and the (secret) result of the comparison, the following index is immediate. Each entry depends on exactly one entry of the previous index, thus a single multiplication per entry suffices. Through secure indexing the overall computation is readily obtained with immediate correctness and privacy.

Complexity is equivalent to the $\lceil \log(p_{max}) \rceil$ comparisons plus the construction and use of the indexes. Noting that each entry of the full arrays may only be tested at *one* level of the search, it can be seen that full index-arrays are not required – at level i only 2^{i-1} candidates exist – the rest are *known* 0. Thus overall only a linear number of secure multiplications are needed for computing and using the secret indexes needed. Finally, round complexity is dominated by the logarithmic number of iterations of the binary search. It is noted that this variation may be combined with the one above.

Chapter 15

Solving Linear Programming Problems

In economics problems may often be phrased as linear programs (LP), i.e. maximising a function given constraints on the variables. Thus the ability to solve LP problems using MPC appears to be *very* useful, e.g. for computing a desired coordination without disclosing information, which may otherwise distort the incentives of the economic system.

One immediate application is benchmarking. Companies may wish to compare themselves to the industry standard, but none are motivated to share their data, which may contain trade secrets. One popular approach to benchmarking, both theoretically and practically, is the Data Envelopment Analysis (DEA) proposed by Charnes *et al.* [CCR78]. DEA models can be phrased as linear programs, thus this provides motivation for a privacy preserving means of solving LP problems, i.e. performing the computation in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model.

This chapter is a light edit of an unpublished paper by the author [Tofb]. Its main idea is to use a variation of simplex as the basis for a privacy preserving protocol solving LP problems. This variation uses integer pivoting at its core, ensuring that all computation may be performed using $\mathcal{F}_{\mathbf{EMPC}}$. The result is a protocol, which appears to be realisable in practice, while revealing a minimal amount of information – the number of iterations performed in the computation. Variations disclosing even less will be considered as well. The complexity of the protocol constructed is equivalent to that of the basis-algorithm in the sense that the number of basic computational operations is only increased by a small constant factor. To the author's knowledge, solving LP problems using MPC has not been considered previously, though it appears to have immediate uses as well as the potential of being a useful primitive in further secure computation.

Section 15.1 gives an overview of the simplex algorithm using integer pivoting – it is intended as a reminder for readers familiar with the algorithm and as a brief, introductory overview for those who are not. The remainder of the chapter consists of translating the algorithm to one working within $\mathcal{F}_{\mathbf{EMPC}}$. Section 15.2 considers the desired task and provides an ideal functionality. This is followed by a translation of the main body of the algorithm in section 15.3. Section 15.4 then provides the full secure computation based on the translation of the body, and the chapter concludes with additional discussion, including securely extracting the output upon protocol termination.

15.1 The Simplex Algorithm

The simplex method is a well-known strategy for solving linear programming problems. Though the worst-case complexity is exponential, it is efficient in practice and used in a large range of applications. This section will contain an brief overview of (the steps of) the algorithm. For a full description as well as explanation of the details, the reader is referred to Chvátal's well-known work [Chv83]. As the focus here is multi-party computation providing (simulated) integer computation, a variation of simplex using integer pivoting is considered. This technique – generally attributed to Edmonds – is described in detail by Rosenberg in [Ros05].

A linear program consists of n variables, $x_1, \ldots, x_n \geq 0$, and m constraints:

$$\sum_{i=1}^{n} c_{j,i} \cdot x_i \leq b_j \text{ for } j \in \{1, \ldots, m\}. \tag{15.1}$$

The goal is to maximise an objective function, f:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i \cdot x_i,$$

where the x_i are subject to the constraints of 15.1. Note that n is *not* used to refer to the number of parties. *No* reference to the number of parties performing the computation is made in this chapter, thus n will be used to denote the number of variables of the LP.

To ensure integer computation, all values, i.e. the $c_{j,i}$ and f_i, are required to be integer. Maximising f is then accomplished by first introducing slack-variables, $x_{n+1}, \ldots x_{n+m} \geq 0$, resulting in equalities in the constraints:

$$x_{n+j} + \sum_{i=1}^{n} c_{j,i} x_i = b_j \text{ for } j \in \{1, \ldots, m\}.$$

A solution will be an assignment to the x_i such that the constraints hold. In simplex, solutions allowing only m variables to be positive are constructed, the current is known as the basis. Each of these variables are associated with a constraint. All other variables (denoted the co-basis) take the value 0. The execution of the algorithm starts by considering an initial solution with the basis consisting of the slack-variables (taking the values b_j).[1] By repeatedly moving variables in and out of the basis, the problem is rephrased and the solution improved (the value obtained when evaluating f increased) with every iteration until one maximising f is found. Consider the linear program written in tableau form

[1] For simplicity it is assumed that the linear program is origin-feasible – i.e. that $(x_1, \ldots, x_n) = (0, \ldots, 0)$ does not violate any constraints. Standard techniques may be applied to avoid this issue.

$$
\begin{array}{ccc|cccccc|c}
c_{1,1} & \dots & c_{1,n} & 1 & 0 & 0 & \dots & 0 & 0 & b_1 \\
c_{2,1} & \dots & c_{2,n} & 0 & 1 & 0 & \dots & 0 & 0 & b_2 \\
 & \ddots & & & & & \ddots & & & \vdots \\
c_{m,1} & \dots & c_{m,n} & 0 & 0 & 0 & \dots & 0 & 1 & b_m \\
\hline
-f_1 & \dots & -f_n & 0 & 0 & 0 & \dots & 0 & 0 & z = 0
\end{array}
$$

where column i, $1 \leq i \leq m + n$, is associated with variable x_i, and row j, $1 \leq j \leq m$ is associated with constraint j.

For the tableau, the right-most column (except the bottom row), will be known as the b-vector, similarly the bottom row (except the right-most element), will be called the f-vector; note $f_{n+1}, \dots, f_{n+m}$ initially set to zero. The sub-matrix of the constraints consisting of the columns of the slack-variables is initialised to the identity matrix. The columns of the tableau associated with the basis variables will always be the columns of I multiplied by a positive integer, p' (the previous pivot element, initially 1, see below). The basis variable of the current solution associated with the j'th constraint (row) takes the value b_j/p' in the solution. The value z in the bottom right-hand corner is the objective function, f, evaluated at the current solution and multiplied by p', initially $f(0, \dots, 0) = 0$. Note that solution and the evaluation of f contain *rational* values. These are easily stored using integers, splitting each value up into two, the numerator and the denominator.

Integer pivoting simplex repeatedly updates the tableau through the following steps; steps I through V of which is known as the *pivot*.

I. Determine a column, C with a negative value in the f-vector. The (guaranteed co-basic) variable associated with C is chosen to enter the basis. C will be referred to as the *pivot column*.

II. Determine a row, R, such that its intersection with C, C_R, is positive (constraining) and b_R/C_R is minimal. This row is called the *pivot row*, the element C_R is called the *pivot element*. The basic variable associated with R is the one selected for leaving the basis; after the current iteration the basic variable associated with row R will be the one associated with C.

III. Multiply all entries in the non-pivot rows by the pivot element.

IV. Subtract a multiple of the pivot row from all non-pivot rows such that the updated pivot column will consist entirely of zeros except for the pivot row.

V. Divide all non-pivot rows by the pivot element of the *previous* iteration (which is initialised to 1)

VI. If the f-vector contains one or more negative values, then goto step I.

A problem with the simplex algorithm is that it may cycle and therefore never terminate. This only occurs in the case of degenerate solutions, i.e. when one or more basic variables are assigned 0. Though cycling is rarely encountered

in practice, for completeness it is necessary to consider the issue. Fortunately from an MPC-point of view, there is a simple way of avoiding it in the form of Bland's rule: When confronted with a choice of entering or leaving variable, always pick the one with the lowest index, see [Chv83]. Naturally, this means that the index of the variable associated with a given row (constraint) must be stored along with it. This value is updated once the entering and leaving variables have been determined at the end of step II.

15.2 Privacy Preserving Simplex

The solving of linear programming problems will be added as a command to $\mathcal{F}_{\mathbf{EMPC}}$ rather than viewing it as an overall, stand-alone application. The task is a general one, and though it has immediate applications, it may also be useful as a sub-task in others. Moreover, it allows the constraints and objective function to depend on inputs in any (efficiently computable) way. Though it may be the case that each value of the tableau will be known by some party, this is an artificial restriction as the computation is constructed in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model.

Negative integer values are required for simplex, these will be encoded in the underlying field $\mathbb{Z}_q$ in the natural way, i.e. the negative $-a$ will be encoded as $q - a \in \mathbb{Z}_q$. This ensures that both addition and multiplication involving negative values work as desired. Essentially the point at which overflow occurs is moved to the "middle" of the field. Naturally minor modification of the inequality test is required.

The initial tableau is simply assumed to be encoded in secret variables of $\mathcal{F}_{\mathbf{EMPC}}$, their origin not explored further. Linear programs of n variables,

$$x_1, \ldots, x_n \geq 0,$$

and m constraints

$$\sum_{i=1}^{n} c_{j,i} \cdot x_i \leq b_j$$

are represented by secret variables encoding the tableau in the natural way, i.e. as

$$[c_{1,1}], \ldots, [c_{m,n}], [b_1], \ldots, [b_m].$$

Additionally, the objective function,

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i \cdot x_i$$

is provided as

$$[f_1], \ldots, [f_n].$$

The matrix $[\mathbf{T}]$ will be used to refer to the tableau form of the problem, allowing ease reference to all of the above variables. For simplicity it is assumed that the LP given is bounded: An optimal solution does exist and no variable may become arbitrarily large.

$\mathcal{F}_{\mathbf{EMPC}}$ is extended with a command – (**let** $([\mathbf{T}], [p']) \leftarrow \mathtt{LP}([\mathbf{T}])$) – instructing the ideal functionality to execute the simplex algorithm. This takes an initial tableau – i.e. the $[c_{j,i}]$, $[f_i]$, and $[b_j]$ – and computes the final, updated tableau (and the final pivot element). This contains the final solution, its extraction is considered in section 15.5 below. Upon receiving this command in round R, $\mathcal{F}_{\mathbf{EMPC}}$ immediately runs the simplex algorithm of section 15.1 on the inputs. The adversary is then notified that the command has been given, as well as the number of iterations (pivots) performed.[2] Then, assuming that each iteration takes r rounds, the variables $[\mathbf{X}]$, $[z]$, and $[p']$ are set in round $R + I \cdot r - 1$, and the parties informed that the command has terminated.

The above command is easily realisable assuming a command for performing pivots – updating the tableau – exists. This is implicitly how command is realised; the following section "translates" steps I through V to MPC. The remaining step – handling termination – is then considered in section 15.4.

In the following sections m will always denote the number of constraints and n the number of initial variables. The overall number of variables is therefore $n + m$. The tableau, $[\mathbf{T}]$, is the matrix consisting of $m + 1$ rows and $m + n + 1$ columns. For ease of notation, $[\mathbf{F}]$ will denote the f-vector of $[\mathbf{T}]$ and $[\mathbf{B}]$ the b-vector. Finally, let $[\mathbf{S}]$ of length m be an array storing the indexes of variables associated with the constraints, i.e. the basis. Variable $x_{\mathrm{val}([\mathbf{S}](j))}$ is associated with row j and takes the value $\frac{\mathrm{val}([\mathbf{B}](j))}{\mathrm{val}([p'])}$ in the current solution, where $[p']$ is the most recent pivot element, initially 1. $[\mathbf{S}]$ is initialised with the slack-variables, $(n + 1, \ldots, n + m)$.

15.3 Translating the Body of Simplex

The translation of the simplex-iteration will be done by considering each of the five steps individually. It is assumed that output from previous steps is available, i.e. that the tableau $[\mathbf{T}]$ available at one step is the one output by the previous.

Step I, Determining the variable to enter the basis. This step consists of determining the pivot column as described in section 15.1. This is done by considering the f-vector and computing the minimal index, $[\mathbf{c}]$, such that $\mathrm{val}([\mathbf{F}]([\mathbf{c}]))$ is negative. This is clearly a candidate selected using Bland's rule, which in addition to ensuring that cycling is avoided is also efficiently implementable. A secure computation determining that index as well as a copy of the pivot column, $[\mathbf{C}]$, is seen as protocol 17. That the correct column is determined is seen as follows. The initial for-loop computes an array, $[\mathbf{D}]$, containing bits, such that $[\mathbf{D}](i)$ is 1 exactly if x_i associated with the i'th column is a candidate for entering the basis. The goal is to set all $[\mathbf{D}](i)$ to zero, except for the first occurring 1. Let i_{min} be the smallest index where a 1 occurs. The output of $\mathtt{pre}_{\vee}([\mathbf{D}])$, $[\mathbf{D}']$, is an array with $\mathrm{val}([\mathbf{D}'](i))$ equal to 0 for $1 \leq i < i_{min}$ and 1 for

[2] Technically, this implies that the ideal functionality is not probabilistic polynomial-time as it should be. However, this is due to the protocol itself not being polynomial-time, thus if the protocol is considered reasonable, then this amount of computation is as well.

Protocol 17 Step I: Selecting the pivot column

Input: The tableau, $[\mathbf{T}]$ (including $[\mathbf{F}]$ by definition).

Output: $[\mathbf{c}]$ and $[\mathbf{C}]$, such that $\mathrm{val}([\mathbf{F}]([\mathbf{c}]))$ is negative, $\mathrm{val}([\mathbf{c}])$ is minimal, and $[\mathbf{C}]$ is a copy of the $\mathrm{val}([\mathbf{c}])$'th column of $[\mathbf{T}]$.

```
for i = 1,...,n+m do
    [D](i) ← [F](i) ?< 0
end for
[D'] ← pre_∨([D])
[c](1) ← [D'](1)
for i = 2,...,n+m do
    [c](i) ← [D'](i) − [D'](i − 1)
end for
[C] ← [T](·, [c])
```

$i_{min} \leq i \leq m+n$. As the array, $[\mathbf{c}]$, is constructed as the difference between an entry of $[\mathbf{D}']$ and its predecessor, for $1 \leq i < i_{min}$ and $i_{min} < i \leq m+n = [\mathbf{c}].\mathrm{len}$ clearly $[\mathbf{c}](i)$ contains 0. It is equally simple to verify that $\mathrm{val}([\mathbf{c}](i_{min})) = 1$. Thus, $[\mathbf{c}]$ contains the index i_{min} of the form introduced in section 12. $[\mathbf{C}]$ is correct by construction.

With regard to complexity, $n+m$ comparisons are performed in the initial for-loop, followed by an execution of $\mathtt{pre}_\vee(\cdot)$ on an input of length $n+m$. The second loop is costless, and the indexing done in order to compute $[\mathbf{C}]$ is equivalent to $m+1$ indexings into arrays of length $n+m$. Overall this is equivalent to $(m+n)(m+1) + 13(m+n) = (m+14)(m+n)$ multiplications and $n+m$ comparisons. For round complexity, note that the body of the initial for-loop does not depend on previous iterations, thus they may be executed concurrently. The remaining $\mathtt{pre}_\vee(\cdot)$ and indexing are also constant rounds, thus disregarding preprocessing the overall complexity is $c_r + 13$ rounds.

Step II, Determining the variable to enter the basis. Recall that the goal of this step is to find the tightest constraint on the variable, $x_{\mathrm{val}([\mathbf{c}])}$, determined in step I above. In other words, this step must result in an index $[\mathbf{r}]$, $1 \leq \mathrm{val}([\mathbf{r}]) \leq m$, such that

$$\mathrm{val}([\mathbf{C}]([\mathbf{r}])) \cdot x_{\mathrm{val}([\mathbf{c}])} = \mathrm{val}([\mathbf{B}]([\mathbf{r}]))$$

implies the smallest, non-negative value of $x_{\mathrm{val}([\mathbf{c}])}$. In addition to this, in order to facilitate the subsequent steps, (copies of) the pivot row and the pivot element, $[\mathbf{R}]$ and $[p] = [\mathbf{C}]([\mathbf{r}])$, must also be obtained.

As $\mathrm{val}([\mathbf{C}](j)) \leq 0$, implies that constraint j does not limit $x_{\mathrm{val}([\mathbf{c}])}$ upwards, such constraints are never determined. Thus, only constraints with $\mathrm{val}([\mathbf{C}](j)) > 0$ are relevant, these will be called *applicable*, the rest will be called *non-applicable*. The goal is therefore to determine the index, $[\mathbf{r}]$, of an applicable constraint with the *rational* value $\frac{\mathrm{val}([\mathbf{B}]([\mathbf{r}]))}{\mathrm{val}([\mathbf{C}]([\mathbf{r}]))}$ minimal. As in step I, ties are broken using Bland's rule.

Overall, the translation of this step consists of defining a comparison operator on constraints and computing the minimal. As noted in chapter 13,

the maximum-protocols may be executed using arbitrary comparison operators, thus simply constructing a comparison operator suffices. Three values of each constraint are needed: For the j'th constraint – i.e. j'th row – $[\mathbf{B}](j)$ and $[\mathbf{C}](j)$ define the constraint and whether it is applicable, while $[\mathbf{S}](j)$, the index of the variable in the basis currently associated with row j, must be used when Bland's rule is applied.

In order to simplify the construction of the comparison operator, the problem is transformed such that all non-applicable constraints become applicable, though less restricting than the originally applicable ones. This is done by replacing the fractional value of non-applicable constraints with one which is larger than the maximal possible, $\frac{\infty}{1}$. Here ∞ simply represents some value larger than the largest possible value of the $[\mathbf{B}](j)$'s, e.g. if the values are k-bit, choosing $\infty = 2^k$ suffices. The updated arrays $[\mathbf{C}']$ and $[\mathbf{B}']$ are computed as

$$[\mathbf{C}'](j) \leftarrow [\mathbf{C}](j) \overset{?}{>} 0 \,?\, [\mathbf{C}](j) : 1$$

and

$$[\mathbf{B}'](j) \leftarrow [\mathbf{C}](j) \overset{?}{>} 0 \,?\, [\mathbf{B}](j) : \infty$$

for $1 \leq j \leq m$. This can be done using $\mathcal{O}(m)$ comparisons and multiplications in a constant number of rounds, and the problem now consists entirely of applicable constraints. That the solution is not changed by the transformation is obvious, the LP was assumed bounded, so some initial constraint was applicable and this represents a tighter restriction than the ones introduced.

Defining a comparison operator for applicable constraints – triples of the form $([\mathbf{C}'](j), [\mathbf{B}'](j), [\mathbf{S}](j))$ – is the next task. This is in essence just a comparison for positive fractions, $\frac{B'}{C'}$, except that in the case of equality, the S values must be compared. Noting that for positive integers a_n, a_d, b_n, and b_d,

$$\frac{a_n}{a_d} < \frac{b_n}{b_d} \Leftrightarrow a_n \cdot b_d < b_n \cdot a_d$$

the desired output may be computed using an integer comparison. As the inequality in the above equation can be replace by equality, Bland's rule applies exactly when the two products are equal. The protocol for constraint comparison based on this discussion is denoted $\overset{?}{\sqsubset}$, details are seen in protocol 18. Correctness is immediate and complexity is three multiplications and three comparisons – the latter may be executed in parallel. It is noted that if the LP is not degenerate, Bland's rule does not need to be invoked. Moreover, non-degeneracy implies that the $[\mathbf{B}](j)$ are non-zero. It can be verified that in this case, the initial transformation of the problem may be ignored as non-positive $[\mathbf{C}](j)$ are implicitly handled through the comparison of the products, $[l]$ and $[r]$.

Having constructed the required comparison operator, computing $[\mathbf{r}]$ is a simple matter of applying the loglog-rounds protocol of chapter 13 to the array consisting of triples

$$\Big(\big([\mathbf{C}'](1), [\mathbf{B}'](1), [\mathbf{S}](1)\big), \ldots, \big([\mathbf{C}'](m), [\mathbf{B}'](m), [\mathbf{S}](m)\big)\Big).$$

Protocol 18 Constraint comparison, $\overset{?}{\sqsubset}$

Input: Triples $([C_i], [B_i], [S_i])$ and $([C_j], [B_j], [S_j])$ representing applicable constraints to be compared – $[C_i]$ and $[C_j]$ represent the relevant entries of the pivot column while $[B_i]$ and $[B_j]$ represent the values of the b-vector. Finally $[S_i]$ and $[S_j]$ are the index-values of the current basis variables associated with the two constraints, i.e. the candidates to leave the basis.

Output: Bit $[b]$ – 1 if the left argument $([C_i], [B_i], [S_i])$ constrains less than the right $([C_j], [B_j], [S_j])$, with ties broken using Bland's rule.

$[l] \leftarrow [B_i] \cdot [C_j]$

$[r] \leftarrow [B_j] \cdot [C_i]$

$[b] \leftarrow ([l] \overset{?}{=} [r])\ ?\ ([S_i] \overset{?}{<} [S_j]) : ([l] \overset{?}{<} [r])$

Executing this protocol returns the desired index, $[\mathbf{r}]$.[3] Obtaining a copy of the pivot row, $[\mathbf{R}]$, is simply an indexing operation,

$$[\mathbf{R}] \leftarrow [\mathbf{T}]([\mathbf{r}], \cdot).$$

The pivot element $[p]$ may be obtained as $[\mathbf{C}]([\mathbf{r}])$. Finally, as the basis is being updated, this must be reflected by $[\mathbf{S}]$, thus $[\mathbf{S}]([\mathbf{r}])$ must be set to $\mathrm{val}([\mathbf{c}])$.

The initial transformation requires m comparisons and $2m$ multiplications in c_r+1 rounds. Determining $[\mathbf{r}]$ requires $2m$ invocations of the $\overset{?}{\sqsubset}$-protocol and $9m$ multiplications[4] in $\mathrm{loglog}(m) \cdot (c_r + 2(c_r + 2) + 4)) = \mathrm{loglog}(m) \cdot (3c_r + 8)$ rounds. Finally, determining $[\mathbf{R}]$ and $[p]$ and updating $[\mathbf{S}]([\mathbf{r}])$ requires $m+n+3$ indexing operations in arrays of length m. Thus overall $2m+(2m \cdot 2+4m)+m(m+n+3) = 13m + m^2 + mn$ multiplications and $7m$ comparisons in $(3\mathrm{loglog}(m) + 1) \cdot c_r + 8\mathrm{loglog}(m) + 2$ rounds are required.

Step III, Multiply all non-pivot rows by the pivot element. Having determined the variables to enter and leave the basis in the form of indexes, $[\mathbf{c}]$ and $[\mathbf{r}]$, as well as having obtained a copy of the pivot element, $[p]$, the third step consists of multiplying all entries of the tableau *except* the ones in the pivot row, by the latter. This is accomplished by protocol 19. For correctness, note that after step 4, all entries in $[\mathbf{M}]$ are $[p]$ except the $[\mathbf{r}]$'th, which is [1]. Thus step 7 multiplies all entries of non-pivot rows by the pivot element, while leaving the pivot row implicitly untouched. Regarding complexity, m multiplications are required to set $[\mathbf{M}]([\mathbf{r}])$, while $(m + 1) \cdot (n + m + 1)$ are needed to update the tableau. However, everything may be updated concurrently, thus only two rounds are required.

Step IV, Subtract a multiple of the pivot row from all non-pivot rows. Step IV consists of subtracting a multiple of the pivot row $[\mathbf{R}]$ from all non-pivot rows (of the updated tableau from step III) such that the pivot column

[3]Note that if the entering variable is unbounded, $\mathrm{val}([\mathbf{B}'][\mathbf{r}])) = \infty$, thus detecting unbounded linear programs is possible.

[4]Note that as arrays of *triples* are considered indexing is more expensive than in the basic computation of chapter 13.

Protocol 19 Step III: Multiplication of all non-pivot rows by the pivot element

Input: The tableau $[\mathbf{T}]$, the index of the pivot row $[\mathbf{r}]$, and the pivot element $[p]$.

Output: The updated tableau $[\mathbf{T}']$ with all non-pivot rows multiplied by $[p]$.

```
for j = 1,...,m+1 do
    [M](j) ← [p]
end for
[M]([r]) ← 1
for j = 1,...,m+1 do
    for i = 1,...,n+m+1 do
        [T'](j,i) ← [M](j) · [T](j,i)
    end for
end for
```

will consist entirely of zeros except in the pivot row. As all non-pivot rows have been multiplied by the pivot element in the previous step, the multiple to subtract from non-pivot row j is the j'th entry of the original pivot column $[\mathbf{C}]$ computed in step I. The required computation is analogous to that of step III; set the multiple to be subtracted from the pivot row to zero and subtract the relevant multiple for each entry in the tableau. Details are seen in protocol 20, correctness is equivalent to step III, as is the complexity obtained: $(m+1) \cdot (n+m+2)$ multiplications in two rounds.

Protocol 20 Step IV: Subtraction of the pivot row from all non-pivot rows

Input: The tableau $[\mathbf{T}]$, the index of the pivot row $[\mathbf{r}]$, the pivot row $[\mathbf{R}]$, and the original pivot column $[\mathbf{C}]$.

Output: The updated tableau $[\mathbf{T}']$, such that all non-pivot rows are zero in the pivot column.

```
[C]([r]) ← 0
for j = 1,...,m+1 do
    for i = 1,...,n+m+1 do
        [T'](j,i) ← [T](j,i) − [C](j) · [R](i)
    end for
end for
```

Step V, Divide non-pivot rows by the previous pivot element. The final step of the iteration consists of dividing all non-pivot rows of the tableau updated in step IV by the pivot element of the previous iteration, $[p']$. Naturally, this is an integer division. With the current state of the art MPC, no truly efficient protocol for performing general integer division exist. However, this is not a general case: It is guaranteed that $[p']$ divides $[\mathbf{T}](j,i)$ for all entries *not* in the pivot row, for a proof of this see [Ros05]. This simplifies the problem, division can be implemented as multiplication by the inverse (in the underlying

field) of the divisor.[5]

Element inversion is possible using the well-known protocol of Bar-Ilan and Beaver [BB89]. Generate a uniformly random mask, multiply this onto the element to be inverted, and output this. Then invert the masked element and securely multiply it onto the mask. Disregarding the random element generation, this is equivalent to one multiplication and two rounds, which may be executed in parallel with the previous step.

Having obtained $[(p')^{-1}]$, this must then be multiplied onto all non-pivot rows. This task is equivalent to that of step III, thus protocol 19 may be used to update the tableau as desired. Correctness follows from the above and the correctness of step III. The final updating of the previous pivot element to the current is costless and inversion highly efficient, thus the complexity will be considered equivalent to that of protocol 19.

Overall complexity. Joining the above analyses, each step performs at most a constant number of multiplications per entry in the tableau – sometimes hidden in indexing operations – as well as up to a constant number of comparisons per variable. Performing a detailed analysis, it can be verified that the overall complexity of steps I through V is bounded by $8m + n$ comparisons and $5mn + 17n + 5m^2 + 22m + 3$ multiplications – $\mathcal{O}(m + n)$ multiplications and $\mathcal{O}(m(m+n))$ multiplications for short, though the above demonstrates that the constants are small.

Round complexity under big-$\mathcal{O}$ is dominated by step II, $\mathcal{O}(\text{loglog}(m))$. A more thorough analysis results in a count of $(3\text{loglog}(m)+2)\cdot c_r+8\text{loglog}(m)+21$ rounds. Recalling the alternative maximal-realisations, a *constant-rounds* pivot computation is found. This comes at a worse complexity though, $\mathcal{O}(m^2 + n)$ comparisons.

Note that the number of operations is similar to what is required in a (worst case) iteration of the algorithm on public data. The number of multiplications is always linear in the size of the tableau, while the number of comparisons is at least linear in m and may be higher than $n+m$. Moreover, the constants are small, thus the secure computation is essentially as efficient as can be hoped for, though minor improvements – which reduce clarity – are possible.

15.4 Iteration and Termination

A high-level view of the protocol will now be taken. This consists of describing the evaluation of the termination condition, as well as determining what to do with it, step VI. The protocol must iterate until all entries of $[\mathbf{F}]$ are non-negative. At this point nothing further should be done. Broadly speaking there are two choices: Leak information on the number of iterations or perform exponentially many. The approach taken for the command of $\mathcal{F}_{\mathbf{EMPC}}$ was the former, which provides the simplest solution.

[5] If computation occurs over a ring this may not be possible. However, for the motivating example based on Paillier encryption the probability of this occurring is negligible.

Obtaining a bit stored in $\mathcal{F}_{\mathbf{EMPC}}$ stating whether to continue comes at no cost at all, except for the final (failing) test. The secure computation required must be invoked anyway in the subsequent iteration. Consider what happens if an optimistic approach is taken and step I is performed with no knowledge of whether or not there is a negative entry. If one exists an index is determined, however, if all entries of $[\mathbf{F}]$ are non-negative, all entries of the bit-vector containing the tests for candidates will be 0. This will also be the case for the "index" determined. Computing a bit stating if a proper index was found is trivial: The sum of the entries is the desired result, thus simply performing step I and computing a (costless) sum suffices.

By section 15.3 and the above discussion, $\mathcal{F}_{\mathbf{EMPC}}$ extended with the command for solving linear programming problems introduced in section 15.2 may be perfectly UC-realised in the basic $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model. The full solution consists of repeatedly computing the index of the pivot row and outputting its sum – this outputs a bit stating whether more iterations should be performed. If so then an iteration is executed, which updates the tableau.

Correctness follows as the secure computation performed is simply a translation of the simplex algorithm. Privacy is immediate as computation occurs in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model, only the termination tests are output, implying that the number of rounds is revealed. Finally, complexity is that of the body of the overall loop times the number of iterations. The latter is exponential in the worst case, however, this is a property of the original algorithm and not a result of performing MPC. The result is summed up in the following theorem.

Theorem 15.1 *$\mathcal{F}_{\mathbf{EMPC}}$ extended with the command* $(\mathbf{let}\ ([\mathbf{T}], [p']) \leftarrow \mathtt{LP}([\mathbf{T}]))$ *may be perfectly UC-realised in the original $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model. Letting n denote the number of variables and m the number of constraints, this may be done through a realisation of the main body of the simplex algorithm requiring $\mathcal{O}(m+n)$ comparisons and $\mathcal{O}(m \cdot (m+n))$ multiplications in $\mathcal{O}(\log\log(m))$ rounds. The full solution is then obtained by publicly testing for termination at every iteration; its complexity is the number of iterations times the complexity of the main body.*

Though the LP-command does so, it may not always be acceptable to leak the number of rounds. This has to do with the overall application rather than the cryptographic aspects though. Hiding an execution path involves performing all possible paths conditionally, but obliviously. For loops this means that an upper bound on the number of iterations, U, must be determined and this many iterations performed. Once the looping condition becomes false, "dummy iterations" with no effect must be executed.

Letting bit $[d]$ obtained from the loop condition denote whether the current iteration is a "dummy." For all variables being updated (the tableau, the basis-array, and previous pivot element) a conditional selection must be performed, choosing the original if d is set. As U is an upper bound, termination occurs within this many rounds and the correct solution is found. Under big-$\mathcal{O}$, the complexity of the loop body does not change, and no information at all is revealed through this. The drawback is that U iterations must always be

executed, which for simplex is exponentially large, i.e. infeasible in general.

Compromises between the two strategies leaking less information is possible. In general simplex is efficient, thus it is very likely that termination occurs swiftly. Choosing a *likely* "upper bound" rather than a guaranteed one, and only verifying publicly that termination has occurred after that many iterations reveals little information. Alternatively, rather than testing publicly at every iteration, dummy pivots may be utilised such that only every k'th is published for some fixed k. This ensures that the exact number of pivots is kept hidden, while roughly the correct number of iterations are performed.

15.5 Remaining Issues

The LP-command computes the final tableau containing the assignment to the variables $x_1, \ldots, x_n$ and the evaluation of f. The solution is immediately obtainable from the basis, the b-vector, the z-value, and the final pivot element $[p']$; $x_{\mathrm{val}([\mathbf{S}](j))}$ is assigned the rational value $\frac{\mathrm{val}([\mathbf{B}](j))}{\mathrm{val}([p'])}$. That the x_i may take non-integer values is not a problem, as they are encoded as a numerator and a denominator.

Unfortunately, outputting $[\mathbf{S}]$, $[\mathbf{B}]$, and $[p']$ discloses additional information. Examples include which basis variables take the value 0, which slack-variables are in the basis, and which basis variable is associated with which constraint. Moreover, if the assignments to specific variables are not obtained, then further computation cannot require these.

Therefore, computing an array $[\mathbf{X}]$ with $[\mathbf{X}](i)$ containing the numerator assigned to x_i is desirable; the denominator is ignored for now as this is always $[p']$. Computing $[\mathbf{X}]$ may be done using protocol 21. Recalling that all co-basis

Protocol 21 Assigning the solution to the variables, x_i

Input: The solution stored as the b-vector $[\mathbf{B}]$ and the list of basis variables $[\mathbf{S}]$.

Output: An array $[\mathbf{X}]$, such that $[\mathbf{X}](i)$ takes the value of the numerator of the value assigned to x_i.

```
for i = 1,...,n do
    [X](i) ← 0
end for
for i = 1,...,m do
    [X]([S](j)) ← [B](j)
end for
```

variables take the value 0, the intuition is that all (non-slack) variables are initialised to 0, and set to the relevant $[\mathbf{B}](j)$. Note that this requires the $[\mathbf{S}](j)$ to be indexes. This is not unacceptable, though the updates of $[\mathbf{S}]$ become more expensive – the complexity is no worse than that of updating the tableau. A better solution would be to store the basis as integers, compute the index representation of each of the $[\mathbf{S}](j)$ from scratch at the end.

Correctness and privacy are immediate, and so is the communication complexity – m indexed assignments are made implying $m \cdot n$ secure multiplications.

The naïve round complexity is not acceptable, however, by noting that no entry of $[\mathbf{X}]$ is updated more than once, all computation may be performed in a single round. Consider the following example of two updates:

$$[\mathbf{A}]([\mathbf{i}]) \leftarrow [x]; \quad [\mathbf{A}]([\mathbf{i}']) \leftarrow [x'].$$

For $\mathrm{val}([\mathbf{i}]) \neq \mathrm{val}([\mathbf{i}'])$ this is equivalent to letting

$$[\mathbf{A}](j) \leftarrow [\mathbf{i}](j) \cdot ([x] - [\mathbf{A}](j)) + [\mathbf{i}'](j) \cdot ([x'] - [\mathbf{A}](j)) + [\mathbf{A}](j)$$

for all j, which allows the multiplications to be performed in parallel. Moreover, this generalises to arbitrary many distinct updates, thus only a single round is required.

Thus, secret variables $[\mathbf{X}]$, $[p']$, and $[z]$ containing the assignments to the x_i and the evaluation of f have been set. Assignments to the x_i may now be used in further computation, however, outputs may still contain undesired information, namely the final pivot element. The fractions may be reducible, thus the final pivot element need not be implied by the solution. Though a minor leak, it may be avoided by reducing all fractions – the assignments to the basis-variables and the evaluation of f. Determining the greatest common divisor of the numerator and denominator of each fraction and dividing both by this (again using Bar-Ilan and Beaver inversion as the remainders are 0) accomplishes this. GCD may be computed e.g. as described in appendix A, included to demonstrate the feasibility of the solution. The assignments to the x_i will then consist of two arrays, the numerators $[\mathbf{X_n}] = [\mathbf{X}]$ and the denominators $[\mathbf{X_d}]$. One subtlety remains, namely that different zeros are possible through denominators; this is easily avoided.

The final point to consider is the choice of field, i.e. the modulus q. Bounds on the values must be provided such that q can be chosen sufficiently larger, such that integer computation is simulated. In [Goe94] broad upper bounds on the maximal bit-length of the b-vector values of the final tableau, as well as the on the value of the final pivot element is provided through Hadamard's inequality. This result may be adapted for the intermediate tableau's. It states that the bit-length required is at most

$$L = \log(det_{max}) + B + F + m + n + 1$$

where B and F are the bit-lengths of the original b_i's and f_i's respectively, and det_{max} is the maximal determinant of an $m \times m$ sub-matrix of constraint-rows. Letting C be the bit-length of the $c_{j,i}$, by Hadamard's inequality,

$$\log(det_{max}) \leq (m \cdot (2C + \log(m)))/2$$

implying a general upper bound. Note that a doubling of this bound is required as fractions are compared using integer comparison.

Though theoretically acceptable, the bound does imply that q must be chosen large, particularly for linear programs with many constraints. However, when an exact solution is provided this is unavoidable, the field must allow sufficient headroom, and values may grow throughout the entire computation. For specific problems, additional domain-specific knowledge may provide a better bounds, though.

Beyond This Thesis

Chapter 16

Concluding Remarks

Recall that the task set out to be accomplished was the application of techniques from multi-party computation to provide solutions to real-world problems. These should be constructed through the use of a general "secure integer computer," allowing the parties themselves – or multiple trusted third parties – to perform the desired computation without disclosing additional information on the inputs. This chapter rounds of the thesis by taking a broad view of the work presented. The results of the previous chapters – both with regard to primitives and applications – are related to the overall goals. Moreover, open questions and possible future directions of research are considered.

16.1 Looking Back

In the preceding chapters, the loosely formulated "secure integer computer" of the introduction was constructed in the form of the ideal functionality $\mathcal{F}_{\mathbf{EMPC}}$; in particular novel protocols providing comparison were presented. This allowed the construction of several efficient, high-level protocols solving complex problems with real-world motivation. The work performed on these primitives represents significant advances in the area of (constant-rounds) multi-party computation. In particular it allows (simpler) construction of large-scale protocols – comparison may be viewed as an available primitive as done in part III of this work.

Regarding the bit-decomposition of chapter 11, this primitive is somewhat overlooked in the present work. It is stressed that though its use here has been limited, it has significant potential as it efficiently – in constant-rounds – bridges the gap from field arithmetic to Boolean computation based on the binary representation of elements. The representation change provides immediate applications, and without it the possibility of obtaining constant rounds solution would be seriously diminished. Importantly, it may be used in the construction of additional integer-primitives, with intuition removed from the arithmetic of the actual field or ring in which computations occur. This was the case for the immediate applications, but also for its use in the GCD-protocol of appendix A – determining the greatest dividing power of 2 is very much an integer problem.

Based on the ideal functionality constructed, several applications were considered. These demonstrated complex tasks performed in the $\mathcal{F}_{\mathbf{MPC}}$-hybrid model, i.e. using multi-party computation. The basic first- and second-price auctions provided immediate applications of the protocol(s) for determining the maximal element, thereby demonstrating the applicability of the protocols of part II. It is noted that both the logarithmic-rounds solution and the constant-rounds solution highlight the ease of using the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model i.e. the advantage of using a clean model of MPC with access to powerful primitives. The loglog-rounds solution represents a minor contribution, as it is merely a direct application of an existing result of parallel algorithms in the present setting.

Both the double auction and the secure solving of LP problems provide further support of this. The solutions for these two tasks also demonstrate that multi-party computation and economics may benefit greatly from each other – with economics providing interesting motivation examples and multi-party computation providing a virtual trusted third party.

However, though the model of computation is powerful, the secure variation of simplex method of chapter 15 demonstrates one weakness: When simulating integer computation, bounds on values must be provided – this may imply that the modulus defining the field in which computation occurs must be chosen quite large. This results in more expensive secure computation, as the transmission of field elements (and also computation on these) require more resources of the realising protocol. However, when a precise solution is required, this cannot be avoided – there must be sufficient precision available to represent all possible values.

One final remark is made, namely that these protocols are (or at least seem) practically realisable. The SCET double auction has been implemented (using an inequality test different from, but related to the ones considered here). Though this has not been touched upon in this work, it has been demonstrated that the protocol is feasible, having a running time measured in minutes. This has been demonstrated in a real-life setting (running over the Internet), see [BDJ$^+$05] for discussion and precise timing data. Though solving LP problems requires a much more complex protocol, the implementation of the double auction does lend weight to the claim that high-level multi-party computation is practically realisable, and that solving LP's securely seems feasible, though running time will most likely be measured in hours or perhaps even days.

16.2 Looking Forward

Concerning possible future avenues of research, there are plenty of open questions. Efficient primitives have been constructed, however, it may be possible to do even better. Alternatively, if not possible then this should be demonstrated, implying optimality of the solutions presented.

Equality testing It is reasonable to conjecture that the probabilistic equality test of chapter 9 is optimal – i.e. that a constant-rounds solution is at least linear

in the security parameter (unless the result is to be output). A solution which is sub-linear in the security parameter *seems* overly optimistic – even when the requirement of a constant-rounds solution is dropped. However, no argument for this is given.

Inequality testing In contrast to the equality test, there are no clear reasons to expect that the complexity of the inequality test should be optimal. The output consists of just one field element, thus it is not clear that the complexity must depend on the underlying field. When the binary representation of elements is considered – as is done in all cases in this work – this seems unavoidable. However, it still leaves open the possibility of protocols based on radically different ideas, which may provide better solutions. Moreover, even if comparison protocols sub-linear in ℓ are *not* possible, there may still be hope for protocols with sub-linear *online* requirements.

Bit-decomposition Regarding bit-decomposition, as the number of output values is linear in the bit-length of the modulus, it is reasonable to expect that this is a lower bound. However, again this is only a conjecture – as only *secure multiplications* are counted and costless computation exists, a sub-linear complexity cannot be ruled out entirely, though it seems *highly* unlikely.

The lower bound considered still leaves a gap, from $\mathcal{O}(\ell)$ to the $\mathcal{O}(\ell \log(\ell))$ solution provided, so in this case, the question of optimality is wide open. Late in the process of writing this thesis, the question has to some extent been answered: It *is* possible to do better, however, the improvement found still does not provide linear complexity. It is possible to perform constant-rounds bit-extraction using only $\mathcal{O}(\ell \cdot \log^{(c)}(\ell))$ multiplications, where for a constant c, $\log^{(c)}$ is the iterated logarithm, i.e. log applied c times. Due to time constraints, this result did not make its way into this thesis, however, the main idea will be sketched here.

Apart from the bit-wise additions required, all of the sub-tasks of the bit-decomposition protocol are linear in the bit-length. Thus, improving the postfix-carry computation implies an immediate overall improvement. Splitting the ℓ-bit addends into blocks of $\ell/\log(\ell)$ bits each, unbounded-fan-in carry propagation is performed on all these blocks. This may be performed in a constant number of rounds using a linear number of secure multiplications. Viewing the outcome as a postfix-carry problem of size $\ell/\log(\ell)$, this may be solved using the original solution of chapter 11 in $\mathcal{O}(\ell)$ multiplications overall as the problem is of smaller size. This represents the solution for every $\log(\ell)$'th carry-bit of the original problem, i.e. it has been transformed to $\ell/\log(\ell)$ sub-problems of size $\log(\ell)$. Applying the original solution to each of these requires only $\mathcal{O}(\ell \cdot \mathrm{loglog}(\ell))$ multiplications in all, this becomes the new overall complexity. Alternatively the idea may be repeated; this results in problems of size $\mathrm{loglog}(\ell)$. Performing $c-1$ such divisions into logarithmic-length sub-problems, provides the stated complexity.

$\log^{(c)}$ is "essentially constant" on all "realistic" inputs – in the same manner that $\log^*$ is "essentially constant." Thus, this nearly bridges the gap between the

conjectured lower bound and the upper bound demonstrated. However, from a theoretic point of view, there is still some distance to go. It is noted that several variations of the above are possible. First, bit-decomposition may now be performed using a linear number of secure multiplications in $\mathcal{O}(\log^{(c)}(\ell))$ rounds for any constant c: Simply reduce the problem sufficiently and perform the logarithmic-rounds solution noted in chapter 11. Alternatively, the improvement may be repeated $\log^*(\ell)$ times. This provides the answer using $\mathcal{O}(\ell \log^*(\ell))$ multiplications using $\mathcal{O}(\log^*(\ell))$ rounds.

Further primitives and applications A second category of open problems consists of the construction of additional protocols, both primitives and applications.

As an example of the former, consider the related problems of modulo reduction and integer division. Though not required in the work presented, these are *must haves* if the integer computer is to be considered complete. As noted in chapter 11 it is possible to realise these in constant rounds based on bit-extraction, however, that solution was presented in order to argue feasibility of a constant-rounds protocol. Thus, proper realisations of modulo- and division-commands should be considered.

The construction of further applications involve identifying problems which may benefit from multi-party computation as well as applying it. The initial motivating examples of the introduction may be seen as inspiration, indeed many of the tasks considered have already been the focus of rigorous studies, particularly elections and auctions. However, papers on privacy preserving data-mining have also appeared. Additional motivation in the form of interesting settings is readily obtainable from economics; MPC provides what is essentially a mediator, which may be used to ensure coordination – assuming that the desired computation may be specified within a model of secure computation, e.g. as provided by $\mathcal{F}_{\mathbf{EMPC}}$.

Bibliography

[ACS02] J. Algesheimer, J. Camenisch, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 417–432. Springer-Verlag, Berlin, Germany, 2002.

[BB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In Piotr Rudnicki, editor, *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 201–209, New York, 1989. ACM Press.

[BBNN07] P. Bogetoft, K. Boye, H. Neergaard-Petersen, and K. Nielsen. Reallocating sugar beet contracts: Can sugar production survive in Denmark? *European Review of Agricultural Economics*, 34(1), 2007. Forthcoming.

[BDJ+05] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications. BRICS Report Series RS-05-18, BRICS, 2005. `http://www.brics.dk/RS/05/18/`.

[Bea00] D. Beaver. Minimal latency secure function evaluation. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 335–350. Springer-Verlag, Berlin, Germany, 2000.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10. ACM Press, 1988.

[Bra01] F. Brandt. Cryptographic protocols for secure second-price auctions. In *Cooperative Information Agents V: 5th International Workshop*, volume 2182, pages 154–165. Springer-Verlag, 2001.

[BS05] F. Brandt and T. Sandholm. Efficient privacy-preserving protocols for multi-unit auctions. In A. Patrick and M. Yung, editors, *Financial Cryptography and Data Security, 9th International Conference,*

FC05, volume 3570 of *Lecture Notes in Computer Science*, pages 298–312. Springer Berlin / Heidelberg, 2005.

[Can00a] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[Can00b] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org/`.

[Can01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–147. IEEE Computer Society Press, 2001.

[CCD88] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19. ACM Press, 1988.

[CCR78] A. Charnes, William W. Cooper, and E. Rhodes. Measuring the efficiency of decision making units. *European Journal of Operational Research*, 2:429–444, 1978.

[CD01] R. Cramer and I. Damgård. Secure distributed linear algebra in a constant number of rounds. In J. Killian, editor, *Advances in Cryptology – Crypto 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 119–136. SPRINGER, 2001.

[CDI05] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Proceedings of the Second Theory of Cryptography Conference*, pages 342–362, 2005.

[CDM00] R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer Berlin / Heidelberg, 2000.

[CDN01] R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045, pages 280–300, 2001.

[CFGN96] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 639–648, New York, NY, USA, 1996. ACM Press.

[CFL83a] A. Chandra, S. Fortune, and R. Lipton. Lower bounds for constant depth circuits for prefix problems. In *Proceedings of ICALP 1983*,

volume 154 of *Lecture Notes in Computer Science*, pages 109–117. Springer-Verlag, 1983.

[CFL83b] A. Chandra, S. Fortune, and R. Lipton. Unbounded fan-in circuits and associative functions. In *15th Annual ACM Symposium on Theory of Computing*, pages 52–60. ACM Press, 1983.

[Chv83] V. Chvátal. *Linear Programming*. W. H. FREEMAN, 1983.

[CR03] R. Canetti and T. Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer Berlin / Heidelberg, 2003.

[DFK+06] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryoptography*, Lecture Notes in Computer Science, 2006.

[DGK] I. Damgård, M. Geisler, and M. Krøigård. Efficient and secure comparison for on-line auctions. Manuscript available from the authors, `mk@daimi.au.dk`.

[DJ01] I. Damgård and M. Jurik. A generalization, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *PKC 2001: 4th International Workshop on Theory and Practice in Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 110–136. Springer-Verlag, Berlin, Germany, 2001.

[DN03] I. Damgård and J. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer Berlin / Heidelberg, 2003.

[Fis01] M. Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In David Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 457–471. Springer-Verlag, Berlin, Germany, 2001.

[FKN94] U. Feige, J. Killian, and M. Naor. A minimal model for secure computation (extended abstract). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 554–563, New York, NY, USA, 1994. ACM Press.

[FR95] M. Franklin and M. Reiter. The Design and Implementation of a Secure Auction Service. In *Proc. IEEE Symp. on Security and Privacy*, pages 2–14, Oakland, Ca, 1995. IEEE Computer Society Press.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[Goe94] M. Goemans. Linear programming, October 1994. Course notes, `http://www-math.mit.edu/~goemans/notes-lp.ps`.

[GRR98] R. Gennaro, M. Rabin, and T. Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 101–111, New York, NY, USA, 1998. ACM Press.

[IK97] Y. Ishai and E. Kushilevitz. Private simultaneous messages protocols with applications. In *Proc. 5th Israel Symposium on Theoretical Comp. Sc. ISTCS*, pages 174–183, 1997.

[IK00] Y. Ishai and E. Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st Annual Symposium on Foundations of Computer Science*, pages 294–304. IEEE Computer Society Press, 2000.

[IK02] Y. Ishai and E. Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In *Proceedings of ICALP 2002*, Lecture Notes in Computer Science, pages 244–256. Springer Berlin / Heidelberg, 2002.

[Jáj92] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[KLM05] E. Kiltz, G. Leander, and J. Malone-Lee. Secure computation of the mean and related statistics. In *TCC 2005: 2nd Theory of Cryptography Conference*, volume 3378 of *Lecture Notes in Computer Science*, pages 283–302. Springer-Verlag, Berlin, Germany, 2005.

[Nie03] J. Nielsen. On protocol security in the cryptographic model. Dissertation Series DS-03-8, BRICS, Department of Computer Science, University of Aarhus, August 2003.

[NO07] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC 2007: 10th International Workshop on Theory and Practice in Public Key Cryptography*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 2007. Forthcoming.

[NPS99] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *1st ACM Conf. on Electronic Commerce*, pages 129–139. ACM Press, 1999.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer Berlin / Heidelberg, 1999.

[Per52] O. Perron. Bemerkungen über die Verteilung der quadratischen Reste. *Mathematische Zeitschrift*, 56(2):122–130, 1952.

[RBO89] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, New York, NY, USA, 1989. ACM Press.

[Ros05] G. Rosenberg. Enumeration of all extreme equlibria of bimatrix games with integer pivoting and improved degeneracy check, 2005. CDAM Research Report LSE-CDAM-2005-18, `http://www.cdam.lse.ac.uk/Reports/Abstracts/cdam-2005-18.html`.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[ST04] B. Schoenmakers and P. Tuyls. Practical two-party computation based on the conditional gate. In Pil Joong Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 119–136. Springer-Verlag, Berlin, Germany, 2004.

[Tofa] T. Toft. An efficient, unconditionally secure equality test for secret shared values. Manuscript available from the author, `tomas@daimi.au.dk`.

[Tofb] T. Toft. Solving linear programs using multiparty computation. Manuscript available from the author, `tomas@daimi.au.dk`.

[Tof05] T. Toft. Secure integer computation with applications in economics, 2005. PhD progress report, available online `http://www.daimi.au.dk/~tomas/publications/progress.pdf`.

[Yao82] A. Yao. Protocols for secure computations (extended abstract). In *23th Annual Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164. IEEE Computer Society Press, 1982.

[Yao86] A. Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, 1986.

Appendices

Appendix A

Greatest Common Divisor

Computing the greatest common divisor of two secret variables can be accomplished based on the binary GCD algorithm. This appendix is a light rewrite of an appendix of an unpublished article by the author [Tofb]. It may be seen as an additional contribution by the author that did not make it into the main part of this thesis. It is intended as a feasibility result, demonstrating that performing a GCD securely in the $\mathcal{F}_{\mathbf{EMPC}}$-hybrid model introduced in chapter 12 is possible. Efficiency of the computation performed has not been a concern.

The details of the computation are seen in protocol 23 below, where U denotes an upper bound on the number of iterations. As one of the two values is halved in each iteration, U is $\mathcal{O}(\ell)$, where ℓ denotes the bit-length of the inputs (and thereby the prime q defining the group). However, before considering GCD, a few additional primitives are introduced.

A protocol for extracting only the lowest bit of a value is needed, this is denoted $\texttt{lowbit}(\cdot)$. The same primitive was considered in the construction of the inequality test in chapter 10; for complexity purposes it will be considered equivalent to a comparison. It is noted that if the values are of bounded size, this may be done more efficiently similarly to the comparison of bounded values. A second primitive is seen as protocol 22 – computing the largest power of two dividing a given value, $\texttt{g2p}(\cdot)$. Correctness is easily verified: $[\mathbf{A}]$ is simply the original value "written backwards." Viewing $[\mathbf{B}]$ in the same way, clearly the greatest dividing power is the same for the two. The overall result is therefore 2 to the power of the number of 0's in $[\mathbf{B}]$. This is computed using only costless operations. The complexity is that of bit-decomposition and prefix-`OR`.

With regard to GCD, protocol 23 lines 1 through 5 compute the greatest common power of two, $[g_2]$, and ensure that the inputs are odd. The divisions can be performed using element inversion. The invariant of the loop is that $[a]$ is odd. At each iteration, based on the least significant bit of $[b]$ one of two things happens: If $[b]$ is even then it is halved, while if it is odd, $[a]$ is set to the minimal of the two values and $[b]$ is assigned their difference, halved. In both cases, the division is multiplication by the inverse. The algorithm "terminates" when the two values are equal; when this occurs in the protocol, the following iteration sets $[b]$ to zero. From then on, dummy iterations are performed, $[b]$ is repeatedly divided by two. The output is the product of the smallest power of two and the remaining GCD, $g_2 \cdot a$.

Protocol 22 $\texttt{g2p}(\cdot)$ – Computing the greatest two-power dividing a value

Input: $[a]$ viewed as an integer.
Output: $[t]$, such that $[t]$ is the greatest two-power dividing $[a]$.

$[a]_B \leftarrow \texttt{bits}([a])$
for i = 1,...,ℓ **do**
  $[\mathbf{A}](i) \leftarrow [a_{i-1}]$
end for
$[\mathbf{B}] \leftarrow \texttt{pre}_\vee([\mathbf{A}])$
$[t] \leftarrow 1$
$c \leftarrow 1$
for i = 1,...,ℓ **do**
  $[t] \leftarrow [t] + \big([\mathbf{B}](i)\ ?\ 0 : c\big)$
  $c \leftarrow 2 \cdot c$
end for

Protocol 23 Computing the GCD of two secret variables

Input: $[a]$ and $[b]$.
Output: $[g]$, such that val($[g]$) is the GCD of val($[a]$) and val($[b]$).

$[a_2] \leftarrow \texttt{g2p}([a])$
$[b_2] \leftarrow \texttt{g2p}([b])$
$[g_2] \leftarrow \big([a_2] \overset{?}{<} [b_2]\big)\ ?\ [a_2] : [b_2]$
$[a] \leftarrow [a]/[a_2]$
$[b] \leftarrow [b]/[b_2]$
for i = 1,...,U **do**
  $[min] \leftarrow \big([a] \overset{?}{<} [b]\big)\ ?\ [a] : [b]$
  $[max] \leftarrow [a] + [b] - [min]$
  $[b_0] \leftarrow \texttt{lowbit}([b])$
  $[a] \leftarrow [b_0]\ ?\ [min] : [a]$
  $[b] \leftarrow [b_0]\ ?\ [max] - [min] : [b]$
  $[b] \leftarrow [b]/2$
end for
$[g] \leftarrow [a] \cdot [g_2]$

Regarding complexity, two bit-decompositions and prefix-OR's are needed initially, while $U \in \mathcal{O}(\ell)$ comparisons and multiplications are needed for the loop. These do *not* parallelise, implying a round complexity of $\mathcal{O}(\ell)$.