Unconditionally Secure Protocols

PhD thesis submitted by
Sigurd Meldgaard

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

To Daffy

Great thanks go to my advisor Ivan Damgaard for his excellent advising during my entire PhD studies.

To Yuval Ishai and the Technion crypto for taking me in and helping me having a fun and fruitful stay in Haifa.

To my co-authors whose work is represented in this thesis.

To the crypto-group in Aarhus — best colleagues ever.

To my family who are there for me.

And to my girlfriend who has supported me greatly and put up with me during the writing of this thesis.

Abstract

This thesis contains research on the theory of secure multi-party computation (MPC). Especially information theoretically (as opposed to computationally) secure protocols.

It contains results from two main lines of work. One line on Information Theoretically Secure Oblivious RAMS (covered in Chapter 3 and 4), and how they are used to speed up secure computation. An Oblivious RAM is a construction for a client with a small $O(1)$ internal memory to store N pieces of data on a server while revealing nothing more than the size of the memory N, and the number of accesses. This specifically includes hiding the access pattern.

We construct an oblivious RAM that hides the client's access pattern with information theoretic security with an amortized $\log^3 N$ query overhead. And how to employ a second server that is guaranteed not to conspire with the first to improve the overhead to $\log^2 N$, while also avoiding the bottleneck of sorting networks. And we show how to utilize this construction for four-player MPC.

Another line of work (covered in Chapter 2) has results about the power of correlated randomness; meaning in a preprocessing phase the participants in a MPC protocol receive samples from some joint distribution to aid them implement the secure computation. Especially we look at the communication complexity of protocols in this model, and perfectly secure protocols.

We show general protocols for any finite functionality with statistical security and optimal communication complexity (but exponential amount of preprocessing). And for two-player functionalities where only one player receives output (sender-receiver functionalities) with perfect security.

We also show protocols for some specific sender-receiver tasks with both optimal communication and small preprocessing.

We show lower bounds on the amount of communication and show the impossibility of general perfect protocols when both parties receive output.

Also we show how to use the sender-receiver protocols with perfect security given correlated randomness to construct secure protocols in the plain model with perfect correctness.

Resume

Denne afhandling indeholder forskning om teorien bag sikker flerpartsberegning (MPC).

Fokus er især på informationsteoretisk (modsat beregningsmæssigt) sikre protokoller.

Resultaterne dækker to hovedområder.

Det ene hovedområde er informationsteoretisk sikker Oblivious RAM (Se kapitel3 og 4) og hvordan det kan bruges til at gøre sikker beregning hurtigere. En oblivious RAM er en konstruktion for en klient med en lille ($O(1)$) intern hukommelse til at gemme N stykker data på en server uden at afsløre mere end størrelsen på hukommelsen N og antallet af opslag i hukommelsen. Dette inkluderer specifikt at skjule tilgangsmønstret.

Vi konstruerer en oblivious RAM som skjuler tilgangsmønstret med informationsteoretisk sikkerhed som amortiseret bruger $O(\log^3 N)$ så mange tilgange. Og hvordan man kan bruge endnu en server som man er sikker på ikke er sammensværget med den første til at forbedre omkostningerne til $O(log^2 N)$ mens den samtidig undgår at bruge sorteringsnetværk som typisk er en flaskehals for oblivous RAM. Og vi beskriver hvordan dette kan bruges til at lave firparts MPC.

Det andet hovedområde (kapitel 2) indeholder resultater om hvor kraftfuld korreleret tilfældighed. D.v.s. at deltagerne i en MPC-protokol modtager prøver (samples) fra en ikke uafhængig distribution der kan hjælpe dem med beregningen i en præprocesseringsfase før input er kendt. Vi ser især på kommunikationskompleksiteten af protokoller i denne model og på protokoller med perfekt sikkerhed.

Vi beskriver generelle protokoller for at beregne en hvilkensomhelst endelig funktionalitet med statistisk sikkerhed og optimal kommunikationskompleksitet men dog med en eksponentiel mængde præprocessering. Og for topartsberegninger hvor kun den ene deltager modtager output (sender-modtager-funktionaliteter) med perfekt sikkerhed.

Vi beskriver også protokoller for visse sender-modtager-funktionaliteter med perfekt sikkerhed og lille præprocessering.

Vi viser nedre grænser (lower bounds) for mængden af kom-

munikation og viser at det er umuligt generelt at lave perfekt sikre protokoller når begge spillere modtager output.

Vi viser også hvordan man kan bruge sender-modtager-protokollerne med perfekt sikkerhed givet korelateret tilfældighed til at lave sikre protokoller i “the plain model”(model uden opsætningsantagelser) som har perfekt korrekthed.

Contents

Contents **vii**

1 Introduction **1**
1.1 Secure computation . . . 1
1.2 Defining Security . . . 2
1.2.1 Adversaries . . . 3
1.2.2 Functionalities . . . 4
1.2.3 Protocols . . . 4
1.2.4 The Ideal world execution . . . 5
1.2.5 Simulation . . . 6
1.2.6 Distinguishing . . . 7
1.3 Computational security . . . 8
1.4 Unconditional security . . . 9
1.4.1 Perfect Security . . . 9
1.5 Composition and Concurrency . . . 10
1.6 Example Functionality: Oblivious transfer . . . 11
1.7 Preprocessing with correlated randomness . . . 11
1.7.1 The power of preprocessing . . . 14
1.8 Oblivious RAM and secure computation . . . 14
1.8.1 Two-server Oblivious RAM . . . 17

2 On The Power of Correlated Randomness **19**
2.1 Introduction . . . 19
2.1.1 Our Results . . . 21
2.1.2 Summary of contributions . . . 23
2.1.3 Related Work . . . 25
2.1.4 Warmup: Equality Test . . . 26
2.2 Preliminaries . . . 28

2.3 Optimal Communication for General Functionalities 32
2.3.1 Upper Bounds on Communication Complexity 33
2.3.2 Semi-Honest Two-Party Protocol via One-Time Truth Tables 34
2.3.3 Malicious Protocol via One-Time Truth Tables 35
2.3.4 Multiparty Generalization 38
2.4 Perfect Security for Sender-Receiver Functionalities 41
2.5 Protocols For Specific Tasks 43
2.5.1 Set Intersection 43
2.5.2 Inner product 46
2.5.2.1 Some Algebraic Preliminaries 46
2.5.2.2 The protocol. 48
2.6 Perfect Security for General Functionalities is Impossible . . . 50
2.6.1 Trade-off between Communication and Statistical Security 52
2.6.2 With expected round complexity 54
2.7 Impossibility of a Perfectly Secure Multi-Party Protocol 55
2.8 Negative Results on Communication and Randomness 56
2.8.1 Communication Lower Bounds 56
2.8.1.1 Proofs 58
2.8.2 A Link Between Storage Complexity And Private Information Retrieval 66
2.9 Perfect Correctness in the Plain Model 67
2.9.1 From the Preprocessing Model to the Plain Model . . . 67
2.9.2 Perfect Correctness is Not Always Possible 69
2.9.3 Applications of Perfect Correctness 70
2.10 Future Directions 72
2.10.1 Combining Multiple Sources — the Commodity Based Model 72
2.10.2 Generalized OT extensions 72
2.10.3 More efficient protocols 73
2.10.4 More Restricted Communication Models 73

3 Information Theoretic Oblivious RAM 75
3.1 Introduction 75
3.2 Related work 77
3.3 Applications 78
3.4 The model 79

3.5 Oblivious sorting and shuffling . . . 81
3.6 The square root algorithm . . . 82
3.6.1 Making a lookup . . . 82
3.6.2 Obliviously shuffling a tree . . . 83
3.6.3 Security . . . 86
3.6.4 Performance . . . 86
3.7 The Polylogarithmic Construction . . . 86
3.7.1 Merging and shuffling of two partial trees . . . 88
3.7.2 Security . . . 93
3.7.3 Performance . . . 94
3.8 Lower bounds on randomness . . . 94
3.8.0.1 Extensions . . . 97
3.8.1 Related bounds . . . 98
3.9 Discussion . . . 98
3.9.1 Random access in the MPC setting . . . 98
3.9.2 Future Directions . . . 99

4 Two-server Information-Theoretic Oblivious RAM 101
4.1 Introduction . . . 101
4.2 Model . . . 102
4.2.1 Avoiding the extended interface . . . 104
4.2.2 . . . 104
4.3 Trivial information theoretic solution . . . 104
4.4 Idea of the construction . . . 105
4.5 Reshuffling . . . 105
4.6 Statement . . . 106
4.6.1 Security . . . 107
4.6.2 Overhead . . . 107
4.7 Why do we not need tags . . . 107
4.8 Use for Multi Party Computation . . . 107
4.9 Round complexity . . . 108
4.10 Future directions . . . 108

Bibliography 111

1 Introduction

1.1 Secure computation

The basic task of secure computation (often called MPC for secure Multi Party Computation) consists of having several parties, the *players*, do a joint computation on their private inputs without revealing more information about those inputs than what the output of the computation reveals anyway.

Because the notion of secure computation is so general it can be hard to visualize when this will be useful. Here are a few examples of situations where secure computation can help.

Auctions An auctioneer wants to sell some item to the highest bidder, but the bidders are unwilling to reveal their pricing strategy.

Matching Two groups of people want to be matched in pairs up in a way respecting everyone's preferences as much as possible. But they do not want to announce their priorities to other participants.

Set membership An airline company wants to find out if the name of a passenger is on the state's list of terrorist suspects. But the airline is not willing to reveal the passenger's identity to the state, and the state is not willing to openly reveal the list of suspects.

Voting A new president has to be chosen, and the voters do not want to reveal who they are voting for — still they want their vote to count.

Shared Research Two hospitals want to do medical research across their patient databases. But each hospital is subject to strict privacy regulations, and cannot freely exchange patient data.

All of these problems could easily be solved if there was a trusted third party willing to do the computation for the mutually distrustful parties. The third party would receive the inputs from all parties, do the computation and send the result back as the output.

In the real world this third party could be (and often is) a lawyer, paid to handle the data confidentially. The problem with such a solution is the salary the lawyer will take just for keeping the data confidential, and also the chances that he might not do it anyway. Some of these problems can sometimes be handled somewhat satisfactory by physical devices (for example voting ballot boxes).

The field of *secure computation* is a subfield of cryptology focusing on the study of *protocols* for performing these computations, that are *as good* as the ideal trusted third party in terms of privacy. In other words protocols that output the desired results to the players, but do not reveal anything else (that could not be computed by the parties anyway). It is a fundamental problem that has been extensively studied since the 1980s, originating from the works [Yao82, GMW87, BGW88, CCD88].

1.2 Defining Security

Here we will present a somewhat informal definition of security in the spirit of [Gol04]. A similar but more terse and formal definition appears in the preliminaries of Chapter 2.

This way of defining security is known as the *stand-alone* model of security, and we will briefly discuss the stricter notion of *UC-security*.

There are many aspects of security for a distributed computation that we might care about, e.g. correctness, privacy, authenticity, anonymity, assurance that the computation terminates, fairness etc. And it is a hard, if not impossible job to enumerate all the different properties that are relevant to a given use-case and proving that the protocol has each property. Therefore cryptographic protocols are often proven secure by direct reference to the

given functionality in a "real-world/ideal-world" argument. Loosely spoken this is a way of saying: we already know what *functionality* we would like a trusted third-party to implement for us (the ideal world), so *anything* an adversary is allowed to do by the protocol (the real world) and anything it is allowed to see that deviates from this specification is considered a breach of security.

This, at first seems to put a lot of pressure on the designer of the functionality — the designer cannot give the responsibility away to a cryptographer to "make the computation secure", but rather has to define rigorously what is and is not allowed behavior or even a secure protocol for the functionality would not be useful. But this really just highlights the necessity that exists anyway, to somewhere decide exactly what is meant by *security*. There exist no cryptographic "pixie dust" one can just powder on any interactive scheme, and call it secure and then there is no way to misuse it. The cryptographer only works to promise that some protocols works as well as the trusted third party.

As a trivial example, one might ask for a *secure* computation of the mapping $f(x, y) \mapsto (x \oplus y, x \oplus y)$ from two players' input bits x, y to letting both players learn the xor of the bits. However, the output of the computation together with one player's own input reveals the other player's input $x \oplus y \oplus x = y$. So there is nothing a cryptographer can do to make something better than a protocol where both players just exchange their inputs. And exactly this protocol would be considered secure from the cryptographer's standpoint.

When that is said, there is still a lot to do for the cryptographer in the general case when receiving a functionality to implement securely, as most functionalities do not give away all information.

1.2.1 Adversaries

To model security we always try look at worst cases and assume as little as possible about our adversaries. To simplify we assume that we are up against a single "mastermind" *adversary* who has he power to *completely* corrupt a number of the players, controlling their every action and seeing all their communications. And then we assume that the rest of the players are honest.

Several models of security have been defined with different assumptions about the abilities of the adversary. We can ask how many of the players (or

more generally which subsets of the players) it can corrupt. If it can corrupt less than half of the players we talk of *honest majority* otherwise *dishonest majority*.

One other important distinction for strength of the adversary is that of *active* or *malicious* versus *passive* or *semi-honest* adversaries.

A *passive adversary* is assumed to always follow the protocol as stated, but is still trying to learn whatever extra information he can from his transcript of the protocol.

Whereas an *active adversary* can do anything, send any message based on all his knowledge at that point in time.

The passive model might seem too weak to realistically model anything in the real world, and it is indeed often considered only a stepping stone for the construction of protocols with active security — first the protocol is designed so it is secure against the passive adversary, and then it is modified in order to force a malicious adversary to behave according to the protocol (most generally by proving in zero-knowledge for every step that he followed the protocol). However in some real-world situations the security promised by a semi-honest protocol might be considered strong enough, and that seems to be the case for the actual uses of secure computation that has been done as in [BCD+09].

1.2.2 Functionalities

By a *functionality* we think of the task we would like the trusted third party to do for us of mapping of the players' inputs (typically in some finite domain) to some output to each player. This mapping does not have to be deterministic, but can include randomness, so we can model it as a function with one extra input that is supposed to be a random bitstring $f(r, x_1, \ldots, x_n) \mapsto (z_1, \ldots, z_n)$.

Many times we will equate the functionality and the function it computes.

1.2.3 Protocols

A protocol π for a functionality f is a specification of the messages that players have to send to each other to compute the same result as the functionality.

A protocol can be specified by a function $next(i, x_i, r_i, m_{i,j}, j)$ determining what the honest player P_i should do in round j depending on every-

thing that player has seen: the input x_i, random tape r_i, and message history seen by that player until now $m_{i,j}$. The value of *next* can be of the form $(\text{"}send\text{"}, v_1, \ldots, v_n)$ to send a message to each player. Or it can be $(\text{"}terminate\text{"}, out)$ telling the player should terminate the protocol and locally output something.

An adversary can corrupt some players to try to change the honest players outcome of the protocol from what they output in the real world and to get more information out of he execution. This can be attempted by specifying any alternative messages to send in each round (these can depend on the compound view from all the corrupted players).

In this work we mostly consider *static adversaries* who have to decide which players to corrupt before the protocol run. We could also consider the harder case of *adaptive adversaries* that can choose whom to corrupt at any point.

The real world *view* of an adversary $\mathcal{A}$, $\mathbf{view}_{\mathcal{A},\pi}(x_1, \ldots, x_n)$ is the concatenation of all the messages seen by the corrupted players during the protocol with those inputs.

The real world *output* of the honest players in the same protocol we call $\mathbf{output}_{\mathcal{A},\pi}(x_1, \ldots, x_n)$.

The real world *execution* $\mathbf{exec}_{\mathcal{A},\pi}(x_1, \ldots, x_n)$ is the concatenation of the view and the output.

1.2.4 The Ideal world execution

The ideal world execution defines exactly what influence the adversary is allowed to have when the protocol runs. The ideal execution is as follows:

- Each player gets their input $(x_1, \ldots, x_n)$
- Each of the corrupted players P_i delivers an input x'_i of their choice (can be related to their original input) to the ideal functionality.
- Each of the honest players deliver their original input.
- The functionality computes the specified randomized mapping f.
- The functionality outputs the computed values to the corrupted players. They can locally output whatever they want based on what they saw until now.

- The functionality outputs to the honest players. They locally output the correct output.

If we want to protect against a majority of corrupted players we typically allow the adversary to interrupt the protocol at any point (thus for example depriving the honest players of their output) this is called *security with abort* if we disallow this we call it *full security*. This is done because as is observed in [Cle86] it is impossible to implement functionalities without abort in this case.

We let the corrupted players input any value to the computation, since it is not really meaningful to restrict players to input something specific.

1.2.5 Simulation

Even when we have defined the ideal security of some interactive scheme in a way that is really hindering misuse of the interaction, the view of the protocol can usually not be made to look exactly like the ideal world execution. Therefore a so called "simulation-based" approach is taken.

The *simulator* $\mathcal{S}$ is an entity (specified as an algorithm) that takes what the adversary can see do in ideal world execution (the ideal view), and from that *simulates* the view that the adversary sees in the real world execution, the *simulated view*.

The *simulated view* of a simulator $\mathcal{S}$, $\mathbf{view}_{\mathcal{S},f}(x_1, \ldots, x_n)$ is what the simulator outputs when run with access to the ideal functionality, corrupting the same players as the real world adversary.

The *ideal world output* of the honest players in the same protocol we call $\mathbf{output}_{\mathcal{S},\pi}(x_1, \ldots, x_n)$.

The *ideal world execution* $\mathbf{exec}_{\mathcal{S},f}(x_1, \ldots, x_n)$ is the concatenation of the simulated view and the ideal world output.

Note that the simulator is allowed access to the program of the adversary to create the view.

If one cannot *distinguish* between what the adversary is seeing in the real world view and a view simulated only from the adversary's view in the ideal world, it follows that in the ideal world (where he learns as little as we might hope) he might as well have run the simulator on his view to get a something as good to him as the real-world view — therefore he did not learn anything new from the protocol execution.

We will discuss more below what is here meant by distinguishing.

1.2.6 Distinguishing

We define security by requiring that the simulator must be able to produce a view such that the real world and the ideal world executions cannot be distinguished from each other. The respective executions can be modeled as *random variables* that each will have a certain distribution.

Given samples from one of two distributions, how easy is it to guess which one you got?

First we need a way to express how different the two distributions (with support D) are. This is called the *statistical distance*:

$$\Delta(D_0, D_1) = \frac{1}{2}\sum_{d \in D} |Pr(D_0 = d) - Pr(D_1 = d)|$$

Now we want to say that the two distributions are *in*distinguishable if the distance between them is small. But how small? Well small enough! indistinguishability is defined in an asymptotical manner, for some security parameter σ (usually signifying the length of the input to the process doing the sampling) and two ensembles of distributions $D_0(\sigma), D_1(\sigma)$, we call D_0, D_1 statistically indistinguishable if the distance between them is expressed by some function $negl(\sigma)$ that is asymptotically smaller than any inverse polynomial in σ.

To define the security of a protocol we let some adversary A process either the real view or the simulated view and then give a guess.

Now we are ready to define security: A protocol is *secure against a class of adversaries* if there for any adversary in that class exists a simulator such that for any input the adversary cannot distinguish between the simulated view and the real-world view.

$$\Delta(A(\mathbf{exec}_{\mathcal{A},\pi}(\sigma)), A(\mathbf{exec}_{\mathcal{S},f}(\sigma))) = negl(\sigma)$$

($A(B)$ where A and B are randomized processes or distributions here means: sample from B and input the result to A).

Different adversarial powers lead to different definitions of secure computations. Typically we are concerned with three kinds of protocols. Those with *computational* security, also called *cryptographically* secure computations, those with *statistical* (also called *unconditional* or *information theoretic*) security and those with *perfect* security.

For *computational* security we restrict the adversary to be in a certain complexity class (typically PPT probabilistic, polynomial time). PPT is

polynomial-time in the input, therefore we often invent a computational security parameter that is (implicitly) passed to all processes in unary, and require the process to be polynomial in that.

To be statistically secure we give the adversary unlimited computational power — the best thing an unlimited adversary can do is to just return his input. Therefore this is equivalent to saying that the real view and the simulated view of the protocol themselves have a negligible distance in the security parameter.

Perfect security is when the two views really are the distributed the same way. That is when $\Delta(exec_{A,\pi}, exec_{\mathcal{S},f}) = 0$.

1.3 Computational security

The security of computationally secure protocols is always relying on the assumed hardness of some underlying computational problem. One of the most well-known such problems is factoring: the task of taking an integer and breaking it down to its prime factors. Much research has been done on this problem, and still no known (non-quantum) algorithm can factor a so called RSA-modulus ($N = p \cdot q$ for correctly chosen primes p, q) in polynomial time in the bit-length of N.

As an example: the assumption that factoring is hard implies that squaring modulo N is a one-way function (and furthermore something that can be used to implement a trap-door, one-way permutation as shown in [Rab79]). And many cryptographic primitives can be based on the existence of one-way functions, and even stronger primitives (for example MPC without honest majority) on the assumption of trap-door one-way permutations.

However, to this date no function has really been *proven* to be one-way in the sense that it is easy to compute but hard to invert. And, as this would imply answering one of the greatest questions of computer science;

Furthermore relying on the computational hardness of some problem implies that a computationally unbounded adversary will always be able to invert the function (given time enough you can invert any function by enumerating all possible inputs to the function).

1.4 Unconditional security

Unconditionally secure protocols are secure in a deeper sense. They do not put the private data in any risk. What an adversary sees contains practically no information besides what can be obtained in the ideal world. No matter what amount of computing power the adversary has available, it will never be able to cheat.

Unfortunately general unconditionally secure protocols are not possible without strong assumptions.

One sufficient set of assumptions for general multiparty computation is secure channels and honest majority.

Secure channels means private, authenticated channels, such that any party can send any other party a message privately, and you can verify that messages received are from the claimed sender. These channels can be implemented with encryption and signatures given a public-key infrastructure, but that would leave the whole protocol computationally secure again.

Honest majority means that the adversary can corrupt at most $t < \frac{n}{2}$ of the players, so for two-party computation we cannot tolerate any players to be adversarial.

As we will see in Chapter 2 other, stronger setups can allow for information theoretically secure computation between two players with one of them corrupted.

1.4.1 Perfect Security

A statistical security of e.g. 2^{-80} gives such a good security that one might ask if designing a protocol with perfect security really has real-world reason. A easily overseen advantage of perfectly secure protocols besides the clarity of the security statement, (even if they are used on top of a computationally secure infrastructure) is that they often lead to inherently efficient algorithms. It can be shown that the asymptotic complexity of any $2^{-\sigma}$-secure protocol must grow at least linearly with σ. As nothing in a perfectly secure protocol can depend on a security parameter nothing will “grow” when the security is raised.

Another advantage is that there is no need to choose security parameters — the protocols are guaranteed to be secure now and forever, in spite of effects like the exponential growth in computing power purported by Moore’s Law.

It can be very hard to estimate the input parameters needed for reaching a given level of computational security.

As a (scaring) real world example, the RSA scheme [RSA78] was initially suggested to be used with a 200-digit public modulus (around 664 bits) for providing a "margin of safety against future attacks". In 2009 the RSA-768 challenge has been broken[1], and 1024 bit keys are no longer being recommended. On the other hand, all secure communication done via one-time pads during the cold war is still secure, and is guaranteed mathematically to remain that way, (unless flaws are found in the management of the one-time pads or the random number source turns out to be flawed - note that these problems (wrong handling of secret keys, bad random number generation) are at least as critical for computational hard protocols, though keys might be easier to distribute.)

1.5 Composition and Concurrency

Security as defined here is often called *stand-alone security*. One problem with this definition is that it only guarantees security for a single instance of the protocol.

Running just two instances of a secure protocol in parallel can completely break security. For example [GK96] gives such a protocol.

An alternative security definition called *Universally Composible security* or *UC-security* is presented by Canetti [Can01] to amend these problems. A protocol is UC-secure if it is secure in relation to any feasible *interactive environment*. The environment is an interactive protocol that provides inputs and receives outputs to the players and can run parallel instances of any protocols at the same time and try to aid the adversary.

Now a protocol is secure if any real-world adversary with help from an environment can be emulated by some ideal-world adversary (the simulator) interacting with the same environment.

A main difference from the stand-alone model is that the simulator has to interact with the environment in the same way as the adversary, and thus cannot "rewind" the environment (the simulation has to be "straight-line").

This way of modeling the security also makes it easier to talk about reactive functionalities that do not only compute a single function, but keep

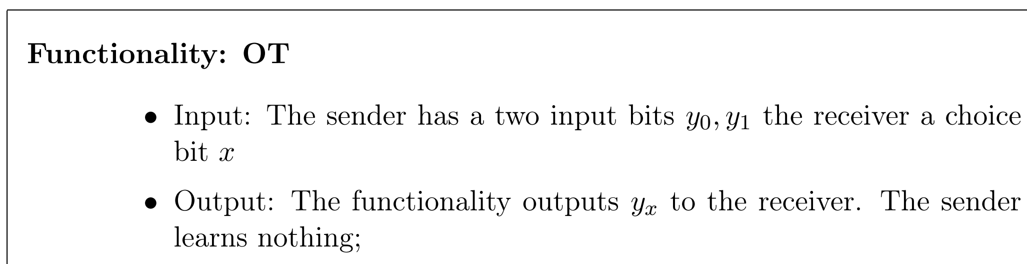
[1] See `http://www.crypto-world.com/FactorRecords.html`

a state between invocations and run a general interactive protocol with the players.

The advantage of protocols secure in this model is that any execution of the protocol can be replaced by an oracle call in a hybrid model to show the security of a composed protocol.

1.6 Example Functionality: Oblivious transfer

One of the simplest yet, it turns out, most useful non-trivial functionalities is called Oblivious transfer (OT). It exists in many variants, one of the simplest variants *one-out-of-two bit-OT* presented in Figure 1.1, is a protocol between two players the *sender* and the *receiver*, and creates an asymmetry of information between the players.



Functionality: OT

- Input: The sender has a two input bits y_0, y_1 the receiver a choice bit x
- Output: The functionality outputs y_x to the receiver. The sender learns nothing;

Figure 1.1: One out of two bit-Oblivious Transfer functionality

Notice that the sender learns absolutely nothing about x and the receiver learns nothing about y_{1-x}. Common variants of OT are: *String-OT* where the secrets n-bit strings. 1-*out-of-k-OT* with k secrets. *Rabin-OT* with 1 secret that is transmitted with probability 1/2 (and no input from the receiver). These can all be reduced to each other.

Many nice properties are known about this primitive functionality. Most important maybe, that it is universal for secure computation[Kil88].

Another nice property of OT is that it can be "extended", so a few OT's can be made into polynomially many OT's. [Bea96, IKNP03, Nie07].

1.7 Preprocessing with correlated randomness

Statistical security is not possible in the plain model if we do not have an honest majority. There are however different set-up assumptions that can

allow us to circumvent this impossibility. One of these set-up assumptions is to give the players samples from a joint distribution. This can be done even before the players know their inputs. One way to create such samples is to get them from a MPC protocol (without inputs) run in a preprocessing phase that outputs the samples distributed as wanted.

Imagine you want to tally the votes for each candidate on election day by a secure computation. The Cryptographic protocols implementing the computation will many times have high communication costs and computation time leading to a long wait for the election results after the votes are cast.

There might however be a long time between elections — during all this time the computer network stands idle.

Here the possibility of "preprocessed" secure computation comes in handy. You trade time spent ahead of the arrival of the inputs to precompute some values distributed amongst the players, that can then be turned into the actual results in the online phase.

We extend the real world execution by starting the protocol with sampling from some given distribution, and giving each player a part of the sample (so that these parts are correlated).

A prime example of a cryptographic task that can benefit from having access to correlated randomness is oblivious transfer (OT) [Rab81, EGL85]. Beaver [Bea95] shows that having access to the inputs and outputs of a random instance of OT can be used to realize OT on any inputs with unconditional security. This very simple reduction is shown in figure 1.2

The correctness of the protocol is immediate.

The security is shown by constructing a simulator that for any adversary $\mathcal{A}$ and access to the ideal execution can make a simulation of the adversary's real-world view such that the two executions are distributed the same.

If none of the players are corrupted the simulator has no work to do as the adversary will have no transcript in the real world.

If they are both corrupted there is no secret to protect and the simulator can just run the program of the adversary to create a transcript.

If only one player is corrupted the simulator as a first step will construct randomness (y'_0, y'_1, z) distributed correctly as in the protocol.

If the sender is corrupted the simulator will run $\mathcal{A}$ as a subprotocol, compute a random u as the first message. Send this u to the real world adversary who responds with some $v = (v_0, v_1)$. Because there are only four different messages of this form this binds the adversary to some choice of input $y = (v_0 \oplus y'_u, v_1 \oplus y'_{1\oplus u})$. The simulator inputs this y to the ideal

Functionality: OT

- The receiver has input $x \in \{0,1\}$, the sender input $y_0, y_1 \in \{0,1\}$;
- The receiver learns y_x. The sender learns nothing;

Preprocessing:

1. Sample three random bits y'_0, y'_1, z;
2. The preprocessing outputs (z, y'_z) to the receiver and (y'_0, y'_1) to the sender;

Protocol:

1. The receiver computes and sends $u = x \oplus z$ to sender;
2. The sender computes and sends $v = (y_0 \oplus y'_u, y_1 \oplus y'_{1\oplus u})$ to the receiver;
3. The receiver outputs $v_x \oplus y'_z = y_x \oplus y'_{u\oplus x} \oplus y'_z = y_x \oplus y'_{x\oplus z\oplus x} \oplus y'_z = y_x$;

Figure 1.2: A reduction of OT to preprocessed randomness

functionality. Now the simulated execution looks as $((v_0, v_1, z, v_z), u, v), y_x$. These values are distributed exactly as in the real protocol execution. (Notice only messages from the corrupted party strictly needs to be included, the honest party's messages are given by the protocol).

If the receiver is corrupted the simulator will get some u as the first message from the adversary. Now compute the real $x = u + z$ and send it to the ideal functionality. The functionality will respond with y_x.

Now we can construct the last message $v_x = y_x \oplus y'_z$ and choose $v_{x\oplus 1}$ at random. Again the execution $((v_0, v_1, z, v_z), u, v), y_x$ is distributed exactly as in the real protocol execution.

This protocol, together with the fact that OT is complete for secure computation ([Kil88, IPS08]), gives that every functionality can be securely computed given access to an appropriate source of correlated randomness and no additional assumptions.

And with constructions for efficiently extending OT's highly efficient general MPC can be based on this idea.

1.7.1 The power of preprocessing

In chapter 2 we look at the communication complexity of reducing secure computations to correlated randomness. We also examine what tasks can be reduced to correlated randomness in a perfectly secure manner instead of with only statistical or computational security.

We show that the secure computation of any functionality against a malicious adversary can be reduced to preprocessed randomness by protocols communicating only as much as the size of the inputs plus a number of bits equal to two times the statistical security parameter. And we show how any two-player functionality where only one party receives output can be perfectly reduced to correlated randomness using communication *exactly equal* to the sum of each player's input.

These protocols have the drawback that, while not depending on the circuit for computing f, the amount of preprocessed data needed is proportional to the size of the function-table of the computed function, and the function table is exponential in the bit-length of the inputs.

The chapter also contains a number of lower bounds, showing this communication complexity to be optimal for any nontrivial functionality, and showing that not all functionalities can be computed perfectly.

Finally we show how to use perfectly secure $sender-receiver$ protocols in the preprocessing model together with (one-sided) perfectly private protocols to implement protocols in the plain model with *perfect correctness.* Meaning that the receivers output is distributed perfectly as in the simulation.

1.8 Oblivious RAM and secure computation

Most protocols for implementing SFE have complexity that grows with the size of the circuit computing the function to be evaluated.

However most algorithms are described in terms of instructions to a RAM-machine. This means the machine has a query operation into the memory that is assumed to be an $O(1)$ operation. For secure multiparty computation most protocols show how to evaluate functions given by a circuit.

Say we want to implement the following functionality between a sender and a receiver:

If this functionality was implemented as a circuit, it would have to be of a size linear in N, as the computation has to involve (to *touch*) all possible

- Inputs: The sender inputs an array of N values A, the receiver inputs index $x \in \{0, ..N-1\}$.
- Outputs: The functionality outputs $A[x]$ to the receiver.

Figure 1.3: A functionality that has big overhead implemented as a circuit

memory locations to guarantee correctness of the output.

Oblivious RAM (ORAM), a definition initially proposed by Goldreich [Gol87] for doing software protection, can be applied to the indexing problem in secure computation and bring down the (amortized) access cost while still maintaining the privacy of the client (basically by adding memory get and set operations to the basic circuit gates).

An ORAM is a way for a client with a limited memory to implement the interface of a RAM given access to an untrusted server with a bigger memory.

The first thing one would think to do to secure the data, is to encrypt it before storing it on the server. But that still does not hide the order in which different parts of the data is accessed. An oblivious RAM also have to hide the *access pattern* to the data. An ORAM is a program using only $O(1)$ internal memory that given some stream of memory accesses maps these to a stream of accesses to the server to implement the RAM in a way so the two are distributed independently.

Given a protocol for evaluating circuits securely we can use an ORAM to securely implement RAM-program computation. We do that by implementing the circuit of the ORAM client securely, then run the circuit parts of the RAM-program, and whenever it accesses the RAM, run the ORAM on that query, and any accesses to the ORAM server are output to one of the parties who acts as the server. This party can respond as input to the next step of the computation. And because the access pattern of the ORAM client to the server is independent of the access pattern of the RAM-program we can easily simulate what the player acting as server sees.

Alternatively we can save the encryption of a value v the client wants to output to the server by *secret sharing* the value by choosing uniformly random shares $v_1 \dots v_n$ under the constraint that they sum up to v and outputting one share to each player.

One drawback most oblivious RAM constructions have for this use in secure multiparty computation is that their security rely on the client computing pseudo-random functions. This results in the protocol having computa-

ORAM	Query overhead	Worst Case	Server Space	Client space
[Gol87]	$O(\sqrt{N}\log N)$	$O(N\log N)$	$O(n+\sqrt{N})$	$O(1)$
[GO96]	$O(\log^3 N)$	$O(N\log^2 N)$	$O(\log N)$	$O(1)$
[NP01][a]	$O(\log^2 N)$	$O(N\log N)$	$O(n+\sqrt{N})$	$O(1)$
[GM11]1	$O(\log^2 N)$	$O(N\log N)$	$O(N)$	$O(1)$
[KLO12]	$O(\log^2 N/\log\log N)$	$O(\log^2 N/\log\log N)$	$O(N)$	$O(1)$
[WS08]	$O(\log^2 N)$	$O(N\log N)$	$O(N)$	$O(\sqrt{N})$
[GM11]2	$O(\log N)$	$O(N)$	$O(N\log N)$	$O(N^\epsilon)$
[GMOT12]1	$O(\log N)$	$O(N\log N)$	$O(N)$	$O(N^\epsilon)$
[Ajt10][b]	$O(\text{polylog}(N))$	$O(N\text{polylog}(N))$	$O(N\text{polylog}N)$	$O(1)$
Chapter 3[b]	$O(\log^3 N)$	$O(N\log N)$	$O(N\log N)$	$O(1)$
[SCSL11]1[b,d]	$O(\log^3 N)$	$O(\log^3 N)$	$O(N\log N)$	$O(1)$
[SCSL11]2[b]	$\tilde{O}(\log^2 N)$	$\tilde{O}(\log^3 N)$	$O(N\log N)$	$O(1)$
[GMOT12]2[b]	$O(\log^2 N)$	$O(N\log N)$	$O(N)$	$O(N^\epsilon)$
[LO13][c,d]	$O(\log N)$	$O(N\log N)$	$O(N)$	$O(1)$
Chapter 4[b,c,d]	$O(\log^2 N)$	$O(N\log N)$	$O(N)$	$O(1)$

[a] Has a non-negligible failure probability.
[b] Has information-theoretic security in hiding the access pattern.
[c] Uses 2 servers.
[d] Avoids sorting networks.

Table 1.1: An overview of different ORAM constructions. $\tilde{O}$ hides polylog log-terms.

tional security, and as the client is implemented by the secure computation, also the pseudo-random function has to be computed securely, and these functions are typically very non-linear and therefore expensive to compute.

In Chapter 3 we show an information theoretically secure ORAM construction achieving perfect security. The proposed construction has an amortized query overhead of $O(\log^3 N)$.

Many different constructions of Oblivious RAMs have been proposed with different parameters and trade-offs. Table 1.1 tries to give an overview of existing constructions.

1.8.1 Two-server Oblivious RAM

Almost all proposed ORAM constructions (including the one in Chapter 3) rely on *oblivious sorting*. That is, a way to *sort* the elements in a way that your access pattern to the elements is independent of the elements themselves. Oblivious sorting that can be implemented by *sorting networks* That is, sequences of pairs of elements to compare and swap if they are out of order. Such a sequence is a sorting network if it sorts all possible input permutations.

One problem with sorting networks is that, while asymptotically efficient constructions exist, the only known (deterministic) constructions have a large multiplicative overhead in the number of operations and are very likely to be the main bottleneck of any oblivious RAM construction. See more discussion about sorting networks in Section 3.5.

Furthermore our ORAM construction uses oblivious sorting, not for sorting itself, but for two distinct tasks: shuffling the elements to a random permutation (by sorting the elements by a random index), and to exclude certain elements of a list from being processed (by assigning them with inf before sorting). Both of these tasks can be done non-obliviously in $O(N)$ steps (versus $O(N \log N)$ steps for sorting the elements.) See section 3.5 for more discussion about techniques for oblivious shuffling.

Lu and Ostrovsky suggest in [LO13] to sidestep the question of how to efficiently sort obliviously, by splitting the work of an ORAM server between two servers, letting one server shuffle the elements locally that the other server will hold. Their solution is dependent on PRF's.

We show in chapter 4 how to transfer our information theoretic construction to the two-server model, and discuss how it can be used for securely running RAM-programs for 4-player MPC with unconditional security and threshold $t = 1$.

It is more fun to talk with someone who doesn't use long, difficult words but rather short, easy words like "What about lunch?"

A.A. Milne, Winnie-the-Pooh

2 On The Power of Correlated Randomness

This chapter reports on joint work with Yuval Ishai, Eyal Kushilevitz, Claudio Orlandi and Anat Paskin-Cherniavsky, a version will appear as [IKM+13].

2.1 Introduction

In this chapter, we study the power of *correlated randomness* in secure computation computation with no honest majority. That is, we consider secure computation with a randomness distribution phase which takes place before the inputs are known. In this phase the parties receive a sample from a predetermined joint distribution. While each party only receives its own random string from this sample, these random strings are correlated as specified by the joint distribution.

From a theoretical point of view, the correlated randomness model is interesting because it can be used to circumvent impossibility results for the plain model such as the impossibility of information-theoretic security, analogously to the use of shared secret randomness for encryption. This model can also be of practical relevance, as it can be instantiated in the

following ways:

- MPC WITH PREPROCESSING. It is often the case that parties can use idle time before they have any input to run a secure "offline protocol" for generating and storing correlated randomness. This correlated randomness is later consumed by an "online protocol" which is executed once the inputs become available. This paradigm for MPC is particularly useful when it is important that the outputs are known shortly after the inputs are (i.e., for low-latency computation). Note that if the online protocol is unconditionally secure, then it has the potential efficiency advantage of not requiring any "cryptographic" operations. If the online protocol is *perfectly secure*, then it has the additional potential advantage of a *finite* complexity that does not grow with a statistical security parameter. From here on we will refer to MPC with correlated randomness also as *MPC with preprocessing*.

- COMMODITY-BASED MPC. In the setting of commodity-based cryptography [Bea97], the parties can "purchase" correlated randomness from some external server. Security in this model is guaranteed as long as the server is honest. In contrast to the obvious solutions of employing a server as a trusted party in an MPC protocol among the parties, the server is only used during an offline phase before the inputs are known.

- HONEST-MAJORITY MPC. Recent large-scale practical employments of MPC [BCD+09, BTW11] used three servers and assumed that at most one of these servers is corrupted by a semi-honest adversary. Protocols in the correlated randomness model can be translated into protocols in this 3-server model by simply letting one server generate the correlated randomness for the other two.

As we saw in the introduction, a prime example of a cryptographic task that can benefit from having access to correlated randomness is oblivious transfer (OT) [Rab81, EGL85]. Beaver [Bea95] shows that having access to the inputs and outputs of a random instance of OT can be used to realize OT on any inputs with unconditional security. This, together with the fact that OT is complete for secure computation [Kil88, IPS08], shows that every functionality can be securely computed given access to an appropriate source of correlated randomness and no additional assumptions.

While the OT protocol from [Bea95] has both *perfect security* and *optimal communication complexity*, the protocols obtained using the compilers of [Kil88, IPS08] "only" achieve *statistical security* and their communication complexity grows linearly with the *circuit size* of the functionality. The same holds for more recent unconditionally secure MPC protocols in the preprocessing model [BDOZ11, DPSZ12]. This leaves open the following natural questions:

Question 1. *What is the* communication complexity *of unconditionally secure computation in the preprocessing model? Can the communication be made independent of the circuit size?*

Question 2. *Are there general protocols in the preprocessing model achieving* perfect security *against malicious parties?*

While the first question is clearly motivated by the goal of (theoretical and practical) efficiency, we argue that this is also the case for the second question. Consider a scenario where two parties wish to securely evaluate a functionality $f(x, y)$ where x and y are taken from small input domains. Viewing the input size as constant, it can be shown that the asymptotic complexity of any statistically secure protocol with simulation error of $2^{-\sigma}$ must grow (at least) linearly with σ, whereas any perfectly secure protocol has constant complexity. Even in practical terms, when the inputs are small, the complexity of current protocols in the preprocessing model is dominated by their dependence on σ. The potential efficiency advantage of perfectly secure protocols becomes more significant when evaluating many independent instances of "small" functionalities. Finally, the question of perfect security is conceptually interesting, as there are very few examples of perfectly secure cryptographic protocols with security against malicious parties.

2.1.1 Our Results

We essentially settle the above questions, obtaining both positive and negative results on unconditionally secure computation with correlated randomness. In doing so, we present a number of efficient protocols that can be useful in practice, especially when securely computing (many instances of) "unstructured" functions on small input domains. Concretely, we obtain the following results.

Communication complexity. We show that any multiparty functionality can be realized, with perfect security against semi-honest parties or statistical security against malicious parties, by a protocol in which the number of bits communicated by each party is linear in its input length. A disadvantage of our protocols is that their storage complexity (i.e., the number of bits each party receives during preprocessing) grows exponentially with the input length. We give evidence that this disadvantage is inherent even when the honest parties are computationally unbounded. Concretely, if every two-party functionality had a protocol with polynomial storage complexity, this would imply an unexpected answer to a longstanding open question on the complexity of information-theoretic private information retrieval [CGKS95] (Theorem 2.8.4).

We also prove a separation between the communication pattern required by unconditionally secure MPC in the preprocessing model and the communication with no security requirement. Concretely, for most functionalities (even ones with a short output) it is essential that the communication by *each party* grows linearly with its input length. In contrast, without security requirements it is always possible to make the communication by one of the parties comparable to the length of the output, independently of the input length. The same is true in the computational model of security under standard cryptographic assumptions. Concretely, such a communication pattern is possible either without preprocessing using fully homomorphic encryption [Gen09], or with preprocessing by using garbled circuits [Yao86] (provided that the inputs are chosen independently of the correlated randomness [BHR12]).

Perfect security. We show that any "sender-receiver" functionality, which takes inputs from both parties and delivers an output only to one — the receiver, can be *perfectly* realized in the preprocessing model. In contrast, we show that perfect security is generally impossible for functionalities which deliver outputs to both parties, even for non-reactive functionalities and even if one settles for "security with abort" without fairness (Thm. 2.6.1). A similar impossibility result for bit commitment (a reactive functionality) was obtained in [BMSW02].

The communication and storage complexities of our perfectly secure protocols are comparable to those of the statistical protocols, except for eliminating the dependence on a security parameter. In particular, the storage

complexity grows exponentially with the bit-length of the inputs. We present storage-efficient protocols for several natural functionalities, including string equality (see Section 2.1.4 below), set intersection, and inner product (Section 2.5).

Perfect correctness in the plain model. We present a somewhat unexpected application of our positive results from the preprocessing model to security in the plain model. Consider the goal of securely evaluating a sender-receiver functionality f. We say that a protocol for f is *perfectly correct* if the effect of any (unbounded) malicious sender strategy on the honest receiver's output can be perfectly simulated, via some distribution over the sender's inputs, in an ideal evaluation of f. For example, consider the string equality functionality $f(x, y)$ which receives an n-bit string from each party, and delivers 1 to the receiver if $x = y$ and 0 otherwise. A perfectly correct protocol for f should guarantee, for instance, that if the honest receiver picks its input at random, then the receiver should output 1 with *exactly* 2^{-n} probability, no matter which strategy the sender uses.

The impossibility of perfectly sound zero-knowledge proofs (which carries over to the preprocessing model, see Theorem 2.9.3) shows that perfect correctness cannot always be achieved when the honest parties are required to be efficient. We complement this by a positive result which applies to *all* functionalities on a small input domain as well as some natural functionalities on a large input domain (like string equality). Our result is based on a general approach for transforming perfectly secure protocols for sender-receiver functionalities in the preprocessing model into (computationally) secure protocols in the plain model which additionally offer perfect correctness against a malicious sender.

2.1.2 Summary of contributions

Upper bounds Our positive results for general functionalities are summarized in Table 2.1, and for specific sender-receiver functionalities in Table 2.2.

Lower bounds

- We show *limits* to what functionalities can be implemented *perfectly*. Theorem 2.6.1 shows that not all two-party functionalities have protocols with perfect security and abort. This is generalized in Theo-

Protocol	Communication	Storage	Parties	Security
[GMW87, Bea95]	$O(s)$	$O(s)$	k	perfect, passive
[Kil88, IPS08]	$O(s) + \mathsf{poly}(\sigma)$	$O(s) + \mathsf{poly}(\sigma)$	k	statistical, active
Theorem 2.3.1	$O(n)$	$O(2^n m)$	k	perfect, passive
Theorem 2.3.2	$O(n + \sigma)$	$O(2^n(m + \sigma))$	k	statistical, active
Theorem 2.3.3	$O(n)$	$O(2^n(m + n))$	2	perfect, active

Table 2.1: Comparison of our positive results with previous work: s is the size of a boolean circuit computing the functionality, n is the length of the inputs, m is the output length, and σ is a statistical security parameter. In the asymptotic complexity expressions, the number of parties k is viewed as constant. The protocol of Theorem 2.3.3 applies only to sender-receiver two-party functionalities.

Protocol	f	Communication	Storage	Computation	Security
Sec. 2.1.4	$x =_? y$	$2\|x\|$	$O(\|x\|)$	$\mathsf{poly}(\|x\|)$	perfect, active
Thm. 2.5.1	$x \cap y$	$\mathsf{poly}(\|x\|) + \|y\|$	$\mathsf{poly}(\|x\|)$	$\exp(\|x\|, \|y\|)$	perfect, active
Thm. 2.5.1	$x \cap y$	$\mathsf{poly}(\|x\|, k) + \|y\|$	$\mathsf{poly}(\|x\|, \sigma)$	$\mathsf{poly}(\|x\|, \|y\|, \sigma)$	statistical, active
Thm. 2.5.2	$\langle x, y \rangle$	$2\|x\|$	$O(\|x\|)$	$\mathsf{superpoly}(\|x\|)$	perfect, active

Table 2.2: Sender-receiver protocols for specific tasks. Two variants of set intersection are given: a perfectly secure with exponential computation, and a statistically secure with efficient computation.

rem 2.6.2 were we show a function that requires $\Omega(\log \frac{1}{\epsilon})$ communication to compute with ϵ-security. Another generalization of Theorem 2.6.2, Theorem 2.6.3 shows that the negative results extends to the case of *expected* round complexity.

- We put a *lower bound* on the amount of *communication* that a secure protocol for a non-trivial functionality must use. Theorem 2.8.1 for the perfect case and Theorems 2.8.2, 2.8.3 for the statistical case show that for general functionalities the communication complexity of our protocols is optimal.

- We give evidence that superpolynomial preprocessing is needed in general. Theorem 2.8.4 explains that improving on the preprocessing

needed for sender-receiver functionalities will imply a breakthrough in information theoretic PIR.

Perfect *correctness* in the *plain model*.

- On the positive side, we show in Theorem 2.9.1 and 2.9.2 how to use perfectly secure sender-receiver protocols in the preprocessing model to implement *perfectly correct* protocols in the plain model (if the preprocessing is small enough).

- On the other hand, Theorem 2.9.3 shows that not all functions can be computed (efficiently) with perfect correctness in the plain model unless $NP \subseteq BPP$.

2.1.3 Related Work

Beaver [Bea95] showed that OT can be realized with perfect security given preprocessing. Later Beaver [Bea97] generalized the above to the *commodity-based model*, a setting where there are multiple servers providing precomputed randomness, only a majority of which are honest (Beaver also notes that perfect security is not possible in general because commitment cannot be realized perfectly, and a proof of this appeared in [BMSW02]. However, the question was left open for standard (non-reactive) functionalities.

Since OT can be precomputed [Bea95] and as it is complete for secure computation [Kil88], it is possible to compute any function with statistical security. The result of [IPS08] improves the asymptotical complexity of [Kil88], while [Bea91, BDOZ11, NNOB12] offer efficient statistically secure protocols in the preprocessing model for arithmetic and Boolean circuits respectively. A recent result [DZ13] shows that this can be done with no overhead during the online phase by giving a protocol with optimal communication complexity for the case of "generic preprocessing" (i.e., the preprocessing does not depend on the function to be evaluated — only on its size). Our results achieve better online communication complexity as we do not rely on a circuit representation.

A protocol for computing secret shares of the inner product against malicious adversaries was proposed in [DvMN10]. In Section 2.5, we give a protocol for computing the inner product where one party learns the output. In the setting of malicious corruptions, it is not trivial to reconstruct

the results from the shares, and therefore our protocol takes a substantially different approach than [DvMN10].

In [TDH+09], a perfectly secure protocol for oblivious polynomial evaluation in the preprocessing model is presented. [TDH+09] also presents a protocol for equality which is claimed to be perfectly secure but it is however not perfectly secure according to the standard simulation-based definition — see Section 2.1.4 below for a perfectly secure protocol for equality.

Naor and Pinkas [NP06] describe a perfectly secure reduction (attributed to Gilboa) from the case of securely evaluating a degree-d polynomial to evaluation to oblivious linear evaluation. While in the case of a malicious sender they do not consider correctness, the reduction can be shown to be perfectly secure in the strongest possible sense.

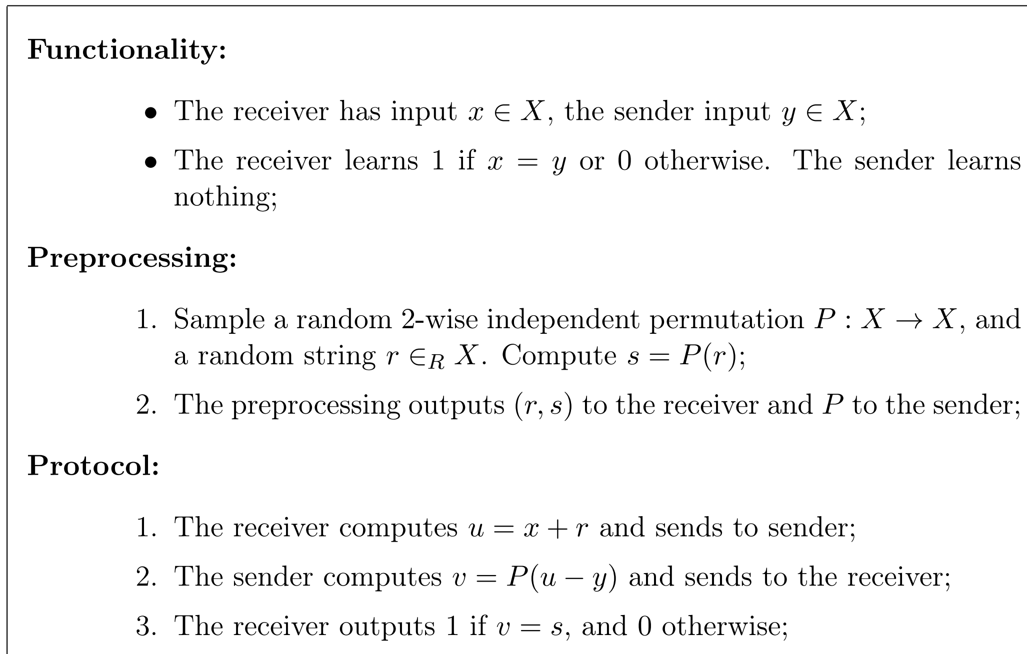
The type of correlated randomness needed for realizing multiparty computation with unconditional security in the presence of an honest majority is studied in [FGMvR02, FWW04]. Statistically secure commitment protocol from correlated randomness are constructed in [Riv99]. Finally [WW10] gives linear lower bounds on the storage complexity of secure computation with correlated randomness.

2.1.4 Warmup: Equality Test

To introduce some of the notation and the techniques that we use later to prove more general results, we describe the simple protocol in Figure 2.1 for equality testing in the preprocessing model.

We consider at the *sender-receiver* version of the functionality, where only the receiver gets output from the protocol. In this setting, we have a receiver and a sender holding respectively x, y in some group X (e.g, $(\{0,1\}^n, \oplus)$, the group of strings of length n with exclusive or $\oplus$ as the operation). At the end of the protocol, the receiver learns whether $x = y$ or not. The protocol achieves perfect security against malicious adversaries and it is optimal in terms of communication complexity. Correctness follows from $v = P(u-y) = P(r + x - y)$, and this is equal to s iff $x = y$.

One can prove that the protocol is perfectly secure by a simulation argument: The simulator has access to all preprocessed information. In case of a corrupted *sender*, the simulator proceeds as follows: the simulator sends a random u to the adversary and, when the adversary replies v, the simulator computes $y = u - P^{-1}(v)$ and inputs it to the ideal functionality. In case of a corrupted *receiver*, the simulator extracts the input string x (using u, r)



Functionality:

- The receiver has input $x \in X$, the sender input $y \in X$;
- The receiver learns 1 if $x = y$ or 0 otherwise. The sender learns nothing;

Preprocessing:

1. Sample a random 2-wise independent permutation $P : X \to X$, and a random string $r \in_R X$. Compute $s = P(r)$;
2. The preprocessing outputs (r, s) to the receiver and P to the sender;

Protocol:

1. The receiver computes $u = x + r$ and sends to sender;
2. The sender computes $v = P(u - y)$ and sends to the receiver;
3. The receiver outputs 1 if $v = s$, and 0 otherwise;

Figure 2.1: A perfectly secure protocol for equality with preprocessing.

and inputs it to the ideal functionality for equality. If the ideal functionality outputs 1, the simulator sends $v = s$ to the corrupted receiver, but if it outputs 0, the simulator chooses $v \in_R X$ such that $v \neq s$. This simulation is perfect, as the adversary's view of the protocol is distributed identically both in the real execution and in the simulation. Note that it is enough for P to be drawn from a family of pairwise independent permutations, since the receiver only learns the permutation at two indices.

The protocol is also UC-secure (the simulation is straight line) and is *adaptively* secure. This is the case for all the protocols presented in this work.

While it is not always true that perfect security implies adaptive security [CDD$^+$04] it is easy to see that in this case the simulator can output random (u, v) to the adversary, and then it can "explain" this view to be consistent with any input pair (x, y) by the appropriate choice of the preprocessing information.

2.2 Preliminaries

Notation. Let $[n]$ denote the set $\{1, 2, \ldots, n\}$. We use $Z^{X\times Y}$ to denote the set of matrices over Z whose rows are labeled by the elements of X and whose columns are labeled by the elements of Y.

Computational model. Since some of our results refer to *perfect* security, we incorporate perfect uniform sampling from $[m]$, for an arbitrary positive integer m, into the computational model as an atomic computational step. Alternatively, this can be implemented by using standard (binary) Turing Machines which run in *expected* polynomial time.

Network model. We consider protocols involving n parties. We denote them $P_1, \ldots, P_n$. The parties communicate over synchronous, secure and authenticated point-to-point channels. In some constructions we also use a broadcast channel. We note that, in the preprocessing model, all these channels can be implemented with unconditional security over insecure point-to-point channels. Specifically, secure channels can be perfectly implemented in the preprocessing model using a one-time pad, authentication (with statistical security) using a one-time message authentication code (MAC), and broadcast (with statistical security) using the protocol of [PW96].
Functionalities. We consider non-reactive secure computation tasks, defined by a deterministic or randomized *functionality* $f : X_1 \times \ldots \times X_n \to Z_1 \times \ldots \times Z_n$. The functionality specifies a mapping from n inputs to n outputs which the parties want to compute. We will often consider a special class of two-party functionalities referred to as *sender-receiver functionalities*. A sender-receiver functionality $f : X \times Y \to Z$ gets an input x from P_1 (the *receiver*), an input y from P_2 (the *sender*) and delivers the output z only to the receiver.
Protocols with preprocessing. An n-party protocol can be formally defined by a *next message function.* This function, on input (i, x_i, r_i, j, m), specifies an n-tuple of messages sent by party P_i in round j, when x_i is its inputs, r_i is its randomness and m describes the messages it received in previous rounds. (If a broadcast channel is used, the next message function also outputs the message broadcasted by P_i in Round j.) The next message function may also instruct P_i to terminate the protocol, in which case it also specifies the output of P_i. In the *preprocessing model*, the specification of a protocol also

includes a joint distribution $\mathcal{D}$ over $R_1 \times R_2 \ldots \times R_n$, where the R_i's are finite randomness domains. This distribution is used for sampling correlated random inputs $(r_1, \ldots, r_n)$ which the parties receive before the beginning of the protocol (in particular, the preprocessing is independent of the inputs). The next message function, in this case, may also depend on the private random input r_i received by P_i from $\mathcal{D}$. We assume that for every possible choice of inputs and random inputs, all parties eventually terminate.
Security definition. We work in the standard ideal-world/real-world simulation paradigm. Our positive results hold for the strongest possible security model, namely UC-security with adaptive corruptions, while our negative results hold for the weaker model of standalone security against static corruptions. We consider both semi-honest (passive) corruptions and malicious (active) corruptions. Using the standard terminology of secure computation, the preprocessing model can be thought of as a *hybrid model* where the parties have a one-time access to an ideal randomized functionality $\mathcal{D}$ (with no inputs) providing them with correlated, private random inputs r_i. We consider by default *full security* (with guaranteed output delivery) for sender-receiver functionalities, and *security with abort* for general functionalities. We mainly focus on the cases of *statistical* or *perfect* security, though some of our results refer to computational security as well. We will sometimes refer separately to *correctness* and *privacy* — the former considers only the effect of the adversary on the outputs and the latter considers only the view of the adversary.

The real model. The real model execution of a protocol π starts with drawing correlated random inputs $(r_1, \ldots, r_n) \leftarrow \mathcal{D}$ and delivering to each P_i its random input r_i along with an input $x_i \in X_i$. The protocol proceeds in rounds, as specified by the next-message function of π. In the 2-party case, only one party speaks in each round (and they take turns speaking). The protocol terminates after a fixed number of rounds and each party P_i outputs some value $z_i \in Z_i$, based on its *view* during the protocol. This view consists of its input x_i, its random input r_i, and all messages received from other parties during the execution. (The outgoing messages are determined by the above information and thus are not included in the view.)

When considering sender-receiver functionalities, we will refer to party 1 as a "receiver" R, and party 2 as the sender, S. We will use x, y to denote the inputs of the sender and the receiver, respectively, and r_x, r_y to denote

their correlated randomness. The protocol is executed in the presence of a real world adversary $\mathcal{A}$ that may corrupt any subset of the parties in a *static* way (i.e., the set of corrupted parties is decided once and for all before the protocol starts) or in an *adaptive* way (i.e., new parties can be corrupted at any time, possibly based on the information gathered by the adversary so far). The adversary learns the entire view of corrupted parties. A *malicious* adversary can send arbitrary messages on behalf of the corrupted parties whereas a *semi-honest* adversary can only observe their views but does not modify the messages they send, including a special message `abort`. We assume by default that the adversary has a *rushing* capability: at any round it can first wait to hear all messages sent by honest parties to the corrupted parties, and use these to determine its own messages. In the end of the execution, the adversary can output any function of its view. We denote by $\mathbf{view}_{\mathcal{A},\pi,\mathcal{D}}(x_1,\ldots,x_n)$ the output of an adversary $\mathcal{A}$ attacking a protocol π with inputs $x_1,\ldots,x_n$, by $\mathbf{output}_{\mathcal{A},\pi,\mathcal{D}}(x_1,\ldots,x_n)$ the output of the honest parties in the same interaction (along with their identities), and by $\mathbf{exec}_{\mathcal{A},\pi,\mathcal{D}}(x_1,\ldots,x_n)$ the concatenation of **view** and **output**.

The ideal model. We compute functionalities of the form

$$f : X_1 \times \ldots \times X_n \to Z_1 \times \ldots \times Z_n,$$

specifying a mapping from the parties' inputs to the parties output. The input and output domains are finite. In the common case where all parties get the same output we use the shorthand $f : X_1 \times \ldots \times X_n \to Z$, where Z is the output domain. One may also consider randomized functionalities, which take an additional random input, however, in this work we by default deal with deterministic functionalities. As a special case, we will consider the case where only one party gets output. We refer to such functionalities as *sender-receiver* functionalities i.e., $f : X \times Y \to Z \times \{\perp\}$. An n-party functionality f defines an ideal function evaluation process in which the parties use a trusted party TP to evaluate f. (We will sometimes refer to TP as the functionality.) This process may be attacked by an ideal-model adversary $\mathcal{S}$, also referred to as a simulator, who may corrupt an arbitrary subset of the parties. An execution begins with each party P_i receiving input x_i. If P_i is not corrupted it simply forwards x_i to TP. If P_i is corrupted, $\mathcal{S}$ can instruct it to forward a different value x_i', possibly as a function of the original value x_i. Then TP computes the output $(z_1,\ldots,z_n) = f(x_1',\ldots,x_n')$ (where $x_i' = x_i$

if P_i is not corrupted). We now distinguish between the case of *full security* and the case of *security with abort*. In the case of full security, TP sends to $\mathcal{S}$ the outputs of the corrupted parties and to each honest party P_i its output z_i. In the case of security with abort, TP starts by sending to $\mathcal{S}$ the outputs of the corrupted parties. If $\mathcal{S}$ replies `continue`, TP sends the honest parties their outputs too, while if $\mathcal{S}$ replies `abort`, TP sends a special symbol $\perp$ to all honest parties instead. (Note that an honest party outputs $\perp$ iff all honest parties output $\perp$.) If $\mathcal{S}$ is *malicious* then it is unrestricted in the previous game, while if it is *semi-honest* it must always use inputs $x'_i = x_i$ and never send any `abort` message. We say that the adversary is *static* if it chooses the set of corrupted parties before the protocol starts or *adaptive* if it can choose which parties to corrupt at any point of the execution. At the end of the execution, $\mathcal{S}$ can output any function of its view. We denote by $\mathbf{view}_{\mathcal{S},f}(x_1, \ldots, x_n)$ the output of the simulator $\mathcal{S}$ interacting with a functionality f on inputs $x_1, \ldots, x_n$ and by $\mathbf{output}_{\mathcal{S},f}(x_1, \ldots, x_n)$ the output of the honest parties in the same interaction (along with their identities), and by $\mathbf{exec}_{\mathcal{S},f}(x_1, \ldots, x_n)$ the concatenation of **view** and **output**.

Security. We now define statistical and perfect security in the preprocessing model. Let $\mathbf{output}_{\mathcal{A}(z),\pi}(x_1, \ldots, x_n)$ denote the output of the honest parties in a run of the protocol π, where party P_i is given input x_i in the presence of an adversary $\mathcal{A}$ with auxiliary input z; Let $\mathbf{output}_{\mathcal{S}(z),f} x_1, \ldots, x_n$ denote the output of the honest parties in the ideal model for computing f, where party P_i is given input x_i in the presence of a simulator $\mathcal{S}$ with auxiliary input z.

We let $\Delta(D, D')$ denote the statistical distance between the two distributions D and D'.

Definition 2.2.1. *We say that a protocol π, given preprocessing $\mathcal{D}$, realizes a functionality $f : X_1 \times \ldots \times X_n \to Z_1 \times \ldots \times Z_n$ with* ϵ-security *if for every real life adversary $\mathcal{A}$ there exists an ideal model simulator $\mathcal{S}$ such that for all inputs $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$,*

$$\Delta(\mathbf{exec}_{\mathcal{A},\pi,\mathcal{D}}(x_1, \ldots, x_n), \mathbf{exec}_{\mathcal{S},f}(x_1, \ldots, x_n)) \leq \epsilon.$$

We say that π is perfectly secure *if $\epsilon = 0$. We distinguish between* semi-honest security *(where both $\mathcal{A}$ and $\mathcal{S}$ are semi-honest) and* malicious security *(where both are malicious). We also distinguish between* full security *and* security with abort*, depending on the variant of the ideal model being used. By*

default, security refers to malicious security, and to full security for sender-receiver functionalities or security with abort for other functionalities.

We say that a protocol has *perfect security* when $\epsilon = 0$, and we say that a protocol has *statistical security* when for all polynomial p and large enough $\ell = |x_1, \ldots, x_n|$ it holds that $\epsilon < 1/p(\ell)$.[1] An equivalent form of Definition 2.2.1 quantifies over all input distributions, rather than all specific choices of inputs. This equivalent form will be useful for proving negative results. We note that we will typically view protocols as finite objects and hence ignore the efficiency of simulation. However, when referring to protocols over large domains we require by default that the complexity of the simulator be polynomial in that of the adversary. We will sometimes use k to denote a statistical security parameter which is given to all parties as an additional input. In such a case, the simulation error ϵ is required to be $2^{-\Omega(k)}$.

We also need the following slightly less standard definitions, which separate "privacy" and "correctness." Note that unlike the above definitions of security, the following notions are not *composable*.

Definition 2.2.2. *We say that a protocol π for f is* ϵ-correct *(resp.,* ϵ-private*) if it is ϵ-secure as in Definition 2.2.1 except that* **exec** *is replaced by* **output** *(resp.,* **view***).*

2.3 Optimal Communication for General Functionalities

In this section, we settle the communication complexity of MPC in the preprocessing model. For simplicity, we restrict the attention to non-reactive functionalities, but the results of this section apply also to reactive functionalities.

In Section 2.8, we prove negative results which complement these positive results. In particular, we show that the communication complexity of the following protocols is optimal (for non-trivial functions) and give evidence that the exponential storage complexity (or randomness complexity) is inherent.

[1] When looking at statistical security, the inputs of the parties contains a security parameter in addition to the actual input i.e., $x'_i = (x_i, 1^k)$.

2.3.1 Upper Bounds on Communication Complexity

The following is a summary of our upper bounds. These will follow from Claims 1, 2, 3, 4 (some of which are in later sections) and by inspection of the protocols.

Theorem 2.3.1. *For any n-party functionality $f : X_1 \times \ldots \times X_n \to Z_1 \times \ldots \times Z_n$, there is a protocol π which realizes f, in the preprocessing model, and has the following features against semi-honest parties:*

- *π is perfectly secure;*
- *It uses two rounds of communication;*
- *Let $\alpha = \sum_{i \in [n]} \log |X_i|$ be the total input length. Then, the total communication complexity is $O(\alpha)$ and the storage complexity is $O(\alpha 2^\alpha)$.*

Theorem 2.3.2. *For any n-party functionality $f : X_1 \times \ldots \times X_n \to Z_1 \times \ldots \times Z_n$ and $\epsilon > 0$, there is a protocol π which realizes f, in the preprocessing model against a malicious adversary, such that:*

- *π is statistically ϵ-secure with abort;*
- *It uses two rounds of communication (given broadcast);*
- *The total communication complexity is $O(\alpha + n \log 1/\epsilon)$ and the storage complexity is $O(2^\alpha \cdot (\alpha + n \log 1/\epsilon))$, where α being the total input length, as above.*

Theorem 2.3.3. *For any 2-party sender-receiver functionality $f : X \times Y \to Z$, there is a protocol π which realizes f, in the preprocessing model against a malicious adversary, such that:*

- *π is perfectly secure;*
- *It uses two rounds of communication;*
- *The total communication complexity is $\log |X| + \log |Y|$ and the storage complexity is $O(|X| \cdot |Y| \cdot \log |Y|)$.*

2.3.2 Semi-Honest Two-Party Protocol via One-Time Truth Tables

For the sake of exposition, we focus on protocols where both parties P_1, P_2 receive the same output $f(x, y) \in Z$, for some function $f : X \times Y \to Z$. We view X, Y and Z as groups and use additive notation for the group operation, i.e., $(X, +), (Y, +), (Z, +)$.

In Figure 2.2 we present a simple protocol that is secure against a semi-honest adversary if the parties have access to a preprocessing functionality dealing correlated randomness. The protocol has communication complexity $\log |X| + \log |Y| + 2 \log |Z|$. A protocol with communication complexity $\log |X| + \log |Y| + \log |Z|$ follows from the protocol in Section 2.4.[2] We start by presenting this slightly less efficient protocol here, as this protocol is easier to generalize for security against malicious parties and for the multiparty case (see protocols in Figure 2.3 and 2.4).

The protocol uses *one-time truth tables* (OTTT). Intuitively, OTTT can be seen as the one-time pad of secure function evaluation. The parties hold shares of a permuted truth-table, and each party knows also the permutation that was used for its input. In the two-party case, the truth-table can be seen as a matrix, where one party knows the permutation of the rows and the other knows the permutation of the columns. In fact, given that every truth table will be only used once, a random cyclic-shift can be used instead of a random permutation.

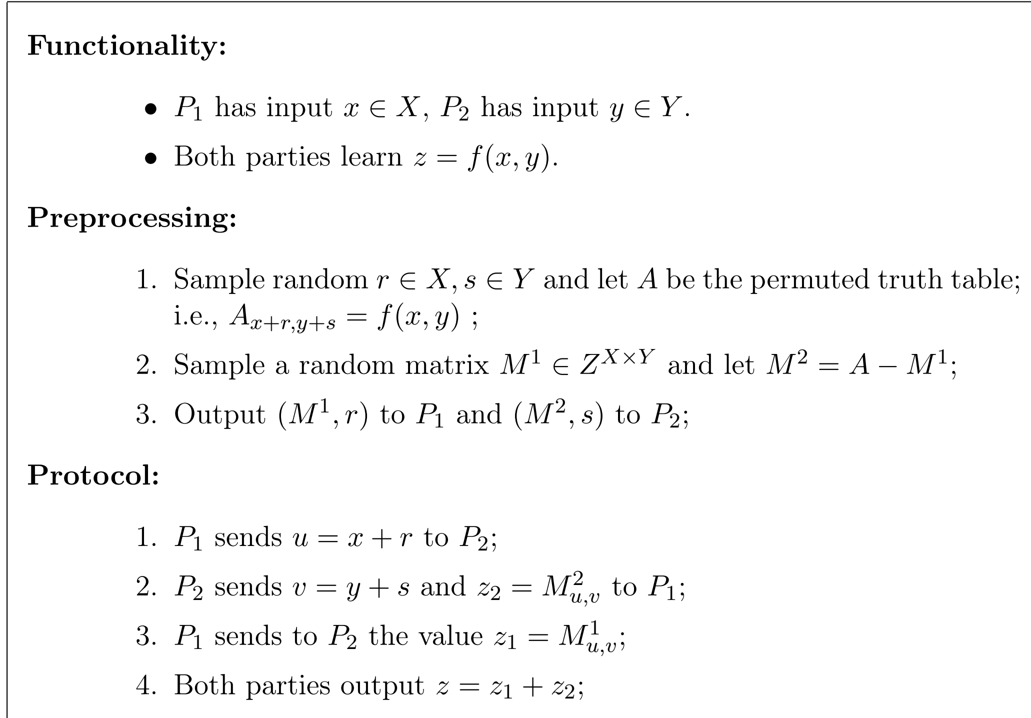
Claim 1. *The protocol in Figure 2.2 securely computes f with perfect security against semi-honest corruptions.*

Proof. When both parties are honest, the protocol indeed outputs the correct value:

$$z = z_1 + z_2 = M^1_{u,v} + M^2_{u,v} = A_{u,v} = A_{x+r,y+s} = f(x, y).$$

Security against semi-honest parties can be argued as follows: the view of P_2 can be simulated by choosing a random $u \in X$ and defining $z_1 = z - M^2_{u,v}$. The view of P_1 can be simulated by choosing a random $v \in Y$ and defining $z_2 = z - M^1_{u,v}$. As both in the simulation and in the real protocol the values u, v, z_1, z_2 are distributed uniformly at random in the corresponding domains, the protocol achieves perfect security. □

[2] The protocol of Section 2.4 has complexity $\log |X| + \log |Y|$ but only one party gets output; however, in the semi-honest case, this party may simply transfer the output ($\log |Z|$ bits) to the other party.



Functionality:

- P_1 has input $x \in X$, P_2 has input $y \in Y$.
- Both parties learn $z = f(x, y)$.

Preprocessing:

1. Sample random $r \in X, s \in Y$ and let A be the permuted truth table; i.e., $A_{x+r,y+s} = f(x, y)$;
2. Sample a random matrix $M^1 \in Z^{X \times Y}$ and let $M^2 = A - M^1$;
3. Output (M^1, r) to P_1 and (M^2, s) to P_2;

Protocol:

1. P_1 sends $u = x + r$ to P_2;
2. P_2 sends $v = y + s$ and $z_2 = M^2_{u,v}$ to P_1;
3. P_1 sends to P_2 the value $z_1 = M^1_{u,v}$;
4. Both parties output $z = z_1 + z_2$;

Figure 2.2: Semi-Honest Secure Protocol using One-Time Truth Table

2.3.3 Malicious Protocol via One-Time Truth Tables

The above protocol is only secure against semi-honest adversaries, as a malicious party P_i could misreport its output share z_i and therefore change the output distribution. To fix this problem, we will enhance the OTTT protocol using information theoretic message authentication codes (MAC): the preprocessing phase will output keys for a one-time MAC to both parties, and will add shares of these MACs to the truth table. The resulting protocol is only statistically secure as an adversary will always have a (negligibly small) probability to output a fake share z_i together with a valid MAC. As we will see later (Section 2.6), this is inherent; i.e., it is impossible to securely compute every function with perfect security, even in the preprocessing model.

Definition 2.3.1 (One-Time MAC)**.** *A pair of efficient algorithms* $(\mathsf{Tag}, \mathsf{Ver})$ *is a* one-time ϵ-secure message authentication code scheme*, (MAC scheme) with key space* $\mathcal{K}$ *and MAC space* $\mathcal{M}$*, if* $\mathsf{Ver}_k(m, \mathsf{Tag}_k(m)) = 1$ *with probability*

1 *and for every (possibly unbounded) adversary* $\mathcal{A}$*:*

$$\Pr[k \leftarrow \mathcal{K}, m \leftarrow \mathcal{A}, (m', t') \leftarrow \mathcal{A}(m, \mathsf{Tag}_k(m)) : \mathsf{Ver}_k(m', t') = 1 \wedge m \neq m'] \ < \ \epsilon \, .$$

The MAC can be instantiated with the standard "$am + b$" construction: Let $\mathbb{F}$ be a finite field of size $|\mathbb{F}| > \epsilon^{-1}$, and let $k = (a, b) \in \mathbb{F}^2$. To compute a MAC tag, let $\mathsf{Tag}_k(m) = am + b$ and for verification compute $\mathsf{Ver}_k(m, t) = 1$ iff $t = am + b$. Without loss of generality, the range of f, the function to be computed, is $Z = \mathbb{F}$. [3] For the purpose of this application, we will write the MAC space $\mathcal{M}$ as an additive group $(\mathcal{M}, +)$.

MAC enhanced OTTT: In Figure 2.3, the protocol for general two-party computation in the preprocessing model using OTTT is presented. Note that, as all the MAC signatures are secret-shared and only one is reconstructed, we can use the same MAC key for all the entries in the matrix. We assume, for notational simplicity, that both parties obtain the same output z; the general case may be handled similarly.

Claim 2. *The protocol in Figure 2.3 computes* f *with* ϵ*-security against a malicious adversary.*

Proof. If P_1 is corrupted, the simulator produces the preprocessing output uniformly at random i.e., it outputs random $r \in_R X$, $k_1 \in_R \mathcal{K}$ and $M^1 \in_R (Z \times \mathcal{M}^2)^{X \times Y}$. When P_1 sends u', the simulator computes $x' = u' - r$ and inputs it to the ideal functionality, thus receiving $f(x', y)$. Now the simulator samples a random $v \in_R Y$, and computes $z_2 = (f(x', y), t_1, t_2) - M^1_{u',v}$, where $t_1 = \mathsf{Tag}_{k_1}(f(x', y))$ and $t_2 = \mathsf{Tag}_{k_2}(f(x', y))$, for a random MAC key k_2. Let z_1' be the last message sent by the corrupted P_1, and $e = z_1' - M^1$ the *error* introduced by the adversary. Let $e = (e_1, e_2, e_3)$ be the three components of the error, with $e_1 \in Z, e_2, e_3 \in \mathcal{M}$. If $(e_1, e_3) = (0, 0)$ the simulator inputs `continue` to the ideal functionality; otherwise, it inputs `abort`.

It can be verified that the view of the adversary is distributed identically in the real protocol and in the simulation: v is uniformly random in the simulation and it is computed as $v = y + s$, with s uniformly random in the protocol. The message z_2 is computed in both cases as $(A - M^1)_{u,v}$. The only difference is in the way that the output of the honest party is computed: in the simulated execution P_2 outputs $\perp$ if $(e_1, e_3) \neq (0, 0)$ and $f(x', y)$

[3] Note that we still only need $\log_2 |\mathrm{Im}(f)|$ bits to encode the output of the function.

Functionality:

- P_1 has input $x \in X$ and P_2 has input $y \in Y$.
- Both parties learn $f(x, y)$.

Preprocessing:

1. Sample random keys for an ϵ-secure MAC scheme $k_1, k_2 \in \mathcal{K}$;
2. Sample random $r \in X, s \in Y$ and let $A \in (Z \times \mathcal{M} \times \mathcal{M})^{X \times Y}$ be a matrix s.t.

$$A_{x+r,y+s} = \big(f(x,y), \mathsf{Tag}_{k_1}(f(x,y)), \mathsf{Tag}_{k_2}(f(x,y))\big) \ ;$$

3. Sample a random matrix $M^1 \in (Z \times \mathcal{M} \times \mathcal{M})^{X \times Y}$ and let $M^2 = A - M^1$;
4. Output (M^1, r, k_1) to P_1 and (M^2, s, k_2) to P_2;

Protocol:

1. P_1 sends $u = x + r$ to P_2;
2. P_2 sends $v = y + s$ and $z_2 = M^2_{u,v}$ to P_1;
3. P_1 sends $z_1 = M^1_{u,v}$ to P_2;
4. Each party P_i parses $z_1 + z_2$ as (z, t_1, t_2);
5. If $\mathsf{Ver}_{k_i}(z, t_i) = 1$, party P_i outputs z, otherwise it outputs $\perp$;

Figure 2.3: Malicious Secure Protocol using One-Time Truth Table

otherwise, while in the protocol P_2 outputs $\perp$ if $\mathsf{Ver}_{k_2}(z + e_1, t_2 + e_3) = 0$ and $z + e_1$ otherwise. By the security of the one-time MAC, the two distributions are ϵ-close.

Security against a corrupted P_2 follows from a similar argument, due to the symmetric structure of the protocol. Finally, we note that the protocol is also secure against adaptive corruptions: the protocol is run over a secure channel, and therefore the simulator does not need to produce any view until the first party is corrupted. When this happens, the simulator produces a random view consistent with the corrupted party's input and output. If the second party is also corrupted, it is always possible to determine values for the preprocessing information that are consistent with the transcript of the

protocol and the inputs and output of the corrupted parties, i.e., $r = u - x$ or $s = v - y$, depending on which party is corrupted first. □

2.3.4 Multiparty Generalization

We now describe a multiparty generalization of the above protocol, which maintains the optimal communication complexity and still requires only two rounds of interaction when broadcast is available.

In this setting we have n parties $P_1, \ldots, P_n$ with inputs $x_1, \ldots, x_n \in X_1 \times X_2 \times \ldots \times X_n$ that want to compute $f : X_1 \times \ldots \times X_n \to Z$. The adversary can maliciously corrupt any number of parties (no honest-majority). In case of semi-honest corruption, it is (essentially) trivial to extend the two-party case, while for malicious corruptions the extension requires more care as, according to the definition of security, we need to make sure that if one honest party outputs $\perp$, then every other honest party does so too: note that it will not be sufficient to add a new MAC for each party, as a malicious adversary can now add different errors in the different coordinates of the output share so that the MAC of a party P_i is invalid (making that party output $\perp$) while another party P_j outputs the correct value z.

To fix this, we use the notion of *unanimously identifiable commitments* from [IOS12, PCR09].

Definition 2.3.2 (Unanimously Identifiable Commitments). *An ϵ-UIC is a scheme consisting of a randomized algorithm* com *and a deterministic algorithm* open *with the following syntax:*

Commit: $\mathsf{com} : S \to C^n \times D$, *takes as input a secret $s \in S$ and outputs n commitments $c_1, \ldots, c_n$ and decommitment information d.*

Open: $\mathsf{open} : C \times D \to S \cup \{\perp\}$, *takes as input a commitment c_i and the decommitment information d and recreates the secret s or outputs $\perp$ indicating failure.*

that satisfies the following properties:

Completeness: *For all $s \in S$, if $(c_1, \ldots, c_n, d) \leftarrow \mathsf{com}(s)$, then $\mathsf{open}(c_i, d) = s$ w.p. 1.*

Secrecy: *For all $s, s' \in S$, if $(c_1, \ldots, c_n, d) \leftarrow \mathsf{com}(s)$ and $(c'_1, \ldots, c'_n, d') \leftarrow \mathsf{com}(s')$ then $(c_1, \ldots, c_n)$ and $(c'_1, \ldots, c'_n)$ are perfectly indistinguishable.*

Binding with Agreement on Abort: *Let $T \subset [n]$ a subset of indexes of $[n]$ and $\vec{c}_{-T}$ the vector containing the commitments c_i for all $i \in [n] \setminus T$. For all (possibly unbounded) adversary $\mathcal{A}$, for all $T \subset [n], s \in S$ and for all $i, j \in T$:*

$$\begin{aligned}\Pr[&(\vec{c}, d) \leftarrow \mathsf{com}(s), d' \leftarrow \mathcal{A}(d, \vec{c}_{-T}) : \\ &(\mathsf{open}(c_i, d') = \mathsf{open}(c_j, d') = s) \\ &\vee (\mathsf{open}(c_i, d') = \mathsf{open}(c_j, d') = \perp))] \geq 1 - \epsilon\end{aligned}$$

Let $\mathbb{F}$ be a field. It is possible to construct an unanimously identifiable commitment scheme as follows: $\mathsf{com}(s)$ samples a random polynomial q over $\mathbb{F}$ of degree n s.t., $q(0) = s$ and outputs $d = q$ to the dealer and $c_i = (a_i, b_i = q(a_i)) \in \mathbb{F}^2$ to P_i, with a_i's uniformly random in $\mathbb{F}$. The function for the decommitment phase $\mathsf{open}(c_i, d)$ parses $c_i = (a_i, b_i)$ and d as a polynomial q, and outputs $s = q(0)$ if $q(a_i) = b_i$ or $\perp$ otherwise. It is possible to prove (see [IOS12]) that this is an unanimously identifiable commitment scheme with $\epsilon = n/|\mathbb{F}|$.

The protocol for multiparty computation with preprocessing, using UIC is presented in Figure 2.4.

Claim 3. *The protocol in Figure 2.4 securely computes f with ϵ-security (with abort) against a malicious adversary.*

Proof. It is easy to check that the parties output the correct output when no parties are corrupted. To argue for security, consider an adversary that corrupts a subset $T \subseteq [n]$ of parties: the simulator will provide the adversary with uniformly random values for the preprocessing of the corrupted parties. In the first round of the protocol, the simulator samples uniformly random $u_i \in_R X_i$ for all $i \in [n] \setminus T$, and receives u_i' for all $i \in T$ from the adversary. The simulator computes $x_i' = u_i' - r_i$ for all $i \in T$ and inputs those values to the ideal functionality. Let $i^* = \min\{[n] \setminus T\}$. Upon receiving the results z' from the ideal functionality, the simulator samples random $z_i \in \mathbb{F}^{n+1}$ for all $i \in [n] \setminus (T \cup \{i^*\})$, and sets $z_{i^*} = Q - \sum_{i \in [n] \setminus T, i \neq i^*} z_i - \sum_{i \in T} M^i_{u_1,\ldots,u_n}$, where Q is a random polynomial of degree n such that $Q(a_i) = b_i$ for all $i \in T$ and $Q(0) = z$. Let z_i' be the messages received by the simulator from the adversary, for all $i \in T$. Then the simulator computes $e = \sum_{i \in T} z_i' - \sum_{i \in T} M^i_{u_1,\ldots,u_n}$. If $e \neq 0$, the simulator inputs `abort` to the ideal functionality or `continue` otherwise.

Functionality:

- For $i = 1, \ldots, n$, P_i has input $x_i \in X_i$;
- All parties output $z = f(x_1, \ldots, x_n)$;

Preprocessing:

1. For $i = 1, \ldots, n$, sample random $r_i \in X_i$;
2. For $i = 1, \ldots, n$, sample random $(a_i, b_i) \in \mathbb{F}^* \times \mathbb{F}$;
3. Let $Q_{x_1,\ldots,x_n} \in (\mathbb{F}^{n+1})^{X_1 \times \ldots \times X_n}$ be a matrix of polynomials of degree n s.t.:
$$\forall i = [n],\ b_i = Q_{x_1,\ldots,x_n}(a_i) \text{ and } Q_{x_1,\ldots,x_n}(0) = f(x, y)$$
4. Let $A \in (\mathbb{F}^{n+1})^{X_1 \times \ldots \times X_n}$ be the matrix s.t.
$$A_{x_1+r_1,\ldots,x_n+r_n} = Q_{x_1,\ldots,x_n}\ ;$$
5. For $i = 1, \ldots, n-1$, sample a random matrix $M^i \in (\mathbb{F}^{n+1})^{X_1 \times \ldots \times X_n}$;
6. Compute $M^n = A - \sum_{i=1}^{n-1} M^i$;
7. For $i = 1, \ldots, n$, output (M^i, r_i, a_i, b_i) to P_i;

Protocol:

1. Each party P_i broadcasts $u_i = x_i + r_i$ to all other parties;
2. Each party P_i broadcasts $z_i = M^i_{u_1,\ldots,u_n}$;
3. Each party P_i computes $\sum_{i=1}^{n} z_i$ and parses it as Q;
4. If $Q(a_i) = b_i$, party P_i outputs $z = Q(0)$, or $\perp$ otherwise;

Figure 2.4: Multiparty Malicious Secure Protocol using One-Time Truth Table

As previously argued for the two-party case, the view of the adversary (the preprocessing information, the u_is and the z_is for the honest parties) is distributed identically in the real protocol and in the simulated execution. The only difference is in the output of the honest parties: let $e = (e_0, e_1, \ldots, e_n)$ the error vector introduced by the adversary with respect to its shares z_is and let $E(X) = \sum_{i=0}^{n} e_i x^i$ be the corresponding polynomial. In the simula-

tion, all honest parties output `abort` if $e \neq 0$ or z' otherwise. In the real protocol instead, each party P_i outputs $(Q+E)(0)$ if $(Q+E)(a_i) = b_i$ or `abort` otherwise. But any attack to the protocol can be reduced to an attack to the underlying UIC scheme. More in details, this is possible because in the protocol even if the adversary corrupts $n-1$ parties, it will only learn one of the polynomial that are entries of the matrix A. Therefore it does not make a difference whether we first sample $Q, a_1, \ldots, a_n$ and compute the b_i's (as in the description of the UIC scheme) or whether (as in the description of the protocol) we first fix $(a_1, b_1, \ldots, a_n, b_n)$ and finally compute the coefficients of Q given those points and the value z. □

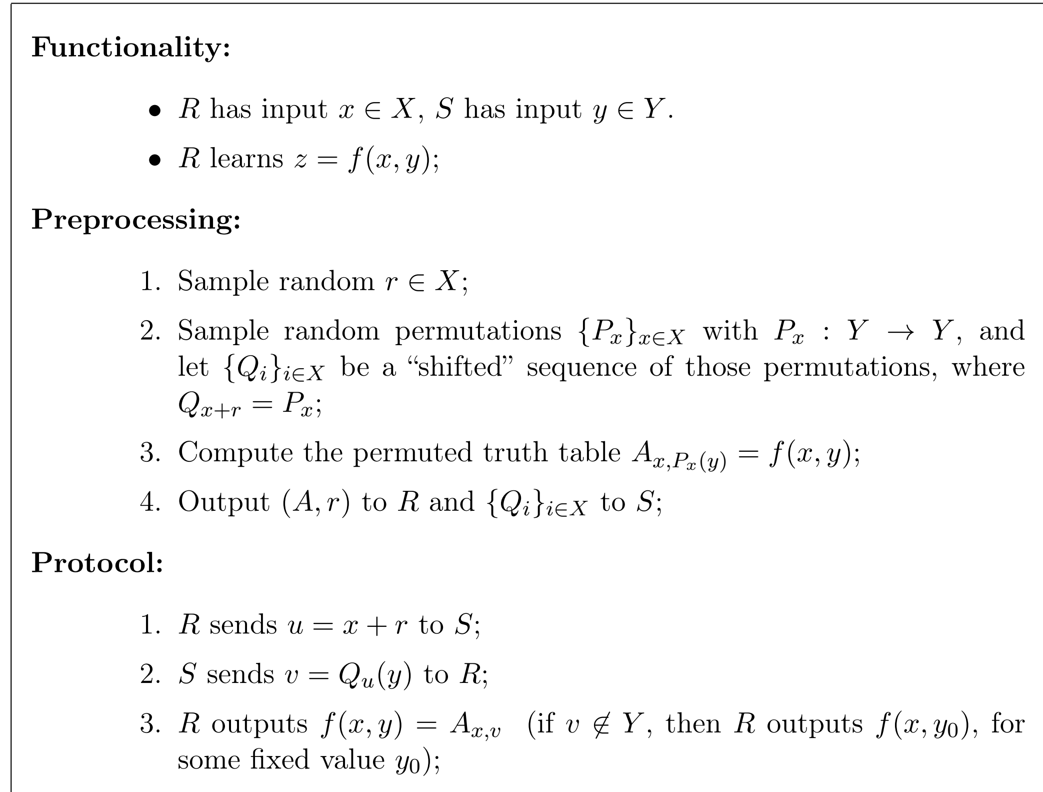
2.4 Perfect Security for Sender-Receiver Functionalities

In this section we show that, if only one party receives output, it is possible to achieve *perfect* security even against a *malicious* adversary. We will show, in Section 2.6, that this is not the case for general functionalities where all parties receive outputs.

The protocol is presented in Figure 2.5. The structure of the protocol is similar to previous constructions, in the sense that the preprocessing samples some random permutations, and then during the online phase the parties apply the random permutations on their inputs and exchange the results. However, the protocol uses the asymmetry between the sender and receiver: every row of the truth table (corresponding to each input of the receiver) is permuted using a different random permutation. The sender learns this set of permutations, permuted under a receiver permutation (implemented by a random circular shift, as in previous constructions). The receiver learns the truth table where each row is permuted according to the corresponding permutation.

In the online phase, the sender uses the first message of the receiver to determine which of the permutation to apply to his input. The receiver, using this value, can perform a look-up in the permuted truth table and output the correct result. The protocol is intuitively perfectly private as both parties only see each other's input through a random permutation. Perfect correctness is achieved because, in contrast to previous constructions, every

message sent by the sender uniquely determines its input (together with the preprocessing information).



Functionality:

- R has input $x \in X$, S has input $y \in Y$.
- R learns $z = f(x, y)$;

Preprocessing:

1. Sample random $r \in X$;
2. Sample random permutations $\{P_x\}_{x \in X}$ with $P_x : Y \to Y$, and let $\{Q_i\}_{i \in X}$ be a "shifted" sequence of those permutations, where $Q_{x+r} = P_x$;
3. Compute the permuted truth table $A_{x,P_x(y)} = f(x, y)$;
4. Output (A, r) to R and $\{Q_i\}_{i \in X}$ to S;

Protocol:

1. R sends $u = x + r$ to S;
2. S sends $v = Q_u(y)$ to R;
3. R outputs $f(x, y) = A_{x,v}$ (if $v \notin Y$, then R outputs $f(x, y_0)$, for some fixed value y_0);

Figure 2.5: Perfect Secure Protocol for Sender-Receiver Functionalities, Malicious Adversaries

Claim 4. *The protocol in Figure 2.5 securely computes the sender-receiver functionality f with perfect security and optimal communication complexity against malicious corruptions.*

Proof. When both parties are honest the output is correct:

$$A_{x,v} = A_{x,Q_u(y)} = A_{x,P_x(y)} = f(x, y).$$

If S^* is a corrupted sender, the simulator samples the preprocessing for S consisting of the permutations $\{Q_i\}_{i \in X}$. The simulator then picks a random message u, as the first message of the protocol. Then, it runs $v \leftarrow S^*(\{Q_i\}, y, u)$,

and it extracts an effective input $y' = Q_u^{-1}(v)$ and inputs y' to the ideal functionality to get the ideal-world output $z = f(x, y')$. The simulator outputs the simulated view $(\{Q_i\}, v)$. Observe that u and $\{Q_i\}$ are distributed as in the real world and independently of x. The simulated view considered jointly with f's output on the effective input (i.e., z) is thus distributed identically to the view of S^* jointly with the receiver's output, in the real-world execution.

For a corrupted receiver R^*, the simulator samples $A, r, \{Q_i\}$, runs $u \leftarrow R^*(A, r, x)$, extracts $x' = u - r$, inputs it to the ideal functionality, receives $z = f(x', y)$, computes $u = A_{x,v}^{-1}(z)$ and outputs the simulated view (A, r, u). This is distributed identically to the real-world view of R^*. □

Note that even for the case of semi-honest security, this protocol is more efficient than a protocol using 1-out-of-n OT, where the sender acts as the transmitter and offers $f(x_1, y), \ldots, f(x_m, y)$ for each possible $x_i \in X$ and the receiver acts as the chooser and selects x. Such protocol would have (online) communication complexity $O(|X| \log |Z|)$, while our protocol requires only $\log(|X|) + \log(|Y|)$ bits of communication.

2.5 Protocols For Specific Tasks

In this section, we present protocols for a number of specific sender-receiver tasks. We focus on the case of perfect security in the malicious model. All of these protocols have a 2-move structure: the receiver sends a message m_X, the sender replies with a message m_Y, and the receiver computes its output.

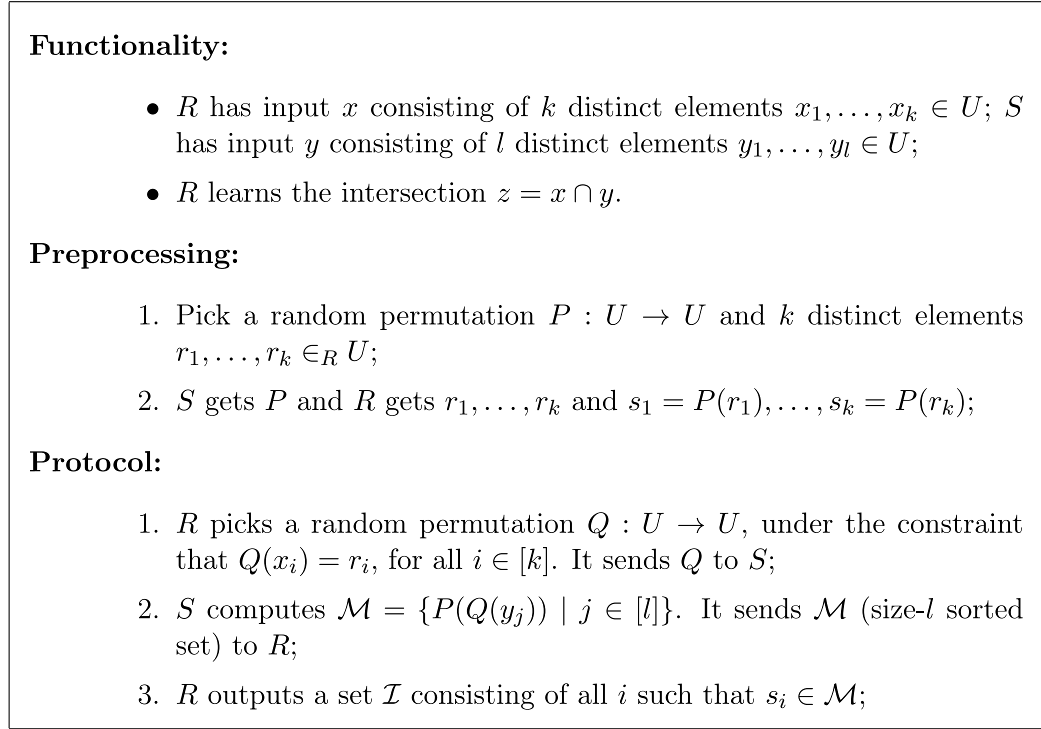
2.5.1 Set Intersection

In Figure 2.6, we present a protocol for computing the intersection of two sets x, y of fixed sizes (k, l, respectively) over some domain U.

Theorem 2.5.1. *The protocol in Figure 2.6 realizes the sender-receiver functionality* set intersection *with perfect security.*

Proof. First, we argue that, when both parties are honest, the output is correct; i.e. $\mathcal{I} = x \cap y$. If $x_i = y_j$, for some i, j, then for some $s \in \mathcal{M}$

$$s = P(Q(y_j)) = P(Q(x_i)) = P(r_i) = s_i.$$



Functionality:

- R has input x consisting of k distinct elements $x_1, \dots, x_k \in U$; S has input y consisting of l distinct elements $y_1, \dots, y_l \in U$;
- R learns the intersection $z = x \cap y$.

Preprocessing:

1. Pick a random permutation $P : U \to U$ and k distinct elements $r_1, \dots, r_k \in_R U$;
2. S gets P and R gets $r_1, \dots, r_k$ and $s_1 = P(r_1), \dots, s_k = P(r_k)$;

Protocol:

1. R picks a random permutation $Q : U \to U$, under the constraint that $Q(x_i) = r_i$, for all $i \in [k]$. It sends Q to S;
2. S computes $\mathcal{M} = \{P(Q(y_j)) \mid j \in [l]\}$. It sends $\mathcal{M}$ (size-l sorted set) to R;
3. R outputs a set $\mathcal{I}$ consisting of all i such that $s_i \in \mathcal{M}$;

Figure 2.6: Protocol for Set Intersection

Conversely, if an element $s \in \mathcal{M}$ is equal to some s_i, then for some y_j

$$y_j = Q^{-1}(P^{-1}(s)) = Q^{-1}(P^{-1}(s_i)) = Q^{-1}(r_i) = x_i.$$

In case of a corrupted sender S^*, the simulator works as follows: first, the simulator picks P as in the protocol. It then picks a random permutation Q and computes a size-l set $\mathcal{M} \leftarrow S^*(P, Q)$. Finally, it computes an effective input set $y' = Q^{-1}(P^{-1}(\mathcal{M}))$, inputs it to the ideal functionality on behalf of S^*, and outputs the simulated view (P, Q). The simulated view, considered jointly with the ideal-world output $((P, Q), f(x, y'))$, is distributed exactly as the view in the protocol joint with the receiver's actual output.

In case of a corrupted receiver R^*, the simulator picks $P, r_1, \dots, r_k$, as in the protocol, and computes a permutation $Q \leftarrow R^*(\{r_i\}_{i\in[k]}, \{P(r_i)\}_{i\in[k]})$. It extracts an effective input $x' = \{Q^{-1}(r_i)\}_{i\in[k]}$, inputs it to the ideal functionality on behalf of R^* and gets $\mathcal{I} = f(x', y)$. It then constructs a message $\mathcal{M}$ as follows: for each x'_i, if $x'_i \in \mathcal{I}$, add $P(r_i)$ to $\mathcal{M}$; otherwise, choose a random element not chosen before and not equal to any $P(r_j)$ and add it to

$\mathcal{M}$. Finally, it outputs $(\{r_i\}_{i\in[k]}, \{P(r_i)\}_{i\in[k]}, \mathcal{M}\})$. This simulated view is distributed exactly as in the real-world execution. □

Optimizing the protocol In the above protocol, both the randomness and the first message have size $O(|U|\log|U|)$ (the space it takes to describe a permutation). This may be super-exponential in the input size. Like for the equality protocol, we can optimize by taking advantage of k-wise independent families of permutations, as these may have smaller descriptions

Definition 2.5.1. *Let S_U be the set of all permutations over U. A family of permutations F over U is said to be* k-wise independent *if, for $P_1 \in_R F, P_2 \in_R S_U$, for all $i_1, \dots, i_k \in U$, the following distributions are identical:*

$$P_1(i_1), \dots, P_1(i_k) \equiv P_2(i_1), \dots, P_2(i_k) \ .$$

Similarly, we say that F is a k-wise, ϵ-independent family of permutations if the statistical difference between the two distributions is $\leq \epsilon$.

In each execution of the above protocol, the receiver only sees $k+l$ evaluations of P (k in the preprocessing stage and l when receiving $\mathcal{M}$). Hence, we pick P in the preprocessing stage from a family F_{k+l} of $k+l$-wise permutations. Modifying the simulator to also pick P from F_{k+l}, still produces a view that is distributed as in the real world.

Similarly, we can replace the receiver's message Q in the protocol by a permutation Q sampled from a k-wise independent family of permutations F_k (subject to $Q(x_i) = r_i$, for all $i \in [k]$). A uniformly sampled permutation Q from F_k still maps $x_1, \dots, x_k$ to uniformly random distinct values. The simulator remains unchanged, except that it also pick its Q from F_k.

Efficient explicit constructions of "small" k-wise independent families of permutations are known for $k = 1, \dots, 3$. Recently, Kuperberg et al. [KLP11] proved the existence of "small" k-wise independent families of permutations, for any k. This is only an existence proof, so this allows us to do set-intersection with efficient communication and randomness, but still super-polynomial computation.

We can also settle for *statistical* security and use families of k-wise *almost* independent permutations. This will make the simulated view statistically close to the real view. Kaplan et al. [KNR05] give efficient explicit constructions of k-wise almost-independent families of permutations, for any k.

2.5.2 Inner product

Let $\mathbb{F}_p$ be a finite field of size p, and let $t \geq 1$. The inner product functionality $\mathrm{IP}_{t,p}$ is as follows:

- S has input $y \in {\mathbb{F}_p}^t$ and R has a linear function $x : {\mathbb{F}_p}^t \to \mathbb{F}_p$, represented by a vector $x \in {\mathbb{F}_p}^t$, so that $x(y) = \langle x, y \rangle = \sum_{i \leq t} x_i y_i$.
- R outputs $x(y)$.

2.5.2.1 Some Algebraic Preliminaries

We need the following background for our protocol and its analysis. Let $e_i \in \mathbb{F}^t$ be the i-th unit vector of length t. Let $\mathbb{F}^*$ be the multiplicative group of a finite field $\mathbb{F}$. By default, vectors are column vectors. $\mathbb{F}^{m \times n}$ is the set of $m \times n$ matrices over $\mathbb{F}$. Let $GL(n,p)$ be the group (under matrix multiplication) of invertible matrices in ${\mathbb{F}_p}^{n \times n}$. M_i is the i'th row of a matrix $M \in \mathbb{F}^{n \times m}$, and M^i is its i'th column. Given vectors $v_1, \ldots, v_n \in \mathbb{F}^m$, let $(v_1; \ldots; v_n)$ denote the matrix $M \in \mathbb{F}^{n \times m}$ with rows $M_i = v_i^T$. We will also need the following algebraic primitive:

Definition 2.5.2 (Good exhaustive operator)**.** *Let $L : {\mathbb{F}_p}^t \to {\mathbb{F}_p}^t$ be the linear, injective operator defined via $L(y) = y^T L$ (L represents both the operator and the matrix implementing it). Consider the (infinite) sequence* $\mathrm{seq}_L = (v, L(v), \ldots, L^{(i)}(v), \ldots)$ *generated by L for some v.*

We say that L is a good exhaustive operator *for $v \in {\mathbb{F}_p}^t$ if:*

1. seq_L *is periodic with period length $p^t - 1$ for v (i.e., all elements in $\mathbb{F}_p^t$, except 0, appear in* seq_L*).*
2. *The the first t elements in* seq_L *forms a basis for ${\mathbb{F}_p}^t$.*

We refer to a *good exhaustive operator* L for $v = e_t$ simply as a *good exhaustive operator*.

Lemma 1 (instantiating *good exhaustive operators*)**.** *Consider the operator L_x where $L_x(y) = x \cdot y$ where x, y are viewed as elements of $\mathbb{F}_{p^t}$ (the multiplication is over $\mathbb{F}_{p^t}$). Viewed as a linear function from ${\mathbb{F}_p}^t$ to ${\mathbb{F}_p}^t$, we have $L_x(y) = y^T L_x$. Let g be a generator of $\mathbb{F}_{p^t}^*$, then L_g is a* good exhaustive operator *for $v = e_t$ (could use any other vector v).*

Proof. Observe that ${L_g}^i = L_{g^i}$, and $L_{a_1 g^{i_1} + a_2 g^{i_2}} = a_1 {L_g}^{i_1} + a_2 {L_g}^{i_2}$ (here L_g is viewed as a matrix).

Since ${L_g}^{p^t-1} = L_{g^{p^t-1}} = I$, the sequence seq_{L_g} is periodic, and its period length divides $p^t - 1$. Its period length is not smaller than $p^t - 1$, since $e_t^T {L_g}^\alpha = 1$ implies $1 \cdot g^\alpha = 1$ (by definition of L_g). Thus, we must have $p^t - 1 | \alpha$ (since g is a generator of $\mathbb{F}_{p^t}^*$).

Next, assume for contradiction that the vectors $e_t^T, e_t^T L_g, \ldots, e_t^T {L_g}^{t-1}$ are not linearly independent. Let $0 \neq q \in {\mathbb{F}_p}^t$ such that $\sum_{i=0}^{t-1} e_t^T q_i {L_g}^i = e_t^T L_{\sum_{i=0}^{t-1} q_i g^i} = 0$. This implies that $q(g) = 0$ for a polynomial $q(x) \in \mathbb{F}_p[x]$ for rank $t' < t$ (with coefficients as in the linear combination q). Thus, g is the root of an irreducible polynomial $q' \in \mathbb{F}_p[x]$ of degree $< t$ (with $q'(x)|q(x)$). Thus, the field extension $\mathbb{F}_p(g)/\mathbb{F}_p$ is of degree $< t$, which is a subfield of $\mathbb{F}_{p^t}$ containing g. Thus, *order*(g) (in $\mathbb{F}_{p^t}^*$) is $\leq p^{t-1} - 1$, contradicting the fact that g generates $\mathbb{F}_{p^t}^*$. □

Functionality:

- Inputs: R gets $x \neq 0 \in \mathbb{F}_p^t$, and S gets $y \in \mathbb{F}_p^t$.
- Output: R outputs $\langle x, y \rangle$.

Primitives: Let $L \in {\mathbb{F}_p}^{t \times t}$ be a *good exhaustive operator* for e_t. For $a \in \mathbb{F}_p^t \setminus \{0\}$, we let *ind*$(a)$ denote the index of the first appearance of a in seq_L.

Preprocessing:

1. S gets a random vector $r_2 = y' \in \mathbb{F}_p^t$.
2. R gets (x', p_2), where x' is randomly chosen at $\mathbb{F}_p^t \setminus \{0\}$ and $p_2 = \langle y', x' \rangle$.

Protocol:

1. R sends $\delta = ind(x') - ind(x) \pmod{p^t - 1}$ to S.
2. S sends the vector $m = (e_t^T L^0(y + L^\delta y'), e_t^T L(y + L^\delta y'), \ldots, e_t^T L^{t-1}(y + L^\delta y'))$.
3. R sets $M = (e_t^T; e_t^T L; \ldots; e_t^T L^{t-1})$, $r = x^T M^{-1}$. It outputs $rm - p_2$.

Figure 2.7: A protocol for $\text{IP}+_{t,p}$

2.5.2.2 The protocol.

In Figure 2.7, we present a protocol for a slightly modified functionality, $\mathrm{IP}+_{t,p}$, where x is restricted to be non-zero. This also implies a protocol for $\mathrm{IP}_{t,p}$, as on input $x = 0$ the receiver can adopt any input $x' \neq 0$, and output 0 at the end, ignoring the communication.

Theorem 2.5.2. *The above protocol is a perfectly secure protocol, with preprocessing, for the functionality $IP+_{t,p}$, for all $t \geq 1$ and prime p. The communication complexity, randomness size and S's work are polynomial in $|x| = t \log p$, while R's computation is as hard as finding discrete log in $\mathbb{F}^*_{p^t}$.*

Security proof. We will start with verifying that the protocol works correctly if both parties are honest. First observe that r is well defined since M is invertible by property 2 of L. We have

$$\begin{aligned} rm - p_2 &= r(e_t^T L^0 y, \ldots, e_t^T L^{t-1} y) + r(e_t^T L^{\delta} y', \ldots, e_t^T L^{\delta+t-1} y') - p_2 \\ &= rMy + rML^{\delta} y' - p_2 \\ &= x^T y + x^T L^{\delta} y' - \langle x', y' \rangle \\ &= \langle y, x \rangle + \langle y', x' \rangle - \langle y', x' \rangle = \langle y, x \rangle, \end{aligned}$$

where the second equality follows since $x^T L^{\delta} = e_t^T L^{ind(x)} L^{ind(x')-ind(x)} = e_t^T L^{ind(x')} = x'$. Next, we show how to simulate a corrupted receiver R^*. The simulator generates a view as follows (on input x):

- Pick x' uniformly at random from $\mathbb{F}_p^t \setminus \{0\}$, and $p_2 = \langle b', x' \rangle$, where b' is picked uniformly at random from $\mathbb{F}_p$ (that is, p_2 is uniform if $b' \neq 0$, and identically 0 otherwise).

- Invoke R^* on x and (x', p_2) to obtain the index δ that R^* sends. Extract x from x', δ. Send x to the TP, and let *out* denote its reply.

- Simulate m as follows. Let $v \in \mathbb{F}_p^t$ denote a linear combination such that $v^T M = x$. Arbitrarily complete $\{v\}$ into a basis of $\mathbb{F}_p^t$ to obtain $\{v_1 = v, v_2, \ldots, v_t\}$, and let $V = (v_1; \ldots; v_t)$. Let $m = V^{-1}(p_2 + out, l_1, \ldots, l_{t-1})$, where the l_i's are picked independently at random.

Clearly, (x', p_2) is distributed as in the real world. Given $x', p_2 = \langle y', x' \rangle$ in the real world, m contains t linear combinations of $(y, y') \in \mathbb{F}_p^{2t}$, specifically $m = (M; M_\delta)(y, y')$. The rows of $(M; M_\delta)$ are independent since $M_\delta = ML^{\delta}$

(where δ is the actual value sent by the receiver) is non-singular, as L is non-singular by definition, and M is non-singular by property 2 of the definition. Only $\langle y', x' \rangle = out$ is known, and otherwise y' is random (that is, random inside an affine sub-space of dimension $t-1$). Thus m (as sent in the real protocol) satisfies $VL_\delta(a, a') = Vm = (p_2 + out, l_1, \ldots, l_{t-1})$ (distributed as in the simulation above), and thus $m = V^{-1}(p_2 + out, l_1, \ldots, l_{t-1})$. To see the protocol is secure against a malicious sender, we proceed by proving it is private against malicious senders, and then prove a lemma, stating that such a protocol, which also satisfies honest correctness (proved above), and some additional properties is secure against malicious senders. We say a (2-round) protocol is perfectly private against malicious senders, if the view of any (possibly malicious) sender S^* is independent of x (observe that for two rounds, this view is independent of the behavior of S).

Claim 5. *Let Π be a 2-round protocol with preprocessing for IP+, which is perfectly private against a malicious sender, and perfectly correct when the parties are honest. If the set of possible Round 2 messages, in honest executions, is of size $|Y|$, then Π is also secure against a malicious sender.*

Proof. Let S^* denote a corrupted sender. Consider the view of S^* jointly with R's output $((r_S, m_1), out)$ (where r_S denotes the randomness given to the sender during preprocessing phase, m_1 denotes the Round 1 message that it gets, and out denotes the receiver's output). An ideal-world adversary can simulate this as follows:

- Pick r_S as in the preprocessing phase.
- Pick m_1 according to the distribution seen in the real protocol given r_S (can be simulated due to privacy - m_1 is independent of x).
- Simulate S' on (r_S, m_1), to obtain a message m_2. Let y^* be the message for which m_2 is sent in the protocol.

The only non-trivial point to show is that y^*, as above, always exists.[4] Observe that if the same m_2 is sent (by the honest S) for some $y_1 \neq y_2$ given (r_S, m_1), then the outputs for these will be the same for all x values. Assuming the function is non-trivial (as IP+), in the sense that for all $y_1 \neq y_2$ there is an x on which $f(y_1, x) \neq f(y_2, x)$, this means that with non-zero

[4]The proof is similar to that of Theorem 2.8.1 below.

probability, the (honest) receiver errs on at least one of the two inputs. This is the case, since given r_S, message m_1 is sent for all values of x (otherwise receiver's privacy would not hold) for some choices of receiver randomness r_R, r'_R, respectively. Thus, in at least one of these executions, R's output must be wrong, contradicting perfect correctness (for honest parties). Finally, since all values of m_2 are sent for all values of r_S in response to m_1, there must be a value of y^* consistent with the selected (r_S, m_1, m_2).

Finally, applying Claim 5 to our case (indeed, the set of possible m_1 messages is of size $|Y| = p^t$), security against malicious senders follows. □

Related work: In [DvMN10], a protocol with perfect security against malicious adversary is given for the following Sender-Receiver functionality:

$$f((x,r),y) = \mathrm{IP}(x,y) + r.$$

Such a protocol allows two parties to compute the additive sharing of the result of the inner product functionality. Note that, in the case of malicious security, it is not possible to reconstruct the result from the shares (MACs could be used like in some of the protocols presented in this chapter — at the cost of losing perfect security) and therefore the protocol presented here accomplishes a strictly more difficult task.

2.6 Perfect Security for General Functionalities is Impossible

The following theorem shows that the above positive result cannot be extended to general functionalities where both players receive output. See section 2.7 for a tight tradeoff between communication and error probability.

Theorem 2.6.1. *Let $f(x_1, x_2) = (x_1 \oplus x_2, x_1 \oplus x_2)$. Then, there is no protocol for f, in the preprocessing model, which is* perfectly secure with abort.

Proof. Assume towards a contradiction that π perfectly realizes f with abort given preprocessing $\mathcal{D}$. Consider the experiment of running π on a uniformly random choice of inputs $(x_1, x_2) \in \{0,1\}^2$ and correlated random inputs (r_1, r_2) drawn from $\mathcal{D}$. Let i_1 be the *minimal* number such that, at the beginning of round i_1, the output of P_1 is always *determined* (over all choices of inputs and random inputs) regardless of subsequent messages (which may

possibly be sent by a malicious P_2^*). That is, when running the above experiment, before round i_1, party P_1 may have an uncertainty about the output; but, at the beginning of round i_1, the view of P_1 always determines a unique output value $b \in \{0, 1\}$ such that P_1 will either output b or $\perp$. The value i_2 is defined symmetrically. Note that i_1, i_2 are well defined, because the outputs are always determined at the end of the execution, and, moreover, they are distinct because only one party sends a message in each round.

Assume, without loss of generality, that $i_1 < i_2$ and, moreover, that in the above experiment there is an execution which terminates with P_2 outputting 0, but where in the beginning of round i_1 the output of P_2 is not yet determined (namely, there are messages of P_1^* that would make it output 1).

We can now describe a malicious strategy P_1^* that would make an honest P_2, on a random input x_2, output 1 with probability $p > 1/2$. Since this is impossible in the ideal model, we get the desired contradiction. The malicious P_1^* proceeds as follows.

- Run the protocol honestly on a random input x_1 until the beginning of round i_1. Let b be the output value determined at this point.

- If $b = 1$, continue running the protocol honestly.

- Otherwise, continue the protocol by sending a uniformly random message in each round.

In the event that $b = 1$, which occurs with probability $1/2$, P_2 will always output 1. In the event that $b = 0$, by the above assumption there exist subsequent messages of P_1^* making P_2 output 1, and hence also in this case P_2 outputs 1 with nonzero probability. Overall P_2 outputs 1 with probability $p > 1/2$. □

Now we will show two extensions to this impossibility result. One that shows that if we go with statistical security there is a trade-off between the communication complexity and the security. And one that takes away the assumption of a fixed round-protocol

2.6.1 Trade-off between Communication and Statistical Security

We first observe that the impossibility result for the perfect case, can be extended into a quantitative bound on the communication complexity for statistically secure protocols with preprocessing and abort.

Theorem 2.6.2. *Every protocol with preprocessing* Π *that* ϵ*-securely computes the functionality* $f(x_1, x_2) = (x_1 \oplus x_2, x_1 \oplus x_2)$ *with abort, has communication complexity* $\Omega(\log \frac{1}{\epsilon})$.

Proof. We use the notation of Section 2.2, to define and analyze protocols with preprocessing. Consider a statistically-secure protocol with preprocessing, Π, with ℓ rounds and communication complexity t. Assume, for simplicity, that the output of both parties in honest executions is always correct. We prove that, when the input of P_1 is uniformly distributed, P_2 can make P_1's output 1 with probability at least $\frac{1}{2} + \tilde{\Omega}(\frac{1}{2^t})$ (while possibly making it output $\perp$ in some cases), or vice versa, swapping the parties' roles. Clearly, this is $\tilde{\Omega}(\frac{1}{2^t})$-far from the distribution achieved by any ideal-world adversary.

By an "honest execution" of Π, we refer to an execution where both parties follow the protocol, for some initial inputs (x, y) and randomness $(r_1, r_2) \in$ *support*$(\mathcal{D})$. For randomness $(r_1, r_2) \in$ *support*$(\mathcal{D})$, inputs (x, y), and round number i in which P_1 speaks, consider the set of honest executions of Π on (r_1, r_2', x, y'), for some r_2', y', such that the communication transcript before round i, is consistent with the honest execution on (r_1, r_2, x, y). Let $i^1_{r_1,r_2,x,y}$ denote the minimal round number i, for which in all executions as above, that correspond to r_1, r_2, x, y, i, party P_1 "knows" the output already at the beginning of round i. That is, in all such executions, P_1's output at the end is the same (and equals $x \oplus y$). We define $i^2_{r_1,r_2,x,y}$ analogously. Given r_1, r_2, x, y, such round indices always exist as at the end each party knows the output (the index may be $\ell + 1$).

Claim 6. *Let* ℓ *be the number of rounds in the protocol. Then, there exists a round number* i_{min}*, such that either* $\Pr_{(r_1,r_2)\leftarrow\mathcal{D}}(i_{min} = i^2_{r_1,r_2,0,0} < i^1_{r_1,r_2,0,0}) \geq \frac{1}{2\ell}$ *(*P_2 *"wins"), or* $\Pr_{(r_1,r_2)\leftarrow\mathcal{D}}(i^1_{i_{min}=r_1,r_2,0,0} < i^2_{r_1,r_2,0,0}) \geq \frac{1}{2\ell}$ *(*P_1 *"wins").*

To prove the claim, observe first that $i^1_{r_1,r_2,0,0}, i^2_{r_1,r_2,0,0}$ are always different, since P_1 speaks in odd rounds, and P_2 speaks in even rounds, or vice versa. For each (r_1, r_2), the honest execution on $(r_1, r_2, 0, 0)$ may have one of ℓ different values for $\min(i^1_{r_1,r_2,0,0}, i^2_{r_1,r_2,0,0})$, and one of 2 values for the identity

of the party p for which $i^p_{r_1,r_2,0,0}$ is smaller – 2ℓ values overall. Thus, at least $\frac{1}{2\ell}$ of the probability weight has the same values, hence the claim follows.

Let i_{min} be as guaranteed in the claim, and assume (without loss of generality) that P_2 wins. We demonstrate a cheating strategy for P_2, that biases P_1's output towards 1 by $\Omega(\frac{1}{\ell 2^t})$: Up to round i_{min}, proceed honestly, on inputs $(r_1, 0)$. Then, in round i_{min}, do the following:

1. If the output *out* of an honest P_2 is still not known, or is known and equals 1, proceed honestly on inputs $(r_1, 0)$.

2. Otherwise, pick a random reply sequence from

$$\left\{(m_{i_{min}}, \ldots, m_\ell) \mid \sum_i |m_i| \leq t\right\}$$

, and respond according to it in subsequent rounds.

Consider an execution of the protocol, between an honest P_1, whose input, x, is uniform over $\{0, 1\}$, and a malicious P_2 as described above. If P_1's input is 1, then by the above strategy for P_2, party P_1's will output 1. This happens with probability 1/2 over r_1, r_2, x. If P_1's input was 0, then with (conditional) probability at least $\frac{1}{2\ell}$ party P_2 "wins", since P_2's "effective" input is also 0 until round i_{min}, and by choice of i_{min}. In that case, for any (r_1, r_2), there is at least one choice of the remaining communication for P_2 that makes P_1 output 1 (note that not all continuations make P_1 output either 0 or $\perp$, since in the definition of i_{min} we are considering only honest executions). P_2 guesses that communication sequence with probability at least $\frac{1}{2^t}$. Thus, the event when P_1's input is 0, contributes at least $\frac{1}{\ell 2^{t+2}}$ to its probability of outputting 1. Overall, P_1 outputs 1 with probability at least $1/2 + \tilde{\Omega}(\frac{1}{2^t})$.

Finally, assume that the probability for a correct reply for each party is $\geq 1 - \epsilon$ (not necessarily 1) on all inputs x, y, assuming both parties are honest. We leave the strategy for P_2 as is. In particular, observe that $i^p_{r_1,r_2,0,0}$ still exist for any honest execution, but the output "known" to P_p may be incorrect for certain (r_1, r_2) pairs. Also, all executions contributing to P_1 outputting 1 corresponded to honest executions. If P_1's input was chosen to be 1, then for all (r_1, r_2) such that the output of both parties is correct in the honest execution $(r_1, r_2, 1, 0)$, party P_2 will proceed honestly. This contributes $\geq (1-2\epsilon)/2$ to P_1's probability of outputting 1. If P_1's input was

0, then with conditional probability at least $\frac{1-\epsilon}{2\ell}$ party P_2 deviates and "wins". Also, by definition of i_{min}, there exists a choice of subsequent communication for P_2, that makes P_1 output 1 (although the "known" output does not need to be always correct, by definition of $i^1_{\dots}$ both outputs can be induced!). Thus, the event that P_1's input is 0, contributes a probability weight of $\frac{1-O(\epsilon)}{2\ell 2^t}$ to the event when P_1 outputs 1. Overall, P_1 outputs 1 with probability at least $p' = 1/2+\tilde{\Omega}(\frac{1}{2^t})-O(\epsilon)$. To meet the security requirement, we get $p' \leq 1/2+\epsilon$, we still get $\epsilon \geq 1/2^{\Omega(t)}$, that is, qualitatively the same result. □

2.6.2 With expected round complexity

We also observe that the impossibility result from Theorem 2.6.1, which assumes strict constant round complexity ℓ, can be extended to the expected case. That is, to protocols where the worst-case communication complexity has expectancy t (over the choice of (r_1, r_2)). More precisely, in this model, the parties get correlated, infinite vectors of bits r_1, r_2, and in every round, the corresponding party reads a bounded prefix of its randomness (of length depending on the round number), and sends a message depending on that prefix, its input, and all previous communication. In particular, the number of rounds of protocols in this model may be unbounded.

Theorem 2.6.3. *Let $f(x_1, x_2) = (x_1 \oplus x_2, x_1 \oplus x_2)$. Then there is no protocol for f in the preprocessing model which is* perfectly secure with abort*, having expected communication complexity t, for any $t \in \mathbb{N}$.*

Proof. The proof can be obtained by a simple modification of the proof of Theorem 2.6.2. Assume such a protocol Π, with expected communication complexity t exists. We define the $i^p_{r_1,r_2,0,0}$ as the minimal over the same set of honest executions as before, in which P_p "knows" its output will be *out* with probability 1. Next, we prove the following analog of Claim 6.

Claim 7. *Let ℓ be the expected number of rounds in the protocol. Then, there exist $i_{min} \leq 100t$ and L, such that the event where the communication complexity of the honest execution $(r_1, r_2, 0, 0)$ is bounded by L, and either $i_{min} = i^2_{r_1,r_2,0,0} < i^1_{r_1,r_2,0,0}$, or $i_{min} = i^1_{r_1,r_2,0,0} < i^2_{r_1,r_2,0,0}$ has non-zero probability.*

As before, the parities speak in alternating rounds. As the expected communication complexity is t, the expected round complexity is some $t' \leq t$ (as at least one bit is sent in each round). By Markov's inequality, the

probability that an honest execution $(r_1, r_2, 0, 0)$ has more that $100t$ rounds is at most 0.01. Thus, with probability at least 0.99, a value i_{min} exists, and is at most $100t$ (by perfect correctness). In other words, one of the 2ℓ values, say $(i_{min}, 2)$ gets at least $\frac{1}{2\ell}$ of the probability weight of the (r_1, r_2)'s respecting this round bound; that is, at least $\frac{0.99}{2\ell}$ of the probability weight. Also, in these executions, the communication complexity is bounded by some number L, which bounds the communication complexity of executions that terminate within $100t$ rounds. The claim follows.

We use the same adversarial strategy for P_2, as in the proof of Theorem 2.6.2, except for letting P_2 pick a message sequence of total length 2^L, and also decide when to terminate (sometime before round $100t$). As before, we let P_1's output be random. If $x = 1$, party P_2 proceed honestly and P_1's output is 1, which occurs with probability 1/2. If $x = 0$, then with non-zero (conditional) probability P_2[5] will pick its messages randomly. By a calculation similar to the fixed-round case, P_2 correctly guesses a subsequent sequence of messages that make P_1 output 1 with non-zero probability (comparable to 2^L). So, overall, the event $x = 0$ contributes a non-zero probability to P_1's outputting 1. Thus, the above strategy allows to bias P_1's output towards 1, which is impossible in the ideal model. □

2.7 Impossibility of a Perfectly Secure Multi-Party Protocol

In this section, we give evidence that

Theorem 2.7.1. *There exist 3-party functionalities that deliver output to only one party and cannot be securely computed with full perfect security (even in the preprocessing model).*

Proof. Our proof We rely on [IKLP06, Thm. 2] that presents a functionality f, whose output goes to a single party (a receiver), and cannot be computed with full security in constant rounds (in fact, the impossibility in [IKLP06] rules out a weaker type of security and is thus stronger than what we need). There are two issues to resolve regarding the impossibility in [IKLP06]: it

[5] Which we have shown to be quite high, but we do not need the exact bound for this proof.

only rules out constant round protocols, and it works in the standard model where no correlated randomness (pre-processing) is available. Suppose there exists a protocol for f with preprocessing and perfect security. Then, we can eliminate the preprocessing under standard cryptographic assumptions (specifically, using OT and collision resistant hash-functions) actually implementing it in the plain model with full security in a constant number of rounds. This is because it is a randomized functionality with no inputs (so if the protocol aborts, one can disqualify a cheater and restart without privacy issues).

Finally, we note that perfect security implies constant number of (online) rounds. More specifically, the negative result in [IKLP06] says that to get ϵ-security, the number of rounds should grow (slowly) with $1/\epsilon$. So if a protocol has r rounds it cannot have better than $\epsilon < 1/r$ security (in particular, it cannot be perfectly secure). □

We note that the above does not rule out the possibility of having either perfect security with abort or statistical full security (without abort).

2.8 Negative Results on Communication and Randomness

2.8.1 Communication Lower Bounds

Let M_X (resp., M_Y) denote the set of all possible message sequences sent by the receiver (resp., sender) in any protocol execution (on any input and any randomness). We start by stating our lower bounds.

Perfect security. The following theorem shows, that under certain "non-triviality" conditions on the sender-receiver functionality f, that $|M_X|, |M_Y|$ are at least as large as the corresponding input domains. Examples of such non-trivial functions include equality, m-out-of-n OT, and essentially any other natural functionality.

Theorem 2.8.1. *Let Π be a protocol for the sender-receiver functionality $f : X \times Y \to Z$, which is perfectly secure in the semi-honest model, with sender message domain M_Y, and receiver message domain M_X. Then,*

- *If for all* $y_1 \neq y_2 \in Y$*, there exists* $x \in X$ *such that* $f(x, y_1) \neq f(x, y_2)$*, then* $|M_Y| \geq |Y|$*.*
- *If for every* $z_1, z_2 \in Z$ *and* $x_1 \neq x_2 \in X$*, we have* $\{y|f(x_1, y) = z_1\} \neq \{y|f(x_2, y) = z_2\}$*, then* $|M_X| \geq |X|$*.*

Statistical security. Next, we show that the above bounds on communication can be generalized to ϵ-secure protocols, for small enough constant ϵ. More concretely, for the proofs to go through, the bounds for the statistical case would pose more stringent requirements on the function f computed, and would be slightly worse (by a constant factor). We split the theorem statement into two, one for each direction of the communication.

Theorem 2.8.2. *Let* $c > 0$ *be a constant, and consider a sender-receiver functionality* $f : X \times Y \to \{0, 1\}$ *where* $\log |X| = n$, $\log |Y| = m$*. Assume there exists a subset* $X' \subseteq X$ *of size* $c \cdot m$*, such that* $\{(x', f(x', I_Y))\}_{x' \in X}$ *determines* I_Y*, for all* $I_Y \in Y$*). Then, there exists* $\epsilon > 0$*, depending only on* c*, such that in any* ϵ*-secure protocol with preprocessing for* f *in the semi-honest model, the sender-side communication is* $\Omega(m)$*.*

The above requirement from f can be viewed as a strengthening of the requirement in the first part of Theorem 2.8.1. Namely, in Theorem 2.8.1 it is required that for all $y_1 \neq y_2$, there is some $x \in X$ distinguishing between them ($f(x, y_1) \neq f(x, y_2)$), while here we require that this x comes from a small subset $X' \subseteq X$ common to all y_1, y_2 pairs.

Theorem 2.8.3. *Let* $c_1, c_2, c_3 > 0$ *be constants such that* $(1+c_3)(1-c_2) < 1$*. Consider a sender-receiver functionality* $f : X \times Y \to \{0, 1\}$ *where* $\log |X| = n$, $\log |Y| = m$ *satisfying*

- *For all* $x \neq x' \in X$*, we have* $H(f(x', I_Y)|f(x, I_Y)) \geq c_1$*, where* I_Y *is picked uniformly from* Y*.*
- *For all* $y \neq y' \in Y$*, we have* $\Pr(f(I_X, y) = f(I_X, y')) \geq c_2$*, where* I_X *is picked uniformly at random.*
- *There exists a subset* $Y' \subseteq Y$ *of size* $(1+c_3)n$*, such that* $\{y, f(I_X, y)\}_{y \in Y}$ *determines* I_X*, for all* $I_X \in X$*).*

Then, there exists $\epsilon > 0$*, depending only on the* c_i*'s, such that in any* ϵ*-secure protocol with preprocessing for* f*, in the semi-honest model, the receiver-side communication is* $\Omega(n)$*.*

To gain some intuition on the preconditions of this theorem, observe that the third condition is analogous to the requirement of Theorem 2.8.2, reversing the roles of x and y. Additionally, we require that learning $f(x,y)$ leaves "a lot" of uncertainty about $f(x',y)$ for any $x \neq x'$, and randomly chosen y. Symmetric conditions are posed on $f(x,y)$ and $f(x,y')$.

These preconditions of Theorem 2.8.3 are satisfied by many "natural" functions. For starters, a random function $f : X \times Y \to \{0,1\}$ on $X \times Y$ for large enough X, and (say) $|Y| \geq 1000 \log |X|$ satisfies the condition for suitable constants c_1, c_2, c_3 with high probability. That is, one can fix c_1, c_2, c_3, and prove that, say, a 0.7 fraction of all functions $f : X \times Y \to \{0,1\}$ satisfy the theorem's preconditions for c_1, c_2, c_2. The proof is standard, and relies on applying Chernoff's bound, combined with the union bound. Additionally requiring that X is not too small relatively to Y, say $|Y| \geq 1000 \log |X|$, will also imply that the preconditions of Theorem 2.8.2 are satisfied, and thus a bound on sender-side communication.

An explicit example of a function for which Theorem 2.8.3 holds, is inner product restricted to $\mathbb{F}_2^n \setminus \{0\} \times \mathbb{F}_2^n$ for $n \geq 3$, which satisfies the theorem with $c_1 = h(3/8), c_2 = 3/8, c_3 = 0.067$. Another example is the function parity-$\binom{m}{n}$-OT (that is, the parity of the OT's output vector), for $n > 2, m < n$, satisfies the theorem with $c_1 = 1, c_2 = 1/2, c_3 = 0$. Specifically, observe that $f(x,y) = \mathrm{IP}_{x,y}$ (over $\mathbb{F}_2$), where y is the characteristic vector of the corresponding size-m set[6]. To verify condition 3, observe that if the set of possible y's has dimension l, then there are exactly 2^l database "classes" (after "shrinking"), and the result follows (in fact, it is not hard to show that $l \geq n-1$).

2.8.1.1 Proofs

Notation. In the following proofs, we let r_X, r_Y be random variables denoting the receiver's and the sender's parts of the randomness respectively. r_x, r_y will typically denote specific values of r_X, r_Y. I_X, I_Y denote random variables for the receiver's and the sender's inputs respectively. Unless stated otherwise, they are assumed to be uniform over the respective domains. The random variable $M = (m_X, m_Y)$ denotes the communication transcript of a

[6] If needed, we "shrink" the database domain so that condition 3 is satisfied (in particular, for all $x \neq x'$, $\exists y | f(x,y) \neq f(x',y)$). An example in which such shrinking is needed can be obtained by taking $m = 2, n = 3$. Then the y-vectors span a subspace of $\mathbb{F}_2^3$ of dimension 2, so even the set of all pairs $(y, \mathrm{IP}_{x,y})$ for $x \in \mathbb{F}_2^3$, does not determine x.

protocol execution. When clear from the context, we sometimes abuse notation, and replace the expression "$V = v$", where V is some random variable by "v". We will need some tools from information theory. One concept we need is the Shannon entropy $H(X)$ of a random variable X. For a random variable X distributed over a size-2 set, assuming one of the values with probability p, we denote $H(X)$ by $h(p)$ (the value is independent of the particular value we choose). Denote $h^{-1}(c) = min_p(h(p) = c)$ ($1 - p$ is also a preimage). We denote the conditional Shannon entropy of X given Y by $H(X|Y)$. See [CT06] for definitions and properties of these measures. Also, we directly use terminology and tools as appearing in [WW10], Appendix B.1. One property of H we need not explicitly appearing there is the following chain rule

Fact 1. $H(X|Y) = H(X,Y) - H(Y)$.

Proof. (of Theorem 2.8.1) To prove the first part, assume for the sake of contradiction that $|M_Y| < |Y|$. Fix some receiver input $x \in X$ and $(r_x, r_y) \in$ *support*$(\mathcal{D})$. By the pigeonhole principle, there exist some $y_1 \neq y_2$, such that the sender-side communication, $m_y \in M_Y$, on (r_x, r_y, x, y_1) and on (r_x, r_y, x, y_2) is the same. Since (r_x, x) in both executions are also the same, the entire communication transcript $m = (m_x, m_y)$ in these executions is the same (by induction on the round number). By the assumption on f, there exists some x', such that $f(x', y_1) \neq f(x', y_2)$. If this holds for x itself then we immediately get a contradiction to correctness; the rest of the proof assumes $x' \neq x$. By receiver's privacy, there exists some r'_x, such that $\Pr(r'_x, r_y) > 0$ and m is the transcript on (r'_x, r_y, x', y_1); otherwise, on input (r_y, y_1), the sender has positive probability of distinguishing x and x' (by observing m). Finally, observe that the communication on (r'_x, r_y, x', y_2) is also m. Again, this holds by induction on the round number; the receiver's message function is the same as on (r'_x, r_y, x', y_2) (same input and randomness), and the sender's messages given m_x are the same as in (r_x, r_y, x', y_2). Since the receiver's output depends only on its inputs (r'_x, x') and on the communication seen, m. And since $f(x', y_1) \neq f(x', y_2)$, then the receiver's output is wrong with non-zero probability (at least $\Pr(r'_x, r_y) > 0$) for at least one of the two inputs, contradicting the perfect correctness of the protocol.

Similarly, to prove the second part, assume for contradiction that the condition on f holds but $|M_X| < |X|$. Fix some $(r_x, r_y) \in$ *support*$(\mathcal{D})$ and some sender input y. Then, by the pigeonhole principle, there exist some $x_1 \neq x_2$, such that receiver-side communication, $m_x \in M_X$, on (r_x, r_y, x_1, y) and on (r_x, r_y, x_2, y) is the same. As before, this implies that the entire

communication $m = (m_x, m_y)$ is the same (as the sender's input (r_y, y) is the same in both cases). By the condition on f, there exists some y' such that $f(x_1, y') = f(x_1, y)$ and $f(x_2, y') \neq f(x_2, y)$, or vice versa. Assume, without loss of generality, that the first case holds. By sender's privacy, the communication is m on (r_x, r'_y, x_1, y'), for some r'_y such that $\Pr(r_x, r'_y) > 0$. Otherwise, on input (r_x, x_1), the receiver has positive probability of distinguishing y and y' (by observing m), although it should have no distinguishing advantage since its output is the same in both cases. Now, observe that the communication transcript is m on (r_x, r'_y, x_2, y') as well. This is so, since the receiver-side response function is the same as in the case of (r_x, r'_y, x_2, y) (since its inputs are the same), and the sender-side responses to m_x are as in (r_x, r'_y, x_1, y') (where it has the same inputs, so it has the same response function).

Since the receiver's output depends only on its input, randomness and the communication, it equals some value v with non-zero probabilities (at least $\Pr(r_x, r'_y) > 0$ and at least $\Pr(r_x, r''_y) > 0$, respectively), on inputs (x_2, y) and (x_2, y'). By the choice of y, y', we have $f(x_2, y') \neq f(x_2, y'')$, contradicting the perfect correctness of the protocol. □

The starting point of our proofs of Theorem 2.8.2 and Theorem 2.8.3 is Lemma 2 in [WW10], and we use similar, information-theory based, techniques.

Proof. (of Theorem 2.8.2) In this direction, the communication bound is almost a direct corollary from Lemma 2 in [WW10], (a special case we will use is cited here for completeness).

Lemma 2. *[WW10] Let $f : X \times Y \to \{0, 1\}$ be a sender-receiver functionality, which is non-trivial in the sense that for all $y \neq y' \in Y$, there exists x such that $f(x, y) \neq f(x, y')$. Then, every statistically ϵ-secure protocol with preprocessing in the semi-honest model for f satisfies that for all $x \in X$,*

$$H(I_Y | r_Y, M, I_X = x) \leq (3|X| - 2)(\epsilon + h(\epsilon)).$$

Intuitively, the entropy bound implies a bound on $|M_Y|$, since only messages sent by the sender reveal information about the value of I_Y. Now, if $H(I_X | r_X, M, I_Y = y)$ is almost 0, it implies that the sender revealed almost

n bits. More formally, for all $x \in X$ we have:

$$\begin{aligned} H(I_Y|r_Y, M, I_X = x) &\geq H(I_Y|r_Y, r_X, M, I_X = x) \\ &= H(I_Y|r_Y, r_X, I_X = x, m_Y) \\ &= \sum_{r_y, r_x} \Pr(r_X = r_x, r_Y = r_y) \\ &\qquad \cdot H(I_Y|m_Y, I_X = x, r_X = r_x, r_Y = r_y) \\ &= \sum_{r_y, r_x} \Pr(r_X = r_x, r_Y = r_y) H(I_Y|m_Y(I_Y, m_x, r_y)) \\ &= \sum_{r_y, r_x} \Pr(r_X = r_x, r_Y = r_y)(H(I_Y) - H(m_Y)) \\ &\geq \sum_{r_y, r_x} \Pr(r_X = r_x, r_Y = r_y) \\ &\qquad \cdot (H(I_Y) - |m_Y|) \;=\; m - |m_Y|. \end{aligned}$$

Let us explain some of the transitions. The first follows from [WW10, Appendix B1],equation (B.3). The second transition follows since m_X is determined by x, r_X (so it adds no information, and may be removed). and the third by [WW10, Appendix B1],equation (B.5). The fourth transition stresses that m_Y depends on I_Y, and other information, which is completely fixed at that point (including m_x, which is determined by I_Y, x, r_x). The following transitions rely on Fact 1, and that m_Y is determined by I_Y (when the rest of the parameters are fixed). Finally, the last equality follows from $H(m_Y) \leq \log |M_Y|$.

Combining the lemma with the above calculation, we get that

$$|m_Y| \geq m - 3|X|(\epsilon + h(\epsilon)).$$

Next, observe that the above inequality likewise holds for any restriction $f'(x, y)$ of f to $X' \times Y$, where $X' \subseteq X$ satisfying the non-triviality property. Since any secure protocol for f is a secure protocol for any restriction f' of it (with the same security parameter), a bound for f' as above implies the same bound for f. Therefore, if there exists some $X' \subseteq X$ of size $c \cdot m$, satisfying the pre-requirement of our theorem, then applying the bound for

f's restriction $f' : X' \to Y$, we get that an ϵ-secure protocol for f satisfies

$$|m_X| \geq m(1 - 3c(\epsilon + h(\epsilon))).$$

Clearly, there indeed exists a small enough ϵ, depending only on c, such that the right-hand side is $\Omega(m)$ (and can in fact be made arbitrarily close to m, by further decreasing ϵ). □

Proof. (of Theorem 2.8.3.) The proof of this direction also closely follows the steps of the proof of [WW10, Lemma 2], with proper modifications (and is slightly more technically involved). We adopt [WW10]'s notation $P(T)$ as the distribution of a random variable T. We fix some ϵ-secure protocol Π for f, and show that ϵ can indeed be fixed to a small enough value, depending only on the c_i's, so that the receiver must send $\Omega(n)$ bits. Let Z denote the receiver's output. For convenience, we define $Z(x, r_x, M) = \$$ ("out of range" value) if the two values (r_x, M) are not possible for receiver's input x. Fix some value y of Y. By the security definition, there exists a simulator S, such that $P(I_X, r_X, M|I_Y = y)$ is ϵ-close to $P(I_X, S(X, f(X, y)))$. We conclude that for all $y \neq y'$,

$$\Delta(P(I_X, r_X, M|I_Y = y), P(I_X, r_X, M|I_Y = y')) \leq 1 - c_2 + 2c_2\epsilon. \quad (2.1)$$

The reason is that $S_= = \{x|f(x, y) = f(x, y')\}$ is of size $c_2|X|$ (by assumption). By ϵ-security, $\Delta(\Pr(I_X, r_X, M|X \in S_=, I_Y = y), \Pr(I_X, r_X, M|X \in S_=, I_Y = y)) \leq \epsilon$, and inequality (2.1) follows. For a given $y \in Y$, let $T_y(M, r_X)$ be an algorithm, "approximating" the output Z of the receiver given r_x, m, by setting it to b, such that $\Pr(Z(I_X, r_X, M) = b|r_X = r_x, M = m) \geq \Pr(Z(I_X, r_X, M) = 1 - b|r_X = r_x, M = m)$. We denote the output of $T_y(M, r_X)$, by the random variable Z'. Clearly, $I_X \leftrightarrow (r_X, M) \leftrightarrow Z'$ since, given r_X, M, the r.v. Z' is independent of I_X. Thus, for all $y \in Y$,

$$H(f(I_X, y)|r_X, M, Y = y) \leq H(f(I_X, y)|Z', Y = y), \quad (2.2)$$

using properties of Markov chains (specifically [WW10, Appendix B1,(B6)]). The main useful observation, based on sender's privacy, is that Z' is a good approximation for Z.

Claim 8. *For all $y \in Y$, we have (where c' is introduced by convenience),*

$$\Delta(P(Z|I_Y = y), P(Z'|I_Y = y)) \leq \frac{3 + h^{-1}(c_1)}{h^{-1}(c_1)}\epsilon = c'\epsilon$$

Proof. Assume, for contradiction, that for some $y \in Y$, we have $\Delta(P(Z|I_Y = y), P(Z'|I_Y = y)) > \frac{3\epsilon}{h^{-1}(c_1)} = c'\epsilon$. For each value (r_x, m) of (r_X, M), let

$$\begin{aligned} X^-_{r_x,m} &= \{x | Z(x, r_x, m) \neq \$, Z(x, r_x, m) \neq Z'(r_x, m)\}, \\ X^+_{r_x,m} &= \{x | f(x, y) = Z(x, r_x, m) \neq Z'(r_x, m)\}. \end{aligned}$$

Then,

$$\begin{aligned} c'\epsilon &\leq \Delta(P(Z|I_Y = y), P(Z'|I_Y = y)) \\ &= \sum_{r_x,m} \Pr(r_x, m|I_Y = y) \Pr(I_X \in X^-_{r_x,m}|I_Y = y, r_X, M = r_x, m) \\ &\leq \sum_{r_x,m} \Pr(r_x, m|I_Y = y) \Pr(I_X \in X^+_{r_x,m}|I_Y = y, r_X, M = r_x, m) + \epsilon, \end{aligned}$$

where the last inequality is by ϵ-correctness (letting I_X be uniform over X, and fixing $I_y = y$). We conclude that

$$\begin{aligned} (c' - 1)\epsilon &\leq \sum_{r_x,m|X^+_{r_x,m} \neq \phi} \Pr[(r_X, M) = (r_x, m)|I_Y = y] \\ &= \sum_{r_x,m,x|X^+_{r_x,m} \neq \phi, x \in X} \Pr[(r_X, M) = (r_x, m)|I_y = y] \\ &\qquad \cdot \Pr[I_X = x|(r_X, M) = (r_x, m), I_y = y] \end{aligned}$$

By an averaging argument, there exist some $x \in X$, such that

$$(c' - 1)\epsilon \ \leq \sum_{r_x,m|X^+_{r_x,m} \neq \{x\}} \Pr(r_x, m|I_Y = y, I_X = x). \tag{2.3}$$

In particular, by definition of Z', it follows that for each (r_x, m) that contributes to the sum above, there exists $x' \neq x$, such that $Z(x', r_X, M) = f(x, y)$. Fix this x. We will need the following observation.

Claim 9. *Fix some $x \in X, y \in Y$. Consider a probability distribution P' on some set $X' \subseteq X \setminus \{x\}$. There exists some y', such that $f(x, y) = f(x, y')$, and*

$$\Pr(f(I_x, y) \neq f(I_x, y')) \ \geq \ h^{-1}(c_1),$$

where I_x is chosen according to P'.

Proof. For $x' \in X', y' \in Y'$, define $\delta_{x',y'}$ to be 1 iff $f(x, y') \neq f(x', y)$. By assumption $E_{y'|f(x,y)=f(x,y')}(\delta_{x',y'}) \geq h^{-1}(c_1)$. Thus,

$$\begin{aligned} h^{-1}(c_1) &\leq \sum_{x' \in X'} E_{y'|f(x,y)=f(x,y')}(P'(x')\delta_{x',y'}) \\ &= E_{y'|f(x,y)=f(x,y')}(\sum_{x' \in X'} P'(x')\delta_{x',y'}). \end{aligned}$$

Note that the argument of the $E_{y'}$, for fixed y', is exactly the probability we are trying to maximize, for that value of y'. Now, since the expected value of that probability (over the various y''s) is $\geq h^{-1}(c_1)$, there must exist some y', for which this probability is at least $h^{-1}(c_1)$, as required. □

Next, we present an adversary attacking the receiver, that on input x, distinguishes between y and y' with probability $> 2\epsilon$, contradicting the security requirement of the protocol. We will conclude that the contradiction assumption must be false, and Claim 8 indeed holds. The (semi-honest) adversary A acts as follows:

- On inputs x, r_x, execute the protocol to obtain a communication transcript m. Check whether $X^+_{r_x,m} \setminus \{x\} \neq \phi$.
  - If so, let $x_{r_x,m}$ be some fixed, arbitrary, value in $X^+_{r_x,m} \setminus \{x\}$. Output $(x_{r_x,m}, r_x, m, Z(x_{r_x,m}, r_x, m))$.
  - Otherwise, output \$.

Let $Good = \{r_x, m | Pr(r_x, m|x, y) > 0, X^+_{r_x,m} \neq \{x\}\}$. It follows from Equation 2.3, that the set *Good* is "hit" by A on inputs (x, y) with probability $\geq (c'-1)\epsilon$. In this case, it outputs some $(x_{r_x,m}, r_x, m, f(x_{r_x,m}, y))$. Let us set y' to a value guaranteed to exist for x, y, and P' is over the $x_{r_x,m}$'s satisfying $X^+_{r_x,m} \neq \{x\}$, where $\Pr(x_{r_x,m}) = P(r_x, m|I_x = x, I_y = y)$ by Claim 9. Let *Bad* denote the set of $x_{r_x,m}$'s in *support*(P'), such that $f(x_{r_m,x}, y) \neq f(x_{r_m,x}, y')$. The probability that $x_{r_x,m}$ in A's output $(x_{r_x,m}, r_x, m, f(x_{r_x,m}, y))$ is in *Bad* if $I_Y = y$ is at least $h^{-1}(c_1)(c'-1)\epsilon$. For the case of $I_Y = y'$, the probability that $(x_{r_x,m}, r_x, m, Z(x_{r_x,m}, r_x, m)) = (x_{r_x,m}, r_x, m, f(x_{r_x,m}, y))$, and $x_{r_x,m} \in Bad$ is at most ϵ, since if $x_{r_x,m} \in Bad$, and $Z = f(x_{r_x,m}, y')$ (the latter has probability at least $1 - \epsilon$, by correctness), the Z part in the output tuple must differ, by the choice of y'. This yields a distance $> (h^{-1}(c_1)(c'-1)-1)\epsilon = 2\epsilon$ between the receiver's views on (x, y) and (x, y'), contradicting ϵ-security. □

We conclude that for all $y \in Y$

$$\begin{aligned} H(f(I_X,y)|Z',Y=y) &\leq H(f(I_X,y)|Z,Y=y) + (c'+1)\epsilon + h((c'+1)\epsilon) \\ &\leq (c'+1)\epsilon + h((c'+1)\epsilon) \end{aligned}$$

As $\Delta(Z,Z') \leq c'\epsilon$, we have $\Delta((f(I_X,y),Z),(f(I_X,y),Z')) \leq c'\epsilon$, thus the first transition holds by [WW10, Lemma B.1]. The second transition is by using $\Delta(f(I_X,y)|Z) \leq \epsilon$ (follows from correctness), and applying [WW10, Appendix B1,(B.9)]. Finally, combining with Equation 2.2, we get that for all $y \in Y$

$$H(f(I_X,y)|r_X,M,I_Y=y) \leq (c'+1)\epsilon + h((c'+1)\epsilon)$$

Finally, let $Y' = \{y'_1,\ldots,y'_t\}$. Since I_X is determined by $\{(y,f(I_X,y))\}_{y\in Y'}$ we have

$$\begin{aligned} H(I_X|r_X,M,I_Y=y'_1) &\leq H(f(I_X,y'_1),\ldots,f(I_X,y'_t)|r_X,M,I_Y=y'_1) \\ &\leq \sum_{i=2}^{t} H(f(I_X,y'_i)|r_X,M,I_Y=y'_1) \\ &\quad + H(f(I_X,y'_1)|r_X,M,I_Y=y'_1) \\ &\leq (1-c+\epsilon)(1+c_2)n + (c'+1)\epsilon + h((c'+1)\epsilon). \end{aligned} \tag{2.4}$$

Here the second inequality is by standard entropy-related arguments (specifically, follows by combining (B.2) and (B.3) in [WW10, Appendix B1]). Clearly, for small enough ϵ, depending only on the c_i's, the right-hand side of the above inequality is $c'n$, where $c' < 1$ is some constant (independent of m,n).

Finally, we deduce a lower bound on $H(I_X|r_X,M,I_Y=y'_1)$ in terms of $|m_X|$ ($|m_X|$ denotes the worst-case sender-side communication complexity).

$$H(I_X|r_X,M,I_Y=y'_1) \;\geq\; n - |m_X|. \tag{2.5}$$

The proof of this bound is similarly to that of Theorem 2.8.2.

Combining the bounds on $H(I_X|r_X,m_X)$ obtained from inequalities (2.4) and (2.5), we get

$$|m_X| \;\geq\; (1-(1-c+\epsilon)(1+c_2))n - (c'+1)\epsilon - h((c'+1)\epsilon),$$

which satisfies the required condition, for small enough ϵ. □

2.8.2 A Link Between Storage Complexity And Private Information Retrieval

Even though our generic protocols are communication-efficient, the amount of correlated randomness that they employ is exponential in the input length. The following theorem shows that obtaining a significant improvement on the amount of randomness for every two-party functionality (or even just sender-receiver functionalities) would imply a breakthrough result on information-theoretic PIR [CGKS95]. This holds even if requiring only semi-honest security and without any computational restrictions on the parties. A similar result for honest-majority MPC in the plain model was presented in [IK04].

Formally, we prove the following theorem:

Theorem 2.8.4. *Assume that there exists a semi-honest statistically secure protocol in the preprocessing model for every sender-receiver functionality* $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ *with correlated randomness complexity* $r(n)$ *(i.e., where* $r_X, r_Y \in \{0,1\}^{r(n)}$*). Then, there exists 3-server, interactive PIR protocol, with communication complexity* $r(\log N) + O(\log N)$*, where* N *is the database size.*

Proof. Let $\mathcal{U}$ denote the user in the PIR protocol, with input $i \in [N]$, and let S_1, S_2, S_3 denote the 3 servers, all holding an identical database $D \in \{0,1\}^N$.

1. $\mathcal{U}$ picks at random $a, b \in [N]$ subject to $a + b = i \bmod N$ (i.e., a, b form an additive secret-sharing of i), and a random bit σ. It sends a, σ to server S_1 and b to server S_2.

2. S_1, S_2 execute the guaranteed sender-receiver protocol for the function

$$f_D((a,\sigma), b) = D_{a+b} \oplus \sigma.$$

   For this, server S_3 first generates correlated randomness (r_X, r_Y) as required by the guaranteed protocol for f_D (note that all servers know D; also note that the correlated randomness can in fact be generated in parallel to Step 1).

3. S_2 (the receiver in the protocol) sends the output $\mathcal{U}$ who masks it with σ to recover D_i.

It is easy to verify that the communication complexity of the constructed PIR protocol is as promised. To argue $\mathcal{U}$'s privacy, observe that the input to

the sender-receiver protocol provides 1-privacy (since it is a 1-private secret sharing of i), the output of this protocol also maintains the privacy since it is masked by a random bit, and the view of each of S_1, S_2 is the view of the corresponding party in the sender-receiver protocol, which also maintains 1-privacy. Finally, note that S_3 only sends messages in this protocol. □

The above theorem implies, in particular, that protocols in the preprocessing model with correlated randomness of polynomial length would imply information-theoretic PIR with a constant number of servers and polylogarithmic communication complexity, a major (and quite unexpected) positive resolution of a longstanding open problem [CGKS95].

We note that [WW10, DPSZ12] contain *unconditional* lower bounds on the amount of correlated randomness required for computing functions in the preprocessing model, however these lower bounds are at most linear in the input length.

2.9 Perfect Correctness in the Plain Model

2.9.1 From the Preprocessing Model to the Plain Model

In Section 2.9.2, we show that the impossibility of perfectly sound zero-knowledge proofs for NP carries over to the preprocessing model.

This implies that some sender-receiver functionalities cannot be securely realized with perfect correctness in the plain model. In this section, we show that the class of functionalities that can be securely realized with perfect correctness is actually quite rich. To the best of our knowledge, this important fundamental question has been neglected in the literature so far.

We present a general transformation from perfect sender-receiver protocols in the preprocessing model, to protocols with perfect correctness in the plain model. This is possible for functionalities for which the preprocessing can be realized in the plain model with perfect privacy (and computational correctness): the main conceptual contribution is to show how we can turn perfect privacy into perfect correctness by using an offline/online protocol.

We start by showing how to do this against semi-honest parties, and then we will argue that this can be boosted using a GMW-like compiler that preserves the perfect privacy of the semi-honest protocol.

The high level idea of this transformation is to use the "reversibility" of correlated randomness for turning perfect privacy in the plain model into

perfect correctness in the plain model. Concretely, let π be a perfectly secure protocol for f in the preprocessing model. Using standard techniques (a combination of perfectly private OT protocols [NP01, AIR01] with an information-theoretic variant of the garbled circuit technique [Yao82, Kil88]), one can get a *perfectly private* protocol π' (with unbounded simulation) for all sender-receiver functionalities in NC^1. We then use π', with the sender in π playing the role of the receiver in π', for generating the correlated randomness required by π. In this subprotocol the receiver picks its randomness r_x from the correct marginal distribution and the sender obtains as its output from π' a random input r_y sampled from the conditional distribution defined by r_x. This subprotocol prevents a malicious sender from learning *any* information about r_x other than what follows from r_y. Running π on top of the correlated randomness (r_x, r_y) generated by the subprotocol gives a perfectly correct protocol for f.

The approach described so far only guarantees security against semi-honest parties (in addition to perfect correctness against a malicious sender); however, using a GMW-style compiler we can get (computational) security against malicious parties while maintaining perfect correctness against a malicious unbounded sender.

Formally, let $\mathcal{D}$ be a distribution over $R_1 \times R_2$ such that all probabilities in the support of $\mathcal{D}$ are rational; i.e., there exists a positive integer M s.t., for all $(r_1, r_2) \in R_1 \times R_2$, there exists an integer p such that $\Pr[\mathcal{D} = (r_1, r_2)] = \frac{p}{M}$.

Let $\mathcal{D}_{r_1}$ be a family of distributions over R_2 such that the two distributions $\{(r_1, r_2) : (r_1, r_2) \leftarrow \mathcal{D}\}$ and $\{(r_1, r_2') : (r_1, r_2) \leftarrow \mathcal{D}, r_2' \leftarrow \mathcal{D}_{r_1}\}$ are identically distributed.

Let $\mathsf{Pre}^{\mathcal{D}} : R_1 \to R_2$ be a randomized functionality[7] that, on input r_1 from party R outputs r_2, sampled according to $\mathcal{D}_{r_1}$ to S (if r_1 is not in the support of $\mathcal{D}$, the function outputs $\perp$).

Theorem 2.9.1. *Let f be a sender-receiver functionality that admits a perfectly secure protocol π_{online}, in the presence of preprocessing $\mathcal{D}$, where all probabilities in support($\mathcal{D}$) are rational. Let π_{pre} be a protocol that realizes $\mathsf{Pre}^{\mathcal{D}}$ which is semi-honest secure and perfectly private against malicious S. Then, it is possible to securely compute f with semi-honest security and* perfect correctness.

[7]As discussed in Section 2.2, our computational model allows perfect sampling from the uniform distribution over $[m]$, for all integers m.

Proof. Let $\pi_{\mathsf{online}} = (R_{\mathsf{online}}, S_{\mathsf{online}})$ and $\pi_{\mathsf{pre}} = (R_{\mathsf{pre}}, S_{\mathsf{pre}})$ be protocols, as assumed by the theorem (where R_{pre} is the party that provides input in π_{pre} and S_{pre} is the party that gets the output). The final protocol π for f works as follows: R samples (r_1, r_2) from $\mathcal{D}$; then, R and S run protocol π_{pre} where R inputs r_1 and S receives r'_2; finally, they run protocol π_{online} using the randomness (r_1, r'_2).

Protocol π is trivially secure for semi-honest parties. The protocol is also *perfectly secure* against a malicious sender (with unbounded simulation), because both π_{pre} and π_{online} have this property; note that, because R has no output in π_{pre}, perfect security against malicious sender and perfect privacy are the same; hence, security follows by applying standard composition theorems (we stress that simulation here is not necessarily efficient). □

Theorem 2.9.2. *Assuming one-way permutations exist, the result of Theorem 2.9.1 holds with security against malicious parties.*

Proof. (Sketch) We apply a GMW-like compiler, with the only difference that the receiver uses only perfectly private primitives: perfectly hiding commitments [Blu82] and perfect zero-knowledge arguments of knowledge for NP [NOVY98], which can be instantiated using one-way permutations. See [Gol04, Section 7.4] for details. □

Applications of the above theorem are discussed in Section 2.9.3 below.

2.9.2 Perfect Correctness is Not Always Possible

Theorem 2.9.3. *If $NP \not\subseteq BPP$, there exists a sender-receiver functionality that cannot be efficiently computed with semi-honest security and perfect correctness (even in the preprocessing model).*

Proof. Let R be any NP relation and $f_R(x, w)$ be the sender-receiver functionality, where the receiver gets output $R(x, w)$ (i.e., the zero-knowledge functionality for the language $L = \{x | \exists w : R(x, w) = 1\}$). Assume, for the sake of contradiction, that it is possible to compute f_R with semi-honest security and perfect correctness. Then, given an instance x, we can decide the language L by running the semi-honest simulator $\mathcal{S}$ on input x and output "$x \in L$" if the zero-knowledge verifier accepts the simulated proof, or "$x \notin L$" otherwise. If the zero-knowledge verifier accepts a simulated proof for an instance $x \notin L$, then there exists malicious prover strategy that makes

the verifier accept a false statement with nonzero probability, contradicting perfect soundness (namely, run the simulator on $x \notin L$ — with some non-zero probability the verifier's messages to the prover are equal to the messages in the simulated execution, and the verifier accepts the simulated proof, contradicting perfect soundness). On the other hand, if the verifier rejects every simulated proof for $x \notin L$, this gives a BPP algorithm that decides L, thus reaching a contradiction with the assumption of the theorem. □

2.9.3 Applications of Perfect Correctness

Every Sender-Receiver Functionality with Polynomial Size Input Space: We have proved in Theorem 2.3.3 that the protocol of Figure 2.5 securely evaluates every sender-receiver functionality f with perfect security. It is therefore left to argue that we can securely implement the functionality $\mathsf{Pre}^{\mathcal{D}}$ with perfect privacy. We will do so using a $\binom{n}{1}$-OT protocol[8] with perfect privacy, such as [AIR01, NP01].

By the fact all the probabilities in *support*$(\mathcal{D})$ are rational, we have that there exists an integer M s.t., all probabilities are multiples of $1/M$. Therefore we can simulate the preprocessing phase using an $\binom{M}{1}$-OT as follows: S, acting as the chooser in the $\binom{M}{1}$-OT will input a random $c \in_R [M]$. R chooses a random sample (r_1, r_2) from $\mathcal{D}$, and then computes values $r_2^1, \ldots, r_2^M \in R_2$ s.t., the two distributions $\{r_2^c : c \in [M]\}$ and $\{r_2 : r_2 \leftarrow \mathcal{D}_{r_1}\}$ are identically distributed. Then R, acting as the transmitter in the $\binom{M}{1}$-OT will input the messages $r_2^1, \ldots, r_2^M$ in random order, and S will retrieve the value r_2^c.

For the sake of concreteness, to implement the preprocessing of the protocol of Figure 2.5 the receiver will choose a random $r \in X$ and a random $P_1, \ldots, P_{|X|}$ with $P_i : Y \to Y$ and will compute the permuted truth table $A_{x,P_x(y)} = f(x,y)$. Then R will compute a set of lists of permutations

$$(Q_1^1, \ldots, Q_{|X|}^1), \ldots, (Q_1^M, \ldots, Q_{|X|}^M)$$

such that,$A_{x,Q_{x+r}(y)} = f(x,y)$.

And the output of S is distributed according to $\mathcal{D}$. Note that the value of M is smaller the "more injective" f is. In the extreme case where for all $(x,y) \in X \times Y$ and $(x',y') \in X \times Y$ with $(x,y) \neq (x',y')$ it holds that $f(x,y) \neq f(x',y')$, there is only one set of possible permutations and $M = 1$.

[8]To avoid confusion, we will refer to the party inputting messages $m_1, \ldots, m_n$ to the $\binom{n}{1}$-OT as the transmitter or T, and the party inputting a choice $c \in [n]$ and receiving m_c as the chooser or C.

Corollary 1. *Every sender-receiver functionality with polynomial size input-space can be securely computed with perfect correctness in the plain model.*

Some Sender-Receiver Functionalities with Exponential Size Input Space: In Section 2.5 we have seen functions with exponential input space that admit perfectly secure sender-receiver protocols in the preprocessing model. This includes protocols for equality, oblivious polynomial evaluation and private set intersection.

All this protocols share the feature that the preprocessing can be computed by an NC1 circuits (the preprocessing only consists of modular additions and multiplications), and we can therefore use the information-theoretic version of Yao's garbled circuits technique [IK02] together with a perfectly private $\binom{2}{1}$-OT to implement the preprocessing, to securely implement $\mathsf{Pre}^{\mathcal{D}}$ with perfect privacy.

More in details, we need that the preprocessing is implementable by an NC1 circuit $\mathsf{Pre}^{\mathcal{D}} : R_1 \times [m] \to R_2$ s.t., the distributions $\{(r_1, r_2) : (r_1, r_2) \leftarrow \mathcal{D}\}$ and $\{(r_1, r_2') : (r_1, r_2) \leftarrow \mathcal{D}, r_2' \leftarrow \mathsf{Pre}^{\mathcal{D}}(r_1, U_{[m]})\}$ are identically distributed. The circuit computing Pre uses randomness ρ from $[m]$, so the garbled circuit will get inputs ρ_1, ρ_2 from the receiver and the sender respectively and compute $\mathsf{Pre}^{\mathcal{D}}$ with randomness $(\rho_1 + \rho_2 \mod m)$ if $\rho_2 \in [m]$ or $\perp$ otherwise.

Corollary 2. *Let f be a sender-receiver functionality that admits a perfectly secure protocol π in the presence of a preprocessing $\mathcal{D}$. If $\mathsf{Pre}^{\mathcal{D}}$ can be computed with an NC1 circuit, f admits a computationally secure protocol with perfect correctness in the plain model.*

In particular equality, oblivious polynomial evaluation and set intersection can be computed with perfect correctness.

For the sake of concreteness, we give an explicit construction of the preprocessing for the equality protocol from Section 2.1.4. The distribution of the preprocessing for computing equality is $((r, s = P(r)), P) \leftarrow \mathcal{D}_{eq}$ where P is a pair-wise independent permutation.

A way to implement a pair-wise independent permutation is using the function $P(x) = ax + b \mod p$ with $(a, b) \in [p]$, for some prime $[p]$. Then the circuit $\mathsf{Pre}_{eq}^{\mathcal{D}}$ gets input $(r, s, \rho_1) \in [p]^3$ from R and $\rho_2 \in [p]$ from S. The circuit computes $a = \rho_1 + \rho_2 \mod p$ and $b = s - ar$ and outputs (a, b) to S.

2.10 Future Directions

2.10.1 Combining Multiple Sources — the Commodity Based Model

In this chapter we mainly viewed the correlated randomness in the preprocessing step as being provided by a trusted offline "dealer," or alternatively generated by the parties by running an interactive protocol in an offline phase. An interesting alternative suggested by Beaver [Bea97] is to use multiple independent "commodity servers" who are unaware of each other. For this approach to be useful, the protocols must offer robustness against some fraction of corrupted servers, even if those can collude with other parties.

We would like a way to combine preprocessed randomness from several sources in a way similar to what Beaver does for OTs. A general approach for combining correlated sources from multiple servers is to use each source to emulate a "virtual server" in an honest-majority (unconditionally secure) MPC protocol. The details of this are still to be figured out.

2.10.2 Generalized OT extensions

While works based directly on OT as [Kil88] are less efficient communication-wise than our protocols, basing secure computation specifically on the preprocessing of OTs have a great advantage: OT extension.

OT extension is a way of "stretching" a few secure OTs to polynomially many (computational) OTs, in a way analogous to a PRG stretching the randomness of a random seed value. It was introduced by Beaver in [Mil96], proving the following:

Theorem 2.10.1. *Let k be a computational security parameter. If one-way functions exist, then for any constant $c > 1$ there exists a protocol for reducing k^c oblivious transfers to k oblivious transfers.*

Later developments [Bon03, Nie07] have given much more efficient constructions for doing OT extensions using slightly stronger primitives, so now it is an attractive way of producing large amounts correlated randomness.

The correlation required for our protocols are more specific. Are there ways of "stretching" correlated randomness in general?

Such stretching would only give computational security, but optimally efficient online protocols.

2.10.3 More efficient protocols

We have shown truly efficient protocols for a few tasks. And shown that in general we cannot hope to achieve efficient preprocessing for general tasks. It would however be interesting to discover even more functionalities that can be computed (perfectly) securely with optimal communication given a polynomial amount of preprocessing as protocols of this kind seems to be highly practically applicable.

And even better would be general protocols with small communication complexity and preprocessing complexity proportional to the size of the circuit computing the function. (Or some other compact function representation).

2.10.4 More Restricted Communication Models

We have explored the possibilities of doing MPC with preprocessing with very restricted communication, just two messages of length proportional to the inputs proves to be enough to compute any functionality.

For general security of even sender-receiver protocols the two-message set-up is indeed the smallest possible: If the sender does not send any message correctness cannot be achieved as the senders input cannot influence the receivers output. If the sender just sends one message, and the receiver has to be able to give the correct output, he must be able to compute the output of the function for any given receiver-input, this is more than the functionality allows, and thus not secure.

But the last case can inspire us to give a relaxed security definition allowing the adversary to learn exactly as much as he can learn from interacting by the functionality varying his own input and no more. (Note this is only interesting for more than two players.) This is known as the "private simultaneous message" (PSM) model [FKN94, IK97], in which a server receives a single message from each other player, and then computes an output. But the earlier works only consider a corrupted server. Is it possible make the model more robust by allowing corruptions also of some client players, and still guarantee some security?

He wondered if the case was really over or if he was not still working on it. He wondered what the map would look like of all the steps he had taken in his life and what word it would spell. When it was dark Quinn slept, and when it was light he ate the food and wrote in his red notebook.

Paul Auster - Ghosts

3 Information Theoretic Oblivious RAM

The contents of this chapter is the result of joint work with Ivan Damgaard and Jesper Buus Nielsen. An earlier version has been published as [DMN11].

3.1 Introduction

In many cases it is attractive to store data at an untrusted place, and only retrieve the parts of it you need. Encryption can help to ensure that the party storing the data has no idea of what he is storing, but it may still be possible to get information about the stored data by analyzing the access pattern.

A trivial solution is to access all the data every time one piece of data is needed. However, many algorithms are designed for being effective in the RAM-model, where access to any word of the memory takes constant time, and so accessing all data for every data access gives an overhead that is linear in the size of the used memory.

This poses the question: is there any way to perfectly hide which data is accessed, while paying a lower overhead cost than for the trivial solution?

Goldreich and Ostrovsky [GO96] solved this problem in a model with a secure CPU, equipped with a random oracle and small (constant size) memory.

The CPU runs a program while using a (large) RAM that is observed by the adversary. The results from [GO96] show that any program in the standard RAM model can be transformed using an "'oblivious RAM simulator" into a program for the oblivious RAM model, where the access pattern is information theoretically hidden. The amortized overhead of this transformation is polylogarithmic in the size of the memory.

Whereas it is not reasonable to assume a random oracle in a real implementation [CGH04], Goldreich and Ostrovsky point out that one can replace it by a pseudo-random function (PRF) that only depends on a short key stored by the CPU. This way, one obtains an oblivious RAM that is computationally secure. But this depends on the existence of computational hardness. Moreover, in applications related to secure multiparty computation (see below), the simulated CPU would need to *securely* compute the pseudo random function, which introduces a very significant overhead.

It is a natural question whether one can completely avoid the random oracle/PRF. One obvious approach is to look for a solution that uses a very small number of random bits. But this is not possible: in this chapter we show a lower bound on the number of secret random bits that an oblivious RAM simulator must use to hide the access pattern information theoretically: the number of random bits used must grow linearly with the number of memory accesses and logarithmically with the memory size. The natural alternative is to generate the random bits on the fly as you need them, and store those you need to remember in the external RAM. This assumes, of course, that the adversary observes only the access pattern and not the data written to the RAM. However, as we discuss below, there are several natural scenarios where this can be assumed, including applications for secure multiparty computation. The advantage of this approach is that it only assumes a source that delivers random bits on demand, which is clearly a more reasonable assumption than a random oracle and much easier to implement in a secure multiparty computation setting.

Using this approach, we progress like [GO96] and first construct an oblivious RAM simulator that can make an access in amortized time $O(\sqrt{N} \log^2 N)$ where N is the size of the memory provided, and next improve that solution ending up with only an overhead of $O(\log^3 N)$ per access. The result remains the same, even if the adversary is told which operations the program is executing, and even if the simulator has no internal memory.

3.2 Related work

Concurrent and independently of this work [Ajt10] Ajtai also deals with oblivious RAM and unconditional security. His result solves essentially the same problem as we are dealing with, but using a completely different technique that does not seem to lead to a zero-error solution. Simplistically speaking, Ajtai's approach is to follow the algorithm of Goldreich and Ostrovsky, but simulating the random oracle by ordinary coin flipping, saving the result of queries in a datastructure stored in the external server's memory. For this modification to work, a certain "error event" must not occur, and Ajtai, solving a rather complicated combinatorial problem, shows that the bad event happens with probability only $n^{-\log n}$.

In comparison, our construction is a more "direct fit" to the model, second it has no failure probability. Finally, we also prove a lower bound on the randomness complexity of any solution.

Shi, Stephanov and Li gives an oblivious RAM with good *worst case* cost in [SCSL11]. Their construction uses a tree of small oblivious RAMs. If the construction given in this chapter is plugged into their construction an information theoretically secure ORAM with amortized cost $\tilde{O}(\log(n^2))$ and worst case cost $\tilde{O}(\log(n^3))$ where the $\tilde{O}$ notation hides $\log\log$ factors.

The asymptotically best ORAM construction (with computational security) is a construction in [KLO12] with a *worst case* query overhead of $\log^2 N/\log\log N$ while using only $O(N)$ server side memory.

For practical memory sizes a client memory of $O(\sqrt{N})$ or even a small linear fraction of N can arguably be acceptable, and this can lead to more efficient constructions. If for example N bytes is 1 terabyte, $\sqrt{N}$ bytes is one megabyte which is small for most purposes. This has been the subject of a number of papers. [WS08, WSC08, GM11, GMOT12] For the purposes of multi-party computation however, every local client-side memory access have to touch all the clients memory. Therefore a larger client-side storage also implies a higher running time.

The costliest part of the oblivious RAM in most constructions (including ours) is the oblivious sorting/shuffling implemented by sorting networks. Several ways of avoiding this have been suggested. Some sort the values on the client side in smaller chunks. This though requires the client to have a non-constant memory. The [SCSL11] construction with the "trivial ORAM" used for implementing the buckets also avoids sorting. [LO13] suggests a two-server model where one server can shuffle values obliviously to the other

server. In the next chapter we show how to adapt our construction to this model.

Also the round complexity of oblivious RAMs is considered. Williams shows [Wil12] how to do *one round* oblivious RAMS, meaning one memory access can be simulated with only a single round. A variant of this result is used in [LO12] to make one-round RAM programs (in the style of Yao's garbled circuits) run with polylogarithmic overhead compared to the RAM-machine program.

See also Table 1.1

3.3 Applications

Software protection: This was the main original application of Goldreich and Ostrovsky. A tamper-proof CPU with an internal secret key and randomness could run an encrypted program stored in an untrusted memory. Now using an oblivious RAM, the observer would only learn the running time and the required memory of the program, and nothing else. Note that, while the adversary would usually be able to see the data written to RAM in such a scenario, this does not have to be the case: if the adversary is doing a side channel attack where he is timing the memory accesses to see if the program hits or misses the cache, he is exactly in a situation where only information on the access pattern leaks, and our solution would give unconditional security.

Secure multiparty computation: If secure multiparty computation is implemented by secret sharing, in the style of [CCD88] and [BGW88], each player will have a share of the inputs, and computations can be done on the shares. We can use the oblivious RAM model to structure the computation by thinking of the players as jointly implementing the ideal CPU (the client), while each memory cell in the physical RAM is represented as a value secret shared between the players. This is another a case where an adversary can observe the access pattern (since the protocol must reveal which shared values we access) but not the data. Using an oblivious RAM, we can hide the access pattern and this allows us, for instance, to do array indexing with secret indices much more efficiently than if we had used the standard approach of writing the desired computation as an arithmetic circuit.

Note that players can generate random shared values very efficiently, so that our solution fits this application much better than an approach where a PRF is used and must be securely computed by the players.

Cloud computing: It is becoming more and more common to outsource data storage to untrusted third parties. And even if the user keeps all data encrypted, analysis of the access patterns can still reveal information about the stored data. Oblivious RAM eliminates this problem, leaving the untrusted party only with knowledge about the size of the stored data, and the access frequency. This type of application was also already proposed by Goldreich and Ostrovsky in [GO96].

3.4 The model

An oblivious RAM simulator is a functionality (some times called the client) that implements the interface of a RAM, using auxiliary access to another RAM, the physical RAM (sometimes called the server). We say that such a simulation securely implements an oblivious RAM, if, for any two access patterns to the simulated RAM, the respective access patterns that the simulation makes to the physical RAM are indistinguishable.

To simplify notation we assume that the interface of a RAM has only one operation, which writes a new value to a memory position and returns the previous value.

We model that the identity of the instructions performed by the simulator leak, but not the operands. We assume that the indices of the memory positions updated by the simulation leak, but not the values being retrieved and stored.

A RAM is modeled as an interactive machine behaving as follows:

1. Set $\mathcal{N}[i] = 0$ for $i \in \mathbb{Z}_N$.
2. On each subsequent input $(\texttt{update}, i, v)$ on the input tape, where $i \in \mathbb{Z}_N$ and $v \in \mathbb{Z}_q$, let $w = \mathcal{N}[i]$, set $\mathcal{N}[i] = v$ and output w on the output tape.

A sequence of update operations is a list

$$U = ((\texttt{update}, i_1, v_1), \ldots, (\texttt{update}, i_\ell, v_\ell))$$

with $i_j \in \mathbb{Z}_N$ and $v_j \in \mathbb{Z}_q$. We let $|U| = \ell$ and we use $\mathrm{IO}_{\mathrm{RAM}}(U) = ((\texttt{update}, i_1, v_1), w_1, \ldots, (\texttt{update}, i_\ell, v_\ell), w_\ell)$ to denote the sequence of inputs and outputs on the input tape and the output tape when the interactive machine is run on the update sequence U.

Formally, an ORAM simulator is a tuple $\mathcal{S} = (\mathcal{C}, N, M, q)$, where $\mathcal{C} = (\mathcal{C}[0], \ldots, \mathcal{C}[|\mathcal{C}| - 1])$ is the finite program code where each $\mathcal{C}[j]$ is one of $(\texttt{random}, i)$, $(\texttt{const}, i, v)$, $(\texttt{+}, t, l, r)$, $(\texttt{-}, t, l, r)$, $(\texttt{*}, t, l, r)$, $(\texttt{=}, t, l, r)$, $(\texttt{<}, t, l, r)$, $(\texttt{goto}, i)$ with $i, t, l, r \in \mathbb{Z}_M$ and $v \in \mathbb{Z}_q$, and $N \in \mathbb{N}$ is the size of the simulated RAM, $M \in \mathbb{N}$ is the size of the physical RAM, and $q \in \mathbb{N}$ indicates the word size of the RAMs: the simulated and the physical RAM store elements of $\mathbb{Z}_q$. We require that $q > \max(N, M)$. We denote the simulated memory by $\mathcal{N} \in \mathbb{Z}_q^N$, indexed by $\{0, 1, \ldots, N-1\} \subseteq \mathbb{Z}_q$. We denote the physical memory by $\mathcal{M} \in \mathbb{Z}_q^M$, indexed by $\{0, 1, \ldots, M - 1\} \subseteq \mathbb{Z}_q$.

The simulation can be interpreted as an interactive machine. It has a register $\mathbf{c}$ and a special *leakage tape*, which we use for modeling purposes. The machine behaves as follows:

1. Set $\mathcal{M}[i] = 0$ for $i \in \mathbb{Z}_M$.

2. Set $\mathbf{c} = 0$.

3. On each subsequent input $(\texttt{update}, i, v)$ on the input tape, where $i \in \mathbb{Z}_M$ and $v \in \mathbb{Z}_q$, proceed as follows:

   (a) Set $\mathcal{M}[0] = i$ and $\mathcal{M}[1] = v$.

   (b) If $\mathbf{c} > |\mathcal{C}|$, then output $\mathcal{M}[2]$ on the output tape, append $(\texttt{return})$ to the leakage tape and halt. Otherwise, execute the instruction $C = \mathcal{C}[\mathbf{c}]$ as described below, let $\mathbf{c} = \mathbf{c} + 1$ and go to Step 3b. Each instruction C is executed as follows:

   - If $C = (\texttt{random}, i)$, then sample a uniformly random $r \in \mathbb{Z}_q$, set $\mathcal{M}[i] = r$ and append $(\texttt{random}, i)$ to the leakage tape.
   - If $C = (\texttt{const}, i, v)$, set $\mathcal{M}[i] = v$ and append $(\texttt{const}, i, v)$ to the leakage tape.
   - If $C = (\texttt{+}, t, l, r)$, set $\mathcal{M}[t] = \mathcal{M}[l] + \mathcal{M}[r] \bmod q$ and append $(\texttt{+}, t, l, r)$ to the leakage tape. The commands - and * are handled similarly.
   - If $C = (\texttt{=}, t, l, r)$, set $\mathcal{M}[t] = 1$ if $\mathcal{M}[l] = \mathcal{M}[r]$ and set $\mathcal{M}[t] = 0$ otherwise, and append $(\texttt{=}, t, l, r)$ to the leakage tape.

- If $C = (<, t, l, r)$, set $\mathcal{M}[t] = 1$ if $\mathcal{M}[l] < \mathcal{M}[r]$ as residues in $\{0, \ldots, q-1\}$ and set $\mathcal{M}[t] = 0$ otherwise, and append $(\texttt{<}, t, l, r)$ to the leakage tape.
- If $C = (\texttt{goto}, i)$, let $\mathbf{c} = \mathcal{M}[i]$ and append $(\texttt{goto}, i, \mathbf{c})$ to the leakage tape.

By $\mathrm{IO}_{\mathcal{S}}(U)$ we denote the random variable describing the sequence of inputs and outputs on the input and output tapes when $\mathcal{S}$ is executed as above on the update sequence U, where the randomness is taken over the values r sampled by the `random`-commands. By $\mathrm{L{\scriptsize EAK}}_{\mathcal{S}}(U)$ we denote the random variable describing the outputs on the leakage tape.

Definition 3.4.1. *We say that $\mathcal{S}$ is an ϵ-secure ORAM simulator if it for all update sequences U the statistical difference $\Delta(\mathrm{IO}_{\mathcal{S}}(U), \mathrm{IO}_{\mathrm{RAM}}(U)) \leq \epsilon$ and it holds for all update sequences U_0 and U_1 with $|U_0| = |U_1|$ that $\Delta(\mathrm{L{\scriptsize EAK}}_{\mathcal{S}}(U_0), \mathrm{L{\scriptsize EAK}}_{\mathcal{S}}(U_1)) \leq \epsilon$. We say that $\mathcal{S}$ is a perfectly secure ORAM simulator if it is a 0-secure ORAM simulator.*

3.5 Oblivious sorting and shuffling

In the following, we will need to shuffle a list of records obliviously. One way to do this, is to assign a random number to each record, and sort them according to this number. If the numbers are large enough we choose distinct numbers for each value with very high probability, and then the permutation we obtain is uniformly chosen among all permutations.

This issue is, in fact, the only source of error in our solution.

If we want to make sure we succeed, we can simply run through the records after sorting to see if all random numbers were different. If not, we choose a new random set of numbers and do another sorting. This will succeed in expected $O(1)$ attempts each taking $O(n)$ time, and so in asymptotically the same (expected) time, we can have a perfect solution.

We can sort obliviously by means of a sorting network. This can be done with $O(n \log n)$ compare-and-switch operations, but a very high constant overhead [AKS83], or more practically with a Batcher's network [Bat68] using $O(n \log^2 n)$ switches.

Each switch can be implemented with two reads and two writes to the memory, and a constant number of primitive operations. Sorting in this

way is oblivious because the accesses are fixed by the size of the data, and therefore independent of the data stored.

By storing the choice bits of each switch we can arrange according to the inverse permutation by running the elements through the switches of the sorting network in backwards order while switching in the opposite direction from before.

3.6 The square root algorithm

As a warm-up we describe in this section an algorithm implementing an oblivious RAM using memory and amortized time in $O(\sqrt{N}\log^2 N)$. The algorithm assumes access to a functionality that shuffles n elements of the physical RAM in time $O(n \log^2 n)$ as can be implemented by a sorting network.

Like the original square root algorithm of Goldreich in [Gol87], the algorithm works by having a randomized dictionary structure that is used to connect an index to a piece of data. At the same time we store a linear cache of previously accessed elements, so we can make sure not to look at the same place twice, we also use dummy elements to hide whether we access the same place twice, and we amortize the time used for searching the increasingly long cache by reshuffling everything for every $\sqrt{N}$ accesses.

But without a random oracle we cannot store a random permutation "for free" (by storing a key to the oracle), so we save the permutation using a binary tree with each level shuffled individually, so each child pointer from a node points to a random location of the next level. Also we make $\sqrt{N}$ "dummy chains", that are also shuffled into the tree. Only the root level is (trivially) not shuffled. The shuffling is depicted in fig. 3.1

3.6.1 Making a lookup

A lookup in the simulated RAM is implemented by making a lookup in the binary search tree. In order to touch every node in the tree only once, we do a (possibly dummy) lookup in the physical RAM on each level of the tree, and for each level we also linearly scan through all of the cache to see if we have accessed the same node earlier.

If we found the element in the cache, the next access in the tree will still be to a dummy node.

The physical memory is split up into $\log_2(N)$ parts, the i'th part is again split in two; $physical[i]$ storing $2^i + \sqrt{N}$ records (the tree, and the dummy chains), and $cache[i]$ storing $\sqrt{N}$ records (the cache).

Each node in the three record stores a .left and .right field for pointing to the next level, and a .bound for directing the search in the tree.

The leaf-records at the lowermost level are different, they contain the .data that are stored, an .index field naming the original index where the data is stored in the simulated RAM, a .used field that is used for reshuffling the data as described below.

An invariant for the outer loop can be phrased:

1. *next* is the real index we are going to look for at *level*

2. *next_in_tree* is the index of the tree where we will look if we do not find the item in the cache. If this is different from next, it is still pointing at a dummy chain.

By changing the index of the cached value to ∞ when it is found at 1 we implicitly invalidate it; it will be sorted last and therefore thrown away when we reshuffle. This is only necessary to do for the cache of the last level of the tree.

3.6.2 Obliviously shuffling a tree

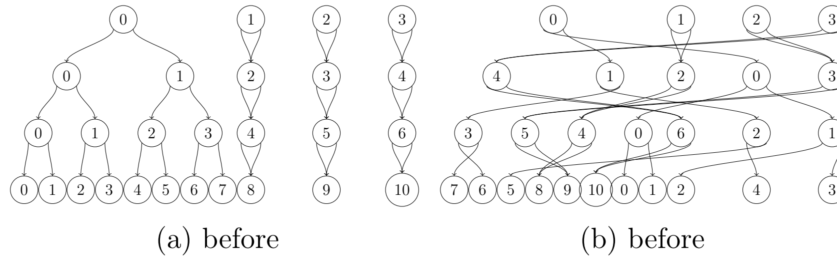
We present an algorithm for shuffling a tree: it works by shuffling one level at a time, starting at the bottom of the tree, and ending with the root-level.

After shuffling a level L with permutation π (so $L'[i] = L[\pi(i)]$), it applies π^{-1} on a sequence $[1, \ldots, n]$, and copies these numbers into the child-pointer fields of the level above. This gives the right result, because what before pointed at k will now point at $\pi^{-1}(k)$. But looking up in the shuffled layer yields the record at $L'[\pi^{-1}(k)] = L[\pi(\pi^{-1}(k))] = L[k]$. We take special care to also shuffle the dummy chains, and ensuring that their left and right pointers

Algorithm 3.6.1: LOOKUP(*key*)

```
procedure DISPATCH(key, record)
  output: The left or right child of record, depending on key
  if key < record.bound
    then return (record.left)
    else return (record.right)

input: key
output: Value stored at key
if count ≥ √n
  then { UNMINGLE()
         SHUFFLE()
         count ← 0
  else count ← count + 1
next ← count
next_from_tree ← count
for level ← 0 to log_2(N) − 1
  do { found ← False
       for i ← 0 to count − 1
         do { k ← cache[level, i]
              if k.index = next
                then { k_from_cache ← k
                       found ← True
                       k.index = ∞                (1)
                       cache[level, i] ← k
       if found
         then next ← next_from_tree
       k_from_tree ← physical[level, next]
       physical[level, next].used = True
       next_from_tree ← DISPATCH(index, k_from_tree)
       next ← DISPATCH(index, k_from_tree)
       if found
         then next ← DISPATCH(index, k_from_tree)
       cache[level, count] ← (next, UPDATE(k))
```



(a) before (b) before

Figure 3.1: Visualization of the memory layout of a tree storing 8 elements, before and after shuffling the tree. The edges indicate child-pointers.

point to the same next node in the chain.

Algorithm 3.6.2: UNMINGLE( )

$a \leftarrow$ Filter out any physical[$\log_2 N$] which has .used= **true**
$b \leftarrow a$ concatenated with cache[$\log_2 N$].
Obliviously sort b according to the original *index* of the records
Remove the last $\sqrt{N}$ records of b
$physical[log_2 N] \leftarrow b$

Algorithm 3.6.3: SHUFFLE( )

for $level \leftarrow log_2(n)$ **downto** 1
do
- Choose a permutaion π uniformly at random
- Shuffle physical[level] according to π
- $temp = [1, 2, \ldots, 2^{level} + \sqrt{N}]$
- Shuffle temp according to π^{-1}
- **for** $i \leftarrow 0$ **to** $2^{level-1}$
  - **do** $physical[level-1, i]$.left $\leftarrow temp[2i]$; $physical[level-1, i]$.right $\leftarrow temp[2i+1]$
- **for** $i \leftarrow 2^{level-1}$ **to** $2^{level-1} + \sqrt{N}$
  - **do** $physical[level-1, i]$.left $\leftarrow temp[2^{level} + i]$; $physical[level-1, i]$.right $\leftarrow temp[2^{level} + i]$

3.6.3 Security

The transcript of a lookup, as seen by the adversary, always consists of the following parts:

- An access at index *count* of the first level of the tree
- An access at a uniformly random location at each lower level of the tree
- A scan of the full cache of each level
- For each $\sqrt{N}$ access, the tree is reshuffled

All these are independent of the access pattern to the original RAM, and thus an eavesdropper will learn nothing whatsoever about the access pattern.

3.6.4 Performance

The time for a single lookup (without the shuffling) is dominated by accessing $\log_2 N$ caches, each of size $O(\sqrt{N})$. For each $\sqrt{N}$ lookups, we perform a shuffle taking time $O(N \log^2 N)$. Giving an amortized running time of $O(\sqrt{N} \log^2 N)$ per lookup.

3.7 The Polylogarithmic Construction

We now present an ORAM construction with an amortized overhead per access of $O(\log^3 N)$.

We follow the basic structure of [GO96] the idea is to have $\log(N)+1$ levels of caches. Cache at level 0 has size $O(N)$, and in general the simulated RAM at level ℓ stores $O(N/2^\ell)$ elements. Just like in [GO96] where each cache-level is a dictionary somewhat reminiscent of the one used in their square root construction. Likewise in our construction, each level is a shuffled tree much as the one used in our square root construction.

The main ideas of the construction are as follows:

- We start with a random shuffled tree at level 0 with all data in this tree and all other levels being empty.

- When a lookup is made, we start at the root node, and follow links as in the square root solution. But now a child pointer can point not only to a node in the same tree, but to a node at any level of the tree with an index closer to 0, and this information is stored with the pointer.

  When some path $(\nabla_1, \ldots, \nabla_m)$ is followed, we first try to insert the nodes ∇_i at the bottom level, i.e., level $\log N$. If this level is empty, the path is shuffled (such that the nodes ∇_i are stored in random positions at the bottom level and such that ∇_i points to the new physical location of ∇_{i+1} at the bottom level) and inserted here. Otherwise it is merged with the tree already on this level, and we try to put the result on the next level and so on until we find an empty level.

  The pointers from the ∇_i to the physical addresses of the siblings of the ∇_i which were not on the path $(\nabla_1, \ldots, \nabla_m)$, and hence were not moved, are just copied along with the nodes during the shuffling, so that the nodes ∇_i at the bottom level might now have pointers to untouched nodes in the trees at higher levels.

Pseudocode for the look-up procedure is shown in Figure 3.2

- For every lookup operation a new path is introduced to the bottom level, if there already is a path there, that and the new one are merged and reshuffled, such that all nodes (except the root of the tree) are placed at new random positions and their intra-pointers are updated. Pointers to untouched nodes at higher levels are kept the same. We describe a procedure for doing this below.

  The merged tree is moved to the level above. If there is a tree at that level, it is again merged with the moved tree and moved up etc. (Like a binary counting with each level representing a bit.)

- During the movement of nodes, either their moving to level $\log N$ after being touched or their being shuffled higher up the levels, it is always remembered where the root node of the tree original at level 1 is stored. This is oblivious, as any search starts at the root no matter the index being searched for.

  Because it is touched as the first node for every look-up, it will always be at the first position of the youngest tree.

- If we were just to start each search at the original root node and then following the updated points to physical addresses it is true that we would never touch the same node twice in the same position, as a touched node is moved down and then shuffled up. The pattern of how the path jumps between the levels would however leak information.[1] We therefore make sure to touch each level once every time we follow one pointer by doing dummy reads at all other levels. If we are following a pointer to level ℓ we do dummy read at levels $i = \log_2 N, \ldots, \ell + 1$, then we read the node at level ℓ, and then we do dummy reads at levels $i = \ell - 1, \ldots, 1$. Only the real node ∇ read at level ℓ is moved to the bottom level.

A sequence of accesses is illustrated in Figure 3.3

The procedure `MergeLayers` is not described in pseudocode, but it is explained in detail in the following section.

3.7.1 Merging and shuffling of two partial trees

We can not not store each layer of a tree in separate parts of the memory as in the square root solution, but with all the nodes of each cache level shuffled amongst each other, this is because the cache does not store the full tree and we cannot show how many nodes we have in that cache.

Each time we make a lookup, we build, in the physical memory, a chain by copying the nodes from the root of the tree to the node we are looking up, and this chain is inserted at the new bottom cache-level. If there is already a bottom level, the nodes from that are merged and inserted one level higher, etc.

We now describe how to merge and shuffle such two partial trees:

First we merge the two lists, ignoring already used nodes (nodes that we have already touched), and observe that now a crucial invariant holds, that enables us to do the shuffling obliviously:

The child-pointers that are not pointing to other nodes inside a cache-level, will always point out of the local cache, and into an *older* cache. We are always merging the two youngest existing caches, and therefore there will

[1] If, e.g., we look up the same leaf node twice, the path to the node is moved to the bottom level in the first update, so the second update would only touch nodes at the bottom level. If we look up a distinct leaf in the second update the pointer jumping would at some point take us back to level 1. This would allow the adversary to distinguish.

Algorithm 3.7.1: LOOKUP(*key*)

```
procedure INSERTPATH(l)
  evel ← log2(N)
  while level ∈ live
     do { path ← MERGELAYERS(path, physical[level])
          live ← live − {level}
          level ← level − 1
  physical[level] ← path
  live ← live ∪ {level}

  input:   key
  output:  Value stored at key
current_level ← log2 N
current_node ← 1
for i ← 1 to log2(N) − 1
  do { for each level ∈ live
         do { if level = current_level
                then { n ← physical[level, current_node]
                       if key < n.bound
                         then { next_level ← n.left.level
                                next_node ← n.left.node
                                n.left.level ← log2(N)
                                n.left.node ← i
                         else {... same thing for right side ...
                else { Dummy lookup:
                       n ← physical[level,
                            physical[level].dummy]
                       physical[level].dummy ← n.left.node
       path ← APPEND(path, n)
       current_level ← next_level
       current_node ← next_node
INSERTPATH(path)
return (n.left, n.right)
```

Figure 3.2: The look-up procedure s pseudocode

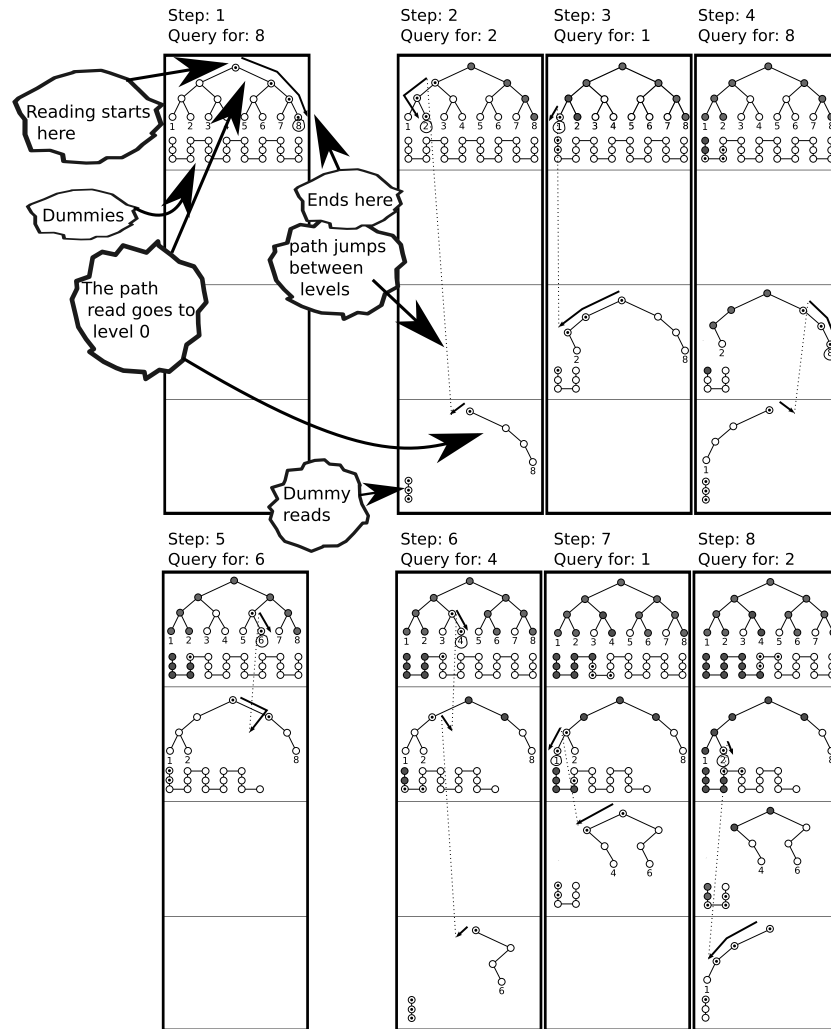


Figure 3.3: Each frame shows the logical (unshuffled) view of the different cache layers of each step of a sequence of accesses to an oblivious RAM storing 8 elements. Dummy elements at each level are shown at the bottom. The arrow shows how the lookup progresses, and nodes with a dot are those read at this step. Used nodes are shown in gray. Notice how no node is ever touched twice, and how, besides the root node, $\log n = 3$ nodes are read from each live level per lookup.

never be pointers from the outside pointing into the merged caches. Also this means that the root of the tree will always be in one of the merged trees (the youngest one).

We will copy all the nodes from both trees that we have not touched yet (these we call *live nodes*) into a new list in the same order as they where before (but with the used nodes removed). We run through the nodes from the second list, and obliviously add to each internal (to that list) child pointer the number of live nodes in the first list, so all internal child-pointers are unique.

Also we connect the two dummy chains, by running through all the nodes of the first tree, and for each obliviously checking if it is the end of a dummy chain, if so, it is set to point at the beginning of the dummy chain of the second tree.

In the two lists there will be a number of live dummies already, and they can be reused but for the larger list we need more dummies (as many as there are already), so we make a chain of new dummies and make the end of that point to the head of the old dummy chain. These new nodes are put at the end of the list of nodes, and we shuffle them together with the rest.

The important invariant is now that every node always has exactly one incoming pointer from the live nodes (they are all unique representatives of nodes in the original tree). They all have one or two children among the copied nodes, the pointers to these needs to be updated, while the (one or zero) child-pointers pointing out of the tree to an older tree from where it was copied should stay the same.

The root-note and the current dummy node are exceptions to this invariant as they have no incoming pointers, and we have to take special care of these two nodes.

Note that we cannot allow the dummy nodes to have two pointers pointing to the next node as in the square-root solution, because that would give them two incoming nodes as well. We solve this by obliviously marking them as dummy nodes, and when following such one, we always follow the only pointer.

We now present a subroutine for updating the pointers while we merge and shuffle the two lists of p nodes at the same time. It is illustrated in figure 3.4

1. For each child pointer in the list in order, we write in a new list a pointer y that, if the child node is internal is a copy of the of the child pointer

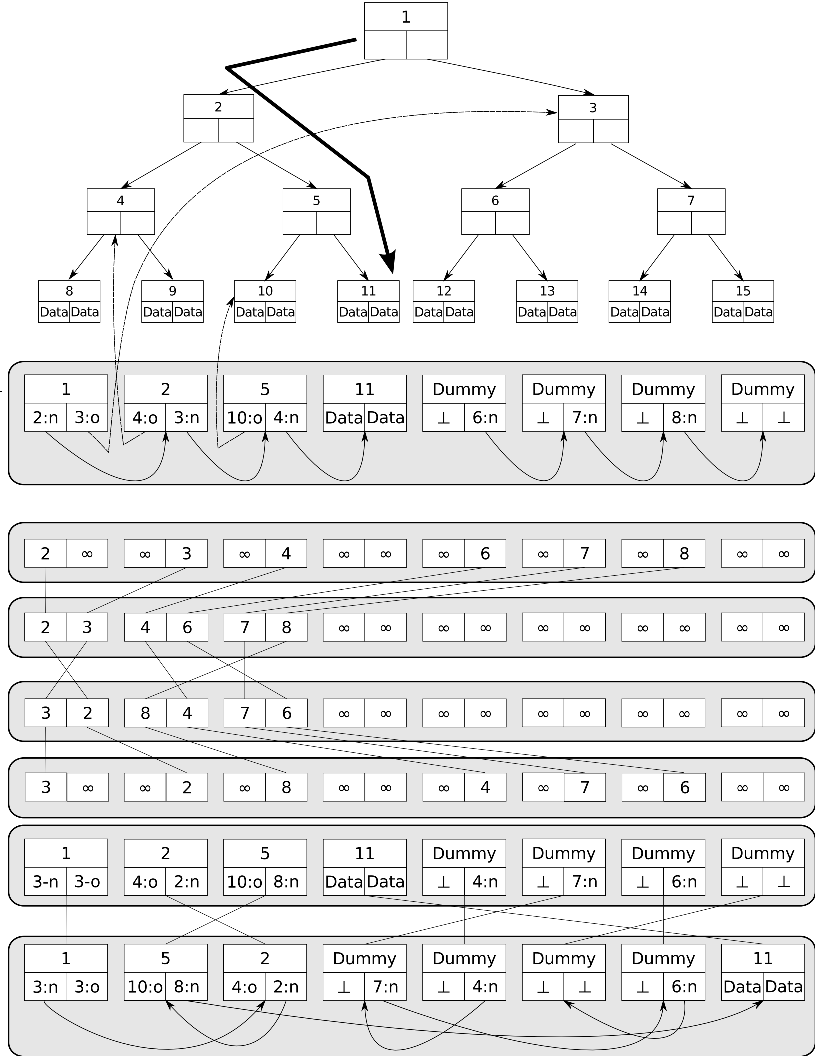


Figure 3.4: Illustration of the shuffling process for partial trees. “-h” here indicates a pointer inside the level, “-1” is a pointer to the first level.

and ∞ otherwise so it will be sorted last. These represent pointers out of the tree, and should not be updated.

When shuffling a list with p live nodes, this list will have $2p$ entries. Approximately half of these will be dummies. It might be more than half, because paths might overlap. This does not affect the shuffling procedure, only the dummy chains will be longer than what can ever be used before reshuffling, but this knowledge is never revealed to the adversary.

2. First sort the row of y's obliviously, and save the sorting permutation as σ. Because each node (except the two special nodes) has exactly one incoming pointer, the fist $p-2$ nodes will be internal, and those are the ones we want to update, so they point to the new locations of those nodes.

3. Now we create the permutation, π, that we want to shuffle the final list in. This permutation should leave location 1 (the root) and the location of the head of the dummy chain the same place, and be uniformly random for all other indices.

4. Now permute a list containing the numbers from 1 to p according to π^{-1}, and remove the first element, and the dummy-head pointer (they stay in place under π). Take this list of $p-2$ elements and concatenate them with $p+2$ ∞'s.

5. Now first undo the sorting by applying σ^{-1} to the list, and use the unsorted list to overwrite the original nodes where they came from (obliviously changing the value only when the new value is $\neq \infty$), now we have updated all the internal pointers, and left the external pointers untouched.

6. Finally shuffle the nodes according to π. We now have a shuffled list with all nodes pointing to the right places.

The time this procedure takes is dominated by the time it takes to sort the list.

3.7.2 Security

A simulated lookup results in

- Access to the root node
- Access to the root-dummy of each cache-level
- $\log(N) - 1$ accesses to uniformly random locations amongst the unvisited nodes of each cache-level
- Some fixed amount of shuffling

All these are independent of the access pattern.

3.7.3 Performance

To see how many log factors we use, we look at the two parts of the algorithm (that takes time). That is the operations for doing the *lookup*, and the operations for doing the *reshuffling*.

Following a path of length $\log N$ requires to look up $\log N$ nodes. For each node we have to touch each level for obliviousness, which gives a total of $\log^2 N$ physical reads.

For reshuffling we saw that the time for shuffling a tree is dominated by sorting it. We have $\log N$ layers with trees. The tree on level i is of size $O(2^i \cdot \log N)$, so that takes $O(2^i \cdot \log N \cdot \log(2^i \cdot \log N)) = O(2^i \cdot \log N \cdot i)$ operations to sort (because i is in $O(\log N)$, so it dominates $\log \log N$). But that level is only shuffled every 2^{i-1}th. lookup. On average we therefore use time for all layers:

$$O\left(\sum_{i=1}^{\log N} \log N \cdot i\right) = O(\log^3 N)$$

Therefore the total amortized time spent per lookup is in $O(\log^3 N)$.

3.8 Lower bounds on randomness

An equivalent way to state the definition of a secure oblivious RAM simulation is that, for simulation of any program running on an standard RAM, the resulting distribution of accesses to physical memory is the same for every choice of input. In particular, if we choose (part of) the input at random, the distribution of memory accesses must remain the same.

Our goal in this section will be to show a lower bound on the number of random bits that a simulation must use in order to be secure. We will for the moment assume that the simulation is *data-oblivious*, i.e., the simulation treats every word it is asked to read or write as a black-box and does not look at the data it contains. Any such word is called a data-word. All known oblivious RAM simulators are data oblivious. Indeed, letting anything the simulator does depend on the content of a data-word would only seem to introduce extra difficulties, since at the end of the day the access pattern must not depend on this content. We have a proof for a weaker lower bound for the non data-oblivious case, it is left out here for space reasons.

To show the lower bound, we consider the program that first writes random data to all N locations in RAM. It then executes d read operations from randomly chosen locations. Let U denote this sequence of update instructions.

Let P be the random variable describing the choice of locations to read from. Clearly $H(P) = d \log N$. Let $C = \text{LEAK}_{\mathcal{S}}(U)$ be the random variable describing the history of the simulation as seen by the adversary. Finally, let K be the random variable describing the random choices made by the simulation during the execution of the program, the r-values of the `random`-commands. We will show a bound on $H(K|C)$, that is, a bound on the number of random bits that are unknown to the adversary.

By construction of the random experiment, K is independent of P and, assuming the simulation is perfectly secure, C and P are independent.

From this it follows by elementary information theory that

$$H(K|C) \geq H(P) - H(P|C,K) = d \log N - H(P|C,K) \ . \qquad (3.1)$$

Now, let us assume that each read operation causes the simulation to access at most n locations in physical RAM. From this we will show an upper bound on $H(P|C,K)$.

Let P_i be the random variable describing the choice of location to read from in the i 'th read. Then we can write $P = (P_d, ..., P_1)$, and we have

$$\begin{aligned} H(P|C,K) &= H(P_1|C,K) + H(P_2|P_1,C,K) + ... + H(P_d|P_{d-1}...P_1,C,K) && (3.2) \\ &\leq H(P_1|C,K) + H(P_2|C,K) + ... + H(P_d|C,K) \ . && (3.3) \end{aligned}$$

The plan is now to bound $H(P_i|C=c, K=k)$ for each i and arbitrary, fixed values c, k, from this will follow a bound on $H(P|C,K)$. We will write $H(P_i|C=c, K=k) = H(P_i|c,k)$ for short in the following.

Note first that once we fix $K = k, C = c$, in particular the value of c specifies a choice of at most n locations that are accessed during the i'th read operation. This will constrain the distribution of P_i to be only over values that cause these locations to be accessed. Let w be the number of remaining possible choices for P_i. This is a set of addresses that we call the *relevant* addresses. Since each address is chosen from a range of N possibilities and there are w relevant addresses, we have $H(P_i|c, k) \leq \log w$.

Let $a = \log_2 q$ be the number of bits in a memory location, and recall that the program initially writes random data to all addresses. Let the random variable $D_{c,k}$ represent the choice of data originally written to the w relevant addresses. Since the simulation is data oblivious, fixing $C = c, K = k$ does not constrain the data stored in the relevant addresses, and hence $D_{c,k}$ is uniform over the 2^{aw} possible values, or equivalently $H(D_{c,k}) = aw$.

Let $R_{c,k}$ represent all the data the simulator accesses while it executes the i'th read operation given the constraints we have defined. Since at most n words from external memory are accessed, we have $H(R_{c,k}) \leq an$.

But on the other hand, $H(R_{c,k})$ must be at least $H(D_{c,k})$, since otherwise the simulator does not have enough information to return a correct result of the read operation. More precisely, since the simulation always returns correct results, we can reconstruct the exact value of $D_{c,k}$ as a deterministic function of $R_{c,k}$ by letting the value of P_i run through all w possibilities and computing in each case what the simulation would return. Since applying a deterministic function can only decrease entropy, it must be that $H(D_{c,k}) \leq H(R_{c,k})$.

We therefore conclude that $aw \leq an$, and from this and $H(P_i|c, k) \leq \log w$ it follows immediately that $H(P_i|c, k) \leq \log n$. By definition of conditional entropy we have $H(P_i|C, K) \leq \log n$ as well, and hence by (3.3) that $H(P|K, C) \leq d \log n$. Combining this with (3.1) we see that $H(K|C) \geq d \log(N/n)$, when d read operations are executed. Thus we have:

Theorem 3.8.1. *Suppose we are given a perfectly secure oblivious RAM simulation of a memory of size N. Assume it accesses at most n locations in pysical RAM per read operation and is is data-oblivious. Then there exist programs such that the simulator must, on average, use at least* $\log(N/n)$ *secret random bits per read operation.*

3.8.0.1 Extensions

Suppose the simulation is not perfect, but leaks a small amount of information. This means that P and C are not independent, rather the distributions of C caused by different choices of P are statistically close. Therefore the information overlap $I(C;P)$ is negligible as a function of the security parameter. If we drop the assumption that P,C are independent (3.3) becomes $H(K|C) \geq d \log N - H(P|C,K) - I(P;C)$. The rest of proof does not depend on C,P being independent, so the lower bound changes by only a negligible amount.

The result also holds even if the adversary does not know which instructions are executed by the simulated program, as the proof does exploit the fact that in our model, he knows this information.

Another remark is that the result assumes that every read operation causes at most n locations to be accessed. This does not, actually, cover the known solutions because they may at times access a very large number of locations, but do so seldom enough to keep the amortized overhead small. So we would like to show that even if we only assume the amortized overhead to be small, a large number of secret random bits is still needed. To this end, consider the same program as before, and suppose we have a simulation that accesses at most n_i bits during the i'th read operation. Let $\bar{n}$ be the average, $\bar{n} = \frac{1}{d}\sum_i n_i$. Then we have

Theorem 3.8.2. *Suppose we are given a perfectly secure oblivious RAM simulation of a memory of size N. Assume it accesses at most n_i locations in physical RAM during the i'th read operation and is data-oblivious. Then there exist programs such that the simulator must, on average, use at least $\log(N/\bar{n})$ secret random bits per read operation.*

Proof. Observe first that the argument in the proof for Theorem 3.8.1 for Equations (3.3) and (3.1) still holds here, and that furthermore the argument for $H(P_i|C,K) \leq \log n$ does not depend on the bound n being the same for every read operation. Hence, under the assumption given here, we get $H(P_i|C,K) \leq \log n_i$, and hence by (3.3), (3.1) that

$$\begin{aligned} H(K|C) &\geq d\log N - \sum_{i=1}^{d} \log n_i = d\log N - d\frac{1}{d}\sum_{i=1}^{d}\log n_i \\ &\geq d\log N - d\log(\frac{1}{d}\sum_{i=1}^{d} n_i) = d\log N - d\log\bar{n} = d\log(N/\bar{n}) \ . \end{aligned}$$

where the last inequality follows from Jensen's inequality. □

3.8.1 Related bounds

Beame and Machmouchi show in [BM11] that an oblivious RAM simulation requires either an $N^{1-o(1)}$ factor increase in space or an $\Omega(\log N \log\log N)$ factor increase in time (corollary 1.2). This bound says nothing about the amount of randomness needed.

3.9 Discussion

3.9.1 Random access in the MPC setting

It is common to measure the complexity of MPC-protocols in rounds, not in field operations. The trivial solution of touching every element in the RAM for each access can be done in $O(1)$ rounds if we have an unlimited fan-in addition primitive. When looking up with a secret index $[i]$ simply compute the sum

$$\sum_j [i] =_? j \cdot A_j$$

Where A_j is the jth element of the RAM.

Our solution on the contrary uses $O(\log(N))$ rounds. If done naively, even the "trivial" solution of touching all elements has a high cost, since secure equality tests are relatively expensive operations. A nice alternative technique is to create an array wih 1 at the index and 0'es everywhere else based on bit-splitting and exponentiation as explained in [SF05] That paper also contains another idea based on mix-nets (or rather shift-nets) and an additive sharing of the index.

In his thesis Peter Thomas Williams has shown [Wil12] how to do 1-round oblivious RAM with a polylogarithmic overhead - and Lu and Ostrovsky shows how to use a similar construction to construct garbled RAM programs, in the sense of Yao's garbled circuits.

It would be interesting to create a similar constant round information theoretic ORAM with polylogarithmic overhead, or show that it cannot be done. [LO12]

3.9.2 Future Directions

The hierarchical ORAM algorithm presented here has the steps of reshuffling as the clear bottleneck due to sorting. We only need sorting for shuffling and for filtering elements. Ways of doing oblivious shuffling without full sorting networks would improve the practicality of the construction.

Also interesting is the possibility of doing the look-up in less than $log^2 N$ steps. That would improve the complexity of the construction in the next chapter where we avoid the oblivious sorting.

So we grow together, Like to a double cherry, seeming parted, But yet an union in partition; Two lovely berries moulded on one stem; So, with two seeming bodies, but one heart; Two of the first, like coats in heraldry, Due but to one and crowned with one crest.

William Shakespeare, A Midsummer Night's Dream

4 Two-server Information-Theoretic Oblivious RAM

The contents of this chapter is unpublished elsewhere.

4.1 Introduction

Lu and Ostrovsky [LO13] introduced the idea of constructing an ORAM by dividing the server's work between two servers, relying on them not to communicate mutually. This model is a great fit for using ORAM together with secure two-party computation. In the two party computation setting we can let each player take the role of a server, and emulate the client in the secure computation. The big advantage of their construction, is that they avoid the costly oblivious sorting/shuffling required by other approaches to oblivious RAM by letting one server shuffle the data that the other server will later hold.

Like the classical construction by Goldreich and Ostrovsky [GO96], their solution is based on hash-tables indexed by a pseudo-random function and thus offers computational security outside the random oracle model.

In this chapter we show an adaption of the unconditionally secure oblivi-

ous RAM of the previous chapter to this two-server setting, thus solving one of the main stated open problems of [LO13]: an information theoretically secure construction in the two server model.

We work in a model where the memory cells of a server holds "closed" entries, and a closed entry can be copied, and transferred via the network, but never opened by anyone else than the client. Also a server cannot recognize a copy of a closed box (think of it as a semantically secure symmetric encryption with rerandomization.)

Notice, that only the access pattern is information-theoretically hidden by this construction. It does not provide unconditional security in the two-party setting of secure computation without assuming unconditionally secure "boxes".

The amortized per-query overhead of our solution is $O(log^2(N))$ (down from $O(log^3(N))$ for the single-server construction). So we do not reach the $\log(N)$ lower bound for one-server oblivious RAMs like [LO13] do, but we shave off one $\log(N)$ factor compared to the one-server version, and avoid the oblivious sorting which is most likely the bottleneck for reasonable Ns.

4.2 Model

A two-server oblivious RAM is a way of translating inputs to a RAM interface into inputs to two independent servers with an extended interface, in a way so any indices of the translated inputs are distributed independently from the original inputs.

A server is an interactive machine with a RAM memory that can be controlled by the client to do different operations. Our construction requires the server to have the following interface:

- set(i, v) Stores an element v at location i in the server's memory.
- get(i) $\to v$ Returns the element v last stored at i.
- shuffle(i, n, $blocksize$) $\to \pi$ Shuffles n sequential pieces of $blocksize$ of the memory starting from memory location i. Returns a counter π, a small identifier naming this particular permutation. This command has complexity $O(n \cdot blocksize)$

- unshuffle(i, n, $blocksize$, π) Permutes n elements of $blocksize$ starting from memory location i according π^{-1}. where π is the identifier returned by some earlier shuffle operation from the same server. Also this command has complexity $O(n \cdot blocksize)$

v and i are considered words of size $\log(M)$, and can thus index the server's entire memory.

Note that this interface is more complicated than that of the single-server model (requiring only the interface of a RAM), but simpler than that of of the servers of [LO13] that must also be able do computation of a *pseudo random function*.

We define the two-server model as a client C defined as an interactive machine as in Section 3.4 on page 80 interacting with two servers S_A and S_B. C is modeled as an interactive Turing machine with a constant sized memory.

Definition 4.2.1. *The* access pattern *of client C to a server S from the execution of a program is the list of commands sent to S and the responses sent back, but with all values v removed.*

For example the communication sequence:
set(3, 4), get(3) -> 4, shuffle(1, 10, 5) -> 1, unshuffle(1, 10, 5, 1)
has the access pattern
set(3, _), get(3) -> _, shuffle(1, 10, 5) -> 1, unshuffle(1, 10, 5, 1).

Definition 4.2.2. *The* two-server model *is a client C interacting with two servers S_A and S_B. C is modeled as an interactive machine with a constant sized memory.*

Definition 4.2.3. *A* two-server ORAM *is a program for a client in the two-server model implementing a RAM interface. We say that an ORAM is secure, if for all access sequences $i_1, \ldots, i_N$ to the client both:*

- *The resulting access sequence $i'_1, \ldots, i'_{N'}$ to S_A is independent of the accesses to the simulated RAM $i_1, \ldots, i_N$,*

- *And likewise the access sequence $i''_1, \ldots, i''_{N''}$ to S_B is independent of $i_1, \ldots, i_N$.*

4.2.1 Avoiding the extended interface

For simplicity we have described the servers as being able to shuffle their data locally. When as here the shuffling does not have to be oblivious this can be implemented by the client doing e.g. a Fisher–Yates shuffle on the server (for $i \in, 0 \ldots, n-1$ swap element i with a random element in $i, \ldots, n-1$), also saving on the server the locations of the swaps used during the shuffle in a separate location and using the address of this location as the id of the permutation. To invert the shuffle do the same swaps in the reverse order.

4.2.2

4.3 Trivial information theoretic solution

First we note that it is trivial to take any one-server ORAM scheme that only has unconditional security in the *random oracle model* and make it information-theoretically secure in the two-server model by using S_B as the random oracle in the following way:

- Let S_A do the work of the original server
- For each oracle-query the client would before do, it instead queries S_B. S_B checks if it had the same query before, if so it returns the stored answer, otherwise it sends a random answer and stores it for later retrieval.

In our model S_B is not allowed to interact with S_A and can therefore not be distinguished from a true random oracle. By using hash-tables the queries to S_B can be implemented with an expected constant number of queries.

The downside is that we do not get improved efficiency. We still have to rely on sorting networks (if the original construction does).

The construction below uses the second server to avoid using sorting networks by using one server to do the shuffling, and at the same time retains the information theoretical security from the construction of the previous chapter.

4.4 Idea of the construction

The construction is an adaptation of the information-theoretic one. S_A stores all data and responds to queries. S_B is only used for doing shuffling obliviously to S_A.

Queries are executed between the client and S_A exactly as in the single-server construction.

During the query, When a path from root to leaf is read, the client copies the followed nodes one by one and stores them on S_A, but updates the followed links, so they form a chain. This is the new "smallest cache" on S_A.

Queries are performed between the client and S_A exactly as in the single-server construction described in Section 3.7, except that during the query, when a path from root to leaf is read, the client copies the followed nodes one by one and stores them on S_A, but updates the followed child pointers, so they form a chain. This chain is then inserted in the smallest cache.

We make sure that the end of a dummy chain is pointing to the head of the dummy chain on the next (bigger) cache-level. This is because we reuse the dummy nodes from two levels together after reshuffling. When two levels are shuffled together they need a dummy chain as long as the sum of the two levels' dummy chains.

When two layers a, b (a being smaller and thus younger than b) are to be reshuffled according to the hierarchical model, their nodes are put in sequence, first all of a's nodes, then all of b's. All child-pointers from a into b will in the following be interpreted as pointing into the newly created layer with an offset of $|a|$.

Notice that there will never be two pointers to the same node, and all nodes except the root and dummy root have exactly one pointer to it. Therefore the internal pointers in the new layer are public information (but their order is not). This can be used to let b do the updating of the pointers.

4.5 Reshuffling

The procedure is described in two parts. First (Step 1–5) we update the internal pointers to point at their new location, and second (Step 6–9) the nodes themselves are shuffled. The procedure is as follows:

1. As noted, the internal pointers between the nodes to be shuffled is public knowledge (where they are pointing from is secret though) be-

cause every node (except the root and the dummy root) have exactly one pointer to them, so the client first sends a list of all the internal pointers in the layers to be shuffled to S_B who stores them sequentially.

2. The client asks S_B to shuffle this list, and gets a permutation ID π, call this permuted list r.

   This list will be used for look-ups.

3. The client ask S_A for all the child-pointers of the nodes to be mixed, including pointers from dummy nodes one by one, stores them back in sequence on S_A, makes S_A shuffle them and gets the permutation ID σ.

4. The client requests each of the shuffled pointers from S_A in order. For each it looks at where the pointer is pointing:

   - If it is a pointer to an *internal node* to one of the layers being shuffled, C requests S_B for the permuted version of the pointer to send back to S_A (by requesting the ith item in the list),
   - Otherwise C just sends it back to S_A.

5. S_A unshuffles the pointers (by π^{-1}) and puts them into their respective nodes.

6. C asks S_A for all the nodes to be shuffled (except the root, and the dummy-root), and sends them to S_B.

7. C asks S_B to unshuffle this list of nodes by π^{-1},

8. C asks S_B for the elements of the "unshuffled" list, and sends them back to to S_A one by one.

9. C asks S_A to inserts the now shuffled nodes with updated pointers at the relevant cache level.

4.6 Statement

Theorem 4.6.1. *The above construction implements a two-server oblivious RAM perfectly hiding the access pattern in the two-server model with an amortized $O(\log^2 N)$ query overhead. (Without using sorting networks).*

4.6.1 Security

Assuming the memory of S_A is set up correctly the query-phase is as secure as it is in the one-server construction.

The shuffling-phase only lets the servers shuffle deterministic intervals of closed values revealing nothing, except in step (*) where C queries S_b for different indices of an array, but these are in a random order. This phase in the end leaves a correctly set up memory on S_A for the next query-phase.

4.6.2 Overhead

As in the previous chapter the overhead of the query-phase is $O(\log^2 N)$, but the shuffle-phase now takes linear time in the $O(\log^2 N)$ nodes to be shuffled on average.

4.7 Why do we not need tags

[LO13] had to use carefully constructed tags constructed from timestamps and cache-level to allow one server to compute the hashes for use on the other server. Instead of just hashing the index of an element as in most one-server constructions. One could call the permutation of the address of an element in our construction our "tag", we ask one server to completely renew this information for each use, and because of the tree-structure, these tags are stored in the tree and can be truly random simplifying our construction.

4.8 Use for Multi Party Computation

One original motivation for two-server ORAM is that it can be used for secure two-party computation; the two parties acts as the servers and the client is simulated by the secure two-player-protocol they are running between themselves.

One main attraction of the information-theoretic solution of the last chapter is that no pseudo-random hash-function has to be computed by the client. Pseudo-random functions are, as mentioned in the previous chapter, typically expensive to compute in a secure model of computation, as they are very non-linear in nature. Instead only comparisons of indices are used in the tree data structure to find where an item is stored.

If we want to use our solution in the plain model, we are still faced with the problem of hiding data before storing it on the servers.

In the two-player setting this is again the job of the client. And if that has to be done by encryption of the values we are not much better off, as encryption also requires computation of pseudo-random functions, and leaves the protocol computationally secure.

But for most settings of information theoretic secure computation we make the assumption of honest majority, requiring more than 2 players.

With 4 players doing (semi-honest) information-theoretically secure MPC with threshold $t = 1$, we can let two players implement S_A and let the other two players implement S_B. In this setting we can use additive secret sharing between two players for implementing "closed box" memory cells. And additive secret sharing is very easy to do as a secure computation.

Whenever the client is supposed to send a value v to a server the MPC protocol implementing the client will choose a random value r and output $v + r$ and r respectively to the two players implementing that server. And likewise when the server has to send a value to the client, they input the stored shares and the client adds them. A shuffle on a server is done by having the one player choose a random permutation and send it to the other. Both players permute the requested values (represented in each player's storage as shares) by this permutation.

4.9 Round complexity

This construction is of course hopelessly inefficient in having to transfer the whole list of elements one by one from server to server many times. It should however be noticed that for cloud-storage purposes where e.g. $O(\sqrt{N})$ client memory seems realistic we can transfer the memory in larger blocks. Also for MPC where

4.10 Future directions

The four players required to implement the two servers seems superfluous, when $t = 1$ we would like to be able to make do with 3 servers. It would be interesting to investigate ways of creating an information theoretically secure ORAM without the overhead of oblivious shuffling in the 3-player

$t = 1$ setting.

Bibliography

[AIR01] William Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2001.

[Ajt10] Miklós Ajtai. Oblivious rams without cryptogrpahic assumptions. In *STOC*, pages 181–190, 2010.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[Bea95] Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 1995.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC*, pages 479–488, 1996.

[Bea97] Donald Beaver. Commodity-based cryptography (extended abstract). In *STOC*, pages 446–455, 1997.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, pages 134–153, 2012.

[Blu82] Manuel Blum. Coin flipping by telephone - a protocol for solving impossible problems. In *COMPCON*, pages 133–137. IEEE Computer Society, 1982.

[BM11] Paul Beame and Widad Machmouchi. Making branching programs oblivious requires superlogarithmic overhead. In *IEEE Conference on Computational Complexity*, pages 12–22, 2011.

[BMSW02] Carlo Blundo, Barbara Masucci, Douglas R. Stinson, and Ruizhong Wei. Constructions and bounds for unconditionally secure non-interactive commitment schemes. *Des. Codes Cryptography*, 26(1-3):97–110, 2002.

[Bon03] Dan Boneh, editor. *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*. Springer, 2003.

[BTW11] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. *IACR Cryptology ePrint Archive*, 2011:662, 2011.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19, 1988.

[CDD+04] Ran Canetti, Ivan Damgård, Stefan Dziembowski, Yuval Ishai, and Tal Malkin. Adaptive versus non-adaptive security of multi-party protocols. *J. Cryptology*, 17(3):153–207, 2004.

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, pages 41–50, 1995.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, pages 364–369, 1986.

[CT06] T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications and Signal Processing. John Wiley & Sons, 2006.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[DvMN10] Rafael Dowsley, Jeroen van de Graaf, Davidson Marques, and Anderson C. A. Nascimento. A two-party protocol with trusted

initializer for computing the inner product. In Yongwha Chung and Moti Yung, editors, *WISA*, volume 6513 of *Lecture Notes in Computer Science*, pages 337–350. Springer, 2010.

[DZ13] Ivan Damgard and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, 2013.

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[FGMvR02] Matthias Fitzi, Nicolas Gisin, Ueli Maurer, and Oliver von Rotz. Unconditional byzantine agreement and multi-party computation secure against dishonest minorities from scratch. In Lars Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 482–501. Springer Berlin / Heidelberg, 2002.

[FKN94] Uriel Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation (extended abstract). In *STOC*, pages 554–563, 1994.

[FWW04] Matthias Fitzi, Stefan Wolf, and Jürg Wullschleger. Pseudo-signatures, broadcast, and multi-party computation from correlated randomness. In Matt Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 505–511. Springer Berlin / Heidelberg, 2004.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GK96] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM J. Comput.*, 25(1):169–192, 1996.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP (2)*, pages 576–587, 2011.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, pages 157–167, 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *STOC*, pages 218–229. ACM, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194, New York, NY, USA, 1987. ACM.

[Gol04] Oded Goldreich. *Foundations of Cryptography Volume 2, Basic Applications*. Cambridge University Press, 2004.

[IK97] Yuval Ishai and Eyal Kushilevitz. Private simultaneous messages protocols with applications. In *ISTCS*, pages 174–184, 1997.

[IK02] Yuval Ishai and Eyal Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 244–256. Springer, 2002.

[IK04] Yuval Ishai and Eyal Kushilevitz. On the hardness of information-theoretic multiparty computation. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 439–455. Springer, 2004.

[IKLP06] Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On combining privacy with guaranteed output delivery

in secure multiparty computation. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, pages 483–500, 2006.

[IKM+13] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness. In *TCC*, 2013.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, pages 145–161, 2003.

[IOS12] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. Identifying cheaters without an honest majority. In *TCC*, pages 21–38, 2012.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 572–591. Springer, 2008.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In Janos Simon, editor, *STOC*, pages 20–31. ACM, 1988.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[KLP11] Greg Kuperberg, Shachar Lovett, and Ron Peled. Probabilistic existence of rigid combinatorial structures. *ArXiv e-prints*, November 2011.

[KNR05] Eyal Kaplan, Moni Naor, and Omer Reingold. Derandomized constructions of k-wise (almost) independent permutations. In Chandra Chekuri, Klaus Jansen, José Rolim, and Luca Trevisan, editors, *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, volume 3624 of *Lecture Notes in Computer Science*, pages 608–608. Springer Berlin / Heidelberg, 2005. 10.1007/11538462_30.

[LO12] Steve Lu and Rafail Ostrovsky. How to garble ram programs. *IACR Cryptology ePrint Archive*, 2012:601, 2012.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *TCC*, 2013.

[Mil96] Gary L. Miller, editor. *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. ACM, 1996.

[Nie07] Jesper Buus Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. *IACR Cryptology ePrint Archive*, 2007:215, 2007.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, pages 681–700, 2012.

[NOVY98] Moni Naor, Rafail Ostrovsky, Ramarathnam Venkatesan, and Moti Yung. Perfect zero-knowledge arguments for *np* using any one-way permutation. *J. Cryptology*, 11(2):87–108, 1998.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In S. Rao Kosaraju, editor, *SODA*, pages 448–457. ACM/SIAM, 2001.

[NP06] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5):1254–1281, 2006.

[PCR09] Arpita Patra, Ashish Choudhary, and C. Pandu Rangan. Round efficient unconditionally secure mpc and multiparty set intersection with optimal resilience. In Bimal K. Roy and Nicolas Sendrier, editors, *INDOCRYPT*, volume 5922 of *Lecture Notes in Computer Science*, pages 398–417. Springer, 2009.

[PW96] Birgit Pfitzmann and Michael Waidner. Information-theoretic pseudosignatures and byzantine agreement for $t \geq n/3$. *IBM Research Report RZ 2882 (#90830)*, 1996.

[Rab79] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1979.

[Rab81] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[Riv99] R.L. Rivest. Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer. *Manuscript*, 1999.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SCSL11] E. Shi, T. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. *Advances in Cryptology–ASIACRYPT 2011*, pages 197–214, 2011.

[SF05] Marius C. Silaghi and Gerhard Friedrich. Secure stochastic multi-party computation for combinatorial problems. Technical report, Florida Institute of Technology, University Klagenfurt, 2005.

[TDH+09] Rafael Tonicelli, Rafael Dowsley, Goichiro Hanaoka, Hideki Imai, Jörn Müller-Quade, Akira Otsuka, and Anderson C. A. Nascimento. Information-theoretically secure oblivious polynomial evaluation in the commodity-based model. *IACR Cryptology ePrint Archive*, 2009:270, 2009.

[Wil12] P.T. Williams. *Oblivious Remote Data Access Made Practical*. PhD thesis, STATE UNIVERSITY OF NEW YORK AT STONY BROOK, 2012.

[WS08] Peter Williams and Radu Sion. Usable pir. In *NDSS*, 2008.

[WSC08] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *ACM Conference on Computer and Communications Security*, pages 139–148, 2008.

[WW10] Severin Winkler and Jürg Wullschleger. On the efficiency of classical and quantum oblivious transfer reductions. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 707–723, Berlin, Heidelberg, 2010. Springer-Verlag.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.