Secure Multi-Party Computation

(Working Draft)

Oded Goldreich

Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot, ISRAEL.

June 9, 1998

Preface

More than ten years have elapsed since the first completeness theorems for two-party and multi-party fault-tolerant computation have been announced (by Yao and Goldreich, Micali and Wigderson, respectively). Analogous theorems have been proven in a variety of models, yet full proofs of the abovementioned basic results (i.e., for the "computational model" as well as for the "private channel model") are not to be found. This manuscript attempts to redeem this sour state of affairs, at least as far as the "computational model" goes.

Acknowledgments

Firstly, I'd like to thank Silvio Micali and Avi Wigderson, my co-authors to the work on which most of this manuscript is based. Secondly, I'd like to thank Ran Canetti for the many things I have learned from him regarding multi-party computation. Lastly, I wish to thank Hagit Attiya, Mihir Bellare, Benny Chor, Shafi Goldwasser, Leonid Levin, and Ronen Vainish for related discussions held throughout the years.

Dedication

To Amir Herzberg and Hugo Krawczyk who demanded that this manuscript be written.

A Warning

This is a working draft. It is certainly full of various minor flaws, but is hoped and believed to contain no serious ones.

Contents

Chapter 1

Introduction and Preliminaries

The current contents of this chapter is tentative. The main part of the introduction is reproduced with minor revisions from [37].

1.1 A Tentative Introduction

A general framework for casting cryptographic (protocol) problems consists of specifying a random process which maps m inputs to m outputs. The inputs to the process are to be thought of as local inputs of m parties, and the m outputs are their corresponding local outputs. The random process describes the desired functionality. That is, if the m parties were to trust each other (or trust some outside party), then they could each send their local input to the trusted party, who would compute the outcome of the process and send each party the corresponding output. The question addressed in this manuscript is to what extent can this trusted party be "emulated" by the mutually distrustful parties themselves.

1.1.1 Overview of the Definitions

For simplicity, we consider in this overview only the special case where the specified process is deterministic and the m outputs are identical. That is, we consider an arbitrary m-ary function and m parties which wish to obtain the value of the function on their m corresponding inputs. Each party wishes to obtain the correct value of the function and prevent any other party from gaining anything else (i.e., anything beyond the value of the function and what is implied by it).

We first observe that (one thing which is unavoidable is that) each party may change its local input before entering the protocol. However, this is unavoidable also when the parties utilize a trusted party. In general, the basic paradigm underlying the definitions of *secure multi-party computations* amounts to saying that situations which may occur in the real protocol, can be simulated in the ideal model (where the parties may employ a trusted party). Thus, the "effective malfunctioning" of parties in secure protocols is restricted to what is postulated in the corresponding ideal model. The specific definitions differ in the specific restrictions and/or requirements placed on the parties in the real computation. This is typically reflected in the definition of the corresponding ideal model – see examples below.

An example – computations with honest majority: Here we consider an ideal model in which any minority group (of the parties) may collude as follows. Firstly this minority shares its original inputs and decided together on replaced inputs[1] to be sent to the trusted party. (The other parties send their respective original inputs to the trusted party.) When the trusted party returns the output, each majority player outputs it locally, whereas the colluding minority may compute outputs based on all they know (i.e., the output and all the local inputs of these parties). A *secure multi-party computation with honest majority* is required to emulate this ideal model. That is, the effect of any feasible adversary which controls a minority of the players in the actual protocol, can be essentially simulated by a (different) feasible adversary which controls the corresponding players in the ideal model. This means that in a secure protocol the effect of each minority group is "essentially restricted" to replacing its own local inputs (independently of the local inputs of the majority players) before the protocol starts, and replacing its own local outputs (depending only on its local inputs and outputs) after the protocol terminates. (We stress that in the real execution the minority players do obtain additional pieces of information; yet in a secure protocol they gain nothing from these additional pieces of information.)

Secure protocols according to the above definition may even tolerate a situation where a minority of the parties aborts the execution. An aborted party (in the real protocol) is simulated by a party (in the ideal model) which aborts the execution either before supplying its input to the trusted party (in which case a default input is used) or after supplying its input. In either case, the majority players (in the real protocol) are able to compute the output although a minority aborted the execution. This cannot be expected to happen when there is no honest majority (e.g., in a two-party computation) [25].

Another example – two-party computations: In light of the above, we consider an ideal model where each of the two parties may "shut-down" the trusted (third) party at any point in time. In particular, this may happen after the trusted party has supplied the outcome of the computation to one party but before it has supplied it to the second. A *secure multi-party computation allowing abort* is required to emulate this ideal model. That is, each party's "effective malfunctioning" in a secure protocol is restricted to supplying an initial input of its choice and aborting the computation at any point in time. We stress that, as above, the choice of the initial input of each party may NOT depend on the input of the other party.

1.1.2 Overview of the Known Results

General plausibility results: Assuming the existence of trapdoor permutations, one may provide secure protocols for ANY two-party computation (allowing abort) [66] as well as for ANY multi-party computations with honest majority [41]. Thus, a host of cryptographic problems are solvable assuming the existence of trapdoor permutations. Specifically, any desired (input–output) functionality can be enforced, provided we are either willing to tolerate "early abort" (as defined above) or can rely on a majority of the parties to follow the protocol. Analogous plausibility results were subsequently obtained in a variety of models. In particular, we mention secure computations in the private channels model [8, 21], in the presence of mobile adversaries [54], and for an adaptively chosen set of corrupted parties [17].

[1] Such replacement may be avoided if the local inputs of parties are verifiable by the other parties. In such a case, a party (in the ideal model) has the choice of either joining the execution of the protocol with its correct local input or not join the execution at all (but it cannot join with a replaced local input). Secure protocols emulating this ideal model can be constructed as well.

We view the above results as asserting that very wide classes of problems are solvable in principle. However, we do not recommend using the solutions derived by these general results in practice. For example, although Threshold Cryptography (cf., [27, 33]) is merely a special case of multi-party computation, it is indeed beneficial to focus on its specifics.

1.1.3 Aims and nature of the current manuscript

Our presentation is aimed at providing an accessible account of the most basic results regarding general secure multi-party computation. We focus on the "computational model", assuming the existence of trapdoor permutations. We provide almost full proofs for the plausibility results mentioned above – secure protocols for ANY two-party (and in fact multi-party) computation allowing abort, as well as for ANY multi-party computations with honest majority. We briefly mention analogous results in other models.

We do not attempt to provide the most general definitions and the most general tools. This choice is best demonstrated in our composition theorems – they are minimal and tailored for our purposes, rather than being general and of utmost applicability. (Actually, in some cases we refrain from presenting an explicit composition theorem and derive a result by implicit composition of a subprotocol inside a bigger one.) Alternative presentations aimed at such generality are provided in [45, 50, 2, 13, 14, 15].

Likewise, no attempt is made to present the most efficient versions possible for the said results. In contrary, in many cases we derive less efficient constructions due to our desire to present the material in a modular manner. This is best demonstrated in our non-optimized compilers – especially those used (on top of one another) in the multi-party case. As we view the general results presented here as mere claims of plausibility (see above), we see little point in trying to optimize them.

1.1.4 Organization of this manuscript

Choices were made here too. In particular, we chose to present the two-party case first (see Chapter 2), and next to extend the ideas to the multi-party case (see Chapter 3). Thus, the reader interested in the multi-party case cannot skip Chapter 2. We hope that such a reader will appreciate that the two-party case is a good warm-up towards the m-party case, for general m. Actually, most ideas required for the latter can be presented in the case $m = 2$, and such a presentation is less cumbersome and allows to focus on the essentials.

Within each chapter, we start with a treatment of the relatively easy case of semi-honest behavior, and next proceed to "force" general malicious parties to behave in a semi-honest manner. We believe that even a reader who views the semi-honest model as merely a mental experiment will appreciate the gain obtained by breaking the presentation in this way.

1.2 Preliminaries (also tentative)

We recall some basic definitions regarding computational complexity and multi-party protocols. More importantly, we present and sustain a stronger than usual definition of proof of knowledge.

1.2.1 Computational complexity

Throughout this manuscript we model adversaries by (possibly non-uniform) families of polynomial-size circuits. Here, we call the circuit family $C = \{C_n\}$ uniform if there exists a poly(n)-time algorithm than on input n produces the circuit C_n. The latter circuit operates on inputs of length n. The non-uniform complexity treatment is much simpler than the uniform analogue for several reasons. Firstly, definitions are simpler – one may quantify over all possible inputs (rather than consider polynomial-time constructible input distributions). Secondly, auxiliary inputs (which are essential for various composition theorems) are implicit in the treatment; they can always be incorporated into non-uniform circuits.

We take the liberty of associating the circuit family $C = \{C_n\}$ with the particular circuit of relevance. That is, we write $C(x)$ rather than $C_{|x|}(x)$; we may actually define $C(x) \stackrel{\text{def}}{=} C_{|x|}(x)$. Furthermore, we talk of polynomial-time transformations of (infinite and possibly non-uniform) circuit families. What we mean by saying that the transformation T maps $\{C_n\}$ into $\{C'_n\}$ is that $C'_n = T(C_n)$, for every n.

Negligible functions. A function $\mu : \mathbb{N} \mapsto [0,1]$ is called negligible if for every positive polynomial p, and all sufficiently large n's, $\mu(n) < 1/p(n)$.

Probability ensembles. A probability ensemble indexed by $S \subseteq \{0,1\}^*$ is a family, $\{X_w\}_{w \in S}$, so that each X_w is a random variable (or distribution) which ranges over (a subset of) $\{0,1\}^{\text{poly}(|w|)}$. Typically, we consider $S = \{0,1\}^*$ and $S = \{1^n : n \in \mathbb{N}\}$ (where, in the latter case, we sometimes write $S = \mathbb{N}$). We say that two such ensembles, $X \stackrel{\text{def}}{=} \{X_w\}_{w \in S}$ and $Y \stackrel{\text{def}}{=} \{Y_w\}_{w \in S}$, are identically distributed, and write $X \equiv Y$. if for every $w \in S$ and every α

$$\mathsf{Pr}[X_w = \alpha] = \mathsf{Pr}[Y_w = \alpha]$$

Such X and Y are said to be statistically indistinguishable if for some negligible function $\mu : \mathbb{N} \mapsto [0,1]$ and all $w \in S$,

$$\sum_{\alpha} |\mathsf{Pr}[X_w = \alpha] - \mathsf{Pr}[Y_w = \alpha]| \ < \ \mu(|w|)$$

In this case we write $X \stackrel{\text{s}}{\equiv} Y$. Clearly, for every probabilistic process F, if $\{X_w\}_{w \in S} \stackrel{\text{s}}{\equiv} \{Y_w\}_{w \in S}$ then $\{F(X_w)\}_{w \in S} \stackrel{\text{s}}{\equiv} \{F(Y_w)\}_{w \in S}$.

Computational indistinguishability. We consider the notion of indistinguishability by (possibly non-uniform) families of polynomial-size circuits.

Definition 1.2.1 (computational indistinguishability): *Let $S \subseteq \{0,1\}^*$. Two ensembles* (indexed by S)*, $X \stackrel{\text{def}}{=} \{X_w\}_{w \in S}$ and $Y \stackrel{\text{def}}{=} \{Y_w\}_{w \in S}$, are* computationally indistinguishable (by circuits) *if for every family of polynomial-size circuits, $\{D_n\}_{n \in \mathbb{N}}$, there exists a negligible function $\mu : \mathbb{N} \mapsto [0,1]$ so that*

$$|\mathsf{Pr}[D_n(w, X_w) = 1] - \mathsf{Pr}[D_n(w, Y_w) = 1]| \ < \ \mu(|w|)$$

In such a case we write $X \stackrel{\text{c}}{\equiv} Y$.

Actually, it is not necessary to provide the distinguishing circuit (i.e., D_n above) with the index of the distribution. That is,

Proposition 1.2.2 *Two ensembles* (indexed by S), $X \stackrel{\text{def}}{=} \{X_w\}_{w\in S}$ *and* $Y \stackrel{\text{def}}{=} \{Y_w\}_{w\in S}$*, are computationally indistinguishable if and only if for every family polynomial-size circuits,* $\{C_n\}_{n\in\mathbb{N}}$*, every polynomial* $p(\cdot)$*, and all sufficiently long* $w \in S$

$$|\mathsf{Pr}\left[C_n(X_w)\!=\!1\right] - \mathsf{Pr}\left[C_n(Y_w)\!=\!1\right]| \; < \; \frac{1}{p(|w|)} \tag{1.1}$$

Proof: Clearly if $X \stackrel{\text{c}}{\equiv} Y$ then Eq. (1.1) holds (otherwise, let $D_n(w,z) \stackrel{\text{def}}{=} C_n(z)$). The other direction is less obvious. Assuming that X and Y are NOT computationally indistinguishable, we will show that, for some polynomial-sized $\{C_n\}_{n\in\mathbb{N}}$, Eq. (1.1) does not hold either. Specifically, let $\{D_n\}_{n\in\mathbb{N}}$ be a family of (polynomial-size) circuits, p be a polynomial, and S' an infinite subset of S so that for every $w \in S'$

$$|\mathsf{Pr}\left[D_n(w,X_w)\!=\!1\right] - \mathsf{Pr}\left[D_n(w,Y_w)\!=\!1\right]| \; \geq \; \frac{1}{p(|w|)}$$

We consider an infinite sequence, $w_1, w_2, \ldots$, so that $w_n \in S'$ if $S' \cap \{0,1\}^n \neq \emptyset$ and $w_n = 0^n$ (or any other n-bit long string) otherwise. Incorporating w_n into D_n, we construct a circuit $C_n(z) \stackrel{\text{def}}{=} D_n(w_n, z)$ for which Eq. (1.1) does not hold. ■

Comments: Computational indistinguishable is a proper relaxation of statistically indistinguishable (i.e., $X \stackrel{\text{s}}{\equiv} Y$ implies $X \stackrel{\text{c}}{\equiv} Y$, but not necessarily the other way around). Also, for every family of polynomial-size circuits, $C = \{C_n\}_{n\in\mathbb{N}}$, if $\{X_w\}_{w\in S} \stackrel{\text{c}}{\equiv} \{Y_w\}_{w\in S}$ then $\{C_{|w|}(X_w)\}_{w\in S} \stackrel{\text{c}}{\equiv} \{C_{|w|}(Y_w)\}_{w\in S}$.

Trapdoor Permutations. A sufficient computational assumption for all constructions used in this text is the existence of trapdoor permutations. Loosely speaking, these are collections of one-way permutations, $\{f_\alpha\}$, with the extra property that f_α is efficiently inverted once given as auxiliary input a "trapdoor" for the index α. The trapdoor of index α, denoted by $t(\alpha)$, *can not* be efficiently computed from α, yet one can efficiently generate corresponding pairs $(\alpha, t(\alpha))$.

Definition 1.2.3 (collection of trapdoor permutations): *A* collection of permutations*, with* indices *in* $I \subseteq \{0,1\}^*$*, is a set* $\{f_\alpha : D_\alpha \mapsto D_\alpha\}_{\alpha\in I}$ *so that each* f_α *is 1-1 on the corresponding* D_α*. Such a collection is called a* trapdoor permutation *if there exists 4 probabilistic polynomial-time algorithms* G, D, F, F^{-1} *so that the following five conditions hold.*

1. (index and trapdoor generation): *For every* n,

$$\mathsf{Pr}[G(1^n) \in I \times \{0,1\}^*] \; > \; 1 - 2^{-n}$$

2. (sampling the domain): *For every* $\alpha \in I$,
   (a) $\mathsf{Pr}[D(\alpha) \in D_\alpha] \; > \; 1 - 2^{-n}$. *Thus, without loss of generality,* $D_\alpha \subseteq \{0,1\}^{\mathrm{poly}(|\alpha|)}$.
   (b) *Conditioned on* $D(\alpha) \in D_\alpha$*, the output is uniformly distributed in* D_α*. That is, for every* $x \in D_\alpha$,

$$\mathsf{Pr}[D(\alpha) = x \,|\, D(\alpha) \in D_\alpha] \; = \; \frac{1}{|D_\alpha|}$$

3. (efficient evaluation): *For every* $\alpha \in I$ *and* $x \in D_\alpha$,

$$\Pr[F(\alpha, x) = f_\alpha(x)] \;>\; 1 - 2^{-n}$$

4. (hard to invert): *For every family of polynomial-size circuits,* $\{C_n\}_{n\in\mathbb{N}}$*, every positive polynomial* $p(\cdot)$*, and all sufficiently large* n*'s*

$$\Pr\left[C_n(f_{I_n}(X_n), I_n) = X_n\right] \;<\; \frac{1}{p(n)}$$

where I_n *is a random variable describing the distribution of the first element in the output of* $G(1^n)$*, and* X_n *is uniformly distributed in* D_{I_n}*.*

5. (inverting with trapdoor): *For every pair* (α, t) *in the range of* G*, and every* $x \in D_\alpha$*,*

$$\Pr[F^{-1}(t, f_\alpha(x)) = x] \;>\; 1 - 2^{-n}$$

We mention that trapdoor permutations can be constructed based on the Intractability of Factoring Assumption (or more precisely the infeasibility of factoring Blum integers; that is, the products of two primes each congruent to 3 mod 4). Any collection as above can be modified to have a (uniform) *hard-core predicate* (cf., [39]); that is, a Boolean function which is easy to compute but hard to predict from the image of the input under the permutation.

Definition 1.2.4 (hard-core for a collection of trapdoor permutations): *Let* $\{f_\alpha : D_\alpha \mapsto D_\alpha\}_{\alpha\in I}$ *be a collection of trapdoor permutations as above. We say that* $b : \{0,1\}^* \mapsto \{0,1\}$ *if a* hard-core *for this collection if the following two conditions hold.*

1. (efficient evaluation): *There exists a polynomial-time algorithm which on input* x *returns* $b(x)$*.*

2. (hard to predict): *For every family of polynomial-size circuits,* $\{C_n\}_{n\in\mathbb{N}}$*, every positive polynomial* $p(\cdot)$*, and all sufficiently large* n*'s*

$$\Pr\left[C_n(f_{I_n}(X_n), I_n) = b(X_n)\right] \;<\; \frac{1}{2} + \frac{1}{p(n)}$$

where I_n *is a random variable describing the distribution of the first element in the output of* $G(1^n)$*, and* X_n *is uniformly distributed in* D_{I_n}*.*

Commitment schemes. For simplicity of exposition, we utilize a stringent notion of a *commitment scheme* – for more general definition see [36]. Loosely speaking, here a commitment scheme is a randomized process which maps a single bit into a bit-string so that (1) the set of possible images of the bit 0 is disjoint from the set of possible images of the bit 1, and yet (2) the commitment to 0 is computationally indistinguishable from the commitment to 1.

Definition 1.2.5 (commitment scheme): *A* commitment scheme *is a uniform family of probabilistic polynomial-size circuits,* $\{C_n\}$*, satisfying the following two conditions.*

1. (perfect unambiguity): *For every* n *the supports of* $C_n(0)$ *and* $C_n(1)$ *are disjoint.*

2. (computational secrecy): *The probability ensembles* $\{C_n(0)\}_{n\in\mathbb{N}}$ *and* $\{C_n(1)\}_{n\in\mathbb{N}}$ *are computationally indistinguishable.*

We denote by $C_n(b,r)$ the output of C_n on input bit b using the random sequence r. Thus, the first item can be reformulated as asserting that for every $n \in \mathbb{N}$ and every $r, s \in \{0,1\}^*$, it holds that $C_n(0,r) \neq C_n(1,s)$.

Commitment schemes can be constructed given any 1-1 one-way function (and in particular given a trapdoor permutation).

1.2.2 Two-party and multi-party protocols

Two-party protocols may be defined as pairs of *interactive* Turing machines (cf., [36]). However, we prefer to use the intuitive notion of a two-party game. This in turn corresponds to the standard message-passing model.

For multi-party protocols we use a *synchronous* model of communication. For simplicity we consider a model in which each pair of parties is connected by a reliable and private (or secret) channel. The issues involved in providing such channels are beyond the scope of this exposition. Some of them – like establishing secret communication over insecure communication lines (i.e., by using encryption schemes), establishing party's identification, and maintaining authenticity of communication – are well-understood (even in case the search for more efficient solutions is very active). In general, as the current exposition does not aim at efficiency (but rather at establishing feasibility) the issue of *practical* emulation of our idealized communication model over a realistic one (rather then the mere feasibility of such emulation) is irrelevant.

To simplify the exposition of some constructions of multi-party protocols (in Section 3.3), we will augment the communication model by a broadcast channel on which each party can send a message which arrives to all parties (together with the sender identity). We assume, without loss of generality, that in every communication round only one (predetermined) party sends messages. Such a broadcast channel can be implemented via an (authenticated) Byzantine Agreement protocol, thus providing an emulation of our model on a more standard one (in which a broadcast channel does not exist).

1.2.3 Strong Proofs of Knowledge

Of the standard definitions of proofs of knowledge, the one most suitable for our purposes is the definition which appears in [5, 36]. (Other definitions, such as of [63, 32], are not adequate at all; see discussion in [5].) However, the definition presented in [5, 36], relies in a fundamental way on the notion of *expected* running-time. We thus prefer the following more stringent definition in which the knowledge extractor is required to run in *strict* polynomial-time (rather than in *expected* polynomial-time).

Definition 1.2.6 (System of strong proofs of knowledge): *Let R be a binary relation. We say that an efficient strategy V is a* strong knowledge verifier for the relation *R if the following two conditions hold.*

- Non-triviality: *There exists an interactive machine P so that for every $(x, y) \in R$ all possible interactions of V with P on common-input x and auxiliary-input y are accepting.*
- Strong Validity: *There exists a negligible function $\mu : \mathbb{N} \mapsto [0,1]$ and a probabilistic* (strict) polynomial-time *oracle machine K such that for every strategy P and every $x, y, r \in \{0,1\}^*$, machine K satisfies the following condition:*

Let $P_{x,y,r}$ be a prover strategy, in which the common input x, auxiliary input y and random-coin sequence r have been fixed, and denote by $p(x)$ the probability that the interactive machine V accepts, on input x, when interacting with the prover specified by $P_{x,y,r}$. Now, if $p(x) > \mu(|x|)$ then, on input x and access to oracle $P_{x,y,r}$, with probability at least $1 - \mu(|x|)$, machine K outputs a solution s for x. That is,[2]

$$\text{If } p(x) > \mu(|x|) \text{ then } \Pr[(x, K^{P_{x,y,r}}(x)) \in R] > 1 - \mu(|x|) \tag{1.2}$$

The oracle machine K is called a strong knowledge extractor*.*

An interactive pair (P, V) so that V is a strong knowledge verifier for a relation R and P is a machine satisfying the non-triviality condition (with respect to V and R) is called a system for strong proofs of knowledge *for the relation R.*

Some zero-knowledge proof (of knowledge) systems for NP are in fact strong proofs of knowledge. In particular, consider n sequential repetitions of the following basic proof system for the *Hamiltonian Cycle* (HC) problem (which is NP-complete). We consider directed graphs (and the existence of directed Hamiltonian cycles), and employ a commitment scheme $\{C_n\}$ as above.

Construction 1.2.7 (Basic proof system for HC):

- Common Input: *a directed graph $G = (V, E)$ with $n \stackrel{\text{def}}{=} |V|$.*
- Auxiliary Input to Prover: *a directed Hamiltonian Cycle, $C \subset E$, in G.*
- Prover's first step (P1): *The prover selects a random permutation, π, of the vertices of G, and commits to the entries of the adjacency matrix of the resulting permuted graph. That is, it sends an n-by-n matrix of commitments so that the $(\pi(i), \pi(j))^{\text{th}}$ entry is $C_n(1)$ if $(i, j) \in E$, and $C_n(0)$ otherwise.*
- Verifier's first step (V1): *The verifier uniformly selects $\sigma \in \{0, 1\}$ and sends it to the prover.*
- Prover's second step (P2): *If $\sigma = 0$ then the prover sends π to the verifier along with the revealing (i.e., preimages) of all n^2 commitments. Otherwise, the prover reveals to the verifier only the commitments to n entries $(\pi(i), \pi(j))$ with $(i, j) \in C$.* (By revealing a commitment c, we mean supply a preimage of c under C_n; that is, a pair (σ, r) so that $c = C_n(\sigma, r)$.)
- Verifier's second step (V2): *If $\sigma = 0$ then the verifier checks that the revealed graph is indeed isomorphic, via π, to G. Otherwise, the verifier just checks that all revealed values are 1 and that the corresponding entries form a simple n-cycle. (Of course in both cases, the verifier checks that the revealed values do fit the commitments.) The verifier accepts if and only if the corresponding condition holds.*

The reader may easily verify that sequentially repeating the above for n times yields a zero-knowledge proof system for HC, with soundness error 2^{-n}. We argue that the resulting system is also a strong proof of knowledge of the Hamiltonian cycle. Intuitively, the key observation is that each application

[2] Our choice to bound the failure probability of the extractor by $\mu(|x|)$ is rather arbitrary. What is important is to have this failure probability be a negligible function of $|x|$. Actually, in case membership in the relation R can be determined in polynomial-time, one may reduce the failure probability from $1 - \frac{1}{\text{poly}(n)}$ to $2^{-\text{poly}(n)}$, while maintaining the polynomial running-time of the extractor.

of the basic proof system results in one of two possible situations depending on the verifier choice, σ. In case the prover answers correctly in both cases, we can retrieve an Hamiltonian cycle in the input graph. On the other hand, in case the prover fails in both cases, the verifier will reject regardless of what the prover does from this point on. This observation suggests the following construction of a strong knowledge extractor (where we refer to repeating the basic proof systems n times and set $\mu(n) = 2^{-n}$).

Strong knowledge extractor for Hamiltonian cycle: On input G and access to the prover-strategy oracle P^*, we proceed in n iterations, starting with $i = 1$. Initially, T (the transcript so far), is empty.

1. Obtain the matrix of commitments, M, from the prover strategy (i.e., $M = P^*(T)$).
2. Extract the prover's answer to both possible verifier moves. Each of these answers may be correct (i.e., passing the corresponding verifier check) or not.
3. If both answers are correct then we recover a Hamiltonian cycle. In this case the extractor outputs the cycle and halts.
4. In case a single answer, say the one for value σ, is correct and $i < n$, we let $T \leftarrow (T, \sigma)$, and proceed to the next iteration (i.e., $i \leftarrow i + 1$). Otherwise, we halt with no output.

It can be easily verified that if the extractor halts with no output in iteration $i < n$ then the verifier (in the real interaction) accepts with probability zero. Similarly, if the extractor halts with no output in iteration n then the verifier (in the real interaction) accepts with probability 2^{-n}. Thus, whenever $p(G) > 2^{-n}$, the extractor succeeds in recovering a Hamiltonian cycle (with probability 1).

Chapter 2

General Two-Party Computation

Our ultimate goal is to design two-party protocols which may withstand any feasible adversarial behavior. We proceed in two steps. First we consider a benign type of adversary, called *semi-honest*, and construct protocols which are secure with respect to such an adversary. Next, we show how to force parties to behave in a semi-honest manner. That is, we show how to transform any protocol, secure in the semi-honest model, into a protocol which is secure against any feasible adversarial behavior.

We note that the semi-honest model is not merely an important methodological locus, but may also provide a good model of certain settings.

Organization In Section 2.1 we define the framework for the entire chapter. In particular, we define two-party functionalities and some simplifying assumptions, the semi-honest model (see Section 2.1.1) and the general malicious model (see Section 2.1.2). In Section 2.2 we describe the construction of protocols for the semi-honest model, and in Section 2.3 a compiler which transforms protocols from the latter model to protocols secure in the general malicious model.

2.1 Definitions

A two-party protocol problem is casted by specifying a random process which maps pairs of inputs (one input per each party) to pairs of outputs (one per each party). We refer to such a process as the desired functionality, denoted $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$. That is, for every pair of inputs (x,y), the desired output-pair is a random variable, $f(x,y)$, ranging over pairs of strings. The first party, holding input x, wishes to obtain the first element in $f(x,y)$; whereas the second party, holding input y, wishes to obtain the second element in $f(x,y)$.

A special case of interest is when both parties wish to obtain a predetermined function, g, of the two inputs. In this case we have

$$f(x,y) \stackrel{\rm def}{=} (g(x,y), g(x,y))$$

Another case of interest is when the two parties merely wish to toss a fair coin. This case can be casted by requiring that, for every input pair (x,y), we have $f(x,y)$ uniformly distributed over $\{(0,0),(1,1)\}$. Finally, as a last example, we mention highly asymmetric functionalities of the form

$f(x,y) \stackrel{\rm def}{=} (f'(x,y),\lambda)$, where $f'\{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$ is a randomized process and λ denotes the empty string.

Whenever we consider a protocol for securely computing f, it is implicitly assumed that the protocol is correct provided that both parties follow the prescribed program. That is, the joint output distribution of the protocol, played by honest parties, on input pair (x,y), equals the distribution of $f(x,y)$.

Simplifying conventions. To simplify the exposition we make the following three assumptions:

1. *The protocol problem has to be solved only for inputs of the same length* (i.e., $|x| = |y|$).
2. *The functionality is computable in time polynomial in the length of the inputs.*
3. *Security is measured in terms of the length of the inputs.*

The above conventions can be greatly relaxed, yet each represent an essential issue which must be addressed.

Observe that making no restriction on the relationship among the lengths of the two inputs, disallows the existence of secure protocols for computing any "non-degenerate" functionality. The reason is that the program of each party (in a protocol for computing the desired functionality) must *either* depend only on the length of the party's input *or* obtain information on the counterpart's input length. In case information of the latter type is not implied by the output value, a secure protocol "cannot afford" to give it away.[1] An alternative to the above convention is to restrict the class of functionalities to such where the length of each party's input is included in the counterpart's output. One can easily verify that the two alternative conventions are in fact equivalent.

We now turn to the second convention (assumption). Certainly, the total running-time of a (secure) two-party protocol for computing the functionality cannot be smaller than the time required to compute the functionality (in the ordinary sense). Arguing as above, one can see that we need an a-priori bound on the complexity of the functionality. A more general approach would be to have this bound given explicitly to both parties as an auxiliary input. In such a case, the protocol can be required to run for time bounded by a fixed polynomial in this auxiliary parameter (i.e., the time-complexity bound of f). Using standard padding and assuming that a good upper bound of the complexity of f is time-constructible, we can reduce this general case to the special case discussed above: Given a general functionality, g, and a time bound $t : \mathbb{N} \mapsto \mathbb{N}$, we introduce the functionality

$$f((x,1^i),(y,1^j)) \stackrel{\rm def}{=} \begin{cases} g(x,y) & \text{if } i=j=t(|x|)=t(|y|) \\ (\perp,\perp) & \text{otherwise} \end{cases}$$

where $\perp$ is a special symbol. Now, the problem of securely computing g reduces to the problem of securely computing f.

Finally, we turn to the third convention (assumption). Indeed, a more general convention would be to have a security parameter which determines the security of the protocol. This general alternative is essential for allowing "secure" computation of finite functionalities (i.e., functionalities defined on finite input domains). We may accommodate the general convention using the special case, postulated above, as follows. Suppose that we want to compute the functionality f, on input pair (x,y) with security (polynomial in) the parameter s. Then we introduce the functionality

$$f'((x,1^s),(y,1^s)) \stackrel{\rm def}{=} f(x,y) \ ,$$

[1] The situation is analogous to the definition of secure encryption, where it is required that the message length be polynomially-related to the key length.

and consider secure protocols for computing f'. Indeed, this reduction corresponds to the realistic setting where the parties first agree on the desired level of security and then proceed to compute the function using this level (of security).

The first convention, revisited. An alternative way of postulating the first convention is to consider only functionalities, $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$, which satisfy $f(x,y) = (\perp, \perp)$ whenever $|x| \neq |y|$. That is, such functionalities have the form

$$f(x,y) \stackrel{\rm def}{=} \begin{cases} f'(x,y) & \text{if } |x| = |y| \\ (\perp,\perp) & \text{otherwise} \end{cases}$$

where f' is an arbitrary functionality. Actually, in some cases it will be more convenient to consider functionalities of arbitrary length relationship, determined by a 1-1 function $\ell : \mathbb{N} \mapsto \mathbb{N}$. Such functionalities have the form

$$f(x,y) \stackrel{\rm def}{=} \begin{cases} f'(x,y) & \text{if } |x| = \ell(|y|) \\ (\perp,\perp) & \text{otherwise} \end{cases}$$

where f' is an arbitrary functionality. Even more generally, we may consider functionalities which are meaningfully defined only for input pairs satisfying certain (polynomial-time computable) relations. Let $R \subseteq \cup_{n \in \mathbb{N}}(\{0,1\}^{\ell(n)} \times \{0,1\}^n)$ be such a relation and f' be as above, then we may consider the functionality

$$f(x,y) \stackrel{\rm def}{=} \begin{cases} f'(x,y) & \text{if } (x,y) \in R \\ (\perp,\perp) & \text{otherwise} \end{cases}$$

2.1.1 The semi-honest model

Loosely speaking, a semi-honest party is one who follows the protocol properly with the exception that it keeps a record of all its intermediate computations. Actually, it suffices to keep the internal coin tosses and all messages received from the other party. In particular, a semi-honest party tosses fair coins (as instructed by its program), and sends messages according to its specified program (i.e., as a function of its input, outcome of coin tosses, and incoming messages). Note that a semi-honest party corresponds to the "honest verifier" in definitions of zero-knowledge.

In addition to the role of honest-parties in our exposition, they do constitute a model of independent interest. In particular, in reality deviating from the specified program – which may be invoked inside a complex application software – may be more difficult than merely recording the contents of some communication registers. Furthermore, records of these registers may be available through some standard activities of the operating system. Thus, whereas totally-honest behavior (rather than semi-honest one) may be hard to enforce, semi-honest behavior may be assumed in many settings.

The semi-honest model is implicit in the following definition of privacy. Loosely speaking, the definition says that a protocol *privately computes* f if whatever a semi-honest party can be obtained after participating in the protocol, could be essentially obtained from the input and output available to that party. This is stated using the simulation paradigm. Furthermore, it suffices to (efficiently) "simulate the view" of each (semi-honest) party, since anything which can be obtain after participating in the protocol is obtainable from the view.

Definition 2.1.1 (privacy w.r.t semi-honest behavior): *Let* $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$ *be a functionality, where* $f_1(x,y)$ (resp., $f_2(x,y)$) *denotes the first* (resp., second) *element of* $f(x,y)$, *and* Π *be a two-party protocol for computing* f.[2] *The* view of the first (resp., second) party *during an execution of* Π *on* (x,y)*, denoted* $\text{VIEW}_1^{\Pi}(x,y)$ (resp., $\text{VIEW}_2^{\Pi}(x,y)$)*, is* $(x,r,m_1,...,m_t)$ (resp., $(y,r,m_1,...,m_t)$)*, where* r *represent the outcome of the first* (resp., second) *party's internal coin tosses, and* m_i *represent the* i^{th} *message it has received. We say that* π privately computes f *if there exist polynomial-time algorithms, denoted* S_1 *and* S_2*, such that*

$$\{S_1(x,f_1(x,\mathsf{y}(x)))\}_{x\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \{\text{VIEW}_1^{\Pi}(x,\mathsf{y}(x))\}_{x\in\{0,1\}^*} \tag{2.1}$$

$$\{S_2(y,f_2(\mathsf{x}(y),y))\}_{y\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \{\text{VIEW}_2^{\Pi}(\mathsf{x}(y),y)\}_{y\in\{0,1\}^*} \tag{2.2}$$

where $\mathsf{x},\mathsf{y}:\{0,1\}^*\mapsto\{0,1\}^*$ *are arbitrary length preserving functions.*

Eq. (2.1) (resp., Eq. (2.2)) asserts that the view of the first (resp., second) party, on each possible input, can be efficiently simulated based solely on its input and output. The functions x,y are used to specify a matched input for the counterpart, per each input considered for the party we simulate.[3]

Alternative formulation. It is instructive to recast the above definitions in terms of the general ("ideal-vs-real") framework discussed in Section 1.1 and used extensively in the case of arbitrary malicious behavior. In this framework we first consider an ideal model in which the (two) real parties are joined by a (third) trusted party, and the computation is performed via this trusted party. Next one considers the real model in which a real (two-party) protocol is executed (and there exist no trusted third parties). A protocol in the real model is said to be *secure with respect to certain adversarial behavior* if the possible real executions with such an adversary can be "simulated" in the ideal model. The notion of simulation here is different than above: The simulation is not of the view of one party via a traditional algorithm, but rather a simulation of the joint view of both parties by the execution of an ideal model protocol.

According to the general methodology (framework), we should first specify the ideal model protocol. Here, it consists of each party sending its input to the trusted party (via a secure private channel), the third party computing the corresponding output-pair and sending each output to the corresponding party. The only adversarial behavior allowed here is for one of the parties to conduct an arbitrary polynomial-time computation based on its input and the output it has received. The other party merely outputs the output it has received.[4] Next, we turn to the real model. Here, there is a two-party protocol and the adversarial behavior is restricted to be semi-honest. That is, one party may conduct an arbitrary polynomial-time computation based on its view of the execution (as defined above). A secure protocol in the (real) semi-honest model is such that for every semi-honest behavior of one of the parties, we can simulate the joint outcome (of their computation) by an execution in the ideal model (where also one party is semi-honest and the other is honest). Actually, we need to augment the definition so to account for a-priori information available to semi-honest parties before the protocol starts. This is done by supplying these parties with auxiliary inputs, or equivalently by viewing them as possibly non-uniform circuits of polynomial-size. Thus, we have –

[2] By saying that Π computes (rather than privately computes) f, we mean that the output distribution of the protocol (when played by honest or semi-honest parties) on input pair (x,y) is identically distributed as $f(x,y)$.

[3] Observe the analogy to the definition of a zero-knowledge protocol (w.r.t honest verifier): The functionality (in this case) is a function $f(x,y)=(\lambda,(x,\chi_L(x)))$, where χ_L is the characteristic function of the language L, the first party is playing the prover, and Π is a zero-knowledge interactive proof for L (augmented by having the prover send $(x,\chi_L(x))$ and abort in case $x\notin L$). Note that the above functionality allows the prover to send x to the verifier which ignores its own input (i.e., y). The standard zero-knowledge condition essentially asserts Eq. (2.2), and Eq. (2.1) holds by the definition of an interactive proof (i.e., specifically, by the requirement that the verifier is polynomial-time).

[4] Thus, unless the party's output incorporates the party's input, this input is not available to an honest party after the computation.

Definition 2.1.2 (security in the semi-honest model): *Let $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$ be a functionality, where $f_1(x,y)$* (resp., $f_2(x,y)$) *denotes the first* (resp., second) *element of $f(x,y)$, and Π be a two-party protocol for computing f.*

- *Let $\overline{C} = (C_1, C_2)$ be a pair of polynomial-size circuit families representing adversaries in the ideal model. Such a pair is* admissible (in the ideal model) *if for at least one C_i we have $C_i(I,O) = O$. The* joint execution under $\overline{C}$ in the ideal model on input pair (x,y), *denoted* $\text{IDEAL}_{f,\overline{C}}(x,y)$, *is defined as* $(C_1(x, f_1(x,y)), C_2(y, f_2(x,y)))$.
  (That is, C_i is honest – it just outputs $f_i(x,y)$).
- *Let $\overline{C} = (C_1, C_2)$ be a pair of polynomial-size circuit families representing adversaries in the real model. Such a pair is* admissible (in the real model) *if for at least one $i \in \{1,2\}$ we have $C_i(V) = O$, where O is the output implicit in the view V. The* joint execution of Π under $\overline{C}$ in the real model on input pair (x,y), *denoted* $\text{REAL}_{\Pi,\overline{C}}(x,y)$, *is defined as* $(C_1(\text{VIEW}_1^{\Pi}(x,y)), C_2(\text{VIEW}_2^{\Pi}(x,y)))$.
  (Again, C_i is honest – it just outputs $f_i(x,y)$).

Protocol Π is said to securely compute f in the semi-honest model (secure w.r.t f and semi-honest behavior) *if there exists a polynomial-time computable transformation of pairs of admissible polynomial-size circuit families $\overline{A} = (A_1, A_2)$ for the real model into pairs of admissible polynomial-size circuit families $\overline{B} = (B_1, B_2)$ for the ideal model so that*

$$\{\text{IDEAL}_{f,\overline{B}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|}$$

Observe that the definition of the joint execution in the real model prohibits both parties (honest and semi-honest) to deviate from the strategies specified by Π. The difference between honest and semi-honest is merely in their actions on the corresponding local views of the execution: An honest party outputs only the output-part of the view (as specified by Π), whereas a semi-honest party may output an arbitrary (feasibly computable) function of the view.

It is not hard to see that Definitions 2.1.1 and 2.1.2 are equivalent. That is,

Proposition 2.1.3 *Let Π be a protocol for computing f. Then, Π privately computes f if and only if Π securely computes f in semi-honest model.*

Proof Sketch: Suppose first that Π securely computes f in semi-honest model (i.e., satisfies Definition 2.1.2). Without loss of generality, we show how to simulate the first party view. We define the following admissible adversary $\overline{A} = (A_1, A_2)$ for the real model: A_1 is merely the identity transformation and A_2 maps its view to the corresponding output (as required by definition of an admissible pair). Let $\overline{B} = (B_1, B_2)$ be the ideal-model adversary guaranteed by Definition 2.1.2. Then, B_1 (in role of S_1) satisfies Eq. (2.1). Note that B_1 is polynomial-time computable from the circuit families A_1, A_2, which in turn are uniform. Thus, the simulation is via a uniform algorithm as required.

Now, suppose that Π privately computes f, and let S_1 and S_2 be as guaranteed in Definition 2.1.1. Let $\overline{A} = (A_1, A_2)$ be an admissible pair for the real-model adversaries. Without loss of generality, we assume that A_2 merely maps the view (of the second party) to the corresponding output (i.e.,

$f_2(x,y))$. Then, we define $\overline{B} = (B_1, B_2)$ so that $B_1(x,z) \stackrel{\rm def}{=} A_1(S_1(x,z))$ and $B_2(y,z) \stackrel{\rm def}{=} z$. Clearly, $\overline{B}$ can be constructed in polynomial-time given $\overline{A}$, and the following holds

$$\begin{aligned}
\text{REAL}_{\Pi,\overline{A}}(x,y) &= (A_1(\text{VIEW}_1^{\Pi}(x,y)), A_2(\text{VIEW}_2^{\Pi}(x,y))) \\
&\stackrel{\rm c}{\equiv} (A_1(S_1(x,f_1(x,y)), f_2(x,y)) \\
&= (B_1(x,f_1(x,y)), B_2(y,f_2(x,y))) \\
&= \text{IDEAL}_{f,\overline{B}}(x,y)
\end{aligned}$$

The above is inaccurate (in its treatment of computational indistinguishability), however, a precise proof can be easily derived following standard paradigms (of dealing with computationally indistinguishable ensembles). ■

Conclusion: The above proof demonstrates that the alternative formulation of Definition 2.1.2 is merely a cumbersome form of the simpler Definition 2.1.1. We stress again that the reason we have presented the cumbersome form is the fact that it follows the general framework of definitions of security which is used for less benign adversarial behavior. In the rest of this chapter, whenever we deal with the semi-honest model (for two-party computation), we will used Definition 2.1.1.

2.1.2 The malicious model

We now turn to consider arbitrary feasible deviation of parties from a specified two-party protocol. A few preliminary comments are in place. Firstly, there is no way to force parties to participate in the protocol. That is, a possible malicious behavior may consists of not starting the execution at all, or, more generally, suspending (or aborting) the execution in any desired point in time. In particular, a party can abort at the first moment when it obtains the desired result of the computed functionality. We stress that our model of communication does not allow to condition the receipt of a message by one party on the *concurrent* sending of a proper message by this party. Thus, no two-party protocol can prevent one of the parties to abort when obtaining the desired result and before its counterpart also obtains the desired result. In other words, it can be shown that perfect fairness – in the sense of both parties obtaining the outcome of the computation concurrently – is not achievable in two-party computation. We thus give up on such fairness altogether. (We comment that partial fairness is achievable, but postpone the discussion of this issue to a later chapter.)

Another point to notice is that there is no way to talk of the *correct input* to the protocol. That is, a party can alway modify its local input, and there is no way for a protocol to prevent this. (We stress that both phenomena did not occur in the semi-honest model, for the obvious reason that parties were postulated not to deviate from the protocol.)

To summarize, there are three things we cannot hope to avoid.

1. Parties refusing to participate in the protocol (when the protocol is first invoked).

2. Parties substituting their local input (and entering the protocol with an input other than the one provided to them).

3. Parties aborting the protocol prematurely (e.g., before sending their last message).

The ideal model. We now translate the above discussion into a definition of an ideal model. That is, we will allow in the ideal model whatever cannot be possibly prevented in any real execution. An alternative way of looking at things is that we assume that the the two parties have at their disposal a trusted third party, but even such a party cannot prevent specific malicious behavior. Specifically, we allow a malicious party in the ideal model to refuse to participate in the protocol or to substitute its local input. (Clearly, neither can be prevent by a trusted third party.) In addition, we postulate that the *first* party has the option of "stopping" the trusted party just after obtaining its part of the output, and before the trusted party sends the other output-part to the second party. Such an option is not given to the second party.[5] Thus, an execution in the ideal model proceeds as follows (where all actions of the both honest and malicious party must be feasible to implement).

Inputs: Each party obtains an input, denoted z.

Send inputs to trusted party: An honest party always sends z to the trusted party. A malicious party may, depending on z, either abort or sends some $z' \in \{0,1\}^{|z|}$ to the trusted party.

Trusted party answers first party: In case it has obtained an input pair, (x, y), the trusted party (for computing f), first replies to the first party with $f_1(x, y)$. Otherwise (i.e., in case it receives only one input), the trusted party replies to both parties with a special symbol, $\perp$.

Trusted party answers second party: In case the first party is malicious it may, depending on its input and the trusted party answer, decide to *stop* the trusted party. In this case the trusted party sends $\perp$ to the second party. Otherwise (i.e., if not stopped), the trusted party sends $f_2(x, y)$ to the second party.

Outputs: An honest party always outputs the message it has obtained from the trusted party. A malicious party may output an arbitrary (polynomial-time computable) function of its initial input and the message it has obtained from the trusted party.

The ideal model computation is captured in the following definition.[6]

Definition 2.1.4 (malicious adversaries, the ideal model): *Let $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$ be a functionality, where $f_1(x, y)$* (resp., $f_2(x, y)$) *denotes the first* (resp., second) *element of $f(x, y)$. Let $\overline{C} = (C_1, C_2)$ be a pair of polynomial-size circuit families representing adversaries in the ideal model. Such a pair is* admissible (in the ideal malicious model) *if for at least one $i \in \{1, 2\}$ we have $C_i(I) = I$ and $C_i(I, O) = O$. The* joint execution under $\overline{C}$ in the ideal model (on input pair (x, y))*, denoted $\text{IDEAL}_{f,\overline{C}}(x, y)$, is defined as follows*

- *In case $C_2(I) = I$ and $C_2(I, O) = O$* (i.e., Party 2 is honest),

$$\begin{array}{cll} (C_1(x, \perp)\,,\, \perp) & \text{if } C_1(x) = \perp & (2.3) \\ (C_1(x, f_1(C_1(x), y), \perp)\,,\, \perp) & \text{if } C_1(x) \neq \perp \text{ and } C_1(x, f_1(C_1(x), y)) = \perp & (2.4) \\ (C_1(x, f_1(C_1(x), y))\,,\, f_2(C_1(x), y)) & \text{otherwise} & (2.5) \end{array}$$

[5] This asymmetry is due to the non-concurrent nature of communication in the model. Since we postulate that the trusted party sends the answer first to the first party, the first party (but not the second) has the option to stop the third party after obtaining its part of the output. The second party, can only stop the third party before obtaining its output, but this is the same as refusing to participate.

[6] In the definition, the circuits C_1 and C_2 represent all possible actions in the model. In particular, $C_1(x) = \perp$ represents a decision of Party 1 not to enter the protocol at all. In this case $C_1(x, \perp)$ represents its local-output. The case $C_1(x) \neq \perp$, represents a decision to hand an input, denoted $C_1(x)$, to the trusted party. Likewise, $C_1(x, z)$ and $C_1(x, z, \perp)$, where z is the answer supplied by the trusted party, represents the actions taken by Party 1 after receiving the trusted party answer.

- *In case* $C_1(I) = I$ *and* $C_1(I,O) = O$ (i.e., Party 1 is honest),

$$\begin{array}{lll} (\perp\,,\,C_2(y,\perp)) & \text{if } C_2(y)=\perp & (2.6) \\ (f_1(x,y)\,,\,C_2(y,f_2(x,C_2(y))) & \text{otherwise} & (2.7) \end{array}$$

Eq. (2.3) represents the case where Party 1 aborts before invoking the trusted party (and outputs a string which only depends on its input; i.e., x). Eq. (2.4) represents the case where Party 1 invokes the trusted party with a possibly substituted input, denoted $C_1(x)$, and aborts while stopping the trusted party right after obtaining the output, $f_1(C_1(x),y)$. In this case the output of Party 1 depends on both its input and the output it has obtained from the trusted party. In both these cases, Party 2 obtains no output (from the trusted party). Eq. (2.5) represents the case where Party 1 invokes the trusted party with a possibly substituted input, and allows the trusted party to answer to both parties (i.e., 1 and 2). In this case, the trusted party computes $f(C_1(x),y)$, and Party 1 outputs a string which depends on both x and $f_1(C(x),y)$. Likewise, Eq. (2.6) and Eq. (2.7) represent malicious behavior of Party 2; however, in accordance to the above discussion, the trusted party first supplies output to Party 1 and so Party 2 does not have an option analogous to Eq. (2.4).

Execution in the real model. We next consider the real model in which a real (two-party) protocol is executed (and there exist no trusted third parties). In this case, a malicious party may follow an arbitrary feasible strategy; that is, any strategy implementable by polynomial-size circuits. In particular, the malicious party may abort the execution at any point in time, and when this happens prematurely, the other party is left with no output. In analogy to the ideal case, we use circuits to define strategies in a protocol.

Definition 2.1.5 (malicious adversaries, the real model): *Let f be as in Definition 2.1.4, and Π be a two-party protocol for computing f. Let $\overline{C} = (C_1, C_2)$ be a pair of polynomial-size circuit families representing adversaries in the real model. Such a pair is* admissible (w.r.t Π) (for the real malicious model) *if at least one C_i coincides with the strategy specified by Π. The* joint execution of Π under $\overline{C}$ in the real model (on input pair (x,y)), *denoted* $\text{REAL}_{\Pi,\overline{C}}(x,y)$, *is defined as the output pair resulting of the interaction between $C_1(x)$ and $C_2(y)$.*

In the sequel, we will assume that the circuit representing the real-model adversary (i.e., the C_i which does not follow Π) is deterministic. This is justified by standard techniques: See discussion following Definition 2.1.6.

Security as emulation of real execution in the ideal model. Having defined the ideal and real models, we obtain the corresponding definition of security. Loosely speaking, the definition asserts that a secure two-party protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated by saying that admissible adversaries in the ideal-model are able to simulate (in the ideal-model) the execution of a secure real-model protocol (with admissible adversaries).

Definition 2.1.6 (security in the malicious model): *Let f and Π be as in Definition 2.1.5, Protocol Π is said to* securely compute f (in the malicious model) *if there exists a polynomial-time computable transformation of pairs of admissible polynomial-size circuit families $\overline{A} = (A_1, A_2)$ for the real model* (of Definition 2.1.5) *into pairs of admissible polynomial-size circuit families* $\overline{B} = (B_1, B_2)$ *for the ideal model* (of Definition 2.1.4) *so that*

$$\{\text{IDEAL}_{f,\overline{B}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|}$$

When the context is clear, we sometimes refer to Π *as an* implementation *of* f.

Implicit in Definition 2.1.6 is a requirement that in a non-aborting (real) execution of a secure protocol, each party "knows" the value of the corresponding input on which the output is obtained. This is implied by the equivalence to the ideal model, in which the party explicitly hands the (possibly modified) input to the trusted party. For example, say Party 1 uses the malicious strategy A_1 and that $\text{REAL}_{\Pi,\overline{A}}(x,y)$ is non-aborting. Then the output values correspond to the input pair $(B_1(x), y)$, where B_1 is the ideal-model adversary derived from the real-model adversarial strategy A_1.

Justification for considering only deterministic real-adversaries. As stated above, we will assume throughout our presentation that the adversaries in the real (execution) model are deterministic. Intuitively, (non-uniform) deterministic adversaries are as powerful, with respect to breach of security, as randomized adversaries – since one may just consider (and fix) the "best possible" choice of coins for a randomized adversary. However, as the above definition of security requires to (efficiently) transform adversaries for the real model into adversaries for the ideal model, one may wonder whether a transformation which applies to deterministic adversaries necessarily applies to randomized adversaries. We claim that this is indeed the case.

Proposition 2.1.7 *Let* T *be a polynomial-time transformation applicable to single-input circuits. Then, there exists a transformation* T' *applicable to two-input circuits so that for every such circuit circuit,* $C(\cdot,\cdot)$*, and for every possible input pair,* (x,r)*, to* C*, it holds*

$$T'(C)(x,r) = T(C_r)(x)$$

where C_r *is the circuit obtained from* C *by fixing the second input to be* r*, and* $T'(C)$ (resp., $T(C_r)$) *is the two-input* (resp., one-input) *circuit obtained by applying the transformation* T' (resp., T) *to the two-input circuit* C (resp., one-input circuit C_r).

Thus, for every transformation for deterministic circuits (modeled above by single-input circuits), we can derive an "equivalent" transformation for randomized circuits (modeled above by two-input circuits).

Proof Sketch: Given a transformation T, consider the *universal function*, $f_T : (\{0,1\}^*)^3 \mapsto \{0,1\}^*$, where f_T is defined as follows, on triples (C, x, r), with C being a two-input circuit.

- Let C_r be the circuit obtained from C by fixing its second input to be r.
- Let $C' = T(C_r)$ be the circuit obtained from C_r by applying the transformation T.
- Then, $f_T(C, x, r)$ equals $C'(x)$.

Note that f_T is computable in (uniform) polynomial-time (and hence circuits computing it can be constructed in polynomial-time). Given a two-input circuit, C, the transformation T' proceeds as follows.

1. Constructs a circuit for computing f_T (on inputs of the adequate length – determined by C).

2. Fix the appropriate inputs of the above circuit to equal the bits in the description of C.

3. Output the resulting circuit, denoted $f_{T,C}$.

Note that $T'(C)(x,r) = f_{T,C}(x,r) = f_T(C,x,r) = T(C_r)(x)$, and so the claim follows. ■

2.2 Secure Protocols for the Semi-Honest Model

We present a method of constructing private protocols (w.r.t semi-honest behavior) for any given functionality. The construction takes a Boolean circuit representing the given functionality and produces a protocol for evaluating this circuit. The circuit evaluation protocol, presented in subsection 2.2.4, scans the circuit from the input wires to the output wires, processing a single gate in each *basic step*. When entering each basic step, the parties hold shares of the values of the input wires, and when the step is completed they hold shares of the output wire. Thus, evaluating the circuit "reduces" to evaluating single gates on values shared by both parties.

Our presentation is modular: We first define an appropriate notion of a reduction, and show how to derive a private protocol for functionality g, given a reduction of the (private computation of) g to the (private computation of) f together with a protocol for privately computing f. In particular, we reduce the private computation of general functionalities to the private computation of deterministic functionalities, and thus focus on the latter.

We next consider, without loss of generality, the evaluation of Boolean circuits with AND and XOR gates of fan-in 2. Actually, we find it more convenient to consider the corresponding arithmetic circuits over GF(2), where multiplication corresponds to AND and addition to XOR. A value v is shared by the two parties in the natural manner (i.e., the sum of the shares is v). Thus, proceeding through an addition gate is trivial, and we concentrate on proceeding through a multiplication gate. The generic case is that the first party holds a_1, b_1 and the second party holds a_2, b_2, where $a_1 + a_2$ is the value of one input wire and $b_1 + b_2$ is the value of the other input wire. What we want is to provide each party with a "random" share of the value of the output wire; that is, a share of the value $(a_1 + a_2) \cdot (b_1 + b_2)$. In other words we are interested in privately computing the following randomized functionality

$$
\begin{aligned}
((a_1, b_1), (a_2, b_2)) \quad \mapsto \quad & (c_1, c_2) && (2.8)\\
& \text{where } c_1 + c_2 = (a_1 + a_2) \cdot (b_1 + b_2). && (2.9)
\end{aligned}
$$

That is, (c_1, c_2) is uniformly chosen among the solutions to $c_1 + c_2 = (a_1 + a_2) \cdot (b_1 + b_2)$. The above functionality has a finite domain, and can be solved (generically) by reduction to a variant of Oblivious Transfer (OT). This variant is defined below, and it is shown that it can be implemented assuming the existence of trapdoor one-way permutations.

The actual presentation proceeds bottom-up. We first define reductions between (two-party) protocol problems (in the semi-honest model). Next we define and implement OT, show how to use it for securely computing a single multiplication gate, and finally for securely computing the entire circuit.

2.2.1 A composition theorem

It is time to define what we mean by saying that private computation of one functionality *reduces* to the private computation of another functionality. Our definition is a natural extension of the standard notion of reduction in the context of ordinary (i.e., one party) computation. Recall that standard reductions are defined in terms of oracle machines. Thus, we need to consider two-party protocols with oracle access. Here the oracle is invoked by both parties, each supplying it with one input (or query), and it responses with a pair of answers, one per each party. We stress that the answer-pair depends on the query-pair.

Definition 2.2.1 (protocols with oracle access): *A* oracle-aided protocol *is a protocol augmented by a pair of oracle-tapes, per each party, and oracle-call steps defined as follows. Each of the parties may send a special* oracle request *message, to the other party, after writing a string – called the* query *– on its write-only oracle-tape. In response, the other party writes a string, its query, on its own oracle-tape and respond to the first party with a* oracle call *message. At this point the oracle is invoked and the result is that a string, not necessarily the same, is written by the oracle on the read-only oracle-tape of each party. This pair of strings is called the* oracle answer*.*

Definition 2.2.2 (reductions):

- *An oracle-aided protocol is said to be* using the oracle-functionality f*, if the oracle answers are according to f. That is, when the oracle is invoked with first party query q_1 and second party query q_2, the answer-pair is distributed as $f(q_1, q_2)$.*
- *An oracle-aided protocol using the oracle-functionality f is said to* privately compute g *if there exist polynomial-time algorithms, denoted S_1 and S_2, satisfying* Eq. (2.1) *and* Eq. (2.2)*, respectively, where the corresponding views are defined in the natural manner.*
- *An oracle-aided protocol is said to* privately reduce g *to f, if it privately computes g when using the oracle-functionality f. In such a case we say that g* is privately reducible to f*,*

We are now ready to state a composition theorem for the semi-honest model.

Theorem 2.2.3 (Composition Theorem for the semi-honest model, two parties): *Suppose that g is privately reducible to f and that there exists a protocol for privately computing f. Then there exists a protocol for privately computing g.*

Proof Sketch: Let $\Pi^{g|f}$ be a oracle-aided protocol which privately reduces g to f, and let Π^f be a protocol which privately computes f. We construct a protocol Π for computing g in the natural manner; that is, starting with $\Pi^{g|f}$, we replace each invocation of the oracle by an execution of Π^f. Denote the resulting protocol by Π. Clearly, Π computes g. We need to show that it privately computes g.

For each $i = 1, 2$, let $S_i^{g|f}$ and S_i^f be the corresponding simulators for the view of party i (i.e., in $\Pi^{g|f}$ and Π^f, respectively). We construct a simulation S_i, for the view of party i in Π, in the natural manner. That is, we first run $S_i^{g|f}$ and obtain the view of party i in $\Pi^{g|f}$. This view includes queries made by party i and corresponding answers. (Recall, we have only the part of party i in the query-answer pair.) Invoking S_i^f on each such "partial query-answer" we fill-in the view of party i for each of these invocations of Π^f.

It is left to show that S_i indeed generates a distribution indistinguishable from the view of party i in actual executions of Π. Towards this end, we introduce an imaginary simulator, denoted I_i. This imaginary simulator invokes $S_i^{g|f}$, but augment the view of party i with views of actual executions of protocol Π^f on the corresponding query-pairs. (The query-pair is completed in an arbitrary consistent way.) Observe that the outputs of I_i and S_i are computationally indistinguishable; or else one may distinguish the distribution produced by S_i^f and the actual view of party i in Π^f (by incorporating a possible output of $S_i^{g|f}$ into the distinguisher). On the other hand, the output of I_i must be computationally indistinguishable from the view of party i in Π; or else one may distinguish the output of $S_i^{g|f}$ from the view of party i in $\Pi^{g|f}$ (by incorporating a possible view of party i in the actual execution of Π^f into the distinguisher). The theorem follows. ∎

Comment: The simplicity of the above proof is due to the fact that semi-honest behavior is rather simple. In particular, the execution of a semi-honest party in an oracle-aided protocol is not effected by the replacement of the oracle by an real subprotocol. (Note that this may not be the case when malicious parties are discussed.)

Application – reducing private computation of general functionalities to deterministic ones. Given a general functionality g, we first write it in a way which makes the randomization explicit. That is, we let $g(r,(x,y))$ denote the value of $g(x,y)$ when using coin tosses $r \in \{0,1\}^{\text{poly}(|x|)}$ (i.e., $g(x,y)$ is the randomized process consisting of uniformly selecting $r \in \{0,1\}^{\text{poly}(|x|)}$, and deterministically computing $g(r,(x,y))$). Next, we privately reduce g to f, where f is defined as follows

$$f((x_1,r_1),(x_2,r_2)) \stackrel{\text{def}}{=} g(r_1 \oplus r_2,(x_1,x_2)) \tag{2.10}$$

Applying Theorem 2.2.3, we conclude that the existence of a protocol for privately computing the deterministic functionality f implies the existence of a protocol for privately computing the randomized functionality g. For sake of future reference, we explicitly state the reduction of privately computing g to privately computing f (i.e, the oracle-aided protocol for g given f).

Proposition 2.2.4 (privately reducing a randomized functionality to deterministic one): *Let g be a randomized functionality, and f be as defined in* Eq. (2.10). *Then the following oracle-aided protocol privately reduces g to f.*

Inputs: *Party i gets input $x_i \in \{0,1\}^n$.*

Step 1: *Party i uniformly selects $r_i \in \{0,1\}^{\text{poly}(|x_i|)}$.*

Step 2: *Party i invokes the oracle with query (x_i,r_i), and records the oracle response.*

Outputs: *Each party outputs the oracle's response.*

Proof: Clearly, the above protocol, denoted Π, computes g. To show that Π privately computes g we need to present a simulator for each party view. The simulator for Party i, denoted S_i, is the obvious one. On input (x_i,v_i), where x_i is the local input to Party i and v_i is its local output, the simulator uniformly selects $r_i \in \{0,1\}^m$, and outputs (x_i,r_i,v_i), where $m = \text{poly}(|x_i|)$. To see that this output is distributed identically to the view of Party i, we note that for every fixed x_1,x_2 and $r \in \{0,1\}^m$, we have $v_i = g_i(r,(x_1,x_2))$ if and only if $v_i = f_i((x_1,r_1),(x_2,r_2))$, for any pair (r_1,r_2) satisfying $r_1 \oplus r_2 = r$. Let ζ_i be a random variable representing the random choice of Party i in Step 1, and ζ_i' denote the corresponding choice made by the simulator S_i. Then, for every fixed x_1,x_2,r_i,v_i

$$\mathsf{Pr}[\text{VIEW}_i^{\Pi}(x_1,x_2) = (x_i,r_i,v_i)] \quad = \quad \mathsf{Pr}[(\zeta_i = r_i) \wedge (f_i((x_1,\zeta_1),(x_2,\zeta_2)) = v_i)]$$

$$
\begin{aligned}
&= 2^{-m} \cdot \frac{|\{r_{3-i} : f_i((x_1, r_1), (x_2, r_2)) = v_i\}|}{2^m} \\
&= 2^{-m} \cdot \frac{|\{r : g_i(r, (x_1, x_2)) = v_i\}|}{2^m} \\
&= \mathsf{Pr}[(\zeta_i' = r_i) \wedge (g_i(x_1, x_2) = v_i)] \\
&= \mathsf{Pr}[S_i(x_i, g_i(x_1, x_2)) = (x_i, r_i, v_i)]
\end{aligned}
$$

and the claim follows. ■

2.2.2 The OT_1^k protocol – definition and construction

The following version of the *Oblivious Transfer* functionality is a main ingredient of our construction. Let k be a fixed integer ($k = 4$ will do for our purpose), and let $b_1, b_2, ..., b_k \in \{0, 1\}$ and $i \in \{1, ..., k\}$. Then, OT_1^k is defined as

$$\mathrm{OT}_1^k((b_1, b_2, ..., b_k), i) \quad = \quad (\lambda, b_i) \tag{2.11}$$

This functionality is clearly asymmetric. Traditionally the first player, holding input $(b_1, b_2, ..., b_k)$ is called the *sender* and the second player, holding the input $i \in \{1, ..., k\}$ is called the *receiver*. Intuitively, the goal is to transfer the i^{th} bit to the receiver, without letting the receiver obtain knowledge of any other bit and without letting the sender obtain knowledge of the identity of the bit required by the receiver.

Using any trapdoor permutation, $\{f_i\}_{i \in I}$, we present a protocol for privately computing OT_1^k. The description below refers to the algorithms guaranteed by such a collection (see Definition 1.2.3), and to a hard-core predicate b for such a collection (see Definition 1.2.4). We denote the sender (first party) by S and the receiver (second party) by R. As discussed in the beginning of this chapter, since we are dealing with a finite functionality, we want the security to be stated in terms of an auxiliary security parameter, n, presented to both parties in unary.

Construction 2.2.5 (Oblivious Transfer protocol for semi-honest model):

Inputs: *The sender has input $(b_1, b_2, ..., b_k) \in \{0, 1\}^k$, the receiver has input $i \in \{1, 2, ..., k\}$, and both parties have the auxiliary security parameter 1^n.*

Step S1: *The sender uniformly selects a trapdoor pair, (α, t), by running the generation algorithm, G, on input 1^n. That is, it uniformly selects a random-pad, r, for G and sets $(\alpha, t) = G(1^n, r)$. It sends α to the receiver.*

Step R1: *The receiver uniformly and independently selects $e_1, ..., e_k \in D_\alpha$, sets $y_i = f_\alpha(e_i)$ and $y_j = e_j$ for every $j \neq i$, and sends $(y_1, y_2, ..., y_k)$ to the sender. That is,*

1. *It uniformly and independently selects $e_1, ..., e_k \in D_\alpha$, by invoking the domain sampling algorithm k times, on input α. Specifically, it selects random pads, r_j's, for D and sets $e_j = D(\alpha, r_j)$, for $j = 1, ..., k$.*
2. *Using the evaluation algorithm, the sender sets $y_i = f_\alpha(e_i)$.*
3. *For $j \neq i$, it sets $y_j = e_j$.*
4. *The receiver sends $(y_1, y_2, ..., y_k)$ to the sender.*

(Thus, the receiver knows $f_\alpha^{-1}(y_i) = e_i$, but cannot predict $b(f_\alpha^{-1}(y_j))$ for any $j \neq i$.)

Step S2: *Upon receiving* $(y_1, y_2, ..., y_k)$, *using the inverting-with-trapdoor algorithm and the trapdoor* t, *the sender computes* $x_j = f_\alpha^{-1}(y_j)$, *for every* $j \in \{1, ..., k\}$. *It sends* $(b_1 \oplus b(x_1), b_2 \oplus b(x_2), ..., b_k \oplus b(x_k))$ *to the receiver.*

Step R2: *Upon receiving* $(c_1, c_2, ..., c_k)$, *the receiver locally outputs* $c_i \oplus b(e_i)$.

We first observe that the above protocol correctly computes OT_1^k: This is the case since the receiver's local output satisfies

$$\begin{aligned} c_i \oplus b(e_i) &= (b_i \oplus b(x_i)) \oplus b(e_i) \\ &= b_i \oplus b(f_\alpha^{-1}(f_\alpha(e_i))) \oplus b(e_i) \\ &= b_i \end{aligned}$$

We show below that the protocol indeed privately computes OT_1^k. Intuitively, the sender gets no information from the execution since, for any possible value of i, the senders sees the same distribution – a sequence of uniformly and independently selected elements of D_α. Intuitively, the receiver gains no computational knowledge from the execution since, for $j \neq i$, the only data it has regarding b_j is the triplet $(\alpha, e_j, b_j \oplus b(f_\alpha^{-1}(e_j)))$, from which it is infeasible to predict b_j better than by a random guess. A formal argument is indeed due and given next.

Proposition 2.2.6 *Suppose that* $\{f_i\}_{i\in I}$ *constitutes a trapdoor permutation. Then, Construction 2.2.5 constitutes a protocol for privately computing* OT_1^k (in the semi-honest model).

Proof Sketch: We will present a simulator for the view of each party. Recall that these simulators are given the local input and output of the party, which by the above includes also the security parameter. We start with the sender. On input $(((b_1, ..., b_k), 1^n), \lambda)$, this simulator selects α (as in Step S1), and uniformly and independently generates $y_1, ..., y_k \in D_\alpha$. Let r denote the sequence of coins used to generate α, and assume without loss of generality that the inverting-with-trapdoor algorithm is deterministic (which is typically the case anyhow). Then the simulator outputs $(((b_1, ..., b_k), 1^n), r, (y_1, ..., y_k))$, where the first element represents the party's input, the second its random choices, and the third – the message it has received. Clearly, this output distribution is *identical* to the view of the sender in the real execution.

We now turn to the receiver. On input $((i, 1^n), b_i)$, the simulator proceeds as follows.

1. Emulating Step S1, the simulator uniformly selects a trapdoor pair, (α, t), by running the generation algorithm on input 1^n.
2. As in Step R1, it uniformly and independently selects $r_1, ..., r_k$ for the domain sampler D, and sets $e_j = D(\alpha, r_j)$ for $j = 1, ..., k$. Next, it sets $y_i = f_\alpha(e_i)$ and $y_j = e_j$, for $j \neq i$.
3. It sets $c_i = b_i \oplus b(e_i)$, and uniformly selects $c_j \in \{0, 1\}$, for $j \neq i$.
4. Finally, it outputs $((i, 1^n), (r_1, ..., r_k), (\alpha, (c_1, ..., c_k)))$, where the first element represents the party's input, the second its random choices, and the third – the two messages it has received.

Note that, except for the sequence of c_j's, this output is distributed identically to the corresponding prefix of the receiver's view in the real execution. Furthermore, the above holds even if we include the bit $c_i = b_i \oplus b(e_i) = b_i \oplus b(f_\alpha^{-1}(y_i))$ (and still exclude the other c_j's). Thus, the two distributions differ only in the following aspect: For $j \neq i$, in the simulation c_j is uniform and independent of anything else, whereas in the real execution c_j equals $b(f_\alpha^{-1}(y_j)) = b(f_\alpha^{-1}(e_j))$. However, it is impossible to distinguish the two cases (as the distinguisher is not given the trapdoor and b is a hard-core predicate of $\{f_\alpha\}_\alpha$). ■

value of (a_2, b_2)	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
OT-input	1	2	3	4
value of output	$c_1 + a_1 b_1$	$c_1 + a_1 \cdot (b_1 + 1)$	$c_1 + (a_1 + 1) \cdot b_1$	$c_1 + (a_1 + 1) \cdot (b_1 + 1)$

Figure 2.1: The value of the output of Party 2 as a function of the values of its own inputs (represented in the columns), and the inputs and output of Party 1 (i.e., a_1, b_1, c_1). The value with which Party 2 enters the Oblivious Transfer protocol (i.e., $1 + 2a_2 + b_2$) is shown in the second row, and the value of the output (of both OT and the entire protocol) is shown in the third. Note that in each case, the output of Party 2 equals $c_1 + (a_1 + a_2) \cdot (b_1 + b_2)$.

2.2.3 Privately computing $c_1 + c_2 = (a_1 + a_2) \cdot (b_1 + b_2)$

We now turn to the functionality defined in Eq. (2.8)–(2.9). Recall that the arithmetics is in GF(2). We privately reduce the computation of this (finite) functionality to the computation of OT_1^4.

Construction 2.2.7 (privately reducing the computation of Eq. (2.8)–(2.9) to OT_1^4):

Inputs: *Party i holds* $(a_i, b_i) \in \{0,1\} \times \{0,1\}$, *for* $i = 1, 2$.

Step 1: *The first party uniformly selects* $c_1 \in \{0,1\}$.

Step 2 – Reduction: *The parties invoke* OT_1^4, *where Party 1 plays the sender and party 2 plays the receiver. The input to the sender is the 4-tuple*

$$(c_1 + a_1 b_1\,,\; c_1 + a_1 \cdot (b_1 + 1)\,,\; c_1 + (a_1 + 1) \cdot b_1\,,\; c_1 + (a_1 + 1) \cdot (b_1 + 1))\,, \tag{2.12}$$

and the input to the receiver is $1 + 2a_2 + b_2 \in \{1, 2, 3, 4\}$.

Outputs: *Party 1 outputs* c_1, *whereas Party 2 output the result obtained from the* OT_1^4 *invocation.*

We first observe that the above reduction is valid; that is, the output of Party 2 equals $c_1 + (a_1 + a_2) \cdot (b_1 + b_2)$. This follows from inspecting the truth table in Figure 2.1, which depicts the value of the output of Party 2, as a function of its own inputs and a_1, b_1, c_1. We stress that the output pair, (c_1, c_2), is uniformly distributed among the pairs for which $c_1 + c_2 = (a_1 + a_2) \cdot (b_1 + b_2)$ holds. Thus, each of the local outputs (i..e, of either Party 1 or Party 2) is uniformly distributed, although the two local-outputs are dependent of one another (as in Eq. (2.9)).
It is also easy to see that the reduction is private. That is,

Proposition 2.2.8 *Construction 2.2.7 privately reduces the computation of* Eq. (2.8)–*(2.9) to* OT_1^4.

Proof Sketch: Simulators for the oracle-aided protocol of Construction 2.2.7 are easily constructed. Specifically, the simulator of the view of Party 1, has input $((a_1, b_1), c_1)$ (i.e., the input and output of Party 1), which is identical to the view of Party 1 in the execution (where c_1 serves as coins to Party 1). Thus the simulation is trivial (i.e., by identity transformation). The same holds also for the simulator of the view of Party 2 – it gets input $((a_1, b_1), c_1 + (a_1 + a_2) \cdot (b_1 + b_2))$, which is identical to the view of Party 2 in the execution (where $c_1 + (a_1 + a_2) \cdot (b_1 + b_2)$ serves as the oracle response to Party 2). We conclude that the view of each party can be perfectly simulated (rather than just be simulated in a computationally indistinguishable manner), and the proposition follows. ■

As an immediate corollary to Propositions 2.2.8 and 2.2.6, and the Composition Theorem (Theorem 2.2.3), we obtain.

Corollary 2.2.9 *Suppose that trapdoor permutation exist. Then the functionality of* Eq. (2.8)–*(2.9) is privately computable* (in the semi-honest model).

2.2.4 The circuit evaluation protocol

We now show that the computation of any deterministic functionality, which is expressed by an arithmetic circuit over GF(2), is privately reducible to the functionality of Eq. (2.8)–(2.9). Recall that the latter functionality corresponds to a private computation of a multiplication gates over inputs shared by both parties. We thus refer to this functionality as the multiplication gate emulation.

Our reduction follows the overview presented in the beginning of this section. In particular, the sharing of a bit-value v between both parties means a uniformly selected pair of bits (v_1, v_2) so that $v = v_1 + v_2$, where first party holds v_1 and the second holds v_2. Our aim is to propagate, via private computation, shares of the input wires of the circuit into shares of all wires of the circuit, so that finally we obtain shares of the output wires of the circuit.

We will consider an enumeration of all wires in the circuit. The input wires of the circuit, n per each party, will be numbered $1, 2...., 2n$ so that, for $j = 1, ..., n$, the j^{th} input of party i corresponds to the $(i-1) \cdot n + j^{\text{th}}$ wire.[7] The wires will be numbered so that the output wires of each gate have a larger numbering than its input wires. The output-wires of the circuit are clearly the last ones. For sake of simplicity we assume that each party obtains n output bits, and that the output bits of the second party correspond to the last n wires.

Construction 2.2.10 (privately reducing any deterministic functionality to multiplication-gate emulation):

Inputs: *Party i holds the bit string $x_i^1 \cdots x_i^n \in \{0,1\}^n$, for $i = 1, 2$.*

Step 1 – Sharing the inputs: Each party splits and shares each of its input bits with the other party. *That is, for every $i = 1, 2$ and $j = 1, ..., n$, party i uniformly selects a bit r_i^j and sends it to the other party as the other party's share of input wire $(i-1) \cdot n + j$. Party i sets its own share of the $(i-1) \cdot n + j^{\text{th}}$ input wire to $x_i^j + r_i^j$.*

Step 2 – Circuit Emulation: Proceeding by the order of wires, the parties use their shares of the two input wires to a gate in order to privately compute shares for the output wire of the gate. *Suppose that the parties hold shares to the two input wires of a gate; that is, Party 1 holds the shares a_1, b_1 and Party 2 holds the shares a_2, b_2, where a_1, a_2 are the shares of the first wire and b_1, b_2 are the shares of the second wire. We consider two cases.*

Emulation of an addition gate: *Party 1 just sets its share of the output wire of the gate to be $a_1 + b_1$, and Party 2 sets its share of the output wire to be $a_2 + b_2$.*

Emulation of a multiplication gate: *Shares of the output wire of the gate are obtained by invoking the oracle for the functionality of* Eq. (2.8)–*(2.9), where Party 1 supplies the input* (query-part) *(a_1, b_1), and Party 2 supplies (a_2, b_2). When the oracle responses, each party sets its share of the output wire of the gate to equal its part of the oracle answer.*

[7] Our treatment ignores the likely case in which the circuit uses the constant 1. (The constant 0 can always be produced by ADDING any GF(2) value to itself.) However, the computation of a circuit which uses the constant 1 can be privately reduced to the computation of a circuit which does not use the constant 1. Alternatively, we may augment Step 1 below so that the shares of the wire carrying the constant 1 are (arbitrarily) computed so that they sum-up to 1 (e.g., set the share of the first party to be 1 and the share of the second party to be 0).

Step 3 – Recovering the output bits: Once the shares of the circuit-output wires are computed, each party sends its share of each such wire to the party with which the wire is associated. *That is, the shares of the last n wires are send by Party 1 to Party 2, whereas the shares of the preceding n wires are sent by Party 2 to Party 1. Each party recovers the corresponding output bits by adding-up the two shares; that is, the share it had obtained in Step 2 and the share it has obtained in the current step.*

Outputs: *Each party locally outputs the bits recovered in Step 3.*

For starters, let us verify that the output is indeed correct. This can be shown by induction on the wires of the circuits. The induction claim is that the shares of each wire sum-up to the correct value of the wire. The base case of the induction are the input wires of the circuits. Specifically, the $(i-1)\cdot n+j^{\text{th}}$ wire has value x_i^j and its shares are r_i^j and $r_i^j+x_i^j$ (indeed summing-up to x_i^j). For the induction step we consider the emulation of a gate. Suppose that the values of the input wires (to the gate) are a and b, and that their shares a_1, a_2 and b_1, b_2 indeed satisfy $a_1+a_2=a$ and $b_1+b_2=b$. In case of an addition gate, the shares of the output wire were set to be a_1+b_1 and a_2+b_2, indeed satisfying

$$(a_1+b_1)+(a_2+b_2) \;=\; (a_1+a_2)+(b_1+b_2) \;=\; a+b$$

In case of a multiplication gate, the shares of the output wire were set to be c_1 and c_2, so that $c_1+c_2=(a_1+a_2)\cdot(b_1+b_2)$. Thus, $c_1+c_2=a\cdot b$ as required.

Privacy of the reduction. We now turn to show that Construction 2.2.10 indeed privately reduces the computation of a circuit to the multiplication-gate emulation. That is,

Proposition 2.2.11 (privately reducing circuit evaluation to multiplication-gate emulation): *Construction 2.2.10 privately reduces the evaluation of arithmetic circuits over* GF(2) *to the functionality of* Eq. (2.8)*–(2.9).*

Proof Sketch: Simulators for the oracle-aided protocol of Construction 2.2.10 are constructed as follows. Without loss of generality we present a simulator for the view of Party 1. This simulator gets the party's input $x_1^1,...,x_1^n$, as well as its output, denoted $y^1,...,y^n$. It operates as follows.

1. The simulator uniformly selects $r_1^1,...,r_1^n$ and $r_2^1,...,r_2^n$, as in Step 1. (The r_1^j's will be used as the coins of Party 1, which are part of the view of the execution, whereas the r_2^j's will be used as the message Party 1 receives at Step 1.) For each $j \le n$, the simulator sets $x_1^j+r_1^j$ as the party's share of the value of the j^{th} wire. Similarly, for $j \le n$, the party's share of the $n+j^{\text{th}}$ wire is set to r_2^j.

   This completes the computation of the party's shares of all circuit-input wires.

2. The party's shares for all other wires are computed, iteratively gate-by-gate, as follows.

   - The share of the output-wire of an addition gate is set to be the sum of the shares of the input-wires of the gate.
   - The share of the output-wire of a multiplication gate is uniformly selected in $\{0,1\}$.

   (The shares computed for output-wires of multiplication gates will be used as the answers obtained, by Party 1, from the oracle.)

3. For each wire corresponding to an output, denoted y^j, available to Party 1, the simulator sets m^j to be the sum of the party's share of the wire with y^j.

4. The simulator outputs

$$(x_1^1, ..., x_1^n), (y^1, ..., y^n), (r_1^1, ..., r_1^n), V^1, V^2, V^3)$$

where $V^1 = (r_2^1, ..., r_2^n)$ correspond to the view of Party 1 in Step 1 of the protocol, the string V^2 equals the concatenation of the bits selected for the output-wires of multiplication gates (corresponding to the party's view of the oracle answers in Step 2), and $V^3 = (m^1, ..., m^n)$ (corresponding to the party's view in Step 3 – that is, the messages it would have obtained from Party 2 in the execution).

We claim that the output of the simulation is distributed identically to the view of Party 1 in the execution of the oracle-aided protocol. The claim clearly holds with respect to the first four parts of the view; that is, the party's input (i.e., $x_1^1, ..., x_1^n$), output (i.e., $y^1, ..., y^n$), internal coin-tosses (i.e., $r_1^1, ..., r_1^n$), and the message obtained from Party 2 in Step 1 (i.e., $r_2^1, ..., r_2^n$). Also, by definition of the functionality of Eq. (2.8)–(2.9), the oracle-answers to each party are uniformly distributed independently of the parts of the party's queries. Thus, this part of the view of Party 1 is uniformly distributed, identically to V^2. It follows, that also all shares held by Party 1, are set by the simulator to have the same distribution as they have in a real execution. This holds, in particular, for the shares of the output wires held by Party 1. Finally, we observe that both in the real execution and in the simulation, these latter shares added to the messages sent by Party 2 in Step 3 (resp., V^3) must yield the corresponding bits of the local-output of Party 1. Thus, conditioned on the view so far, V^3 is distributed identically to the messages sent by Party 2 in Step 3. We conclude that the simulation is perfect (not only computationally indistinguishable), and so the proposition follows. ■

Conclusion. As an immediate corollary to Proposition 2.2.11, Corollary 2.2.9, and the Composition Theorem (Theorem 2.2.3), we obtain.

Corollary 2.2.12 *Suppose that trapdoor permutation exist. Then any* deterministic *functionality is privately computable* (in the semi-honest model).

Thus, by the discussion following Theorem 2.2.3 (i.e., specifically, combining Proposition 2.2.4, Corollary 2.2.12, and Theorem 2.2.3), we have

Theorem 2.2.13 *Suppose that trapdoor permutation exist. Then any functionality is privately computable* (in the semi-honest model).

2.3 Forcing Semi-Honest Behavior

Our aim is to use Theorem 2.2.13 in order to establish the main result of this chapter; that is,

Theorem 2.3.1 (main result for two-party case): *Suppose that trapdoor permutation exist. Then any two-party functionality can be securely computable* (in the malicious model).

This theorem will be established by compiling any protocol for the semi-honest model into an "equivalent" protocol for the malicious model. Loosely speaking, the compiler works by introducing macros which force each party to either behave in a semi-honest manner or be detected – hence the title of the current section. (In case a party is detected as cheating, the protocol aborts.)

2.3.1 The compiler – motivation and tools

We are given a protocol for the semi-honest model. In this protocol, each party has a local input and uses a uniformly distributed local random-pad. Such a protocol may be used to privately compute a functionality (either a deterministic or a probabilistic one), but the compiler does not refer to this functionality. The compiler is supposed to produce an equivalent protocol for the malicious model. So let us start by considering what a malicious party may do (beyond whatever a semi-honest party can do).

1. A malicious party may enter the actual execution of the protocol with an input different from the one it is given (i.e., "substitute its input"). As discussed in Section 2.1.2, this is unavoidable. What we need to guarantee is that this substitution is done obliviously of the input of the other party; that is, that the substitution only depends on the original input.

   Jumping ahead, we mention that the *input-commitment* phase of the compiler is aimed at achieving this goal. The tools used here are *commitment schemes* (see Definition 1.2.5) and *strong zero-knowledge proofs of knowledge* (see Section 1.2.3).

2. A malicious party may try to use a random-pad which is not uniformly distributed as postulated in the semi-honest model. What we need to do is force the party to use a random-pad (for the emulated semi-honest protocol) which is uniformly distributed.

   The *coin-generation* phase of the compiler is aimed at achieving this goal. The tool used here is a *coin-tossing into the well* protocol, which in turn uses tools as above.

3. A malicious party may try to send messages different than the ones specified by the original (semi-honest model) protocol. So we need to force the party to send messages as specified by its (already committed) local-input and random-pad.

   The *protocol emulation* phase of the compiler is aimed at achieving this goal. The tool used here is *zero-knowledge proof systems* (for NP-statements).

Before presenting the compiler, let us recall some tools we will use, all are known to exist assuming the existence of one-way 1-1 functions.

- *Commitment schemes* as defined in Definition 1.2.5. We denote by $C_n(b, r)$ the commitment to the bit b using security parameter n and randomness $r \in \{0,1\}^n$. Here we assume, for simplicity, that on security parameter n the commitment scheme utilizes exactly n random bits.

- *Zero-knowledge proofs of NP-assertions.* We rely on the fact that there exists such proof systems in which the prover strategy can be implemented in probabilistic polynomial-time, when given an NP-witness as auxiliary input. We stress that by the above we mean proof systems with negligible soundness error.

- *Zero-knowledge proofs of knowledge of NP-witnesses.* We will use the definition of a *strong* proof of knowledge (see Definition 1.2.6). We again rely on the analogous fact regarding the complexity of adequate prover strategies: That is, strong proofs of knowledge which are zero-knowledge exists for any NP-relation, and furthermore, the prover strategy can be implemented in probabilistic polynomial-time, when given an NP-witness as auxiliary input (see Construction 1.2.7).

Another tool which we will use is an augmented version of *coin-tossing into the well.* For sake of self-containment, we first present the definition and implementation of the standard (vanilla) notion. The augmented version is presented in the next subsection.

Definition 2.3.2 (coin-tossing into the well, vanilla version): *A* coin-tossing into the well *protocol is a two-party protocol for securely computing* (in the malicious model) *the randomized functionality* $(1^n, 1^n) \mapsto (b, b)$*, where* b *is uniformly distributed in* $\{0,1\}$*.*

Thus, in spite of malicious behavior by any one party, a non-aborting execution of a coin-tossing-into-the-well protocol ends with both parties holding the same uniformly selected bit b. Recall that our definition of security allows $(b, \perp)$ to appear as output in case Party 1 aborts. (It would have been impossible to securely implement this functionality if the definition had not allowed this slackness; see [25].) The following protocol and its proof of security are not used in the rest of this manuscript. However, we believe that they are instructive towards what follows.[8]

Construction 2.3.3 (a coin-tossing-into-the-well protocol): *Using a commitment scheme,* $\{C_n\}_{n\in\mathbb{N}}$.

Inputs: *Both parties get security parameter* 1^n.

Step C1: *Party 1 uniformly selects* $\sigma \in \{0,1\}$ *and* $s \in \{0,1\}^n$*, and sends* $c \stackrel{\text{def}}{=} C_n(\sigma, s)$ *to Party 2.*

Step C2: *Party 2 uniformly selects* $\sigma' \in \{0,1\}$*, and sends* σ' *to Party 1. (We stress than any possible response – including abort – of Party 2, will be interpreted by Party 1 as a bit.)*[9]

Step C3: *Party 1 sets* $b = \sigma \oplus \sigma'$*, and sends* (σ, s, b) *to Party 2.*

Step C4: *Party 2 verifiers that indeed* $b = \sigma \oplus \sigma'$ *and* $c = C_n(\sigma, s)$*. Otherwise, it aborts with output* $\perp$*.*

Outputs: *Both parties sets their local output to* b*.*

Intuitively, Steps C1–C2 are to be viewed as "tossing a coin into the well". At this point the value of the coin is determined (as either a random value or a illegal one), but only one party knows ("can see") this value. Clearly, if both parties are honest then they both output the same uniformly chosen bit, recovered in Steps C3–C4.

Proposition 2.3.4 *Suppose that* $\{C_n\}_{n\in\mathbb{N}}$ *is a commitment scheme. Then, Construction 2.3.3 constitutes a coin-tossing-into-the-well protocol.*

Proof Sketch: We need to show how to (efficiently) transform any admissible circuit pair, (A_1, A_2), for the real model into a corresponding pair, (B_1, B_2), for the ideal model. We treat separately each of the two cases – defined by which of the parties is honest. Recall that we may assume for simplicity that the adversary circuit is deterministic (see discussion at the end of Section 2.1.2). Also, for simplicity, we omit the input 1^n is some places.

We start with the case that the first party is honest. In this case B_1 is determined, and we transform the real-model adversary A_2 into an ideal-model adversary B_2. Machine B_2 will run machine A_2 locally, obtaining the messages A_2 would have sent in a real execution of the protocol and feeding it with messages that it expects to receive. Recall that B_2 gets input 1^n.

[8] The uninterested reader may skip to Section 2.3.2.
[9] Thus, by convention, we prevent Party 2 from aborting the execution.

1. B_2 send 1^n to the trusted party and obtain the answer bit b (which is uniformly distributed).

2. B_2 tries to generate an execution view (of A_2) ending with output b. This is done by repeating the following steps at most n times:

   (a) B_2 uniformly select $\sigma \in \{0,1\}$ and $s \in \{0,1\}^n$, and feeds A_2 with $c \stackrel{\text{def}}{=} C_n(\sigma, s)$. Recall that A_2 always responds with a bit, denoted σ'.

   (b) If $\sigma \oplus \sigma' = b$ then B_2 feed A_2 with the supposedly execution view, $(c, (\sigma, s, b))$, and outputs whatever A_2 does. Otherwise, it continues to the next iteration.

   In case all n iterations were completed unsuccessfully (i.e., without output), B_2 outputs a special **failure** symbol.

We need to show that for the coin-tossing functionality, f, and Π of Construction 2.3.3,

$$\{\text{IDEAL}_{f,\overline{B}}(1^n, 1^n)\}_{n \in \mathbb{N}} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}(1^n, 1^n)\}_{n \in \mathbb{N}}$$

In fact, we will show that the two ensembles are statistically indistinguishable. We start by showing that the probability that B_2 outputs **failure** is exponentially small. This is shown by proving that for every $b \in \{0,1\}$, each iteration of Step 2 succeeds with probability approximately 1/2. Such an iteration succeeds if and only if $\sigma \oplus \sigma' = b$, that is, if $A_2(C_n(\sigma, s)) = b \oplus \sigma$, where $(\sigma, s) \in \{0,1\} \times \{0,1\}^n$ is uniformly chosen. We have

$$\begin{aligned} \mathsf{Pr}_{\sigma,s}[A_2(C_n(\sigma,s)) = b \oplus \sigma] &= \frac{1}{2} \cdot \mathsf{Pr}[A_2(C_n(0)) = b] + \frac{1}{2} \cdot \mathsf{Pr}[A_2(C_n(1)) = b \oplus 1] \\ &= \frac{1}{2} + \frac{1}{2} \cdot (\mathsf{Pr}[A_2(C_n(0)) = b] - \mathsf{Pr}[A_2(C_n(1)) = b]) \end{aligned}$$

Using the hypothesis that C_n is a commitment scheme, the second term above is a negligible function in n, and so our claim regarding the probability that B_2 outputs **failure** follows. Next, we show that conditioned on B_2 not outputting **failure**, the distribution $\text{IDEAL}_{f,\overline{B}}(1^n, 1^n)$ is statistically indistinguishable from the distribution $\text{REAL}_{\Pi,\overline{A}}(1^n, 1^n)$. Both distributions have the form $(b\,,\, A_2(C_n(\sigma, s), (\sigma, s, b)))$, with $b = \sigma \oplus A_2(C_n(\sigma, s))$, and thus both are determined by the (σ, s)-pairs. In $\text{REAL}_{\Pi,\overline{A}}(1^n, 1^n)$, all pairs are equally likely (i.e., each appears with probability $2^{-(n+1)}$); whereas in $\text{IDEAL}_{f,\overline{B}}(1^n, 1^n)$ each pair (σ, s) appears with probability

$$\frac{1}{2} \cdot \frac{1}{|S_{A_2(C_n(\sigma,s)) \oplus \sigma}|} \tag{2.13}$$

where $S_b \stackrel{\text{def}}{=} \{(x, y) : A_2(C_n(x, y)) \oplus x = b\}$.[10] Observe that (by the above), for every $b \in \{0,1\}$ and uniformly distributed $(\sigma, s) \in \{0,1\} \times \{0,1\}^n$, the event $(\sigma, s) \in S_b$ occurs with probability which is negligiblly close to 1/2. Thus, $|S_{A_2(C_n(\sigma,s)) \oplus \sigma}| = (1 \pm \mu(n)) \cdot \frac{1}{2} \cdot 2^{n+1}$, where μ is a negligible function. It follows that the value of Eq. (2.13) is $(1 \pm \mu(n)) \cdot 2^{-(n+1)}$, and so $\text{REAL}_{\Pi,\overline{A}}(1^n, 1^n)$ and $\text{IDEAL}_{f,\overline{B}}(1^n, 1^n)$ are statistically indistinguishable.

[10] The pair (σ, s) appears as output iff the trusted party answers with $A_2(C_n(\sigma, s)) \oplus \sigma$ (which happens with probability 1/2) and the pair is selected in Step 2a. Note that the successful pairs, selected in Step 2a and passing the condition in Step 2b, are uniformly distributed in $S_{A_2(C_n(\sigma,s)) \oplus \sigma}$.

We now turn to the case where the second party is honest. In this case B_2 is determined, and we transform A_1 into B_1 (for the ideal model). On input 1^n, machine B_1 runs machine A_1 locally, obtaining messages A_1 would have sent in a real execution of the protocol and feeding it with messages that it expects to receive.

1. B_1 invokes A_1 (on input 1^n). In case A_1 aborts (or acts improperly) in Step C1, we let B_1 abort before invoking the trusted party. Otherwise, suppose that A_1 sends message c (supposedly c is a commitment by C_n). Recall that c may be in the range of $C_n(\sigma)$ for at most one $\sigma \in \{0,1\}$.

2. Machine B_1 tries to obtain the answers of A_1 (in Step C3) to both possible messages sent in Step C2.

   (a) B_1 feeds A_1 with the message 0 and records the answer which is either abort or (σ, s_0, b_0). The case in which either $c \neq C_n(\sigma, s_0)$ or $b_0 \neq \sigma \oplus 0$ is treated as if A_1 has aborted.

   (b) Rewinding A_1 to the beginning of Step C2, machine B_1 feeds A_1 with the message 1 and records the answer which is either abort or (σ, s_1, b_1). (Again, the case in which either $c \neq C_n(\sigma, s_1)$ or $b_1 \neq \sigma \oplus 1$ is treated as abort.)

   If A_1 aborts in both cases, machine B_1 aborts (before invoking the trusted party). Otherwise, we proceed as follows, distinguishing two cases.

   Case 1: A_1 answers properly (in the above experiment) for a single 0-1 value, denoted σ'.

   Case 2: A_1 answers properly for both values. (Note that the value σ returned in both cases is identical since c must be in the range of $C_n(\sigma)$.)

3. Machine B_1 sends 1^n to the trusted party, which responses with a uniformly selected value $b \in \{0,1\}$. Recall that the trusted party has not responded to Party 2 yet, and that B_1 has the option of stopping the trusted party before it does so.

4. In Case 1, machine B_1 stops the trusted party if $b \neq \sigma \oplus \sigma'$, and otherwise allows it to send b to Party 2. In Case 2, machine B_1 sets $\sigma' = b \oplus \sigma$, and allows the trusted party to send b to Party 2. Next, B_1 feeds σ' to A_1, which responds with the Step C3 message $(\sigma, s_{\sigma'}, b_{\sigma'})$, where $b_{\sigma'} = \sigma \oplus \sigma' = b$.

5. Finally, B_1 feed A_1 with the execution view, $(1^n, \sigma')$, and outputs whatever A_1 does.

We now show that $\text{IDEAL}_{f,\overline{B}}(1^n, 1^n)$ and $\text{REAL}_{\Pi,\overline{A}}(1^n, 1^n)$ are actually identically distributed. Consider first the case where A_1 (and so B_1) never aborts. In this case we have,

$$\begin{aligned} \text{IDEAL}_{f,\overline{B}}(1^n, 1^n) &= (A_1(1^n, \sigma \oplus b)\,,\, b) \\ \text{REAL}_{\Pi,\overline{A}}(1^n, 1^n) &= (A_1(1^n, \sigma')\,,\, \sigma \oplus \sigma') \end{aligned}$$

where σ' and b are uniformly distributed in $\{0,1\}$, and σ is determined by $c = A_1(1^n)$. Observe that σ' is distributed uniformly independently of σ, and so $\sigma \oplus \sigma'$ is uniformly distributed over $\{0,1\}$. We conclude that $(A_1(1^n, \sigma \oplus b)\,,\, b)$ and $(A_1(1^n, \sigma \oplus (\sigma \oplus \sigma'))\,,\, \sigma \oplus \sigma')$ are identically distributed.

Next, consider the case that B_1 always aborts (due to improper A_1 behavior in either Step C1 or Step C3). In this case, B_1 aborts before invoking the trusted party, and so both ensembles are identical (i.e., both equal $(A_1(1^n, \perp), \perp)$). Since A_1 is deterministic (see above), the only case left to consider is where A_1 acts properly in Step C1 and responses properly (in Step C3) to a single value, denoted σ'. In this case, the real execution of Π is completed only if Party 2 sends σ' as its Step C2 message (which happens with probability 1/2). Similarly, in the ideal model, the execution is completed (without B_1 aborting) if the trusted party answers with $b = \sigma \oplus \sigma'$ (which happens with probability 1/2).[11] In both cases, the complete joint execution equals $(A_1(1^n, \sigma'), \sigma \oplus \sigma')$, whereas the aborted joint execution equals $(A_1(1^n, \sigma' \oplus 1, \perp), \perp)$. ■

2.3.2 The compiler – the components

In analogy to the three phases mentioned in the motivating discussion, we present subprotocol for *input-commitment*, *coin-generation*, and *emulation of a single step*. We start with the coin-generation protocol, which is actually an augmentation of the above *coin-tossing into the well* protocol. (Alternatively, the reader may start with the simpler input-commitment and single-step-emulation protocols, presented in §2.3.2.2 and §2.3.2.3, respectively.)

We note that (like the functionality of Definition 2.3.2) all functionalities defined in this subsection are easy to compute privately (i.e., to compute securely in the semi-honest model). Our aim, however, is to present (for later use in the compiler) protocols for securely computing these functionalities in the malicious model.

All the construction presented in this subsection utilize zero-knowledge proofs of various types, which in turn exists under the assumption that commitment schemes exists. We neglect to explicitly state this condition in the propositions, which should be read as stating the security of the corresponding constructions given proof systems as explicitly specified in the latter.

2.3.2.1 Augmented coin-tossing into the well

We augment the above coin-tossing-into-the-well protocol so to fit our purposes. The augmentation is in providing the second party (as output) with a commitment to the coin-outcome obtained by the first party, rather than providing it with coin outcome itself.[12]

Definition 2.3.5 (coin-tossing into the well, augmented): *An* augmented coin-tossing into the well *protocol is a two-party protocol for securely computing* (in the malicious model) *the following randomized functionality with respect to* some fixed *commitment scheme,* $\{C_n\}_{n\in\mathbb{N}}$,

$$(1^n, 1^n) \mapsto ((b, r), C_n(b, r)) \tag{2.14}$$

where (b, r) *is uniformly distributed in* $\{0,1\} \times \{0,1\}^n$.

Eq. (2.14) actually specifies *coin-tossing with respect to a commitment scheme* $\{C_n\}$. Definition 2.3.5 allows $\{C_n\}$ to be arbitrary, but fixed for the entire protocol. The string r included in the output of Party 1, allows it to later prove (in zero-knowledge) statements regarding the actual bit-value, b, committed (to Party 2).

In the following construction the commitment scheme $\{C_n\}$ of Eq. (2.14) is used for internal steps (i.e., Step C1) of the protocol (in addition to determining the output).[13]

[11] Recall that σ and σ' are determined by the Step C1 message.

[12] The reason we use the term 'augmentation' rather than 'modification' is apparent from the implementation below: The augmented protocol is actually obtained by augmenting the vanilla protocol. Furthermore, it is easy to obtain the vanilla version from the augmented one, and going the other way around requires more work (as can be seen below).

[13] Clearly, one could have used a different commitment scheme for Step C1.

Construction 2.3.6 (an augmented coin-tossing-into-the-well protocol):

Inputs: *both parties get security parameter* 1^n.

Step C1: *The parties invoke a truncated/augmented version of the vanilla coin-tossing protocol* $n+1$ *times so to generate uniformly distributed bits,* $b_0, b_1, ..., b_n$*, known to Party 1.*

Specifically, for $j = 0, 1, ..., n$*, the parties execute the following four steps:*[14]

Step C1.1: *Party 1 uniformly selects* $(\sigma_j, s_j) \in \{0,1\} \times \{0,1\}^n$*, and sends* $c_j \stackrel{\text{def}}{=} C_n(\sigma_j, s_j)$ *to Party 2.*

Step C1.2: *The parties invoke a zero-knowledge strong-proof-of-knowledge so that Party 1 plays the prover and Party 2 plays the verifier. The common input to the proof system is* c_j*, the prover gets auxiliary input* (σ_j, s_j)*, and its objective is to prove that it knows* (x, y) *such that*

$$c_j = C_n(x, y) \tag{2.15}$$

In case the verifier rejects the proof, Party 2 aborts with output $\perp$.

(As in Construction 2.3.3, any possible response – including abort – of Party 2 during the execution of the protocol – and specifically this step – will be interpreted by a honest Party 1 as a canonical legitimate message.)

Step C1.3: *Party 2 uniformly selects* $\sigma'_j \in \{0,1\}$*, and sends* σ'_j *to Party 1. (Again, any possible response – including abort – of Party 2, will be interpreted by Party 1 as a bit.)*

Step C1.4: *Party 1 sets* $b_j = \sigma_j \oplus \sigma'_j$.

Step C2: *Party 1 sets* $b = b_0$ *and* $r = b_1 b_2 \cdots b_n$*, and sends* $c \stackrel{\text{def}}{=} C_n(b, r)$ *to Party 2.*

Step C3: *The parties invoke a zero-knowledge proof system so that Party 1 plays the prover and Party 2 plays the verifier. The common input to the proof system is* $(c_0, c_1, ..., c_n, \sigma'_0, \sigma'_1, ..., \sigma'_n, c)$*, the prover gets auxiliary input* $(\sigma_0, \sigma_1, ..., \sigma_n, s_0, s_1, ..., s_n)$*, and its objective is to prove that there exists* $(x_0, x_1, ..., x_n, y_0, y_1, ..., y_n)$ *such that*

$$(\forall j \;\; c_j = C_n(x_j, y_j)) \wedge (c = C_n(x_0 \oplus \sigma'_0, (x_1 \oplus \sigma'_1) \cdots (x_n \oplus \sigma'_n))) \tag{2.16}$$

In case the verifier rejects the proof, Party 2 aborts with output $\perp$ *(otherwise its output will be* c*). (Again, any possible response – including abort – of Party 2 during the execution of this step, will be interpreted by Party 1 as a canonical legitimate message.)*

Outputs: *Party 1 sets its local output to* (b, r)*, and Party 2 sets its local output to* c.

Observe that the specified strategies are indeed implementable in polynomial-time. In particular, in Steps C1.2 and C3, Party 1 supplies the prover subroutine with the adequate NP-witnesses which indeed satisfy the corresponding claims. We rely on the fact that given an NP-witness as auxiliary input, a prover strategy which always convinces the prescribed verifier can be implemented in probabilistic polynomial-time. It follows that if both parties are honest then neither aborts and the output is as required by Eq. (2.14).

[14] Reversing the order of Steps C1.2 and C1.3 makes each iteration, as well as its emulation in the proof of security below, more similar to the vanilla coin-tossing protocol of Construction 2.3.3. However, the proof of security is somewhat simplified by the order used here.

Proposition 2.3.7 *Suppose that $\{C_n\}_{n\in\mathbb{N}}$ is a commitment scheme. Then Construction 2.3.6 constitutes an augmented coin-tossing-into-the-well protocol.*

Proof Sketch: We need to show how to (efficiently) transform any admissible circuit pair, (A_1, A_2), for the real model into a corresponding pair, (B_1, B_2), for the ideal model. We treat separately each of the two cases – defined by which of the parties is honest.

We start with the case that the first party is honest. In this case B_1 is determined, and we transform (the real-model adversary) A_2 into (an ideal-model adversary) B_2. Machine B_2 will run machine A_2 locally, obtaining the messages A_2 would have sent in a real execution of the protocol and feeding it with messages that it expects to receive. The following construction is different from the analogous construction used in the proof of of Proposition 2.3.4. Recall that B_2 gets input 1^n.

1. B_2 send 1^n to the trusted party and obtain the answer c (where $c = C_n(b, r)$ for a uniformly distributed $(b, r) \in \{0,1\} \times \{0,1\}^n$).

2. B_2 generates a transcript which seems computationally indistinguishable from an execution view (of A_2) ending with output c as above. This is done by emulating Steps C1 and C3 as follows.

   Emulating Step C1: For $j = 0, 1, ..., n$, machine B_2 proceeds as follows

   (a) B_2 uniformly select $\sigma_j \in \{0,1\}$ and $s_j \in \{0,1\}^n$, and feeds A_2 with $c_j \stackrel{\text{def}}{=} C_n(\sigma_j, s_j)$.

   (b) B_2 invokes the simulator guaranteed for the *zero-knowledge* proof-of-knowledge system (of Step C1.2), on input c_j, using $A_2(T_{j-1}, c_j)$ as a possible malicious verifier, where $A_2(T_{j-1}, c_j)$ denotes the behavior of A_2 in the j^{th} iteration of Step C1.2, given that it has received c_j in the current iteration of Step C1.1 and that T_{j-1} denotes the transcript of the previous iterations of Step C1.2.
   Denote the obtained simulation transcript by $T_j = T_j(c_j, T_{j-1})$.

   (c) Next, B_2 obtains from A_2 its Step C1.3 message, which by our convention is always a bit, denoted σ'_j. (We may consider this bit to be a part of T_j.)

   Emulating Step C3: B_2 invokes the simulator guaranteed for the *zero-knowledge* proof system (of Step C3), on input c, using $A_2(T_n, c)$ as a possible malicious verifier, where $A_2(T_n, c)$ denotes the behavior of A_2 in Step C3, given that it has received c in Step C2, and that T_n denotes the transcript of (all iterations of) Step C1. Denote the obtained simulation transcript by $T = T(c, T_n)$.

3. Finally, B_2 feed A_2 with T, and outputs whatever A_2 does.

We need to show that for the functionality, f, of Eq. (2.14) and Π of Construction 2.3.6,

$$\{\text{IDEAL}_{f,\overline{B}}(1^n\,,\,1^n)\}_{n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n)\}_{n\in\mathbb{N}} \tag{2.17}$$

There are two differences between $\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n)$ and $\text{IDEAL}_{f,\overline{B}}(1^n\,,\,1^n)$. Firstly, in the real execution the output of Party 1 (i.e.,(b, r)) equals $(\sigma_0 \oplus \sigma'_0, (\sigma_1 \oplus \sigma'_1) \cdots (\sigma_n \oplus \sigma'_n))$, whereas in the ideal-model it (i.e., (b, r)) is uniformly distributed independently of everything else. Secondly, in the ideal model simulations of zero-knowledge proof systems replace their actual execution. To show that the two ensemble are nevertheless computationally indistinguishable we consider a hybrid ensemble, denoted $\text{MENTAL}_{\overline{B}}(1^n)$, which is defined by the following mental experiment. Loosely speaking, the mental experiment behaves as B_2 except that it obtains the pair (b, r) corresponding to the trusted party answer $c = C_n(b, r)$ and emulates Step C1 so that $\sigma_0 \oplus \sigma'_0 = b$ and $(\sigma_1 \oplus \sigma'_1) \cdots (\sigma_n \oplus \sigma'_n) = r$, rather than being independent as in the execution of B_2. Note that B_2 does not get the pair (b, r), and so could not possibly perform the procedure defined as a mental experiment below.

The mental experiment differs from B_2 only in the emulation of Step C1, which is conducted given (b, r) as auxiliary input. We set $b_0 = b$ and b_j to be the j^{th} bit of r, for $j = 1, ..., n$.

> For $j = 0, 1, ..., n$, given b_j, we try to generate an execution view (of A_2 in the j^{th} iteration of Step C1) ending with outcome b_j (for Party 1). This is done by repeating the following steps at most n times:
>
> (a) We uniformly select $\sigma_j \in \{0, 1\}$ and $s_j \in \{0, 1\}^n$, and feeds A_2 with $c_j \stackrel{\text{def}}{=} C_n(\sigma_j, s_j)$.
>
> (b) We run the zero-knowledge simulator for $A_2(T_{j-1}, c_j)$, as B_2 does, and obtain from A_2 its Step C1.3 message, denoted σ'_j.
>
> (c) If $\sigma_j \oplus \sigma'_j = b_j$ then we record the values c_j and T_j (as B_2 does), and successfully complete the emulation of the current (i.e., j^{th}) iteration of Step C1.
>
> Otherwise, we continue to the next attempt to generate such an emulation.
>
> In case all n attempts (for some $j \in \{0, 1, ..., n\}$) were completed unsuccessfully (i.e., without recording a pair (c_j, T_j)), the mental experiment is said to have failed.

By the proof of Proposition 2.3.4, each attempt succeeds with probability approximately $1/2$,[15] and so we may ignore the exponentially (in n) rare event in which the mental experiment fails. Thus, we may write

$$\text{MENTAL}_{\overline{B}}(1^n) \;=\; ((b, r)\,,\, M_{A_2}(b, r))$$

where (b, r) is uniformly distributed and $M_{A_2}(b, r)$ is the outcome of the mental experiment when given (the auxiliary input) (b, r). Turning to $\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n)$ and using again the proof of Proposition 2.3.4,[16] we recall that each of the bits in the output of Party 1 (i.e., (b, r)) is distributed almost uniformly in $\{0, 1\}$, and the same holds for each bit conditioned on the value of all previous bits. Thus, we may write

$$\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n) \;=\; ((b, r)\,,\, R_{A_2}(b, r))$$

where the distribution of (b, r) is statistically indistinguishable from the uniform distribution over $\{0, 1\} \times \{0, 1\}^n$, and $R_{A_2}(b, r)$ is the output of the second party in $\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n)$ conditioned on the first party outputting (b, r). The only difference between $M_{A_2}(b, r)$ and $R_{A_2}(b, r)$ is that in the first distribution the output of zero-knowledge simulators replace the transcript of real executions appearing in the second distribution. Thus, the two ensembles are computationally indistinguishable. Specifically, we use the fact that the standard formulation of zero-knowledge guarantees computationally indistinguishable simulations also in the presence of auxiliary inputs. Considering (b, r) as auxiliary input, it follows that for every fixed (b, r) the distributions $M_{A_2}(b, r)$ and $R_{A_2}(b, r)$ are indistinguishable by polynomial-size circuits. Thus, we have

$$\{\text{MENTAL}_{\overline{B}}(1^n)\}_{n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n)\}_{n\in\mathbb{N}} \tag{2.18}$$

[15] Specifically, we use the fact that $|S_a| \approx 2^n$, where $S_a \stackrel{\text{def}}{=} \{(x, y) : x \oplus A_2(C_n(x, y)) = a\} \subset \{0,1\} \times \{0,1\}^n$.

[16] Specifically, we use the fact that $|S_a| = (1 \pm \mu(n)) \cdot 2^n$, where S_a is as in the previous footnote and μ is a negligible function.

On the other hand, using the hypothesis that the commitment scheme used in Step C1 is secure, one can prove that

$$\{\text{MENTAL}_{\overline{B}}(1^n)\}_{n\in\mathbb{N}} \stackrel{c}{\equiv} \{\text{IDEAL}_{f,\overline{B}}(1^n\,,\,1^n)\}_{n\in\mathbb{N}} \tag{2.19}$$

First, we write

$$\text{IDEAL}_{f,\overline{B}}(1^n\,,\,1^n) \;=\; ((b,r)\,,\,I_{A_2}(b,r))$$

where (b,r) is uniformly distributed and $I_{A_2}(b,r)$ is the output of B_2 (equiv., A_2) conditioned on the trusted party answering Party 1 with (b,r). We will show that for any fixed $(b,r)\in\{0,1\}\times\{0,1\}^n$, no poly($n$)-circuit can distinguish $I_{A_2}(b,r)$ from $M_{A_2}(b,r)$. Recall that $I_{A_2}(b,r)$ from $M_{A_2}(b,r)$ are identical except to the way $c_0,c_1,...,c_n$ are generated. In the first distribution each c_j is generated by uniformly selecting $(\sigma,s)\in\{0,1\}\times\{0,1\}^n$, and setting $c_j=C_n(\sigma,s)$; whereas in the second distribution c_j is generated by uniformly selecting among the (σ,s)'s (in $\{0,1\}\times\{0,1\}^n$) which satisfy $\sigma\oplus A_2(T_{j-1},C_n(\sigma,s))=b_j$ (and setting $c_j=C_n(\sigma,s)$).[17] The rest of the argument is aimed at showing that these two types of distributions (on commitments) are computationally indistinguishable. This is quite intuitive; yet a proof is provided below. Consequently, no poly(n)-circuit can distinguish $I_{A_2}(b,r)$ from $M_{A_2}(b,r)$, and Eq. (2.19) follows.

Abusing notation a little,[18] we need to prove that X_n^a and Y_n are computationally indistinguishable, where $S_a \stackrel{\text{def}}{=} \{(x,y) : A_2(C_n(x,y))\oplus x = a\}$, and X_n^a (resp., Y_n) denote the distribution of $C_n(\sigma,s)$ where (σ,s) is uniformly distributed in S_b (resp., in $\{0,1\}\times\{0,1\}^n$). (Y_n represents the way each c_j is distributed in $I_{A_2}(b,r)$, whereas $X_n^{b_j}$ represents the way c_j is distributed in $M_{A_2}(b,r)$.) To prove the latter assertion, let D be an arbitrary polynomial-size circuit representing a potential distinguisher, and let $A\stackrel{\text{def}}{=}A_2$. Then, for some negligible function μ, we have

$$\begin{aligned}
\mathsf{Pr}[D(X_n^a)=1] &= \mathsf{Pr}_{(\sigma,s)\in\{0,1\}\times\{0,1\}^n}[D(C_n(\sigma,s))=1\,|\,A(C_n(\sigma,s))=\sigma\oplus a] \\
&= 2\cdot\mathsf{Pr}_{(\sigma,s)\in\{0,1\}\times\{0,1\}^n}[(D(C_n(\sigma,s))=1)\,\wedge\,(A(C_n(\sigma,s))=\sigma\oplus a)]\pm\mu(n)
\end{aligned}$$

where the second equality is due to $\mathsf{Pr}_{(\sigma,s)\in\{0,1\}\times\{0,1\}^n}[A(C_n(\sigma,s))=\sigma\oplus a] \;=\; \frac{1}{2\pm 2\mu(n)}$. Thus,

$$\begin{aligned}
\mathsf{Pr}[D(X_n^a)=1] &= \mathsf{Pr}_{s\in\{0,1\}^n}[(D(C_n(0,s))=1)\,\wedge\,(A(C_n(0,s))=0\oplus a)] \\
&\quad + \mathsf{Pr}_{s\in\{0,1\}^n}[(D(C_n(1,s))=1)\,\wedge\,(A(C_n(1,s))=1\oplus a)]\pm\mu(n) \\
&= \mathsf{Pr}_{s\in\{0,1\}^n}[D(C_n(1,s))=1]+\Delta(n)\pm\mu(n)
\end{aligned}$$

where $\Delta(n)$ is defined as the difference between $\mathsf{Pr}_{s\in\{0,1\}^n}[(D(C_n(0,s))=1)\,\wedge\,(A(C_n(0,s))=a)]$ and $\mathsf{Pr}_{s\in\{0,1\}^n}[(D(C_n(1,s))=1)\,\wedge\,(A(C_n(1,s))=a)]$. Observe that $|\Delta(n)|$ is a negligible function, or else one may combine A and D and obtain a small circuit distinguishing $C_n(0)$ from $C_n(1)$ (in contradiction to our hypothesis regarding the commitment scheme C_n). Thus, for some negligible function μ', we have

$$\begin{aligned}
\mathsf{Pr}[D(X_n^a)=1] &= \mathsf{Pr}[D(C_n(1))=1]\pm\mu'(n) \\
&= \mathsf{Pr}_{\sigma\in\{0,1\}}[D(C_n(\sigma))=1]\pm\mu'(n)\pm\mu'(n) \\
&= \mathsf{Pr}[D(Y_n)=1]\pm 2\mu'(n)
\end{aligned}$$

and so, by the above discussion, Eq. (2.19) follows. Combining Eq. (2.18) and Eq. (2.19), we establish Eq. (2.17) as required.

[17] Recall $b_0=b$ and b_j is the j^{th} bit of r, for $j=1,...,n$.

[18] The abuse in in writing $A_2(c)$ as a shorthand for $A_2(T_{j-1},c)$.

We now turn to the case where the second party is honest. In this case B_2 is determined, and we transform (real-model) A_1 into (ideal-model) B_1. Machine B_1 will run machine A_1 locally, obtaining message it would have sent in a real execution of the protocol and feeding it with messages that it expects to receive. Our construction augments the one presented in the proof of Proposition 2.3.4, by using the strong proof-of-knowledge in order to extract, for each j, the bit σ_j being committed in Step C1.1 (by the corresponding c_j). The regular proof system is used as a guarantee that the commitment, c, sent in Step C3 indeed satisfies $c = C_n(b_0, b_1 \cdots b_n)$. Recall that B_1 gets input 1^n.

1. B_1 sends 1^n to the trusted party and obtains a uniformly distributed $(b, r) \in \{0,1\} \times \{0,1\}^n$. We stress that the trusted party has not answered to Party 2 yet, and that B_1 still has the option of stopping the trusted party before it does so.

2. B_1 sets $b_0 = b$ and b_j as the j^{th} bit of r, for $j = 1, ..., n$. It now tries to generate an execution of Step C1 which matches these bits (i.e., with respect to the setting in Step C1.3). Specifically, for each $j = 0, 1, ..., n$, it tries to extract σ_j, by using the strong knowledge extractor associated with the proof system of Step C1.2, and sets the σ'_j accordingly. (We remark that since b_j is uniformly distributed so is σ'_j.) Alongside, machine B_1 produces a view of A_1 of the execution of Step C1. Details follow.

   For $j = 0, 1, ..., n$, machine B_1 proceeds as follows (in emulating the j^{th} iteration of Step C1):

   (a) B_1 obtains from A_1 the Step C1.1 message, denoted c_j. In case A_1 aborts (or acts improperly) in the current iteration of Step C1.1, we let B_1 abort (outputting the transcript of the truncated execution).

   (b) B_1 emulates the verifier in an execution of the strong proof-of-knowledge of Step C1.2 using A_1 as the prover. In case the verifier rejects, B_1 aborts (outputting the transcript of the truncated execution). Otherwise, it records the transcript of the execution (of the proof-of-knowledge system), denoted T_j.

   (c) B_1 invokes the strong knowledge extractor to obtain a pair, (σ_j, s_j), so that $c_j = C_n(\sigma_j, s_j)$. In case the extraction fails, B_1 aborts.

   (d) B_1 sets $\sigma'_j = \sigma_j \oplus b_j$, and feeds it (together with T_j) to A_1. This sets A_1 for the next iteration.

3. In case A_1 aborts (or acts improperly) in Step C2, we let B_1 abort (outputting the transcript of the truncated execution). Otherwise, suppose that A_1 sends message c (supposedly $c = C_n(b, r)$).

4. B_1 emulates the verifier in an execution of the proof system of Step C3 using A_1 as the prover. In case the verifier rejects, B_1 aborts (outputting the transcript of the truncated execution). Otherwise, machine B_1 records the transcript of the execution (of the proof system), denoted T.

5. In case machine B_1 did not abort so far, it allows the trusted party to answer to Party 2.

6. Finally, B_1 feeds A_1 with the execution view so far (i.e., T), and outputs whatever A_1 does.

   Recall that in case B_1 has aborted due to the emulated Party 2 detecting improper behavior of A_1, it did so while outputting A_1's view of an aborting execution.

We now show that
$$\{\text{IDEAL}_{f,\overline{B}}(1^n\,,\,1^n)\}_{n\in\mathbb{N}} \stackrel{\text{s}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}(1^n\,,\,1^n)\}_{n\in\mathbb{N}} \tag{2.20}$$
The statistical difference is due to two cases corresponding to the two proof systems in use. The first case is that A_1 succeeds to convince the strong knowledge-verifier (played by Party 2 in Step C1) that it knows an opening of some c_j (i.e., a preimage $(\sigma_j\,,s_j)$ of c_j under C_n), and still the knowledge-extractor failed to find such an opening. The second case is that A_1 succeeds to convince Party 2 playing the verifier of Step C3 that $c = C_n(b_0, b_1 \cdots b_n)$ (where the b_j's are as above – equal $\sigma_j \oplus \sigma'_j$), and yet this is not the case. By definition of these proof systems, such events may occur only with negligible probability. Details follow.

Discussion – the statistical difference in Eq. (2.20): As stated above, the potential difference is due to two sources (or cases). The first case is that A_1 convinces A_2 in the real execution of some iteration of Step C1.2, but B_1 (using the strong knowledge-extractor) fails to extract the corresponding NP-witness. Let μ be the negligible function referred to in Definition 1.2.6. Then there are two sub-cases to consider.

1. A_1 convinces A_2 with probability at most $\mu(n)$. In this case there is no guarantee with respect to extraction of the NP-witness. However, in this case, with probability at least $1 - \mu(n)$, Party 2 aborts in the real model. Thus, the fact that in the ideal model, Party 2 aborts with probability at least $1-\mu(n)$ raises no problem. To summarize, the statistical difference in this case is bounded above by $\mu(n)$.

2. A_1 convinces A_2 with probability greater than $\mu(n)$. In this case, we are guaranteed that the extraction succeeds with very high probability; specifically, with probability at least $1 - \mu(n)$. Thus, ignoring the negligiblly-rare event of failure, in this case we can match each non-aborting execution in the real model by a corresponding non-aborting execution in the ideal model. Note that the unambiguity property of the commitment scheme guarantees that each extracted bit, σ_j, is the correct one. To summarize, the statistical difference in this case is also bounded above by $\mu(n)$.

We stress the essential role of the strong notion of a proof-of-knowledge (as defined in Definition 1.2.6) in the above argument. This definition provides a negligible function, denoted μ, so that whenever the convincing probability exceeds $\mu(n)$ – extraction succeeds with overwhelmingly high probability. For further discussion see Section 1.2.3. The second potential source of statistical difference in Eq. (2.20) is that A_1 convinces A_2 in the real execution of Step C3, but yet $c \neq C_n(b, r)$, where (b, r) are as are supposed to be (uniquely) determined in Step C2. By the soundness property of the proof system used in Step C3, in the latter case (i.e., $c \neq C_n(b, r)$) the real execution is non-aborting with negligible probability and the same holds for the simulation in the ideal model.

Discussion – the case where A_1 never lies in the proof systems: We next consider the case where A_1 never tries to prove false statements in either Step C1 or Step C3. Furthermore, we assume that in this case the extraction always succeeds (which is indeed the case when using an extractor of zero failure probability, as provided in Section 1.2.3). In this case, we show that $\text{IDEAL}_{f,\overline{B}}(1^n, 1^n)$ and $\text{REAL}_{\Pi,\overline{A}}(1^n, 1^n)$ are identically distributed. We first use the hypothesis that A_1 does not try to lie in the proof system of Step C3. Using the the hypothesis that the commitment scheme is unambiguous, it follows that the (c_j, σ'_j) pairs sent in Step C1 uniquely define the b_j's, and so uniquely define a value $c = C_n(b, r)$ for which Eq. (2.16) can be satisfied. Thus, both in the real execution and in the ideal model, Party 2 outputs the Step C2 message of A_1; that is, $c = C_n(b, r)$, where b and r are as determined is Step C1. Also note that both in the real execution and in the ideal model, the pair

(b, r) is uniformly distributed over $\{0,1\} \times \{0,1\}^n$. As for the output of Party 1, we claim that B_1 exactly emulates A_1. Looking at the construction of B_1, we note that the only possible deviation of B_1 from emulating A_1 may occur when it tries to extract the bits σ_j, for $j = 0, 1, ..., n$. We first note that in case extraction succeeds, it always yields the correct value (again, by unambiguity of the commitment scheme). Finally, by the case hypothesis, A_1 always convinces the verifier (in the iterations of Step C1.2) and extraction always succeeds.

The actual proof of Eq. (2.20): The real argument proceeds top down. That is, we start by considering what happens in the iterations of the real execution of Step C1 versus its emulation. Suppose that we are now at the j^{th} iteration (and that B_1 did not abort so far). The message c_j obtained by B_1 from A_1 is exactly the one sent by A_1 in the current iteration of Step C1.1. We now consider the probability, denoted p_j, that A_1 convinces the verifier in the strong proof-of-knowledge conducted in the current iteration of Step C1.2. (Indeed, p_j depends on the view of A_1 of the execution so far, but we omit this dependency from the notation.) We consider two cases, with respect to the negligible function μ referred to in Definition 1.2.6.

1. *Suppose* $p_t > \mu(n)$. In this case, with probability at least $1 - \mu(n)$, machine B_1 succeeds (using the strong knowledge-extractor) to obtain the (unique) bit σ_j so that c_j is in the range of $C_n(\sigma_j)$. In such a case, setting $\sigma'_j = \sigma_j \oplus b_j$, where b_j as obtained from the trusted party is uniformly distributed, machine B_1 perfectly emulates Step C1.3. Thus, with probability at least $(1 - \mu(n)) \cdot p_t$, machine B_1 perfectly emulates a non-aborting execution of the current iteration (of Step C1) by A_1. Also, with probability at least $(1 - p_t)$, machine B_1 perfectly emulates an aborting execution of the current iteration (of Step C1) by A_1. Thus, the emulation of the current iteration of Step C1 is statistically indistinguishable from the real execution (i.e., the statistical difference is at most $\mu(n)$).

2. *Suppose* $p_t \leq \mu(n)$. Again, real execution of the current iteration of Step C1 aborts with probability $1 - p_t$, which in this case is negligiblly close to 1. In emulating the current iteration of Step C1, with probability $1 - p_j$ we perfectly emulate an aborting execution, but there is no guarantee as to what happens otherwise. However, the uncontrolled behavior occurs only with probability $p_t \leq \mu(n)$. Thus, again, the emulation of the current iteration of Step C1 is statistically indistinguishable from the real execution.

We conclude that the emulation of Step C1 by B_2 is statistically indistinguishable from the real execution of Step C1 by A_1. We next turn to consider the execution (and emulation) of Step C3, assuming – off course – that the execution (and emulation) of Step C1 did not abort. Let $b_0, b_1, ..., b_n$ be bits as determined in a correct execution of Step C1.4. Note that assuming that the emulation did not abort, these bits are well-defined and actually equal the bits provided (to B_1) by the trusted party. Let c be the message sent by A_1 in Step C2. We consider two cases.

1. $c = C_n(b_0, b_1 \cdots b_n)$. In this case the emulation of Steps C2 and C3, as conducted by B_1, is perfect. Note that this does not necessarily mean that the emulation does not abort, as it may abort whenever the real execution does. (This may happen when A_1 fails to convince Party 2 in the real execution, an event which may happen as A_1 is arbitrary.) We stress that in case B_1 does not abort, the trusted party hands $C_n(b_0, b_1 \cdots b_n) = c$ to Party 2 (in the ideal model), and so B_2 outputs c – exactly as A_2 does in the real execution.

2. $c \neq C_n(b_0, b_1 \cdots b_n)$. In this case, the emulation of rejecting (and so aborting) executions of Step C3 is perfect. Recall that by the soundness of the proof system accepting executions

occur only with negligible probability. Indeed, these executions are not correctly emulated by B_1 (as the answer provided to Party 2 in the ideal model differs from the message A_2 receives from A_1, and consequently the output of Party 2 differ in the two models). However, since non-aborting executions in this case occur with negligible probability, the emulation of the execution is statistically indistinguishable from the real execution.

Thus, in the worst case, the emulation conducted by B_1 is statistically indistinguishable from the real execution as viewed by A_1. Eq. (2.20) follows and does the proposition. ■

2.3.2.2 Input Commitment Protocol

Let $\{C_n\}_{n\in\mathbb{N}}$ be a commitment scheme. Our goal is to have Party 1 commit to its input using this scheme. To facilitate the implementation we make the randomization to be used for the commitment be outside the protocol (functionality). In typical applications, the input x will be given by a high-level protocol which also generates r at random. For simplicity, we consider the basic case where x is a bit.

$$((x,r),1^n) \mapsto (\lambda, C_n(x,r)) \tag{2.21}$$

At first glance, one may say that Eq. (2.21) is obviously implementable by just letting Party 1 apply the commitment scheme to its input and send the result to Party 2. However, this naive suggestion does not guarantee that the output is in the range of the commitment scheme, and so is not secure in the malicious model. Furthermore, a secure implementation of the functionality requires that Party 1 "knows" a preimage of the commitment value output by Party 2 (see discussion following Definition 2.1.6). Thus, the naive protocol must be augmented by Party 1 proving to Party 2 (in zero-knowledge) that it knows such a preimage. The resulting protocol follows.

Construction 2.3.8 (input-bit commitment protocol):

Inputs: *Party 1 gets input $(\sigma,r)\in\{0,1\}\times\{0,1\}^n$, and Party 2 gets input 1^n.*

Step C1: *Party 1 sends $c \stackrel{\text{def}}{=} C_n(\sigma,r)$ to Party 2.*

Step C2: *The parties invoke a zero-knowledge strong-proof-of-knowledge so that Party 1 plays the prover and Party 2 plays the verifier. The common input to the proof system is c, the prover gets auxiliary inputs (σ,r), and its objective is to prove that it knows (x,y) such that*

$$c = C_n(x,y) \tag{2.22}$$

In case the verifier rejects the proof, Party 2 aborts with output $\perp$ (otherwise the output will be c). (Again, any possible response – including abort – of Party 2 during the execution of this step, will be interpreted by Party 1 as a canonical legitimate message.)

Outputs: *Party 2 sets its local output to c.* (Party 1 has no output.)

Observe that the specified strategies are indeed implementable in polynomial-time. In particular, in Step C2, Party 1 supplies the prover subroutine with the NP-witness (σ,r) which indeed satisfies Eq. (2.22) (with $x=\sigma$ and $y=r$). Also, using the non-triviality condition of the proof system it follows that if both parties are honest then neither aborts and the output is as required. We comment that the above protocol does not rely on $\{C_n\}$ being a commitment scheme, and remains valid for any family of functions $\{f_n : \{0,1\}\times\{0,1\}^n \mapsto \{0,1\}^{\text{poly}(n)}\}_{n\in\mathbb{N}}$.

Proposition 2.3.9 *Construction 2.3.8 securely computes* (in the malicious model) *the functionality* Eq. (2.21)*, where* $\{C_n : \{0,1\} \times \{0,1\}^n \mapsto \{0,1\}^{\mathrm{poly}(n)}\}_{n\in\mathbb{N}}$.

Proof Sketch: Again, we need to show how to (efficiently) transform any admissible circuit pair, (A_1, A_2), for the real model into a corresponding pair, (B_1, B_2), for the ideal model. We treat separately each of the two cases – defined by which of the parties is honest.

We start with the case that the first party is honest. In this case B_1 is determined, and we transform (the real-model adversary) A_2 into (an ideal-model adversary) B_2, which uses A_2 as a subroutine. Recall that B_2 gets input 1^n.

1. B_2 send 1^n to the trusted party and obtain the commitment value c (which equals $C_n(\sigma, r)$ for (σ, r) handed by Party 1).

2. B_2 invokes the simulator guaranteed for the zero-knowledge proof system, on input c, using A_2 as a possible malicious verifier. Denote the obtained simulation transcript by $S = S(c)$.

3. Finally, B_2 feed A_2 with the supposedly execution view, (c, S) and outputs whatever A_2 does.

We need to show that for the functionality, f, of Eq. (2.21) and Π of Construction 2.3.8,

$$\{\text{IDEAL}_{f,\overline{B}}((\sigma, r)\,,\,1^n)\}_{n\in\mathbb{N},\,(\sigma,r)\in\{0,1\}\times\{0,1\}^n} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}((\sigma, r)\,,\,1^n)\}_{n\in\mathbb{N},\,(\sigma,r)\in\{0,1\}\times\{0,1\}^n} \quad (2.23)$$

Let $R(\sigma, r)$ denote the verifier view of the real interaction on common input $C_n(\sigma, r)$, prover's auxiliary input (σ, r), and verifier played by B_2. Then,

$$\begin{aligned}
\text{REAL}_{\Pi,\overline{A}}((\sigma, r)\,,\,1^n) &= (\perp\,,\,A_2(R(\sigma, r))) \\
\text{IDEAL}_{f,\overline{B}}((\sigma, r)\,,\,1^n) &= (\perp\,,\,A_2(S(C_n(\sigma, r))))
\end{aligned}$$

However, by the standard formulation of zero-knowledge – which guarantees computationally indistinguishable simulations also in the presence of auxiliary inputs – we have that $((\sigma, r), S(C_n(\sigma, r)))$ and $((\sigma, r), R(\sigma, r))$ are computationally indistinguishable for any fixed (σ, r), and so Eq. (2.23) follows.

We now turn to the case where the second party is honest. In this case B_2 is determined, and we transform (real-model) A_1 into (ideal-model) B_1, which uses A_1 as a subroutine. Recall that B_1 gets input $(\sigma, r) \in \{0,1\} \times \{0,1\}^n$.

1. B_1 invokes A_1 on input (σ, r). In case A_1 aborts (or acts improperly) in Step C1, we let B_1 abort before invoking the trusted party. Otherwise, suppose that A_1 sends message c (i.e., $c = A_1(\sigma, r)$). (Supposedly c is in the range of C_n.)

2. Machine B_1 tries to obtain the a preimage of c under C_n. Towards this end, B_1 uses the knowledge-extractor associated with the proof system of Step C2. Specifically, using the strong knowledge-extractor, B_1 tries to extract from A_1 a pair (x, y) satisfying Eq. (2.22). In case the extractor succeeds, B_1 sets $\sigma' = x$ and $r' = y$. If the extraction fails, machine B_1 aborts (before invoking the trusted party). Otherwise, we proceed as follows.

3. Machine B_1 now emulates an execution of Step C2. Specifically, it lets $A_1(\sigma, r)$ play the prover and emulates the (honest) verifier interacting with it (i.e., behaves as A_2). In case the emulated verifier rejects, machine B_1 aborts (before invoking the trusted party). Otherwise, it sends (σ', r') to the trusted party, and allows it to respond to Party 2. (The response will be $C_n(\sigma', r') = c$.)

4. Finally, B_1 feed A_1 with the execution view, which consists of the prover's view of the emulation of Step C2 (produced in Step 3 above), and outputs whatever A_1 does.

We now show that

$$\{\text{IDEAL}_{f,\overline{B}}((\sigma, r)\,,\,1^n)\}_{n\in\mathbb{N},\,(\sigma,r)\in\{0,1\}\times\{0,1\}^n} \stackrel{\text{s}}{\equiv} \{\text{REAL}_{\Pi,\overline{A}}((\sigma, r)\,,\,1^n)\}_{n\in\mathbb{N},\,(\sigma,r)\in\{0,1\}\times\{0,1\}^n} \quad (2.24)$$

The statistical difference is due to the case where A_1 succeeds to convince the strong knowledge-verifier (played by A_2) that it knows a preimage of c under C_n and still the knowledge-extractor failed to find such a preimage. By definition of strong knowledge-verifiers, such an event may occur only with negligible probability. Loosely speaking, the rest of the argument shows that, ignoring the rare case in which extraction fails although the knowledge-verifier (played by A_2) is convinced, the distributions $\text{IDEAL}_{f,\overline{B}}((\sigma, r), 1^n)$ and $\text{REAL}_{\Pi,\overline{A}}((\sigma, r), 1^n)$ are identical.

Consider first, for simplicity, the case where B_1 never aborts (i.e., never stops the trusted party). In this case, both in the real execution and in the ideal model, Party 2 outputs the Step C1 message of A_1; that is, $A_1(\sigma, r)$. Thus, they both equal $(A_1((\sigma, r), T)\,,\,A_1(\sigma, r))$, where T represents the (distribution of the) prover's view of an execution of Step C2, on common input c, in which the prover is played by $A_1(\sigma, r)$.

Next, consider the case that A_1 always aborts (i.e., either it aborts in Step C1 or it never convinces the verifier in Step C2). In this case, B_1 aborts before invoking the trusted party, and so both ensembles are identical (i.e., both equal $(A_1((\sigma, r), \perp), \perp)$). Since A_1 is deterministic, we are left with the case in which A_1 appears to behave properly in Step C1 and, in Step C2, machine $A_1(\sigma, r)$ convinces Party 2 with some probability, denoted p, taken over the moves of Party 2. We consider two cases, with respect to the negligible function μ referred to in Definition 1.2.6.

1. Suppose $p > \mu(n)$. In this case, by definition of a strong proof of knowledge, with probability at least $1 - \mu(n)$, machine B_1 has successfully extracted (σ', r') in Step 2. Thus, the situation is as in the simple case (above), except that with probability $1 - p$, the joint execution in the real model ends up aborting. In the ideal model a joint execution is aborting with probability $1 - p \pm \mu(n)$ (actually, the probability is at least $1 - p$ and at most $1 - p + \mu(n)$). As in the simple case (above), non-aborting executions are distributed identically in both models. (The same holds with respect to aborting executions which equal $(A_1((\sigma, r), \perp), \perp)$ in both models.)

2. Suppose that $p \leq \mu(n)$. Again, in the real model the abort probability is $1 - p$, which in this case is negligiblly close to 1. In the ideal model we are only guaranteed that aborting executions occur with probability at least $1 - p$, which suffices for us (recalling that aborting executions are equal in both models, and noting that they occur with probability at least $1 - \mu(n)$ in both models).

We conclude that in both cases the distributions are statistically indistinguishable, and the proposition follows. ■

2.3.2.3 Authenticated Computation Protocol

Let $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$ and $h : \{0,1\}^* \mapsto \{0,1\}^*$ be polynomial-time computable. Intuitively, our goal is to force Party 1 to send $f(\alpha, \beta)$ to Party 2, where β is known to both parties, α is known to Party 1, and $h(\alpha)$ – which determines α in case h is 1-1 – is known to Party 2. That is, we are interested in the functionality

$$((\alpha, \beta), (h(\alpha), \beta)) \;\mapsto\; (\lambda, f(\alpha, \beta)) \quad (2.25)$$

The above formulation makes acute a issue which is present also in all previous functionalities considered: What happens if the parties provide inputs which do not satisfy the relations postulated above (i.e., Party 1 provides input (α, β) and Party 2 provides (γ', β') where either $\beta \neq \beta'$ or $h(\alpha) \neq \gamma'$). Our convention is that in this case the output is $(\perp, \perp)$ (see discussion in the preamble to Section 2.1).

To facilitate the implementation, we assume that the function h is one-to-one, as will be the case in our applications. This allows us to use (ordinary) zero-knowledge proofs, rather than strong (zero-knowledge) proofs-of-knowledge. We also assume, for simplicity, that for some polynomial p and all α's, the function h satisfies $|h(\alpha)| = p(|\alpha|)$.[19]

The functionality of Eq. (2.25) is implemented by having Party 1 send $f(\alpha, \beta)$ to Party 2, and then prove in zero-knowledge the correctness of the value sent (with respect to the common input $(h(\alpha), \beta)$). Note that this statement is of the NP-type and that Party 1 has the NP-witness. Actually, the following protocol is *the archetypical* application of zero-knowledge proof systems.

Construction 2.3.10 (authenticated computation protocol):

Inputs: *Party 1 gets input $(\alpha, \beta) \in \{0,1\}^* \times \{0,1\}^*$, and Party 2 gets input (u, β), where $u = h(\alpha)$.*

Step C1: *Party 1 sends $v \stackrel{\text{def}}{=} f(\alpha, \beta)$ to Party 2.*

Step C2: *The parties invoke a zero-knowledge proof system so that Party 1 plays the prover and Party 2 plays the verifier. The common input to the proof system is (v, u, β), the prover gets auxiliary inputs α, and its objective is to prove that*

$$\exists x \text{ s.t. } (u = h(x)) \wedge (v = f(x, \beta)) \tag{2.26}$$

(We stress that the common input is supplied by the verifier, which sets the first element to be the message received in Step C1, and the two other elements to be as in its input.) *The proof system employed has negligible soundness error. In case the verifier rejects the proof, Party 2 aborts with output $\perp$ (otherwise the output will be v). (Again, any possible response – including abort – of Party 2 during the execution of this step, will be interpreted by Party 1 as a canonical legitimate message.)*

Outputs: *Party 2 sets its local output to v.* (Party 1 has no output.)

Observe that the specified strategies are indeed implementable in polynomial-time. In particular, in Step C2, Party 1 supplies the prover subroutine with the NP-witness α so that Eq. (2.26) is satisfied with $x = \alpha$. Also, using the completeness condition of the proof system it follows that if both parties are honest then neither aborts and the output is as required. We stress that, unlike the previous two protocols, the current protocol only utilizes an ordinary (zero-knowledge) proof system (rather than a strong proof-of-knowledge).

Proposition 2.3.11 *Suppose that the function h is one-to-one. Then, Construction 2.3.10 securely computes* (in the malicious model) *the functionality* Eq. (2.25).

[19] This assumption can be enforced by redefining h so that $h(\alpha) \stackrel{\text{def}}{=} h(\alpha) \cdot 0^{p(|\alpha|)-|h(\alpha)|}$, where $p(|\alpha|) - 1$ is an upper bound on the time-complexity of the original h.

Proof Sketch: Again, we need to show how to (efficiently) transform any admissible circuit pair, (A_1, A_2), for the real model into a corresponding pair, (B_1, B_2), for the ideal model. We treat separately each of the two cases – defined by which of the parties is honest. Assume, for simplicity, that $|\alpha| = |\beta|$.

We start with the case that the first party is honest. In this case B_1 is determined, and we transform (the real-model adversary) A_2 into (an ideal-model adversary) B_2, which uses A_2 as a subroutine. Recall that B_2 gets input (u, β), where $u = h(\alpha)$.

1. B_2 send (u, β) to the trusted party and obtain the value v, which equals $f(\alpha, \beta)$ for (α, β) handed by Party 1.

2. B_2 invokes the simulator guaranteed for the zero-knowledge proof system, on input v, using A_2 as a possible malicious verifier. Denote the obtained simulation transcript by $S = S(v)$.

3. Finally, B_2 feed A_2 with the supposedly execution view, (v, S) and outputs whatever A_2 does.

Repeating the analogous arguments of the previous proofs, we conclude that for the functionality, f, of Eq. (2.25) and Π of Construction 2.3.10,

$$\{\mathrm{IDEAL}_{f,\overline{B}}((\alpha,\beta)\,,\,(h(\alpha),\beta))\}_{n\in\mathbb{N},\ \alpha,\beta\in\{0,1\}^n} \stackrel{\mathrm{c}}{\equiv} \{\mathrm{REAL}_{\Pi,\overline{A}}((\alpha,\beta)\,,\,(h(\alpha),\beta))\}_{n\in\mathbb{N},\ \alpha,\beta\in\{0,1\}^n}$$

We now turn to the case where the second party is honest. In this case B_2 is determined, and we transform (real-model) A_1 into (ideal-model) B_1, which uses A_1 as a subroutine. Recall that B_1 gets input $(\alpha, \beta) \in \{0,1\}^n \times \{0,1\}^n$.

1. B_1 invokes A_1 on input (α, β). In case A_1 aborts (or acts improperly) in Step C1, we let B_1 abort before invoking the trusted party. Otherwise, suppose that A_1 sends message v (i.e., $v = A_1(\alpha, \beta)$).

2. Machine B_1 checks that v supplied in Step 1 indeed satisfies Eq. (2.26) with respect to $u = h(\alpha)$, where (α, β) are as above (i.e., the input to B_1). This is done by emulating the proof system of Step C2 so that $A_1(\alpha, \beta)$ plays the prover and B_1 plays the (honest) verifier (i.e., behaves as A_2). Recall that this proof system has negligible soundness error, and so if v does not satisfy Eq. (2.26) this is detected with probability $1 - \mu(n)$, where μ is a negligible function. If the verifier (played by B_1) rejects then machine B_1 aborts (before invoking the trusted party). Otherwise, we proceed assuming that v satisfies Eq. (2.26). Note that since h is 1-1 and Eq. (2.26) is satisfied it must be the case that $v = f(h^{-1}(u), \beta) = f(\alpha, \beta)$.[20]

3. Machine B_1 sends (α, β) to the trusted party, and allows it to respond to Party 2. (The response will be $f(\alpha, \beta) = v$.)

4. Finally, B_1 feed A_1 with the execution view, which consists of the prover's view of the emulation of Step C2 (produced in Step 2 above), and outputs whatever A_1 does.

We now show that

$$\{\mathrm{IDEAL}_{f,\overline{B}}((\alpha,\beta)\,,\,(h(\alpha),\beta))\}_{n\in\mathbb{N},\ \alpha,\beta\in\{0,1\}^n} \stackrel{\mathrm{s}}{\equiv} \{\mathrm{REAL}_{\Pi,\overline{A}}((\alpha,\beta)\,,\,(h(\alpha),\beta))\}_{n\in\mathbb{N},\ \alpha,\beta\in\{0,1\}^n} \quad (2.27)$$

[20] We comment that if h were not 1-1 and a strong proof-of-knowledge (rather than an ordinary proof system) was used in Step C2 then one could have inferred that Party 1 knows an α' so that $h(\alpha') = u$ and $v = f(\alpha', \beta)$, but α' does not necessarily equal α. Sending (α', β) to the trusted party in the next step, we would have been fine, as it would have (also) meant that the trusted party's respond to Party 2 is v.

The statistical difference is due to the case where A_1 succeeds to convince the verifier (played by A_2) that it v satisfies Eq. (2.26), and yet this claim is false. By soundness of the proof system, this event happens only with negligible probability. The rest of the argument is a simplified version of the corresponding parts of the previous proofs. Specifically, assuming that v satisfies Eq. (2.26), we show that $\text{IDEAL}_{f,\overline{B}}((\alpha,\beta),(h(\alpha),\beta))$ and $\text{REAL}_{\Pi,\overline{A}}((\alpha,\beta),(h(\alpha),\beta))$ are identically distributed.

Consider first, the case that A_1 always aborts in Step C1 (or is detected to behave improperly – which is treated as abort). In this case, B_1 aborts before invoking the trusted party, and so both ensembles are identical (i.e., both equal $(A_1((\alpha,\beta),\perp),\perp)$). Since A_1 is deterministic, we are left with the case in which A_1 appears to behave properly in Step C1 and, in Step C2, machine $A_1(\alpha,\beta)$ convinces Party 2 with some probability, denoted p, taken over the moves of Party 2. We consider two cases, with respect to the soundness error-bound function μ associated with the proof system. We stress that such an explicit function can be associated with all standard zero-knowledge proof systems, and here we use a system for which μ is negligible. For example, we may use a system with error bound $\mu(n) \stackrel{\text{def}}{=} 2^{-n}$.

1. Suppose $p > \mu(n)$. In this case, by the soundness condition, it must be the case that $A_1(\alpha,\beta) = f(\alpha,\beta)$ (since in this case $v \stackrel{\text{def}}{=} A_1(\alpha,\beta)$ satisfies Eq. (2.26) and so $v = f(h^{-1}(\alpha),\beta) = f(\alpha,\beta)$). Thus, in both the real and the ideal model, with probability p, the joint execution is non-aborting and equals $(A_1((\alpha,\beta),T)\,,\,A_1(\alpha,\beta))$, where T represents the (distribution of the) prover's view of an execution of Step C2, on common input $(h(\alpha),\beta,f(\alpha,\beta))$, in which the prover is played by $A_1(\alpha,\beta)$. Also, in both models, with probability $1-p$, the joint execution is aborting and equal $(A_1((\alpha,\beta),\perp),\perp)$. Thus, in this case the distributions in Eq. (2.27) are identical.

2. Suppose that $p \leq \mu(n)$. Again, in both models aborting executions are identical and occur with probability $1-p$ (as the ideal model aborts only during a single emulation of the real model). In this case we have no handle on the non-aborting executions in the ideal model (as $A_1(\alpha,\beta)$ may be arbitrary), but we do not care since these occur with negligible probability (i.e., $p \leq \mu(n)$). Thus, in this case the distributions in Eq. (2.27) are statistically indistinguishable.

The proposition follows. ■

Authenticated Computation Protocol, generalized. Actually, we will use a slightly more general functionality in which h is a randomized process rather than a function. Alternatively, we consider a two-argument function h (rather than a single argument one), and the following functionality.

$$((\alpha,r,\beta),(h(\alpha,r),\beta)) \;\mapsto\; (\lambda, f(\alpha,\beta)) \tag{2.28}$$

Analogously to above, we make the assumption that h is 1-1 with respect to its first argument; that is, for every $\alpha \neq \alpha'$ and any r, r' we have $h(\alpha,r) \neq h(\alpha',r')$. Construction 2.3.10 generalizes in the obvious way and we obtain.

Proposition 2.3.12 *Suppose that the function $h : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$ satisfies that for every $\alpha \neq \alpha'$, the sets $\{h(\alpha,r) : r \in \{0,1\}^*\}$ and $\{h(\alpha',r) : r \in \{0,1\}^*\}$ are disjoint. Then, the functionality of* Eq. (2.28) *can be securely computed* (in the malicious model).

Proof Sketch: For clarity, we reproduce the generalized protocol.

Inputs: Party 1 gets input $(\alpha,r,\beta) \in (\{0,1\}^*)^3$, and Party 2 gets input (u,β), where $u = h(\alpha,r)$.

Step C1: As before, Party 1 sends $v \stackrel{\text{def}}{=} f(\alpha, \beta)$ to Party 2.

Step C2: As before, the parties invoke a zero-knowledge proof system so that Party 1 plays the prover and Party 2 plays the verifier. The common input to the proof system is (v, u, β), the prover gets auxiliary inputs (α, r), and its objective is to prove that

$$\exists x, y \text{ s.t. } (u = h(x, y)) \wedge (v = f(x, \beta)) \tag{2.29}$$

In case the verifier rejects the proof, Party 2 aborts with output $\perp$ (otherwise the output will be v). (Again, any possible response – including abort – of Party 2 during the execution of this step, will be interpreted by Party 1 as a canonical legitimate message.)

Outputs: As before, Party 2 sets its local output to v. (Party 1 has no output.)

The fact that this generalized protocol securely computes the functionality Eq. (2.28) is proven by following the proof of Proposition 2.3.11. The only thing to notice is that the first element of a preimage in the range of h is still uniquely defined. ■

2.3.3 The compiler itself

We are now ready to present the compiler. Recall that we are given a protocol, Π, for the semi-honest model, and we want to generate an "equivalent" protocol Π' for the malicious model. The meaning of the term 'equivalent' will be clarified below. We assume, without loss of generality, that on any input of length n, each party to Π tosses $c(n) = \text{poly}(n)$ coins.

Construction 2.3.13 (The two-party compiler): *Given a protocol, Π, for the semi-honest model, the compiler produces a protocol, Π', for the malicious model. Following is a specification of the resulting protocol Π'.*

Inputs: *Party 1 gets input $x = x_1 x_2 \cdots x_n \in \{0,1\}^n$ and Party 2 gets input $y = y_1 y_2 \cdots y_n \in \{0,1\}^n$.*

Input-commitment phase: *Each of the parties commits to each of its input bits by using a secure implementation of the input-commitment functionality of* Eq. (2.21). *Recall that these executions should be preceded by the "committing party" selecting a randomization for the commitment scheme C_n. That is, for $i = 1$ to n, the parties do:*[21]

- *Party 1 uniformly selects $\rho_i^1 \in \{0,1\}^n$, and invokes a secure implementation of the input-commitment functionality of* Eq. (2.21), *playing Party 1 with input (x_i, ρ_i^1). Party 2 plays the role of Party 2 in* Eq. (2.21) *with input 1^n. Party 2 obtains the output $C_n(x_i, \rho_i^1)$.*
- *Analogously, Party 2 uniformly selects $\rho_i^2 \in \{0,1\}^n$, and invokes a secure implementation of the input-commitment functionality of* Eq. (2.21), *playing Party 1 with input (y_i, ρ_i^2). Party 1 plays the role of Party 2 in* Eq. (2.21) *Party 1 obtains the output $C_n(y_i, \rho_i^2)$.*

Note that each party now holds a string which uniquely determines the n-bit long input of the other party. Specifically, Party 1 (resp., Party 2) *holds $C_n(y_1, \rho_1^2), ..., C_n(y_n, \rho_n^2)$* (resp., *$C_n(x_1, \rho_1^1), ..., C_n(x_n, \rho_n^1)$). In addition, each party, holds an NP-witness for the value of the input committed to by the sequence held by the other party; that is, Party i holds the witness $\rho_1^i, ..., \rho_n^i$.*

[21] The order in which these $2n$ commitments are run is immaterial. Here we chose an arbitrary one. The same holds for the protocols in the next phase.

Coin-generation phase: *The parties generate random-pad for the emulation of* Π*. Each party obtains the bits of the random-pad to be held by it, whereas the other party obtains commitments to these bits. The party holding the bit also obtains the randomization used in these commitments, to be used as an NP-witness to the correctness of the committed value. This is done by invoking a secure implementation of the* (augmented) *coin-tossing functionality of* Eq. (2.14)*. Specifically, the coin-tossing protocol is invoked* $2c(n)$ *times,* $c(n)$ *times in each of the two directions.*

That is, for $i = 1$ *to* $c(n)$*, the parties do*

- *Party 1 invokes a secure implementation of the coin-tossing functionality of* Eq. (2.14) *playing Party 1 with input* 1^n*. Party 2 plays the role of Party 2 in* Eq. (2.14) *with input* 1^n*. Party 1 obtains a pair,* (r_i^1, ω_i^1)*, and Party 2 obtains the corresponding output* $C_n(r_i^1, \omega_i^1)$*.*

  Party 1 sets the i^{th} *bit of the random-pad for the emulation of* Π *to be* r_i^1*, and records the corresponding NP-witness. Party 2 records* $C_n(r_i^1, \omega_i^1)$*.*
- *Party 2 invokes a secure implementation of the coin-tossing functionality of* Eq. (2.14) *playing Party 1 with input* 1^n*. Party 1 plays the role of Party 2 in* Eq. (2.14) *with input* 1^n*. Party 2 obtains a pair,* (r_i^2, ω_i^2)*, and Party 1 obtains the corresponding output* $C_n(r_i^2, \omega_i^2)$*.*

  Party 2 sets the i^{th} *bit of the random-pad for the emulation of* Π *to be* r_i^2*, and records the corresponding NP-witness. Party 1 records* $C_n(r_i^2, \omega_i^2)$*.*

Each party, sets the random-pad for Π *to be the concatenation of the corresponding bits. That is, for* $j = 1, 2$*, Party* j *sets* $r^j = r_1^j r_2^j \cdots r_{c(n)}^j$*.*

Note that each party holds a string which uniquely determines the random-pad of the other party.

Protocol emulation phase: *The parties use a secure implementation of the authenticated-computation functionality of* Eq. (2.28) *in order to emulate each step of protocol* Π*. The party which is supposed to send a message plays the role of Party 1 in* Eq. (2.28) *and the party which is supposed to receive it plays the role of Party 2. The inputs* α, r, β *and the functions* h, f*, for the functionality of* Eq. (2.28)*, are set as follows:*

- *The string* α *is set to equal the concatenation of the party's original input and its random-pad, the string* r *is set to be the concatenation of the corresponding randomizations used in the commitments and* $h(\alpha, r)$ *equals the concatenation of the commitments themselves. That is, suppose the message is supposed to be sent by Party* j *in* Π *and that its input is* z *(i.e.,* $z = x$ *if* $j = 1$ *and* $z = y$ *otherwise). Then*

$$\begin{aligned}
\alpha &= (z, r^j)\,, \text{ where } r^j = r_1^j r_2^j \cdots r_{c(n)}^j \\
r &= (\rho_1^j \rho_2^j \cdots \rho_n^j\,, \omega_1^j \omega_2^j \cdots \omega_{c(n)}^j) \\
h(\alpha, r) &= (C_n(z_1, \rho_1^j), C_n(z_2, \rho_2^j), ..., C_n(z_n, \rho_n^j)\,, \\
&\qquad C_n(r_1^j, \omega_1^j), C_n(r_2^j, \omega_2^j), ..., C_n(r_{c(n)}^j, \omega_{c(n)}^j))
\end{aligned}$$

  Note that h *indeed satisfies* $h(\alpha, r) \neq h(\alpha', r')$ *for all* $\alpha \neq \alpha'$ *and all* r, r'*.*

- *The string β is set to equal the concatenation of all previous messages sent by the other party.*
- *The function f is set to be the computation which determines the message to be sent in Π. Note that this message is computable in polynomial-time from the party's input* (denoted z above), *its random-pad* (denoted r^j), *and the messages it has received so far* (i.e., β).

Aborting: *In case any of the protocols invoked in any of the above phases terminates in an abort state, the party (or parties) obtaining this indication aborts the execution, and sets its output to $\perp$. Otherwise, outputs are as follows.*

Outputs: *At the end of the emulation phase, each party holds the corresponding output of the party in protocol Π. The party just locally outputs this value.*

We note that the compiler is efficient. That is, given the code of a protocol Π, the compiler produces the code of Π' in polynomial-time. Also, in case both parties are honest, the input-output relation of Π' is identical to that of Π.

2.3.3.1 The effect of the compiler

As will be shown below, given a protocol as underlying the proof of Theorem 2.2.13, the compiler produces a protocol which securely computes the same function. Thus, for any functionality f, the compiler transforms a protocol for *privately* computing f (in the semi-honest model) into a protocol for *securely* computing f (in the malicious model). The above suffices to establish our main result (i.e., Theorem 2.3.1), yet it does not say what the compiler does when given an arbitrary protocol (i.e., one not produced as above). In order to analyze the action of the compiler, in general, we introduce the following model which is a hybrid of the semi-honest and the malicious models.[22] We call this new model, which may be of independent interest, the *augmented semi-honest* model.

Definition 2.3.14 (the augmented semi-honest model): *Let Π be a two-party protocol. An* augmented semi-honest behavior (w.r.t Π) *for one of the parties is a* (feasible) *strategy which satisfies the following conditions.*

Entering the execution: *Depending on its initial input, denoted z, the party may abort before taking any step in the execution of Π. Otherwise, again depending on z, it enter the execution with any input $z' \in \{0,1\}^{|z|}$ of its choice. From this point on z' is fixed.*

Proper selection of random-pad: *The party selects the random-pad to be used in Π uniformly among all strings of the length specified by Π. That is, the selection of the random-pad is exactly as specified by Π.*

Proper message transmission or abort: *In each step of Π, depending on its view so far, the party may either abort or send a message as instructed by Π. We stress that the message is computed as Π instructs based on input z', the random-pad selected above, and all messages received so far.*

[22] Indeed, Theorem 2.3.1 will follow as a special case of the general analysis of the compiler provided below. Our treatment decouples the effect of the compiler from properties of protocols which when compiled (by the compiler) yield a secure in the malicious model implementation of a desired functionality. This footnote is clarified by the text below.

Output: *At the end of the interaction, the party produces an output depending on its entire view of the interaction. We stress that the view consists of the initial input z, the random-pad selected above, and all messages received so far.*

A pair of polynomial-size circuit families, $\overline{C} = (C_1, C_2)$, is admissible w.r.t Π in the augmented semi-honest model *if one family implements Π and the other implements an augmented semi-honest behavior w.r.t Π.*

Intuitively, the compiler transforms any protocol Π into a protocol Π' so that executions of Π' in the malicious model correspond to executions of Π in the augmented semi-honest model. That is,

Proposition 2.3.15 (general analysis of the two-party compiler): *Let Π' be the protocol produced by the compiler of Construction 2.3.13, when given the protocol Π. Then, there exists a polynomial-time computable transformation of pairs of polynomial-size circuit families $\overline{A} = (A_1, A_2)$ admissible* (w.r.t Π') *for the* (real) *malicious model* (of Definition 2.1.5) *into pairs of polynomial-size circuit families $\overline{B} = (B_1, B_2)$ admissible* w.r.t Π *for the augmented semi-honest model* (of Definition 2.3.14) *so that*

$$\{\text{REAL}_{\Pi,\overline{B}}(x, y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi',\overline{A}}(x, y)\}_{x,y \text{ s.t. } |x|=|y|}$$

Proposition 2.3.15 will be applied to protocols as underlying the proof of Theorem 2.2.13. As we shall see (in §2.3.3.2 below), for these *specific* protocols, the augmented semi-honest model (of Definition 2.3.14) can be emulated by the ideal malicious model (of Definition 2.1.4). Thus, Theorem 2.3.1 will follow (since, schematically speaking, for every functionality f there exist Π and Π' so that $\text{IDEAL}_{f,\text{malicious}}(x, y)$ equals $\text{REAL}_{\Pi,\text{aug-semi-honest}}(x, y)$, which in turn equals $\text{REAL}_{\Pi',\text{malicious}}(x, y)$). Thus, Theorem 2.3.1 is proven by combining the properties of the compiler, as stated in Proposition 2.3.15, with the properties of specific protocols to be compiled by it. We believe that this decoupling clarifies the proof. We start by establishing Proposition 2.3.15.

Proof Sketch: Given a circuit pair, (A_1, A_2), admissible w.r.t Π' for the real malicious model, we present a corresponding pair, (B_1, B_2), admissible w.r.t Π for the augmented semi-honest model. Denote by `hon` the identity of the honest party and by `mal` the identity of the malicious party (`mal` $= 1$ if `hon` $= 2$ and `mal` $= 2$ otherwise). Then, $B_{\texttt{hon}}$ is determined, and we transform (the malicious adversary) $A_{\texttt{mal}}$ into (an augmented semi-honest adversary) $B_{\texttt{mal}}$, which uses $A_{\texttt{mal}}$ as a subroutine. Actually, machine $B_{\texttt{mal}}$ will use $A_{\texttt{mal}}$ as well as the ideal-model (malicious) adversaries derived from the behavior of $A_{\texttt{mal}}$ in the various subprotocols invoked by Π'. Furthermore, machine $B_{\texttt{mal}}$ will also emulate the behavior of the trusted party in these ideal-model emulations (without communicating with any trusted party – there is no trusted party in the augmented semi-honest model). Thus, the following description contains an implicit special-purpose composition theorem.[23]

On input $z = z_1 z_2 \cdots z_n \in \{0,1\}^n$, machine $B_{\texttt{mal}}$ behaves as follows.

Entering the execution: $B_{\texttt{mal}}$ invokes $A_{\texttt{mal}}$ on input z, and decides whether to enter the protocol, and if so – with what input. Towards this end, machine $B_{\texttt{mal}}$ emulates execution of the input-committing phase of Π', using $A_{\texttt{mal}}$ (as subroutine). Machine $B_{\texttt{mal}}$ supplies $A_{\texttt{mal}}$ with the messages it expects to see, thus emulating a honest Party `hon` in Π', and obtains the messages

[23] It is indeed our choice neither to make this composition theorem explicit nor to state a general-purpose composition theorem for the malicious model. We believe that the implicit composition is easy to understand, whereas an explicit statement would require some technicalities which – at a last account – will not make the proof easier to follow.

sent by $A_{\mathtt{mal}}$. Specifically, it emulates the executions of the input-commitment protocol, which securely computes the functionality Eq. (2.21), in attempt to obtain the bits committed to by $A_{\mathtt{mal}}$. The emulation of each such execution is done by using the malicious ideal-model adversary derived from (the real malicious adversary) $A_{\mathtt{mal}}$. Details follow.

- In an execution of the input-commitment protocol where Party **hon** commits to an input bit, say its i^{th} bit, machine $B_{\mathtt{mal}}$ tries to obtain the corresponding commitment (for future usage in emulation of message-transmission steps). First $B_{\mathtt{mal}}$ emulates the uniform selection (by Party **hon**) of $\rho_i^{\mathtt{hon}} \in \{0,1\}^n$. Machine $B_{\mathtt{mal}}$ will use an arbitrary value, say 0, for the i^{th} bit of Party **hon** (as the real value is unknown to $B_{\mathtt{mal}}$). Next, machine $B_{\mathtt{mal}}$ derives the ideal-model adversary, denoted $A'_{\mathtt{mal}}$, which corresponds to the behavior of $A_{\mathtt{mal}}$ – given the history so far – in the corresponding execution of the input-commitment protocol.

  Invoking the ideal-model adversary $A'_{\mathtt{mal}}$, and emulating both the honest (ideal-model) Party **hon** and the trusted party, machine $B_{\mathtt{mal}}$ obtains the outputs of both parties (i.e., the commitment handed to Party **mal**). That is, machine $B_{\mathtt{mal}}$ obtains the message that $A'_{\mathtt{mal}}$ would have sent to the trusted party (i.e., 1^n), emulate the sending of message $(0, \rho_i^{\mathtt{hon}})$ by Party **hon**, and emulates the response of the trusted oracle, $\widetilde{c}_i^{\mathtt{hon}}$, where $\widetilde{c}_i^{\mathtt{hon}} = C_n(0, \rho_i^{\mathtt{hon}})$. (See definition of the functionality Eq. (2.21).)

  In case the emulated machines did not abort, machine $B_{\mathtt{mal}}$ records $\rho_i^{\mathtt{hon}}$, and concatenates the emulation of the input-commitment protocol (i.e., the final view of Party **mal** as output by $A'_{\mathtt{mal}}$) to the history of the execution of $A_{\mathtt{mal}}$. (Indeed, the emulated text may not be distributed as a transcript of a prefix of real execution of $A_{\mathtt{mal}}$, but the former is computationally indistinguishable from the latter.)

- In an execution of the input-commitment protocol where Party **mal** commits to an input bit, say its i^{th} bit, machine $B_{\mathtt{mal}}$ tries to obtain the corresponding bit as well as the commitment to it. First $B_{\mathtt{mal}}$ derives the ideal-model adversary, denoted $A'_{\mathtt{mal}}$, which corresponds to the behavior of $A_{\mathtt{mal}}$ – given the history so far – in the corresponding execution of the input-commitment protocol.

  Machine $B_{\mathtt{mal}}$ uniformly selects $\rho_i^{\mathtt{mal}} \in \{0,1\}^n$, invokes $A'_{\mathtt{mal}}$ on input $(z_i, \rho_i^{\mathtt{mal}})$,[24] and emulating both the honest (ideal-model) Party **hon** and the trusted party, machine $B_{\mathtt{mal}}$ obtains the outputs of both parties (i.e., the commitment handed to Party **hon**). A key point is that machine $B_{\mathtt{mal}}$ has obtained, while emulating the trusted party, the input handed by $A'_{\mathtt{mal}}$ to the trusted party. We denote this input by (z'_i, s). That is, machine $B_{\mathtt{mal}}$ obtains the message (z'_i, s) that $A'_{\mathtt{mal}}$ would have sent to the trusted party (which may differ from $(z_i, \rho_i^{\mathtt{mal}})$), emulates the sending of message 1^n by Party **hon**, and emulates the response of the trusted oracle, $\widetilde{c}_i^{\mathtt{mal}} = C_n(z'_i, s)$.

  In case the emulated machines did not abort, machine $B_{\mathtt{mal}}$ records the pair $(z'_i, \widetilde{c}_i^{\mathtt{mal}})$, and concatenates the emulation of the input-commitment protocol (i.e., the final view of Party **mal** as output by $A'_{\mathtt{mal}}$) to the history of the execution of $A_{\mathtt{mal}}$.

- If $A_{\mathtt{mal}}$ aborts in any of these executions then $B_{\mathtt{mal}}$ aborts the entire execution. Otherwise, $B_{\mathtt{mal}}$ sets $z' = z'_1 z'_2 \cdots z'_n$, $\widetilde{c}^{\mathtt{mal}} = \widetilde{c}_1^{\mathtt{mal}} \widetilde{c}_2^{\mathtt{mal}} \cdots \widetilde{c}_n^{\mathtt{mal}}$, and $\rho^{\mathtt{hon}} = \rho_1^{\mathtt{hon}} \rho_2^{\mathtt{hon}} \cdots \rho_n^{\mathtt{hon}}$.

---

[24] Machine $A'_{\mathtt{mal}}$ may, indeed, ignore this input altogether and act according to other strings which may be incorporated in its description. Giving input to $A'_{\mathtt{mal}}$ merely mimics the situation in which this party is honest, which is not the case here. In fact, one could have defined adversaries to have no input at all (as they are non-uniform and so can incorporate whatever input we wish anyhow).

In case $B_{\mathtt{mal}}$ did not abort, it enters protocol Π with input z'.

Note that this entire step is implemented in polynomial-time, and the resulting z' depends only on z (the initial input of $B_{\mathtt{mal}}$).

Selection of random-pad: $B_{\mathtt{mal}}$ selects its random-pad uniformly in $\{0,1\}^{c(n)}$ (as specifies by Π), and emulates the execution of the coin-generation phase of Π' ending with this outcome, so as to place $A_{\mathtt{mal}}$ in the appropriate state towards the protocol-emulation phase. To achieve the latter goal, machine $B_{\mathtt{mal}}$ supplies $A_{\mathtt{mal}}$ with the messages it expects to see, thus emulating a honest Party `hon` in Π', and obtains the messages sent by $A_{\mathtt{mal}}$. Specifically, it emulates the executions of the (augmented) coin-tossing protocol, which securely computes the functionality Eq. (2.14), so that these executions end with the desired coin outcome. The emulation of each such execution is done by using the malicious ideal-model adversary derived from (the real malicious adversary) $A_{\mathtt{mal}}$. The fact that in these emulations machine $B_{\mathtt{mal}}$ also emulates the trusted party allows it to set the outcome of the coin-tossing to fit the above selection of the random-pad. Alternatively, one may think of $B_{\mathtt{mal}}$ as "honestly" emulating the trusted party (i.e., which sets the outcome uniformly), and setting the random-pad to equal the result of these random outcomes. In any case, the random-pad is selected uniformly and independently of any thing else. Details follow.

- Machine $B_{\mathtt{mal}}$ selects its random-pad, $r^{\mathtt{mal}} = r_1^{\mathtt{mal}} r_2^{\mathtt{mal}} \cdots r_{c(n)}^{\mathtt{mal}}$, uniformly in $\{0,1\}^{c(n)}$.
- In i^{th} execution of the coin-tossing protocol in which Party `hon` obtains the outcome of the coin-toss, machine $B_{\mathtt{mal}}$ tries to obtain the outcome as well as the randomness used by Party `hon` when committing to it. First, machine $B_{\mathtt{mal}}$ derives the ideal-model adversary, denoted $A'_{\mathtt{mal}}$, which corresponds to the behavior of $A_{\mathtt{mal}}$ – given the history so far – in the corresponding execution of the coin-tossing protocol. Invoking the ideal-model adversary $A'_{\mathtt{mal}}$, and emulating both the honest (ideal-model) Party `hon` and the trusted party, machine $B_{\mathtt{mal}}$ obtains the outputs of both parties (i.e., both the coin value and the randomness handed to Party `hon` and a commitment handed to Party `mal`).

  That is, machine $B_{\mathtt{mal}}$ obtains the message that $A'_{\mathtt{mal}}$ would have sent to the trusted party (i.e., 1^n), emulates the sending of message 1^n by Party `hon`, and emulates the response of the trusted oracle, $((r_i^{\mathtt{hon}}, \omega_i^{\mathtt{hon}}), \overline{c}_i^{\mathtt{hon}})$, where $(r_i^{\mathtt{hon}}, \omega_i^{\mathtt{hon}}) \in \{0,1\} \times \{0,1\}^n$ is uniformly distributed and $\overline{c}_i^{\mathtt{hon}} = C_n(r_i^{\mathtt{hon}}, \omega_i^{\mathtt{hon}})$. (See definition of the functionality Eq. (2.14).)

  In case the emulated machines did not abort, machine $B_{\mathtt{mal}}$ records the pair $(r_i^{\mathtt{hon}}, \omega_i^{\mathtt{hon}})$, and concatenates the emulation of the coin-tossing protocol (i.e., the final view of Party `mal` as output by $A'_{\mathtt{mal}}$) to the history of the execution of $A_{\mathtt{mal}}$.
- In i^{th} execution of the coin-tossing protocol in which Party `mal` is supposed to obtain the outcome of the coin-toss, machine $B_{\mathtt{mal}}$ tries to generate an execution ending with the corresponding bit of $r^{\mathtt{mal}}$. First $B_{\mathtt{mal}}$ derives the ideal-model adversary, denoted $A'_{\mathtt{mal}}$, which corresponds to the behavior of $A_{\mathtt{mal}}$ – given the history so far – in the corresponding execution of the coin-tossing protocol. It invokes $A'_{\mathtt{mal}}$ and emulating both the honest (ideal-model) Party `hon` and the trusted party, machine $B_{\mathtt{mal}}$ obtains the outputs of both parties (i.e., both the coin value handed to Party `mal` and a commitment handed to Party `hon`).

  That is, machine $B_{\mathtt{mal}}$ obtains the message that $A'_{\mathtt{mal}}$ would have sent to the trusted party (i.e., 1^n), emulates the sending of message 1^n by Party `hon`, and emulates the response of the trusted oracle, $((r_i^{\mathtt{mal}}, \omega_i^{\mathtt{mal}}), \overline{c}_i^{\mathtt{mal}})$, where $(r_i^{\mathtt{mal}}, \omega_i^{\mathtt{mal}}) \in \{0,1\} \times \{0,1\}^n$ is uniformly distributed and $\overline{c}_i^{\mathtt{mal}} = C_n(r_i^{\mathtt{mal}}, \omega_i^{\mathtt{mal}})$.

In case the emulated machines did not abort, machine $B_{\mathtt{mal}}$ records the value $\overline{c}_i^{\mathtt{mal}}$, and concatenates the emulation of the coin-tossing protocol (i.e., the final view of Party `mal` as output by $A'_{\mathtt{mal}}$) to the history of the execution of $A_{\mathtt{mal}}$.

- If $A_{\mathtt{mal}}$ aborts in any of these executions then $B_{\mathtt{mal}}$ aborts the entire execution.

In case $B_{\mathtt{mal}}$ did not abort, it will use $r^{\mathtt{mal}}$ as its random-pad in its the subsequent steps of protocol Π. It also sets $\overline{c}^{\mathtt{mal}} = \overline{c}_1^{\mathtt{mal}} \overline{c}_2^{\mathtt{mal}} \cdots \overline{c}_{c(n)}^{\mathtt{mal}}$ and $\omega^{\mathtt{hon}} = \omega_1^{\mathtt{hon}} \omega_2^{\mathtt{hon}} \cdots \omega_{c(n)}^{\mathtt{hon}}$.

Note that this entire step is implemented in polynomial-time, and $r^{\mathtt{mal}}$ is selected uniformly in $\{0,1\}^{c(n)}$ independent of anything else.

Subsequent steps – message transmission: Machine $B_{\mathtt{mal}}$ now enters the actual execution of Π. It proceeds in this real execution along with emulating the corresponding executions of the authenticated-computation functionality of Eq. (2.28). In a message-transmission step by Party `hon` in Π, machine $B_{\mathtt{mal}}$ obtains from Party `hon` (in the real execution of Π) a message, and emulates an execution of the authenticated-computation protocol resulting in this message as output. In a message-transmission step by Party `mal` in Π, machine $B_{\mathtt{mal}}$ computes the message to be sent to Party `hon` (in Π) as instructed by Π, based on the input z' determined above, the random-pad $r^{\mathtt{mal}}$ selected above, and the messages obtained so far from Party `hon` (in Π). In addition, $B_{\mathtt{mal}}$ emulates an execution of the authenticated-computation protocol resulting in this message as output. The emulation of each execution of the authenticated-computation protocol, which securely computes the functionality Eq. (2.28), is done by using the malicious ideal-model adversary derived from (the real malicious adversary) $A_{\mathtt{mal}}$. The fact that in these emulations machine $B_{\mathtt{mal}}$ also emulates the trusted party allows it to set the outcome of the authenticated-computation protocol to fit the message being delivered. Details follow.

- In a message-transmission step by Party `hon` in Π, machine $B_{\mathtt{mal}}$ first obtains from Party `hon` (in the real execution of Π) a message, denoted `msg`. Next, machine $B_{\mathtt{mal}}$ derives the ideal-model adversary, denoted $A'_{\mathtt{mal}}$, which corresponds to the behavior of $A_{\mathtt{mal}}$ – given the history so far – in the corresponding execution of the authenticated-computation protocol (executed by protocol Π').

  Invoking the ideal-model adversary $A'_{\mathtt{mal}}$, and emulating both the honest (ideal-model) Party `hon` and the trusted party, machine $B_{\mathtt{mal}}$ sets the trusted-party reply to equal `msg`. When emulating Party `hon`, machine $B_{\mathtt{mal}}$ sends the trusted party the message $((0^n, r^{\mathtt{hon}}), (\rho^{\mathtt{hon}}, \omega^{\mathtt{hon}}), \beta)$, where 0^n is the dummy input used for Party `hon`, the string $r^{\mathtt{hon}}$ represents the random-pad (as recorded above), $\rho^{\mathtt{hon}}, \omega^{\mathtt{hon}}$ are randomizations used in the corresponding commitments, and β represents the the messages received received so far by Party `hon` (as resulted in the previous emulated executions).

  We comment that the emulation is carried out so to produce output `msg` which does not necessarily equal the output of the authenticated-computation functionality of Eq. (2.28) on the corresponding inputs. However, the machine $A'_{\mathtt{mal}}$ used in the emulation cannot distinguish the two cases (since the inputs which it gets in the two cases – commitments to the corresponding inputs of Party `hon` – are computationally indistinguishable).

  In case machine $A'_{\mathtt{mal}}$ aborts the emulation, machine $B_{\mathtt{mal}}$ aborts the entire execution of Π. Finally, $B_{\mathtt{mal}}$ concatenates the emulation of the authenticated-computation protocol (i.e., the final view of Party $\mathtt{mal}$ as output by $A'_{\mathtt{mal}}$) to the history of the execution of $A_{\mathtt{mal}}$.

- In a message-transmission step by Party $\mathtt{mal}$ in Π, machine $B_{\mathtt{mal}}$ first computes the message to be sent according to Π. This message is computed based on the input z' determined above, the random-pad $r^{\mathtt{mal}}$ (as recorded above), and the messages received so far (from Party $\mathtt{hon}$ in execution of Π). Denote the resulting message by $\mathtt{msg}$. Next, machine $B_{\mathtt{mal}}$ derives the ideal-model adversary, denoted $A'_{\mathtt{mal}}$, which corresponds to the behavior of $A_{\mathtt{mal}}$ – given the history so far – in the corresponding execution of the authenticated-computation protocol.

  Invoking the ideal-model adversary $A'_{\mathtt{mal}}$, and emulating both the honest (ideal-model) Party $\mathtt{hon}$ and the trusted party, machine $B_{\mathtt{mal}}$ determines the answer of the trusted party. When emulating Party $\mathtt{hon}$, machine $B_{\mathtt{mal}}$ sends the trusted party the message $((\widetilde{c}^{\mathtt{mal}}, \overline{c}^{\mathtt{mal}}), \beta)$ where $\widetilde{c}^{\mathtt{mal}}, \overline{c}^{\mathtt{mal}}$ are the commitments recorded above, and β represents the the messages received received so far by Party $\mathtt{mal}$ (as resulted in the previous emulated executions).

  In case the answer of the trusted party (emulated by $B_{\mathtt{mal}}$) differs from $\mathtt{msg}$, machine $B_{\mathtt{mal}}$ aborts the entire execution of Π.[25] Otherwise, $B_{\mathtt{mal}}$ sends $\mathtt{msg}$ to Party $\mathtt{hon}$ (in Π), and concatenates the emulation of the authenticated-computation protocol (i.e., the final view of Party $\mathtt{mal}$ as output by $A'_{\mathtt{mal}}$) to the history of the execution of $A_{\mathtt{mal}}$.

- If $A_{\mathtt{mal}}$ aborts in any of these executions then $B_{\mathtt{mal}}$ aborts the entire execution.

Note that each message-transmission step is implemented in polynomial-time. Each message sent by $B_{\mathtt{mal}}$ is computed as instructed by Π, and the decision whether to abort or proceed is taken by $B_{\mathtt{mal}}$ based on its input, its random-pad, and the messages it has received so far.

Output: Assuming machine $B_{\mathtt{mal}}$ has not aborted the execution, it just outputs whatever machine $A_{\mathtt{mal}}$ outputs given the execution history composed above.

Clearly, machine $B_{\mathtt{mal}}$ (described above) implements an augmented semi-honest behavior with respect to Π. It is left to show that

$$\{\text{REAL}_{\Pi',\overline{A}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,\overline{B}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \tag{2.30}$$

There are two differences between the two ensembles referred to in Eq. (2.30):

1. In the first distribution (i.e., $\text{REAL}_{\Pi',\overline{A}}(x,y)$), secure protocols implementing the input-commitment, coin-tossing and authenticated-computation functionalities (of Eq. (2.21), Eq. (2.14) and Eq. (2.28), respectively) are executed; whereas in the second distribution (i.e., $\text{REAL}_{\Pi,\overline{B}}(x,y)$) these executions are emulated using the corresponding ideal-model adversaries.

2. The emulation of Eq. (2.28) (in $\text{REAL}_{\Pi,\overline{B}}(x,y)$) is performed with a potentially wrong Party $\mathtt{mal}$ input.

However, by the fact that the above protocols are secure, all emulations are computationally indistinguishable from the real executions. Furthermore, the inputs given to Party $\mathtt{mal}$ in the emulation

[25] Alternatively, we may abort whenever $A_{\mathtt{mal}}$ supplies the trusted party (emulated by $B_{\mathtt{mal}}$) with input which does not fit the input computed by $B_{\mathtt{mal}}$ based on z' and $r^{\mathtt{mal}}$ recorded above and the messages obtained so far from Party $\mathtt{hon}$.

of Eq. (2.28) are computationally indistinguishable from the correct ones, and so the corresponding outputs are computational indistinguishable too. Observing that the output of Party `hon` in both cases is merely the corresponding output of Π on input (x', y'), where $(x', y') = (x, z')$ if $\texttt{hon} = 1$ and $(x', y') = (z', y)$ otherwise, Eq. (2.30) follows. ■

2.3.3.2 On the protocols underlying the proof of Theorem 2.2.13

We now show that for the protocols underlying the proof of Theorem 2.2.13, there is an clear correspondence between the augmented-semi-honest model and the malicious-ideal-model. Recall that each such protocol is designed (and guaranteed) to privately compute some desired functionality. Thus, a real semi-honest execution of this protocol corresponds to an ideal semi-honest computation of the functionality. However, these protocol have the salient property of allowing to transform the wider class of augmented-semi-honest executions into the wider class of ideal malicious computations. Recall that the augmented semi-honest model allows two things which go beyond the semi-honest model: (1) oblivious substitution of inputs, and (2) abort. The first type of behavior has a correspondence in the malicious ideal model, and so poses no problem. To account for the second type of behavior, we need to match an aborting execution in the augmented semi-honest model with an aborting execution in the ideal malicious model. Here is where the extra property of the specific protocols, underlying the proof of Theorem 2.2.13, comes to help – see below.

Proposition 2.3.16 (on the protocols underlying the proof of Theorem 2.2.13): *Let Π be a protocol which privately computes the functionality f. Furthermore, suppose that Π was produced as follows.*

1. *First, the private computation of f was reduced to the private computation of a deterministic functionality, f', using the protocol of Proposition 2.2.4.*
2. *Next, Construction 2.2.10 was applied to a circuit computing f', resulting in an oracle-aided protocol.*
3. *Finally, the oracle was implemented using Corollary 2.2.9.*

Then, there exists a polynomial-time computable transformation of pairs of polynomial-size circuit families $\overline{B} = (B_1, B_2)$ admissible w.r.t Π *for the augmented semi-honest model* (of Definition 2.3.14) *into pairs of polynomial-size circuit families $\overline{C} = (C_1, C_2)$ admissible for the ideal malicious model* (of Definition 2.1.4) *so that*

$$\{\text{REAL}_{\Pi,\overline{B}}(x, y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\text{IDEAL}_{f,\overline{C}}(x, y)\}_{x,y \text{ s.t. } |x|=|y|}$$

Proof Sketch: We use the following property of the simulators of the (view of a semi-honest party) in protocol Π (produced as above). These simulators, hereafter referred to as *two-stage simulators*, acts as follows.

Input to simulator: A pair (z, v), where z is the initial input of the semi-honest party and v the corresponding local output.

Simulation Stage 1: Based on z, the simulator generates a transcript corresponding to the view of the semi-honest party in a truncated execution of Π, where the execution is truncated just before the last message is received by the semi-honest party.

We stress that this truncated view, denoted T, is produced without using v.

Simulation Stage 2: Based on T and v, the simulator produces a string corresponding to the last message received by the semi-honest party. The simulator then outputs the concatenation of T and this message.

The reader may easily verify that protocol Π, produced as in the hypothesis of this proposition, indeed has two-stage simulators. This is done by observing that the simulators for Π are basically derived from the simulators of Construction 2.2.10. (The simulators used in Proposition 2.2.4 and Corollary 2.2.9 merely prepend and expand, respectively, the transcripts produced by the simulator of Construction 2.2.10.) Turning to the protocol of Construction 2.2.10, we note that Steps 1 and 2 of this protocol are simulated without having the corresponding output (see the proof of Proposition 2.2.11). This corresponds to Stage 1 in the definition of a two-stage simulator. The output is only needed to simulate Step 3 which consists of two messages-transmissions (one from Party 2 to Party 1 and the second in the other direction). The latter corresponds to Stage 2 in the definition of a two-stage simulator.

Next we show that for any protocol having two-stage simulators, the transformation claimed in the current proposition holds. Given a circuit pair, (B_1, B_2), admissible w.r.t Π for the augmented semi-honest model, we construct a circuit pair, (C_1, C_2), which is admissible for the ideal malicious model as follows. We distinguish two cases – according to which of the parties is honest. The difference between these cases amount to the possibility of (meaningfully) aborting the execution after receiving the last message – a possibility which exists for a dishonest Party 1 but not for a dishonest Party 2.

We start with the case where Party 2 is totally honest (and Party 1 possibly dishonest). In this case C_2 is determined, and we need to transform the augmented semi-honest real adversary B_1 into a malicious ideal-model adversary C_1. The latter operates as follows, using the two-stage simulator, denoted S_1, provided for semi-honest executions of Π (which privately computes f). Recall that C_1 gets input $x \in \{0,1\}^n$.

1. First, C_1 computes the substituted input with which B_1 enters Π. That is, $x' = B_1(x)$.

2. Next, C_1 invokes the first stage of the simulator S_1, to obtain the view of a truncated execution of Π by a semi-honest party having input x'. That is, $T = S_1(x')$.

   Machine C_1 extracts from T the random-pad, denoted r, of Party 1. This pad correspond to the random-pad used by B_1.

3. Using T, machine C_1 emulates the execution of B_1 on input x' and random-pad r, up to the point where Party 1 is to receive the last message. Towards this end, C_1 feeds B_1 with input x' and random-pad r (i.e., it substitutes r as the random-pad of B_1 making it deterministic), and sends B_1 messages as appearing in the corresponding locations in T.

   Note that B_1 may abort in such an execution, but in case it does not abort the messages it sends equal the corresponding messages in T (as otherwise one could efficiently distinguish the simulation from the real view).

4. In case B_1 has aborted the execution, machine C_1 aborts the execution before invoking the trusted party. Otherwise, it invokes the trusted party with input x', and obtains a response, denoted v.

   We stress that C_1 still has the option of stopping the trusted party before it answers Party 2.

5. Next, C_1 invokes the second stage of the simulator S_1, to obtain the last message sent to Party 1. It supplies the simulator with the input x' and the output v and obtains the last message, denoted `msg`.

6. Machine C_1 now emulates the last step of B_1 by supplying it with the message `msg`. In case B_1 aborts, machine C_1 prevents the trusted party from answering Party 2, and aborts. Otherwise, machine C_1 allows the trusted party to answer Party 2.

7. The output of C_1 is set to be the output of B_1, regardless if B_1 has aborted or completed the execution.

We need to show that

$$\{\text{REAL}_{\Pi,\overline{B}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \stackrel{\text{c}}{\equiv} \{\text{IDEAL}_{f,\overline{C}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \tag{2.31}$$

Suppose first, for simplicity, that machine B_1 *never aborts*. In such a case, by definition of S_1,

$$\begin{aligned}
\{\text{REAL}_{\Pi,\overline{B}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} &\equiv \{(B_1(\text{VIEW}_1^{\Pi}(B_1(x),y))\,,\, f_2(B_1(x),y))\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\stackrel{\text{c}}{\equiv} \{(B_1(S_1(B_1(x), f_1(B_1(x),y)))\,,\, f_2(B_1(x),y))\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{(C_1(x, f_1(C_1(x),y))\,,\, f_2(C_1(x),y))\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{\text{IDEAL}_{f,\overline{C}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n}
\end{aligned}$$

Next, suppose that B_1 always aborts *after receiving the last message*, and before sending its last message to Party 2. In this case, we have

$$\begin{aligned}
\{\text{REAL}_{\Pi,\overline{B}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} &\equiv \{(B_1(\text{VIEW}_1^{\Pi}(B_1(x),y))\,,\, \perp)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\stackrel{\text{c}}{\equiv} \{(B_1(S_1(B_1(x), f_1(B_1(x),y)))\,,\, \perp)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{(C_1(x, f_1(C_1(x),y), \perp)\,,\, \perp)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{\text{IDEAL}_{f,\overline{C}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n}
\end{aligned}$$

As a final illustration, consider the third extreme case in which B_1 always aborts *before* receiving the last message. Here

$$\begin{aligned}
\{\text{REAL}_{\Pi,\overline{B}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} &\equiv \{(B_1(\text{truncated-view}_1^{\Pi}(B_1(x),y))\,,\, \perp)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\stackrel{\text{c}}{\equiv} \{(B_1(S_1(B_1(x))\,,\, \perp)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{(C_1(x,\perp)\,,\, \perp)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{\text{IDEAL}_{f,\overline{C}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n}
\end{aligned}$$

In the general case, machine B_1 may abort in certain executions in varying places – in particular sometimes before obtaining the last message or just after it (and before sending its last message). The first type of abort depends on the view of B_1 in partial executions truncated before it receives the last message, whereas the second type depends also on the last message it receives. For both type of abort, the behavior in the two cases ($\text{REAL}_{\Pi,\overline{B}}(x,y)$ and $\text{IDEAL}_{f,\overline{C}}(x,y)$) is determined by B_1 based on a pair of computational indistinguishable ensembles (i.e., the real view of an execution versus a simulated one). Thus, Eq. (2.31) follows.

Next, suppose that Party 1 is honest. In this case C_1 is determined, and we need to transform the augmented semi-honest real adversary B_2 into a malicious ideal-model adversary C_2. The latter operates as follows, using the two-stage simulator, denoted S_2, provided for semi-honest executions of the private computation of f. (The difference w.r.t the previous case is in the last 3 steps of the emulation.) Recall that C_2 gets input $y \in \{0,1\}^n$.

1. First, C_2 computes the substituted input with which B_2 enters Π. That is, $y' = B_2(y)$.

2. Next, C_2 invokes the first stage of the simulator S_2, to obtain the view of a truncated execution of Π by a semi-honest party having input y'. That is, $T = S_2(y')$.

   Machine C_2 extracts from T the random-pad, denoted r, of Party 2. This pad correspond to the random-pad used by B_2.

3. Using T, machine C_2 emulates the execution of B_2 on input y' and random-pad r, up to the point where Party 2 is to receive the last message. Towards this end, C_2 feeds B_2 with input y' and random-pad r (i.e., it substitutes r as the random-pad of B_2 making it deterministic), and sends B_2 messages as appearing in the corresponding locations in T.

   Note that B_2 may abort in such an execution, but in case it does not abort the messages it sends equal the corresponding messages in T (as otherwise one could efficiently distinguish the simulation from the real view).

4. In case B_2 has aborted the execution, machine C_2 aborts the execution before invoking the trusted party. Otherwise, it invokes the trusted party with input y', and obtains a response, denoted v.

   (Unlike the case where Party 1 is semi-honest, since the trusted party answers Party 1 first, Party 2 does not have the option of stopping the trusted party before it answers Party 2. Yet, we do not need this option either, since in case.)

5. Next, C_2 invokes the second stage of the simulator S_2, to obtain the last message sent to Party 2. It supplies the simulator with the input y' and the output v and obtains the last message, denoted `msg`.

   (Note that Party 2 has already sent its last message, and so the execution of C_2 ends here.)

6. The output of C_2 is set to be the output of B_2, regardless if B_2 has aborted or completed the execution.

We again need to show that Eq. (2.31) holds. The argument is analogous to the one applied for Party 1. Specifically, in the simple case where machine B_1 never aborts, we have

$$\begin{aligned}
\{\mathrm{REAL}_{\Pi,\overline{B}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} &\equiv \{(f_1(x,B_2(y))\,,\,B_2(\mathrm{VIEW}_2^{\Pi}(x,B_2(y))))\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\stackrel{\mathrm{c}}{\equiv} \{(f_1(x,B_2(y))\,,\,B_2(S_2(y,f_2(x,B_2(y)))))\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{(f_1(x,C_2(y))\,,\,C_2(y,f_2(x,C_2(y))))\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n} \\
&\equiv \{\mathrm{IDEAL}_{f,\overline{C}}(x,y)\}_{n\in\mathbb{N},\, x,y\in\{0,1\}^n}
\end{aligned}$$

and the proposition follows. ■

2.3.3.3 Conclusion – Proof of Theorem 2.3.1

Theorem 2.3.1 follow by combining Propositions 2.3.15 and 2.3.16. Specifically, let Π be the protocol produced as in Proposition 2.3.16 when given the functionality f, and Π' be the protocol compiled from Π by Construction 2.3.13. Furthermore, let $\overline{A}$ be admissible for the real *malicious* model, let $\overline{B}$ be (admissible w.r.t Π in the augmented semi-honest model) produced by the transformation in Proposition 2.3.15, and $\overline{C}$ be (admissible for the ideal malicious model) produced by the transformation in Proposition 2.3.16. Then

$$\begin{aligned}\{\text{IDEAL}_{f,\overline{C}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} &\stackrel{c}{\equiv} \{\text{REAL}_{\Pi,\overline{B}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|} \\ &\stackrel{c}{\equiv} \{\text{REAL}_{\Pi',\overline{A}}(x,y)\}_{x,y \text{ s.t. } |x|=|y|}\end{aligned}$$

as required by Theorem 2.3.1. ■

Chapter 3

General Multi-Party Computation

Our presentation proceeds as in the previous chapter. Again, our ultimate goal is to design protocols which may withstand any feasible adversarial behavior. We proceed in two steps. First we consider a benign type of adversary, called *semi-honest*, and construct protocols which are secure with respect to such an adversary. The definition of this type of adversary is very much the same as in the two-party case. However, in case of general adversary behavior we consider two models. The *first model of malicious behavior* mimics the treatment of adversaries in the two-party case; it allows the adversary to control even a majority of the parties, but does not consider the unavoidable early abort phenomena as a violation of security. The *second model of malicious behavior* we assume that the adversary can control only a strict minority of the parties. In this model, which would have been vacuous in the two-party case, early abort phenomena may be effectively prevented. We show how to transform protocols secure in the semi-honest model into protocols secure in each of the two malicious-behavior models. As in the two-party case, this is done by forcing parties (in each of the latter models) to behave in an effectively semi-honest manner.

The constructions are obtained by suitable modifications of the constructions used in the two-party case. Actually, the construction of multi-party protocols for the semi-honest model is a minor modification of the construction used in the two-party case. The same holds for the compilation of protocols for the semi-honest model into protocols for the *first* malicious model. In compiling protocols for the semi-honest model into protocols for the *second* malicious model, a new ingredient – Verifiable Secret Sharing (VSS) – is used to "effectively prevent" minority parties from aborting the protocol prematurely. Actually, we shall compile protocols secure in the *first* malicious model into protocols secure in the *second* malicious model.

As in the two-party case, we believe that the semi-honest model is not merely an important methodological locus, but also provides a good model of certain settings.

Organization: In Section 3.1 we define the framework for the entire chapter. In particular, we define multi-party functionalities, the semi-honest model, and the two malicious models. In Section 3.2 we describe the construction of protocols for the semi-honest model, and in Section 3.3 compilers which transform protocols from the latter model to protocols secure in each of the two malicious models.

3.1 Definitions

A multi-party protocol problem is casted by specifying a random process which maps sequences of inputs (one input per each party) to sequences of outputs (one per each party). Let m denote the number of parties. It will be convenient to think of m as being fixed, alas one can certainly think of it as an additional parameter. An m-ary functionality, denoted $f : (\{0,1\}^*)^m \mapsto (\{0,1\}^*)^m$, is thus a random process mapping string sequences of the form $\overline{x} = (x_1, ..., x_m)$ into sequences of random variables, $f_1(\overline{x}), ..., f_m(\overline{x})$. The semantics is that, for every i, the i^{th} party, initially holds an input x_i, and wishes to obtain the i^{th} element in $f(x_1, ..., x_m)$, denoted $f_i(x_1, ..., x_m)$. The discussions and simplifying conventions made in Section 2.1 apply in the current context too.

We comment that it is natural to discuss multi-party functionalities which are "uniform" in the sense that there exists an algorithm for uniformly computing them for each value of m (and of course each m-sequence). One such functionality is the "universal functionality" which is given a description of a circuit as well as a corresponding sequence of inputs. (For example, the circuit may be part of the input of each party, and in case these circuits are not identical the value of the functionality is defined as a sequence of $\perp$'s.) Indeed, a universal functionality is natural to consider also in the two-party case, but here (in view of the extra parameter m) its appeal is enhanced.

The definitions presented below (both for the semi-honest and the two malicious models) presuppose that honest parties may communicate in secrecy (i.e., or put differently, we assume that adversaries do not tape communication lines between honest parties). This assumption can be removed at the expense of further complicating the notations. Furthermore, the issue of providing secret communication (via encryption schemes) is well understood, and may thus be decoupled from the current exposition. Specifically, this means that protocols constructed in the sequel need to be further compiled using encryption schemes if one wishes to withstand wire-tapping attacks by an adversary. Similarly, we assume that messages sent between honest parties arrive intact, whereas one may want to consider adversaries which may inject messages on the communication line between honest parties. Again, this can be counteracted by use of well-understood paradigms – in this case the use of signature schemes.

The definitions presented below are all "static" in the sense that the set of dishonest parties is fixed before the execution of the protocol starts, rather than being determined *adaptively* during the execution of the protocol. (We stress that in either cases honest parties may not necessarily know which parties are dishonest.) The difference between the static model of security considered in this chapter and the "adaptive" model (considered in Section 4.3) becomes crucial when the number of parties (i.e., m) is treated as a parameter, rather than being fixed.

For simplicity of exposition, we assume throughout our exposition that m is fixed. At the end of each subsection, we comment on what is needed in order to derive definitions when m is a parameter.

3.1.1 The semi-honest model

This model is defined exactly as in the two-party case. Recall that a semi-honest party is one who follows the protocol properly with the exception that it keeps a record of all its intermediate computations. Loosely speaking, a multi-party protocol *privately computes* f if whatever a *set* (or *coalition*) of semi-honest parties can be obtained after participating in the protocol, could be essentially obtained from the input and output available to these very parties. Thus, the only difference between the current definition and the one used in the two-party case is that we consider the gain of a coalition (rather than of a single player) from participating in the protocol.

Definition 3.1.1 (privacy w.r.t semi-honest behavior): *Let* $f : (\{0,1\}^*)^m \mapsto (\{0,1\}^*)^m$ *be an m-ary functionality, where* $f_i(x_1,...,x_m)$*, denotes the* i^{th} *element of* $f(x_1,..,x_m)$*. For* $I = \{i_1,...,i_t\} \subset [m] \stackrel{\text{def}}{=} \{1,...,m\}$*, we let* $f_I(x_1,...,x_m)$ *denote the subsequence* $f_{i_1}(x_1,...,x_m),...,f_{i_t}(x_1,...,x_m)$*. Let* Π *be an m-party protocol for computing f.*[1] *The* view *of the* i^{th} *party during an execution of* Π *on* $\overline{x} = (x_1,...,x_m)$*, denoted* $\text{VIEW}_i^\Pi(\overline{x})$*, is defined as in Definition 2.1.1, and for* $I = \{i_1,...,i_t\}$*, we let* $\text{VIEW}_I^\Pi(\overline{x}) \stackrel{\text{def}}{=} (I, \text{VIEW}_{i_1}^\Pi(\overline{x}),...,\text{VIEW}_{i_t}^\Pi(\overline{x}))$*. We say that* π privately computes *f if there exist polynomial-time algorithm, denoted S, such that for every I as above*

$$\{S(I,(x_{i_1},...,x_{i_t}),f_I(\overline{x}))\}_{\overline{x}\in(\{0,1\}^*)^m} \stackrel{\text{c}}{\equiv} \{\text{VIEW}_I^\Pi(\overline{x})\}_{\overline{x}\in(\{0,1\}^*)^m} \tag{3.1}$$

Eq. (3.1) asserts that the view of the parties in I can be efficiently simulated based solely on their inputs and outputs. The definition above can be easily adapted to deal with a varying parameter m. This is hinted by our order of quantification (i.e., "exists an algorithm S so that for any I").[2] We also note that the simulator can certainly handle the trivial cases in which either $I = [m]$ or $I = \emptyset$.

3.1.2 The two malicious models

We now turn to consider arbitrary feasible deviation of parties from a specified multi-party protocol. As mentioned above, one may consider two alternative models:

1. A model in which the number of parties which deviate from the protocol is arbitrary. The treatment of this case follows the treatment given in the two-party case. In particular, in this model one cannot prevent malicious parties from aborting the protocol prematurely, and the definition of security has to account for this if it is to have a chance of being met.

2. A model in which the number of parties which deviate from the protocol is strictly less than half the total number of parties. The definitional treatment of this case is simpler than the treatment given in the two-party case. In particular, one may – in some sense – (effectively) prevent malicious parties from aborting the protocol prematurely.[3] Consequently, the definition of security is "freed" from the need to account for early stopping, and thus is simpler.

We further assume (towards achieving a higher level of security) that malicious parties may communicate (without being detected by the honest parties), and may thus coordinate their malicious actions. Actually, it will be instructive to think of all malicious parties as being controlled by one adversary. Our presentation follows the ideal-vs-real emulation paradigm introduced in the previous chapters. The difference between the two malicious models is reflected in a difference in the corresponding *ideal models*, which capture the behavior which a secure protocol is aimed at achieving. The different bound on the number of malicious parties (in the two model) is translated into the only difference between the corresponding *real models* (or, rather, a difference in the adversaries allowed as per each malicious model).

[1] As in Section 2.1, by saying that Π computes (rather than privately computes) f, we mean that the output distribution of the protocol (when played by honest or semi-honest parties) on the input sequence $(x_1,...,x_m)$ is identically distributed as $f(x_1,...,x_m)$.

[2] Note that for a fixed m it may make as much sense to reverse the order of quantifiers (i.e., require that "for every I exists an algorithm S_I").

[3] As we shall see, the assumption that malicious parties are in minority opens the door to effectively preventing them from aborting the protocol immaturely. This will be achieved by having the majority players have (together!) enough information so to be able to emulate the minority players in case the latter have decided to abort.

Discussion. The above alternative models gives rise to two appealing and yet fundamentally incomparable notions of security. Put in other words, there is a trade-off between willing to put-up with early-abort (i.e., not consider it a breach of security), and requiring the protocol to be robust against malicious coalitions controlling a majority of all parties. The question of which notion of security to prefer depends on the application or the setting. In some settings one may prefer to be protected from malicious majorities, while giving-up the guarantee that parties cannot abort the protocol prematurely (while being detected doing so). On the other hand, in settings in which a strict majority of the parties can be trusted to follow the protocol, one may obtain the benefit of effectively preventing parties to abort the protocol prematurely.

Convention. The adversary will be represented as a family of polynomial-size circuits. Such a circuit will capture the actions of the adversary in each of the models. Typically, the adversary will be given as input the set of parties it controls, denoted I, the local inputs of these parties, denoted $\overline{x}_I$, and additional inputs as adequate (e.g., the local outputs of parties, or messages they have received in the past, etc.). However, we will omit I from the list of inputs to the circuit. (Alternatively, I could be incorporated into the circuit, but we prefer to have it explicit so that one can refer to it.)

3.1.2.1 The first malicious model

Following the discussion in Section 2.1.2, we conclude that three things cannot be avoided in the first malicious model:

1. Malicious parties may refuse to participate in the protocol (when the protocol is first invoked).
2. Malicious parties may substituting their local input (and enter the protocol with an input other than the one provided to them from the outside).
3. Malicious parties may abort the protocol prematurely (e.g., before sending their last message).

Accordingly, the ideal model is derived by a straightforward generalization of Definition 2.1.4. In light of this similarity, we allow ourself to be quite terse. To simplify the exposition, we assume that, for every I, first the trusted party supplies the adversary with the I-part of the output (i.e., the value of f_I), and only then may answer the other parties (at the adversary's discretion).[4] Actually, as in the two-party case, the adversary has the ability to prevent the trusted party from answering all parties only in case it controls Party 1.

Definition 3.1.2 (malicious adversaries, the ideal model – first model): *Let $f : (\{0,1\}^*)^m \mapsto (\{0,1\}^*)^m$ be an m-ary functionality, $I = \{i_1, ..., i_t\} \subset [m]$, and $(x_1, ..., x_m)_I = (x_{i_1}, ..., x_{i_t})$. A pair (I, C), where $I \subset [m]$ and C is a polynomial-size circuit family represents an adversary in the ideal model. The* joint execution under (I, C) in the ideal model *(on input $\overline{x} = (x_1, ..., x_m)$), denoted $\text{IDEAL}^{(1)}_{f,(I,C)}(\overline{x})$, is defined as follows*

$$(C(\overline{x}_I, \perp), \perp, ..., \perp) \qquad \text{if } C(\overline{x}_I) = \perp \tag{3.2}$$

[4] A less significant simplification is having the m-sequence of outputs not be presented in the "correct" order; that is, the outputs are presented so that the outputs of malicious parties appear first followed by the outputs of honest parties, whereas (unless $I = \{1, ..., t\}$) the order should have been different (i.e., the output of party i should have been in location i).

$$
\begin{array}{lll}
(C(\overline{x}_I, f_I(C(\overline{x}_I), \overline{x}_{\bar{I}}), \perp)\,, \perp, ..., \perp) & \text{if } C(\overline{x}_I) \neq \perp,\ 1 \in I \text{ and } \overline{y}_I = \perp & (3.3) \\
 & \text{where } \overline{y}_I \stackrel{\rm def}{=} C(\overline{x}_I, f_I(C(\overline{x}_I), \overline{x}_{\bar{I}})). & \\
(C(\overline{x}_I, f_I(C(\overline{x}_I), \overline{x}_{\bar{I}}))\,, f_{\bar{I}}(C(\overline{x}_I), \overline{x}_{\bar{I}})) & \text{otherwise} & (3.4)
\end{array}
$$

where $\bar{I} \stackrel{\rm def}{=} [m] \setminus I$.

Eq. (3.2) represents the case where the adversary makes some party (it controls) abort before invoking the trusted party. Eq. (3.3) represents the case where the trusted party is invoked with possibly substituted inputs, denoted $C(\overline{x}_I)$, and is halted right after supplying the adversary with the I-part of the output, denoted $\overline{y}_I = f_I(C(\overline{x}_I), \overline{x}_{\bar{I}})$. This case is allowed only when $1 \in I$, and so Party 1 can always be "blamed" when this happens.[5] Eq. (3.4) represents the case where the trusted party is invoked with possibly substituted inputs (as above), but is allowed to answer all parties.

Definition 3.1.3 (malicious adversaries, the real model): *Let f be as in Definition 3.1.2, and Π be an m-party protocol for computing f. The* joint execution of Π under (I, C) in the real model (on input sequence $\overline{x} = (x_1, ..., x_m)$), *denoted* $\text{REAL}_{\Pi,(I,C)}(\overline{x})$, *is defined as the output sequence resulting of the interaction between the m parties where the messages of parties in I are computed according to C and the messages of parties not in I are computed according to Π.*

In the sequel, we will assume that the circuit representing the real-model adversary is deterministic. This is justified by standard techniques: See discussion following Definition 2.1.6. Having defined the ideal and real models, we obtain the corresponding definition of security.

Definition 3.1.4 (security in the first malicious model): *Let f and Π be as in Definition 3.1.3, Protocol Π is said to* securely compute f (in the first malicious) *if there exists a polynomial-time computable transformation of polynomial-size circuit families $A = \{A_n\}$ for the real model* (of Definition 3.1.3) *into polynomial-size circuit families $B = \{B_n\}$ for the ideal model* (of Definition 3.1.2) *so that for every $I \subset [m]$*

$$
\{\text{IDEAL}^{(1)}_{f,(I,B)}(\overline{x})\}_{n \in \mathbb{N},\, \overline{x} \in (\{0,1\}^n)^m} \stackrel{\rm c}{\equiv} \{\text{REAL}_{\Pi,(I,A)}(\overline{x})\}_{n \in \mathbb{N},\, \overline{x} \in (\{0,1\}^n)^m}
$$

When the context is clear, we sometimes refer to Π as an implementation *of f.*

We stress that the resulting adversary in the ideal model (i.e., B) controls exactly the same set of parties (i.e., I) as the adversary in the real model (i.e., A).

3.1.2.2 The second malicious model

In the second model, where malicious players are in strict minority, the early-abort phenomena can be effectively prevented. Thus, in this case, there is no need to "tolerate" early-abort and consequently our definition of security requires "proper termination" of executions. This is reflected in the definition of the ideal model, which actually becomes simpler. However, since the definition differs more substantially from the two-party one, we present it in more detail (than done in the presentation of the first malicious model).

[5] In fact, in the protocols presented below, early abort is always due to malicious behavior of Party 1. By Definition 3.1.4 (below), this translates to malicious behavior of Party 1 in the ideal model.

The ideal model. Again, we will allow in the ideal model whatever cannot be possibly prevented in any real execution.[6] Specifically, we allow a malicious party in the ideal model to refuse to participate in the protocol or to substitute its local input. (Clearly, neither can be prevent by a trusted third party.) Thus, an execution in the ideal model proceeds as follows (where all actions of the both honest and malicious parties must be feasible to implement).

Inputs: Each party obtains an input; the one of Party i is denoted z_i.

Send inputs to trusted party: An honest party always sends z to the trusted party. The malicious minority parties may, depending on their inputs, $z_1, ..., z_t$, either abort or sends modified $z_i' \in \{0,1\}^{|z_i|}$ to the trusted party.

Trusted party answers the parties: In case it has obtained a valid input sequence, $\overline{x} = (x_1, ..., x_m)$, the trusted party computes $f(\overline{x})$, and replies to the i^{th} party with $f_i(\overline{x})$, for $i = 1, ..., m$. Otherwise, the trusted party replies to all parties with a special symbol, $\perp$.

Outputs: An honest party always outputs the message it has obtained from the trusted party. The malicious minority parties may output an arbitrary (polynomial-time computable) function of their initial inputs and the messages they have obtained from the trusted party.

The ideal model computation is captured in the following definition, where the circuit C represent the coordinated activities of all malicious parties as impersonated by a single adversary. To simplify the exposition, we treat the case in which malicious parties refuse to enter the protocol as if they have substituted their inputs by some special value, denoted $\perp$. (The functionality f can be extended so that if any of the inputs equals $\perp$ then all outputs are set to $\perp$.) Thus, there is a single case to consider: All parties send (possibly substituted) inputs to the trusted party, who always responses.

Definition 3.1.5 (malicious adversaries, the ideal model – second model): *Let $f : (\{0,1\}^*)^m \mapsto (\{0,1\}^*)^m$ be an m-ary functionality, $I = \{i_1, ..., i_t\} \subset [m]$, and $(x_1, ..., x_m)_I = (x_{i_1}, ..., x_{i_t})$. A pair (I, C), is called* admissible *if $t < n/2$ and $C = \{C_n\}_{n \in \mathbb{N}}$ is a family of polynomial-size circuits. The* joint execution under (I, C) in the ideal model (on input sequence $\overline{x} = (x_1, ..., x_m)$)*, denoted $\text{IDEAL}^{(2)}_{f,(I,C)}(\overline{x})$, is defined as follows*

$$(C(\overline{x}_I, f_I(C(\overline{x}_I), \overline{x}_{\bar{I}})) \, , \, f_{\bar{I}}(C(\overline{x}_I), \overline{x}_{\bar{I}})) \tag{3.5}$$

where $\bar{I} \stackrel{\text{def}}{=} [m] \setminus I$.

Note that (again) the m-sequence of outputs is not presented in the "correct" order; that is, the outputs are presented so that the outputs of malicious parties appear first followed by the outputs of honest parties, whereas (unless $I = \{1, ..., t\}$) the order should have been different (i.e., the output of party i should have been in location i). This convention simplifies the presentation, while having no significant impact on the essence. In the sequel we will refer to the pair (I, C) as an adversary. (Note that I can indeed be incorporated into C.)

[6] Recall that an alternative way of looking at things is that we assume that the the parties have at their disposal a trusted third party, but even such a party cannot prevent specific malicious behavior.

Execution in the real model. We next consider the real model in which a real (multi-party) protocol is executed (and there exist no trusted third parties). In this case, a malicious parties may follow an arbitrary feasible strategy; that is, any strategy implementable by polynomial-size circuits. Again, we consider these parties as being controlled by a single adversary, which is represented by a family of polynomial-size circuits. The resulting definition is exactly the one used in the first malicious model (i.e., Definition 3.1.3), except that here we will only consider minority coalitions (i.e., $|I| < m/2$).

Security as emulation of real execution in the ideal model. Having defined the ideal and real models, we obtain the corresponding definition of security. Loosely speaking, the definition asserts that a secure multi-party protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated by saying that admissible adversaries in the ideal-model are able to simulate (in the ideal-model) the execution of a secure real-model protocol (with admissible adversaries). Note that the following definition differs from Definition 3.1.4 in two aspects: Firstly, it quantifies only on minority collisions (i.e., $|I| < m/2$); and, secondly, it refers to the second ideal model (i.e., $\text{IDEAL}^{(2)}$) rather than to the first (i.e., $\text{IDEAL}^{(1)}$).

Definition 3.1.6 (security in the second malicious model, assuming honest majority): *Let f and Π be as in Definition 3.1.3, Protocol Π is said to* securely compute *f* (in the second malicious model) *if there exists a polynomial-time computable transformation of polynomial-size circuit families $A = \{A_n\}$ for the real model* (of Definition 3.1.3) *into polynomial-size circuit families $B = \{B_n\}$ for the ideal model* (of Definition 3.1.5) *so that for every $I \subset [m]$ with $|I| < m/2$*

$$\{\text{IDEAL}^{(2)}_{f,(I,B)}(\overline{x})\}_{n\in\mathbb{N},\, \overline{x}\in(\{0,1\}^n)^m} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi,(I,A)}(\overline{x})\}_{n\in\mathbb{N},\, \overline{x}\in(\{0,1\}^n)^m}$$

When the context is clear, we sometimes refer to Π as an implementation *of f.*

To deal with m as a parameter (rather than a fixed constant), one needs to consider sequences (of strings) so that both the length of individual strings as well as the number of strings may vary. Adversaries will be defined as families of circuits having two parameters (i.e., $C = \{C_{m,n}\}_{n,m\in\mathbb{N}}$), and polynomial-size would mean polynomial in both n and m. Clearly, all these extensions pose no real problem (beyond the usage of even more cumbersome notations).

3.2 Construction for the Semi-Honest Model

Our construction of private multi-party protocols (i.e., secure versus semi-honest behavior) for any given multi-argument functionality follows the presentation of the two-party case. For simplicity, we think of the number of parties m as being fixed. The reader may verify that the dependency of our constructions on m is at most polynomial.

Our protocol construction adapts the one used in the two-party case (see Section 2.2). That is, we consider a GF(2) circuit for evaluating the m-ary functionality f, and start by letting each party share its input bits with all other parties so that the sum of all shares equals the input bit. Going from the input wires to the output wires, we proceed to privately compute shares of each wire in the circuit so that the sum of the shares equals the correct value. We are faced with only one problem: When evaluating a multiplication gate of the circuit, we have party i holding bits a_i and b_i, and we need to conduct a private computation so that this party ends-up with a random bit c_i

and $(\sum_{i=1}^m a_i) \cdot (\sum_{i=1}^m b_i) = \sum_{i=1}^m c_i$ holds. More precisely, we are interested in privately computing the following randomized m-ary functionality

$$((a_1, b_1), ..., (a_m, b_m)) \mapsto (c_1, ..., c_m) \text{ uniformly in } \{0,1\}^m \tag{3.6}$$
$$\text{subject to } \sum_{i=1}^m c_i = (\sum_{i=1}^m a_i) \cdot (\sum_{i=1}^m b_i). \tag{3.7}$$

Thus, all that we need to do on top of Section 2.2 is to provide a private m-party computation of the above functionality. This is done by privately reducing, for arbitrary m, the computation of Eq. (3.6)–(3.7) to the computation of the same functionality in case $m = 2$, which in turn coincides with Eq. (2.8)–(2.9). But first we need to define an appropriate notion of reduction. Indeed, the new notion of reduction is merely a generalization of the notion presented in Section 2.2.

3.2.1 A composition theorem

We wish to generalize the notion of reduction presented in Section 2.2 (in the context of two-party (semi-honest) computation). Here the reduction is an m-party protocol which may invoke a k-ary functionality in its oracle calls, where $k \leq m$. In case $k < m$, an oracle call needs to specify also the set of parties who are to provide the corresponding k inputs. Actually, the oracle call needs to specify the order of these parties (i.e., which party should supply which input, etc.). (We note that the ordering of parties needs to be specified also in case $k = m$, and indeed this was done implicitly in Section 2.2, where the convention was that the party who makes the oracle is request is the one supplying the first input. In case $k > 2$ such a convention does not determine the correspondence between parties and roles, and thus we use below an explicit mechanism for defining the correspondence.)

Definition 3.2.1 (m-party protocols with k-ary oracle access): *As in the two-party case, a* oracle-aided protocol *is a protocol augmented by a pair of oracle-tapes, per each party, and oracle-call steps defined as follows. Each of the m parties may send a special* oracle request *message, to all other parties. The oracle request message contains a sequence of k distinct parties, called the* request sequence*, which are to supply queries in the current oracle call. In response, each party specified in the request sequence writes a string, called its* query*, on its own write-only oracle-tape. At this point the oracle is invoked and the result is that a string, not necessarily the same, is written by the oracle on the read-only oracle-tape of each of the k specified parties. This k-sequence of strings is called the* oracle answer*.*

One may assume, without loss of generality, that the party who invokes the oracle is the one who plays the role of the first party in the reduction (i.e., the first element in the request sequence is always the identity of the party which requests the current oracle call).

Definition 3.2.2 (reductions):

- *An m-party oracle-aided protocol is said to be* using the k-ary oracle-functionality f, *if the oracle answers are according to f. That is, when the oracle is invoked with request sequence $(i_1, ..., i_k)$, and the query-sequence $q_1, ..., q_k$ is supplied by parties $i_1, ..., i_k$, the answer-sequence is distributed as $f(q_1, ..., q_k)$. Specifically, party i_j in the m-party protocol* (the one which supplied q_j)*, is the one which obtains the answer part $f_j(q_1, ..., q_k)$.*
- *An m-party oracle-aided protocol using the k-ary oracle-functionality f is said to* privately compute *g if there exists a polynomial-time algorithm, denoted S, satisfying* Eq. (3.1)*, where the corresponding views are defined in the natural manner.*

- *An m-party oracle-aided protocol is said to* privately reduce *the m-ary functionality g to the k-ary functionality f, if it privately computes g when using the oracle-functionality f. In such a case we say that g* is privately reducible to *f,*

We are now ready to generalize Theorem 2.2.3:

Theorem 3.2.3 (Composition Theorem for the semi-honest model, multi-party case): *Suppose that the m-ary functionality g is privately reducible to the k-ary functionality f, and that there exists a k-party protocol for privately computing f. Then there exists an m-party protocol for privately computing g.*

Proof Sketch: The construction supporting the theorem is identical to the one used in the proof of Theorem 2.2.3: Let $\Pi^{g|f}$ be a oracle-aided protocol which privately reduces g to f, and let Π^f be a protocol which privately computes f. Then, a protocol Π for computing g is derived by starting with $\Pi^{g|f}$, and replacing each invocation of the oracle by an execution of Π^f. Clearly, Π computes g. We need to show that it privately computes g.

We consider an arbitrary set $I \subset [m]$ of semi-honest parties in the execution of Π. Note that, for $k < m$ (unlike the situation in the two-party case), the set I may induce different sets of semi-honest parties in the different executions of Π^f (replacing different invocations of the oracle). Still our "uniform" definition of simulation (i.e., uniform over all possible sets of semi-honest parties) keeps us away from trouble. Specifically, let $S^{g|f}$ and S^f be the simulators guaranteed for $\Pi^{g|f}$ and Π^f, respectively. We construct a simulation S, for Π, in the natural manner. On input $(I, \overline{x}_I, f_I(\overline{x}))$ (see Definition 3.1.1), we first run $S^{g|f}(I, \overline{x}_I, f_I(\overline{x}))$, and obtain the view of the semi-honest coalition I in $\Pi^{g|f}$. This view includes sequence of all oracle-call requests made during the execution as well as the sequence of parties which supplies query-parts in each such call. The view also contains the query-parts supplied by the parties in I as well as the corresponding responses. For each such oracle-call, we denote by J the subset of I which supplied query-parts in this call, and just invoke S^f providing it with the subset J as well as with the corresponding J-parts of queries and answers. Thus, we fill-up the view of I in the current execution of Π^f. (Recall that S^f can also handle the trivial cases in which either $|J| = k$ or $|J| = 0$.)

It is left to show that S indeed generates a distribution indistinguishable from the view of semi-honest parties in actual executions of Π. As in the proof of Theorem 2.2.3, this is done by introducing an imaginary simulator, denoted S'. This imaginary simulator invokes $S^{g|f}$, but augment the view of the semi-honest parties with views of actual executions of protocol Π^f on the corresponding query-sequences. (The query-sequences is completed in an arbitrary consistent way.) As in the proof of Theorem 2.2.3, one can show that the outputs of S' and S are computationally indistinguishable and that the output of S' is computationally indistinguishable from the view of the semi-honest parties in Π. The theorem follows. ■

3.2.2 Privately computing $\sum_i c_i = (\sum_i a_i) \cdot (\sum_i b_i)$

We now turn to the m-ary functionality defined in Eq. (3.6)–(3.7). Recall that the arithmetic is that of GF(2) (and so $-1 = +1$). The key observation is that

$$\left(\sum_{i=1}^m a_i\right) \cdot \left(\sum_{i=1}^m b_i\right) = \sum_{i=1}^m a_i b_i + \sum_{1 \le i < j \le m} (a_i b_j + a_j b_i) \tag{3.8}$$

$$= \quad (m-2) \cdot \sum_{i=1}^{m} a_i b_i + \sum_{1 \le i < j \le m} (a_i + a_j) \cdot (b_i + b_j) \qquad (3.9)$$

Now, looking at Eq. (3.9), we observe that each party, i, may compute (by itself) the term $(m-2)a_i b_i$, whereas each 2-subset, $\{i,j\}$, may privately compute shares to the term $(a_i+a_j)\cdot(b_i+b_j)$, by invoking Corollary 2.2.9. This leads to the following construction.

Construction 3.2.4 (privately reducing the m-party computation of Eq. (3.6)–(3.7) to the two-party computation of Eq. (2.8)–(2.9)):

Inputs: *Party i holds $(a_i, b_i) \in \{0,1\} \times \{0,1\}$, for $i = 1, ..., m$.*

Step 1 – Reduction: *Each pair of parties, (i,j), where $i < j$, invokes the 2-ary functionality of* Eq. (2.8)*–(2.9). Party i provides the input pair, (a_i, b_i), whereas Party j provides (a_j, b_j). Let us denote the oracle respond to Party i by $c_i^{\{i,j\}}$, and the respond to Party j by $c_j^{\{i,j\}}$.*

Step 2: *Party i sets $c_i = (m-2)a_i b_i + \sum_{j \neq i} c_i^{\{i,j\}}$.*

Outputs: *Party i outputs c_i.*

We first observe that the above reduction is valid; that is, the output of all parties indeed sum-up to what they should. It is also easy to see that the reduction is private. That is,

Proposition 3.2.5 *Construction 3.2.4 privately reduces the computation of the m-ary functionality given by* Eq. (3.6)*–(3.7) to the computation of the 2-ary functionality given by* Eq. (2.8)*–(2.9).*

Proof Sketch: We construct a simulator, denoted S, for the view of parties in the oracle-aided protocol, denoted Π, of Construction 2.2.7. Given a set of semi-honest parties, $I = \{i_1, ..., i_t\}$ (with $t < m$), and a sequence of inputs $(a_{i_1}, b_{i_1}), ..., (a_{i_t}, b_{i_t})$ and outputs $c_{i_1}, ..., c_{i_t}$, the simulator proceeds as follows.

1. For each pair, (i,j), where both $i, j \in I$, it uniformly selects $c_i^{\{i,j\}} \in \{0,1\}$ and sets $c_j^{\{i,j\}} = c_i^{\{i,j\}} + (a_i + a_j) \cdot (b_i + b_j)$.
2. Let $\bar{I} \stackrel{\rm def}{=} [m] \setminus I$, and let ℓ be the largest element in $\bar{I}$. (Such an $\ell \in [m]$ exists since $|I| < m$). For each $i \in I$ and each $j \in \bar{I} \setminus \{\ell\}$, the simulator uniformly selects $c_i^{\{i,j\}} \in \{0,1\}$. Finally, it sets $c_i^{\{i,\ell\}}$ to $c_i + (m-2)a_i b_i + \sum_{j \notin \{i,\ell\}} c_i^{\{i,j\}}$.
3. The simulator outputs all $c_i^{\{i,j\}}$'s generated above.

We claim that the output of the simulator is distributed identically to the view of the parties in I during the execution of the oracle-aided protocol. That is, we claim that for every such I and every $\overline{x} = ((a_1, b_1), ..., (a_m, b_m))$,

$$S(I, \overline{x}_I, f_I(\overline{x})) \equiv \textsc{view}_I^{\Pi}(\overline{x}) \qquad (3.10)$$

Note that $f_I(\overline{x})$ is uniformly distributed over $\{0,1\}^t$. The same holds also for the outputs of I in Π (by looking at the contribution of the $c_i^{i,\ell}$'s to the output of $i \in I$). So we may consider the two distributions in Eq. (3.10), when conditioned on any sequence of parties' outputs, $c_{i_1}, ..., c_{i_t}$. In such a case, we show that the views of parties in I (during an execution of Π) are distributed exactly as in the simulation. Specifically, for $i, j \in I$, the oracle answer on $((a_i, b_i), (a_j, b_j))$ is uniformly distributed over a pair of bits summing-up to $(a_i + a_j) \cdot (b_i + b_j)$ (which is exactly what happens in the simulation). Similarly, for every $i \in I$, the answers obtained in the $m-1$ oracle invocations will be uniform over the sequences agreeing with the above and summing-up to $c_i + (m-2)a_i b_i$. The proposition follows. ■

As an immediate corollary to Proposition 3.2.5, Corollary 2.2.9, and the Composition Theorem (Theorem 3.2.3), we obtain

Corollary 3.2.6 *Suppose that trapdoor permutation exist. Then the m-ary functionality of* Eq. (3.6)–*(3.7) is privately computable* (in the m-party semi-honest model).

3.2.3 The multi-party circuit evaluation protocol

For sake of completeness, we explicitly present the m-party analogue of the protocol of Section 2.2.4. Specifically, we show that the computation of any deterministic functionality, which is expressed by an arithmetic circuit over GF(2), is privately reducible to the functionality of Eq. (3.6)–(3.7).

Our reduction follows the overview presented in the beginning of this section. In particular, the sharing of a bit-value v between m parties means a uniformly selected m-sequence of bits $(v_1, ..., v_m)$ so that $v = \sum_{i=1}^{m} v_i$, where the i^{th} party holds v_i. Our aim is to propagate, via private computation, shares of the input wires of the circuit into shares of all wires of the circuit, so that finally we obtain shares of the output wires of the circuit.

We will consider an enumeration of all wires in the circuit. The input wires of the circuit, n per each party, will be numbered $1, 2...., m \cdot n$ so that, for $j = 1, ..., n$, the j^{th} input of Party i corresponds to the $(i-1) \cdot n + j^{\text{th}}$ wire. The wires will be numbered so that the output wires of each gate have a larger numbering than its input wires. The output-wires of the circuit are the last ones. For sake of simplicity we assume that each party obtains n output bits, and that the j^{th} output bit of the i^{th} party corresponds to wire $N - (m+1-i) \cdot n + j$, where N denotes the size of the circuit.

Construction 3.2.7 (privately reducing any deterministic m-ary functionality to the functionality of Eq. (3.6)–(3.7), for any $m \geq 2$):

Inputs: *Party i holds the bit string $x_i^1 \cdots x_i^n \in \{0,1\}^n$, for $i = 1, ..., m$.*

Step 1 – Sharing the inputs: Each party splits and shares each of its input bits with all other parties. *That is, for every $i = 1, ..., m$ and $j = 1, ..., n$, and every $k \neq i$, party i uniformly selects a bit $r_k^{(i-1)n+j}$ and sends it to party k as the party's share of input wire $(i-1) \cdot n + j$. Party i sets its own share of the $(i-1) \cdot n + j^{\text{th}}$ input wire to $x_i^j + \sum_{k \neq i} r_k^{(i-1)n+j}$.*

Step 2 – Circuit Emulation: Proceeding by the order of wires, the parties use their shares of the two input wires to a gate in order to privately compute shares for the output wire of the gate. *Suppose that the parties hold shares to the two input wires of a gate; that is, for $i = 1, ..., m$, Party i holds the shares a_i, b_i, where $a_1, ..., a_m$ are the shares of the first wire and $b_1, ..., b_m$ are the shares of the second wire. We consider two cases.*

Emulation of an addition gate: *Each party, i, just sets its share of the output wire of the gate to be $a_i + b_i$.*

Emulation of a multiplication gate: *Shares of the output wire of the gate are obtained by invoking the oracle for the functionality of* Eq. (3.6)–*(3.7), where Party i supplies the input* (query-part) (a_i, b_i). *When the oracle responses, each party sets its share of the output wire of the gate to equal its part of the oracle answer.*

Step 3 – Recovering the output bits: Once the shares of the circuit-output wires are computed, each party sends its share of each such wire to the party with which the wire is associated. *That is, for $i = 1, ..., m$ and $j = 1, ..., n$, each party sends its share of wire $N - (m+1-i) \cdot n + j$ to Party i. Each party recovers the corresponding output bits by adding-up the corresponding m shares; that is, the share it had obtained in Step 2 and the $m-1$ shares it has obtained in the current step.*

Outputs: *Each party locally outputs the bits recovered in Step 3.*

For starters, let us verify that the output is indeed correct. This can be shown by induction on the wires of the circuits. The induction claim is that the shares of each wire sum-up to the correct value of the wire. The base case of the induction are the input wires of the circuits. Specifically, the $(i-1) \cdot n + j^{\text{th}}$ wire has value x_i^j and its shares indeed sum-up to x_i^j. For the induction step we consider the emulation of a gate. Suppose that the values of the input wires (to the gate) are a and b, and that their shares $a_1, ..., a_m$ and $b_1, ..., b_m$ indeed satisfy $\sum_i a_i = a$ and $\sum_i b_i = b$. In case of an addition gate, the shares of the output wire set to be $a_1 + b_1, ..., a_m + b_m$ indeed satisfying

$$\sum_i (a_i + b_i) \;=\; \left(\sum_i a_i\right) + \left(\sum_i b_i\right) \;=\; a + b$$

In case of a multiplication gate, the shares of the output wire were set to be $c_1, ..., c_m$, so that $\sum_i c_i = (\sum_i a_i) \cdot (\sum_i b_i)$ holds. Thus, $\sum_i c_i = a \cdot b$ as required.

Privacy of the reduction. Analogously to Proposition 2.2.11, we now show that Construction 3.2.7 indeed privately reduces the computation of a circuit to the multiplication-gate emulation. That is,

Proposition 3.2.8 *Construction 3.2.7 privately reduces the evaluation of arithmetic circuits over* GF(2)*, representing an m-ary deterministic functionality, to the functionality of* Eq. (3.6)–*(3.7).*

Proof Sketch: In proving this proposition, for any set $I \subset [m]$, we need to simulate the joint view of the parties in $I = \{i_1, ..., i_t\}$. This is done by direct analogy to the proof of Proposition 2.2.11, where $m = 2$ (and where we have considered, without loss of generality, $I = \{1\}$).

The simulator gets the inputs, denoted $x_{i_1}^1, ..., x_{i_1}^n, ..., x_{i_t}^1, ..., x_{i_t}^n$, as well as the outputs, denoted $y_{i_1}^1, ..., y_{i_1}^n, ..., y_{i_t}^1, ..., y_{i_t}^n$, to the parties in I. It operates as follows.

1. The simulator uniformly selects coins for each party in I, as done in Step 1 of the protocol. These coins determine the messages to be sent to other parties in I as well as the party's shares of its own inputs (as in Step 1). Specifically, the share of Party i, where $i \in I$, of the input wire corresponding to x_i^j is set to equal the sum of all other shares (of this wires) with x_i^j. Finally, the simulator selects uniformly values to be used as Step 1 incoming-messages each party in I has received from parties in $\bar{I} \stackrel{\text{def}}{=} [m] \setminus I$ (as shares of these parties' inputs).

   This completes the computation of the shares of all circuit-input wires held by parties in I.

2. The shares, held by the parties in I, of all other wires are computed, iteratively gate-by-gate, as follows.

   - The share of each party $i \in I$ in the output-wire of an addition gate is set to be the sum of its shares of the input-wires of the gate.
   - The share of each party $i \in I$ in the output-wire of a multiplication gate is uniformly selected in $\{0,1\}$.

   (The shares computed for output-wires of multiplication gates will be used as the answers obtained, by parties in I, from the oracle.)

3. For each wire corresponding to an output, denoted y_i^j, available to Party i in I, the simulator uniformly selects a sequence of m bits among the sequences which match the shares (of this wire) held by parties in I and sum-up to y_i^j. (In case $|I| = m-1$, this sequence is uniquely determined.)

4. The simulator outputs the concatenation of the above x_i^j's and y_i^j's with the coins generated for each party in I and the incoming-messages and oracle-answers generated for it. In particular, the latter include the messages generated in Step 1 (simulating Step 1 of the protocol), the concatenation of the bits selected for the output-wires of multiplication gates (corresponding to the party's view of the oracle answers in Step 2), and the sequences generated in Step 3 (corresponding to the party's view in Step 3 of the protocol).

Analogously to the proof of Proposition 2.2.11, one may verify that the output of the simulation is distributed identically to the view of parties I in the execution of the oracle-aided protocol. The proposition follows. ■

Conclusion. As an immediate corollary to Proposition 3.2.8, Corollary 3.2.6, and the Composition Theorem (Theorem 3.2.3), we obtain.

Corollary 3.2.9 *Suppose that trapdoor permutation exist. Then any* deterministic *m-ary functionality is privately computable* (in the m-party semi-honest model).

Furthermore, as in Section 2.2, we may privately reduce the computation of a general (randomized) m-ary functionality, g, to the computation of the deterministic m-ary functionality, f, defined by

$$f((x_1, r_1), ..., (x_m, r_m)) \stackrel{\text{def}}{=} g(\oplus_{i=1}^m r_i, (x_1, ..., x_m)) \tag{3.11}$$

where $g(r, \overline{x})$ denote the value of $g(\overline{x})$ when using coin tosses $r \in \{0,1\}^{\text{poly}(|\overline{x}|)}$ (i.e., $g(\overline{x})$ is the randomized process consisting of uniformly selecting $r \in \{0,1\}^{\text{poly}(|\overline{x}|)}$, and deterministically computing $g(r, \overline{x})$). Combining this reduction with Corollary 3.2.9 and Theorem 3.2.3, we have

Theorem 3.2.10 *Suppose that trapdoor permutation exist. Then any m-ary functionality is privately computable* (in the m-party semi-honest model).

3.3 Forcing Semi-Honest Behavior

Our aim is to use Theorem 3.2.10 in order to establish the main result of this chapter; that is,

Theorem 3.3.1 (main result for multi-party case): *Suppose that trapdoor permutation exist. Then any m-ary functionality can be securely computable in each of the two malicious models.*

The theorem will be established in two steps. Firstly, we compile any protocol for the semi-honest model into an "equivalent" protocol for the first malicious model. The compiler is very similar to the one used in the two-party case. Next, we compile any protocol for the first malicious model into an "equivalent" protocol for the second malicious model. The heart of the second compiler is a primitive alien to the two-party case – Verifiable Secret Sharing (VSS).

For simplicity, we again think of the number of parties m as being fixed. The reader may again verify that the dependency of our constructions on m is at most polynomial.

3.3.1 Changing the communication model

To simplify the exposition of the multi-party compilers, we describe them as producing protocols for a communication model consisting of a single broadcast channel (and no point-to-point links). We assume, without loss of generality, that in each communication round only one (predetermined) party may send a message and that this message arrives to all processors. Such a broadcast channel can be implemented via an (authenticated) Byzantine Agreement protocol, thus providing an emulation of our model on the standard point-to-point model (in which a broadcast channel does not exist).

We stress that indeed the first compiler, as presented below, transforms protocols which are secure in the semi-honest point-to-point model (of private channels) into protocols secure in the (first) malicious broadcast model. Actually, we first preprocess protocols secure in the semi-honest point-to-point model into protocols secure in the semi-honest broadcast-channel model, and only then apply the two compilers, each taking and producing protocols in the broadcast-channel model (alas secure against different types of adversaries). Thus, the full sequence of transformations establishing Theorem 3.3.1 based on Theorem 3.2.10 is as follows

- *Precompiling* a protocol Π_0 which privately computes a functionality f in the point-to-point model (of the previous section), into a protocol Π'_0 which privately computes f in the broadcast model (where no private point-to-point channels exist).

- Using the first compiler (of Section 3.3.2) to transform Π'_0 (secure in the semi-honest model) into a protocol Π'_1 secure in the first malicious model.

  We stress that both Π'_0 and Π'_1 operate and are evaluated for security in a communication model consisting of a single broadcast channel. The same holds also for Π'_2 mentioned next.

- Using the second compiler (of Section 3.3.3) to transform Π'_1 (secure in the first malicious model) into a protocol Π'_2 secure in the second malicious model.

- *Postcompiling* each of the protocols Π'_1 and Π'_2, which are secure in the first and second malicious models when communication is via a broadcast channel, into corresponding protocols, Π_1 and Π_2, for the standard point-to-point model. That is, Π_1 (resp., Π_2) securely computes f in the first (resp., second) malicious model in which communication is via point-to-point channels.

We note that security holds even if the adversary is allowed to wire-tap the (point-to-point) communication lines between honest parties.

We start by discussing the security definitions for the broadcast communication model, and presenting the precompiler and the postcompiler mentioned above. Once this is done, we turn to the real core of this section – the two compilers which operate on protocols in the broadcast channel.

Definitions. Indeed, security in the broadcast model was not defined above. However, the three relevant definitions for the broadcast communication model are easily derived from the corresponding definitions given in Section 3.1, assuming a point-to-point communication model. Specifically, in defining security in the semi-honest model one merely includes the entire transcript of the communication over the (single) broadcast channel in the party's view. Similarly, when defining security in the two malicious models one merely notes that the "real execution model" (i.e., $\text{REAL}_{\Pi,(I,C)}$) changes (as the protocol is now executed over a different communication media), whereas the "ideal model" (i.e., $\text{IDEAL}^{(1)}_{f,(I,C)}$ or $\text{IDEAL}^{(2)}_{f,(I,C)}$) remains intact.

Precompiler. It is easy to (securely) emulate over a (single) broadcast channel any protocol Π for the (private) point-to-point communication model. All one needs to do is use a secure public-key encryption scheme. That is, each party randomly generates a pair of encryption/decryption keys, posts the encryption-key on the broadcast channel, and keeps the decryption-key secret. Any party instructed (by Π) to send a message, `msg`, to Party i, encrypts `msg` using the encryption-key posted by Party i, and places the resulting ciphertext on the broadcast channel (indicating that it is intended for Party i). Party i recovers `msg` by using its decryption-key, and proceeds as directed by Π. Denote the resulting protocol by Π'. Note that we merely consider the effect of this transformation in the semi-honest model.

Proposition 3.3.2 (precompiler): *Suppose that trapdoor permutation exist. Then any m-ary functionality is privately computable in the broadcast communication model.*

Proof Sketch: Let f be an m-ary functionality, and Π be a protocol (guaranteed by Theorem 3.2.10) for privately computing f in the point-to-point communication model. Given a trapdoor permutation, we construct a secure public-key encryption scheme and use it to transform Π into Π' as described above. To simulate the view of parties in an execution of Π' (taking place in the broadcast communication model), we first simulate their view in an execution of Π (taking place in the point-to-point communication model). We then encrypt each message sent by a party in the semi-honest coalition, as this would be done in an execution of Π'. Note that we know both the message and the corresponding encryption-key. We do the same for messages received by semi-honest parties. All that remain is to deal with messages, which we may not know, sent between two honest parties. Here we merely place an encryption of an arbitrary message. This concludes the description of the "broadcast-model" simulator.

The analysis of the latter simulator combines the guarantee given for the "point-to-point simulator" and the guarantee that the encryption scheme is secure. That is, ability to distinguish the output of the "broadcast-model" simulator from the execution view (in the broadcast model) yields either (1) ability to distinguish the output of the "point-to-point" simulator from the execution view (in the point-to-point model) or (2) ability to distinguish encryptions under the above public-key encryption scheme. In both cases we reach contradiction to our hypothesis. ■

Postcompiler. Here we go the other way around. We are given a protocol which securely computes (in one of the two malicious models) some functionality, where the protocol uses a broadcast channel. We wish to convert this protocol into an equivalent one which works in a point-to-point communication model. (Actually, we do not go all the way back, as we do not assume these point-to-point lines to provide private communication.) Thus, all we need to do is emulate a broadcast channel over a point-to-point network and in the presence of malicious parties – which reduces to solving the celebrated Byzantine Agreement problem. However, we have signature schemes at our disposal and so we merely need to solve the much easier problem known as *authenticated Byzantine Agreement*. For sake of self-containment we define the problem and present a solution.

Authenticated Byzantine Agreement: Suppose a synchronous point-to-point model of communication and a signature scheme infrastructure. That is, each party knows the verification-key of all other parties. Party 1 has an input bit, denoted σ, and the objective is to let all honest parties agree on the value of this bit. In case Party 1 is honest, they must agree on its actual input, but otherwise they may agree on any value (as long as they agree).

Construction 3.3.3 (Authenticated Byzantine Agreement): *Let m denote the number of parties. We assume that the signature scheme in use has signature length which depends only of the security parameter, and not on the length of the message to be signed.*[7]

1. Phase 0: *Party 1 should sign its input and sends it to all parties.*
2. Phase $i = 1, ..., m$: *Each honest party* (other than Party 1) *proceeds as follows:*
   (a) *It inspects the messages it has received at Phase $i-1$. Such a message is* admissible *if it has the form $(v, s_{p_0}, s_{p_1}, ..., s_{p_{i-1}})$, where $p_0 = 1$, all p_j's are distinct, and for every $j = 0, ..., i-1$, the string s_{p_j} is accepted as a signature to $(v, s_{p_0}, s_{p_1}, ..., s_{p_{j-1}})$ relative to the verification key of party p_j. Such an admissible message is called an* authentic $(v, i-1)$-message *or an* authentic v-message*.*

      (We comment that p_{i-1} is different from the identity of the processing party, receiving the message.)

   (b) *If the party finds an authentic $(v, i-1)$-message among these messages then it signs this authentic $(v, i-1)$-message, appends the signature to it, and sends the resulting message to all parties.*[8]

      Note that the resulting message is an authentic (v, i)-message.
3. *Each honest party* (other than Party 1) *evaluates the situation as follows:*
   - *If it has received both an authentic 0-message and an authentic 1-message then it decides that Party 1 is malicious and outputs a default value, say 0.*
   - *If for a single $v \in \{0,1\}$ it has received an authentic v-message then it outputs the value v.*

[7] Such a signature scheme can be constructed given any one-way function. In particular, one signs the hash-value of the message under a universal one-way hashing function [52]. Maintaining short signatures is important in this application since we are going to iteratively sign messages consisting of the concatenation of an original message and prior signatures.

[8] For sake of efficiency, one may instruct the party not to process this authentic $(v, i-1)$-message in case it has seen an authentic (v, j)-message for any $j < i-1$ in a prior phase.

- *If it has never received an authentic v-message, for any $v \in \{0,1\}$, then it decides that Party 1 is malicious and outputs a default value, say 0.*

The protocol can be easily adapted to handle non-binary input values. For sake of efficiency, one may instruct honest parties to forward at most two authentic messages (as this suffices to establish that Party 1 is malicious).

Proposition 3.3.4 (Authenticated Byzantine Agreement): *Assuming that the signature scheme in use is unforgeable, Construction 3.3.3 satisfies the following two conditions:*

1. *It is infeasible to make any two honest parties output different values.*
2. *If Party 1 is honest then it is infeasible to make any honest party output a value different from the input of Party 1.*

Proof Sketch: Suppose that in Phase i, some honest party sees an authentic $(v, i-1)$-message. For this to happen we must have $i - 1 \leq m$. Then, it will send an authentic (v, i)-message in this phase and so all honest parties will see an authentic (v, i)-message in Phase $i + 1$, where $i + 1 \leq m$. Thus, if an honest party see a single (or both possible) authentic v-message then so do all other honest parties, and Part 1 follows. Part 2 follows by noting that if Party 1 is honest and has input v then all honest parties see an authentic $(v, 0)$-message. Furthermore, none can see an authentic v'-message, for $v' \neq v$. ■

3.3.2 The first complier

We follow the basic structure of the compiler presented in Section 2.3, and the reader is referred there for further discussion. Adapting that compiler to the multi-party setting merely requires generalizing the implementation of each of the three phases. Following is a high-level description of the multi-party protocols generated by the (multi-party) compiler. Recall that all communication, both in the input protocol as well as in the one resulting from the compilation, is conducted merely by posting messages on a single broadcast channel.

Input-commitment phase: Each of the parties commits to each of its input bits. This will be done using a multi-party version of the input-commitment functionality of Eq. (2.21).

Intuitively, malicious parties may (abort or) substitute their inputs during this phase, but they may do so depending only on the value of the inputs held by (all) malicious parties.

Coin-generation phase: The parties generate random-pads for each of the parties. These pads are intended to serve as the coins of the corresponding parties in their emulation of the semi-honest protocol. Each party obtains the bits of the random-pad to be held by it, whereas the other parties obtains commitments to these bits. This will be done using a multi-party version of coin-tossing functionality of Eq. (2.14).

Intuitively, malicious parties may abort during this phase, but otherwise they end-up with a uniformly distributed random-pad.

Protocol emulation phase: The parties emulate the execution of the semi-honest protocol with respect to the input committed in the first phase and the random-pads selected in the second phase. This will be done using a multi-party version of the authenticated-computation functionality of Eq. (2.28).

In order to implement the above phases, we define the natural extensions of the coin-tossing, input-commitment and authenticated-computation functionalities (of the two-party case), and present secure implementations of them in the current (first malicious) multi-party model. The original definitions and constructions are obtained by setting $m = 2$.

3.3.2.1 Multi-party coin-tossing into the well

We extend Definition 2.3.5 (from $m = 2$) to arbitrary m, as follows.

Definition 3.3.5 (coin-tossing into the well, multi-party version): *An m-*party coin-tossing into the well *is an m-party protocol for securely computing* (in the first malicious model) *the following randomized functionality with respect to* some fixed *commitment scheme,* $\{C_n\}_{n\in\mathbb{N}}$,

$$(1^n, ..., 1^n) \mapsto ((b, r), C_n(b, r), ..., C_n(b, r)) \tag{3.12}$$

where (b, r) is uniformly distributed in $\{0,1\} \times \{0,1\}^n$.

Construction 2.3.6 generalizes naturally to the multi-party setting. In the generalization we use the fact that the zero-knowledge proofs (and proof of knowledge) employed are of the public-coin (a.k.a Arthur-Merlin) type. That is, the role of the verifier in these proof systems is restricted to tossing coins, sending their outcome to the prover, and evaluating a predetermined predicate at the end of the interaction. Thus, anybody seeing the transcript of the interaction (i.e., the sequence of messages exchanged over the broadcast channel) can determine whether the verifier has accepted or rejected the proof.

Construction 3.3.6 (Construction 2.3.6, generalized):

Inputs: *Each party gets security parameter 1^n.*

Convention: *Any deviation from the protocol, by a party other than Party 1, will be interpreted as a canonical legitimate message. In case Party 1 aborts or is detected cheating, all honest parties halt outputting the special symbol $\perp$.*

Step C1: *The parties generate uniformly distributed bits, $b_0, b_1, ..., b_n$, known to Party 1.*

Specifically, for $j = 0, 1, ..., n$, the parties execute the following four steps:

Step C1.1: *Party 1 uniformly selects $(\sigma_j, s_j) \in \{0,1\} \times \{0,1\}^n$, and places $c_j \stackrel{\text{def}}{=} C_n(\sigma_j, s_j)$ on the broadcast channel.*

Step C1.2: *The parties invoke $m-1$ instances of a zero-knowledge strong-proof-of-knowledge so that Party 1 plays the prover and each of the other $m-1$ parties plays the verifier in one of these invocations (i.e., in the i^{th} invocation Party $i+1$ plays the verifier). The common input to the proof system is c_j, the prover gets auxiliary input (σ_j, s_j), and its objective is to prove that it knows (x, y) such that*

$$c_j = C_n(x, y) \tag{3.13}$$

We stress that all $m-1$ invocations of the proof system takes place over the broadcast channel, and so all parties may determine if the verifier should accept or reject in each of them. In case the verifier should reject in any of these invocations of the proof system, all parties aborts with output $\perp$.

Step C1.3: *For $i = 2, ..., m$, Party i uniformly selects $\sigma_j^{(i)} \in \{0,1\}$, and places $\sigma_j^{(i)}$ on the channel.*

Step C1.4: *Party 1 sets $b_j = \sigma_j \oplus (\oplus_{i=2}^m \sigma_j^{(i)})$.*

Step C2: *Party 1 sets $b = b_0$ and $r = b_1 b_2 \cdots b_n$, and places $c \stackrel{\rm def}{=} C_n(b,r)$ on the channel.*

Step C3: *The parties invoke $m-1$ instances of a zero-knowledge proof system so that Party 1 plays the prover and each of the other $m-1$ parties plays the verifier (i.e., in the $i^{\rm th}$ instance Party $i+1$ plays the verifier). The common input to the proof system is $(c_0, c_1, ..., c_n, \sigma'_0, \sigma'_1, ..., \sigma'_n, c)$, where $\sigma'_j \stackrel{\rm def}{=} \oplus_{i=2}^m \sigma_j^{(i)}$, the prover gets auxiliary input $(\sigma_0, \sigma_1, ..., \sigma_n, s_0, s_1, ..., s_n)$, and its objective is to prove that there exists $(x_0, x_1, ..., x_n, y_0, y_1, ..., y_n)$ such that*

$$(\forall j \;\; c_j = C_n(x_j, y_j)) \wedge (c = C_n(x_0 \oplus \sigma'_0, (x_1 \oplus \sigma'_1) \cdots (x_n \oplus \sigma'_n))) \tag{3.14}$$

Again, all $m-1$ invocations of the proof system takes place over the broadcast channel, and so all parties may determine if the verifier should accept or reject in each of them. In case the verifier should reject in any of these invocations of the proof system, all parties aborts with output $\perp$.

Outputs: *Party 1 sets its local output to (b,r), and each other party sets its local output to c, provided they did not halt with output $\perp$ before.*

The fact that the above protocol constitute an m-party coin-tossing protocol (as in Definition 3.3.5) is established analogously to the proof of Proposition 2.3.7. Specifically, one distinguishes the case in which Party 1 is honest (i.e., $1 \notin I$) from the case in which Party 1 belongs to the malicious coalition. In the first case, all honest parties are shown to produce a proper output (and towards this end one relies again on the perfect completeness[9] of the proof systems in use). In the second case, the special symbol $\perp$ may occur as output, but this is allowed by Definition 3.1.2. Thus, we have

Proposition 3.3.7 *Suppose that $\{C_n\}_{n\in\mathbb{N}}$ is a commitment scheme. Then Construction 3.3.6 securely implements the coin-tossing functionality of* Eq. (3.12) (in the first malicious model).

Proof Sketch: We employs one of the two strategies used in the proof of Proposition 2.3.7, depending on whether Party 1 is honest (i.e., $1 \notin I$) or not (i.e., $1 \in I$).

In case Party 1 is honest, the adversary strategy is transformed (from the real model to the ideal one) as in the first transformation in the proof of Proposition 2.3.7. Note that the only "effective" communication in the protocol is between Party 1 and each of the other parties. Thus the argument is essentially as in the two-party case.

In case Party 1 belongs to the malicious parties (i.e., $1 \in I$), the adversary strategy is obtained analogous to the second transformation in the proof of Proposition 2.3.7. In this case we merely rely on the fact that there exists $i \notin I$, and conclude that since this party has executed the role of verifier properly the verifier-acceptance indicates validity of the corresponding claim. The rest of the argument is essentially as in the two-party case, with one additional concern: Using the fact that honest parties abort based on the publically known decisions of the verifier in all instances of the proof systems employed, we conclude that they either all output $c = C_n(b,r)$ or all abort (i.e., output $\perp$). ■

[9] By *perfect completeness* we mean that, whenever the corresponding assertion does holds, the prover may convince the verifier with probability 1.

3.3.2.2 Multi-party input-commitment protocol

We extend the definition of the bit-committing functionality of Eq. (2.21) (from $m = 2$) to arbitrary m. Recall that, as before, $\{C_n\}_{n\in\mathbb{N}}$ is an arbitrary commitment scheme.

$$((x, r), 1^n, ..., 1^n) \;\mapsto\; (\lambda, C_n(x, r), ..., C_n(x, r)) \tag{3.15}$$

Construction 2.3.8 generalizes naturally to the multi-party setting.

Construction 3.3.8 (multi-party input-bit commitment protocol):

Inputs: *Party 1 gets input $(\sigma, r) \in \{0,1\} \times \{0,1\}^n$, all other parties gets input 1^n.*

Conventions: *As in Construction 3.3.6.*

Step C1: *Party 1 posts $c \stackrel{\text{def}}{=} C_n(\sigma, r)$ on the broadcast channel.*

Step C2: *The parties invoke $m-1$ instances of a zero-knowledge strong-proof-of-knowledge so that Party 1 plays the prover and each of the other $m-1$ parties plays the verifier in one of the invocations. The common input to the proof system is c, the prover gets auxiliary inputs (σ, r), and its objective is to prove that it knows (x, y) such that $c = C_n(x, y)$. In case the verifier rejects the proof, all parties abort with output $\perp$ (otherwise the output will be c).*

Outputs: *For $i = 2, .., m$, Party i sets its local output to c.*

Again, correctness is established analogously to the two-party case (i.e., Proposition 2.3.9).

Proposition 3.3.9 *Suppose that trapdoor permutation exist. Then, Construction 3.3.8 securely computes* (in the first malicious model) *the functionality* Eq. (3.15).

Proof Sketch: Again, we employs one of the two strategies used in the proof of the corresponding two-party case (i.e., Proposition 2.3.9), depending on whether Party 1 is honest (i.e., $1 \notin I$) or not (i.e., $1 \in I$). The adaptation is analogous to what was done in the proof of Proposition 3.3.7. ■

3.3.2.3 Multi-party authenticated-computation protocol

Finally, we extend the definition of the authenticated-computation functionality of Eq. (2.28) (from $m = 2$) to arbitrary m. As in Eq. (2.28), we consider two two-argument functions, $f, h : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$, each being polynomial-time computable. Recall that h captures information on α available to all parties, whereas f captured the desired computation (which may also depend on an auxiliary input β).

$$((\alpha, r, \beta), (h(\alpha, r), \beta), ..., (h(\alpha, r), \beta)) \;\mapsto\; (\lambda, f(\alpha, \beta), ..., f(\alpha, \beta)) \tag{3.16}$$

As before, we make the simplifying assumption that h is 1-1 with respect to its first argument; that is, for every $\alpha \neq \alpha'$ and any r, r' we have $h(\alpha, r) \neq h(\alpha', r')$. The construction used in the proof of Proposition 2.3.12 generalizes in the obvious way and we obtain.

Proposition 3.3.10 *Suppose that trapdoor permutation exist, and that the function $h : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$ satisfies that for every $\alpha \neq \alpha'$, the sets $\{h(\alpha, r) : r \in \{0,1\}^*\}$ and $\{h(\alpha', r) : r \in \{0,1\}^*\}$ are disjoint. Then, the functionality of* Eq. (3.16) *can be securely computed* (in the first malicious model).

Proof Sketch: The desired protocol follows (using conventions as above).

Inputs: Party 1 gets input (α, r, β), and each other party gets input (u, β), where $u = h(\alpha, r)$.

Step C1: Party 1 posts $v \stackrel{\text{def}}{=} f(\alpha, \beta)$ on the broadcast channel.

Step C2: The parties invoke $m-1$ instances of a zero-knowledge proof system so that Party 1 plays the prover and each of the other $m-1$ parties plays the verifier in one of the invocations. The common input to the proof system is (v, u, β), the prover gets auxiliary inputs (α, r), and its objective is to prove that Eq. (2.29) holds (i.e., there exists x, y s.t $(u = h(x, y)) \wedge (v = f(x, \beta))$). In case the verifier rejects the proof, all parties abort with output $\perp$ (otherwise the output will be c).

Outputs: For every $i = 2, ..., m$, Party i sets its local output to v.

The fact that the above protocol securely computed the functionality of Eq. (3.16) is established by adapting (as above) the proof presented in the two-party case. ■

3.3.2.4 The compiler itself

We are now ready to present the compiler. Recall that we are given a multi-party protocol, Π, for the semi-honest model, and we want to generate an "equivalent" protocol Π' for the first malicious model. Recall that both the given protocol and the one generated operate in a communication model consisting of a single broadcast channel. The compiler is a generalization of the one presented in Construction 2.3.13 (for $m = 2$), and the reader is referred there for additional clarifications.

Construction 3.3.11 (The first multi-party compiler): *Given an m-party protocol, Π, for the semi-honest model, the compiler produces the following m-party protocol, denoted Π', for the first malicious model.*

Inputs: *Party i gets input $x^i = x_1^i x_2^i \cdots x_n^i \in \{0,1\}^n$.*

Input-commitment phase: *Each of the parties commits to each of its input bits by using a secure implementation of the input-commitment functionality of* Eq. (3.15). *These executions are preceded by the "committing party" selecting a randomization for the commitment scheme C_n.*

That is, for $i = 1, ..., m$ and $j = 1, ..., n$, Party i uniformly selects $\rho_j^i \in \{0,1\}^n$, and invokes a secure implementation of the input-commitment functionality of Eq. (3.15), *playing Party 1 with input (x_j^i, ρ_j^i). The other parties play the role of the other parties in* Eq. (2.21) *with input 1^n, and obtain the output $C_n(x_j^i, \rho_j^i)$. Party i records ρ_j^i, and each other party record $C_n(x_j^i, \rho_j^i)$.*

Coin-generation phase: *The parties generate random-pads for the emulation of Π. Each party obtains the bits of the random-pad to be held by it, whereas the other party obtains commitments to these bits. This is done by invoking a secure implementation of the coin-tossing functionality of* Eq. (3.12). *Specifically, the coin-tossing protocol is invoked $m \cdot c(n)$ times, where $c(n)$ is the length of the random-pad required by one party in Π.*

That is, for $i = 1, ..., m$ and $j = 1, ..., c(n)$, Party i invokes a secure implementation of the coin-tossing functionality of Eq. (3.12) *playing Party 1 with input 1^n, and the other parties play the other roles. Party i obtains a pair, (r_j^i, ω_j^i), and each other party obtains the corresponding* *output* $C_n(r_j^i, \omega_j^i)$. *Party* i *sets the* j^{th} *bit of the random-pad for the emulation of* Π *to be* r_j^i*, and records the corresponding NP-witness* (i.e., ω_j^i). *Each other party records* $C_n(r_j^i, \omega_j^i)$. *In the sequel, we let* $r^i = r_1^i r_2^i \cdots r_{c(n)}^i$ *denote the random-pad generated for Party* i.

Protocol emulation phase: *The parties use a secure implementation of the authenticated-computation functionality of* Eq. (3.16) *in order to emulate each step of protocol* Π. *The party which is supposed to send a message plays the role of Party 1 in* Eq. (3.16) *and the other parties play the other roles. The inputs* α, r, β *and the functions* h, f*, for the functionality of* Eq. (3.16)*, are set as follows:*

- *The string* α *is set to equal the concatenation of the party's original input and its random-pad, the string* r *is set to be the concatenation of the corresponding randomizations used in the commitments and* $h(\alpha, r)$ *equals the concatenation of the commitments themselves. That is, suppose the message is supposed to be sent by Party* j *in* Π. *Then*

$$\begin{aligned}
\alpha &= (x^j, r^j)\,, \text{ where } r^j = r_1^j r_2^j \cdots r_{c(n)}^j \text{ and } x^j = x_1^j x_2^j \cdots x_n^j \\
r &= (\rho_1^j \rho_2^j \cdots \rho_n^j, \omega_1^j \omega_2^j \cdots \omega_{c(n)}^j) \\
h(\alpha, r) &= (C_n(x_1^j, \rho_1^j), C_n(x_2^j, \rho_2^j), ..., C_n(x_n^j, \rho_n^j)\,, \\
&\qquad C_n(r_1^j, \omega_1^j), C_n(r_2^j, \omega_2^j), ..., C_n(r_{c(n)}^j, \omega_{c(n)}^j))
\end{aligned}$$

  Note that h *indeed satisfies* $h(\alpha, r) \neq h(\alpha', r')$ *for all* $\alpha \neq \alpha'$ *and all* r, r'.

- *The string* β *is set to equal the concatenation of all previous messages sent* (over the broadcast channel) *by all other parties.*

- *The function* f *is set to be the computation which determines the message to be sent in* Π. *Note that this message is computable in polynomial-time from the party's input* (denoted x^j above)*, its random-pad* (denoted r^j)*, and the previous messages posted so far* (i.e., β).

Aborting: *In case any of the protocols invoked in any of the above phases terminates in an abort state, the party (or parties) obtaining this indication aborts the execution, and sets its output to* $\perp$. *Otherwise, outputs are as follows.*

Outputs: *At the end of the emulation phase, each party holds the corresponding output of the party in protocol* Π. *The party just locally outputs this value.*

We note that both the compiler and the protocols produced by it are efficient, and that their dependence on m is polynomially bounded.

3.3.2.5 Analysis of the compiler

The effect of Construction 3.3.11 will be analyzed analogously to the effect of Construction 2.3.13. In view of this similarity we combine the two main steps in this analysis, and state the end result –

Theorem 3.3.12 (Restating half of Theorem 3.3.1): *Suppose that trapdoor permutation exist. Then any* m*-ary functionality can be securely computable in the* first *malicious model* (using only point-to-point communication lines)*. Furthermore, security holds even if the adversary can read all communication among honest players.*

Proof Sketch: We start by noting that the definition of the *augmented semi-honest model* (i.e., Definition 2.3.14) applies with any change to the multi-party context, also in case the communication is via a single broadcast channel. Recall that the *augmented semi-honest model* allows parties to enter the protocol with modified inputs (rather than the original ones), and abort the execution at any point in time. We stress that in the multi-party augmented semi-honest model, an adversary controls all non-honest parties and coordinates their input modifications and abort decisions. As in the two-party case, other than these non-proper actions, the non-honest parties follow the protocol (as in the semi-honest model).

We stress that unless stated differently, all subsequent statements will refer to the single broadcast channel communication model. (Only at the very end of this proof, we pass to the point-to-point communication model.)

The first significant part of the proof is showing that the compiler of Construction 3.3.11 transforms any protocol Π into a protocol Π' so that executions of Π' in the (real) first malicious model can be emulated by executions of Π in the augmented semi-honest model. This part is analogous to Proposition 2.3.15, and its proof is analogous to the proof presented in the two-party case. That is, we transform any malicious adversary (A, I) (for executions of Π') into an augmented semi-honest adversary, (B, I). The construction of B out of A in analogous to the construction of $B_{\mathtt{mal}}$ out of $A_{\mathtt{mal}}$ (carried out in the proof of Proposition 2.3.15): Specifically, B modifies inputs according to the queries A makes in input-committing phase, uniformly selects random-pad (in accordance to the coin-generation phase), and aborts in case the emulated machine does so. Thus, B which is an augmented semi-honest adversary emulates the malicious adversary A.

The second significant part of the proof is essentially showing that the protocols generated by Construction 3.2.7 have the property that their execution in the augmented semi-honest model can be emulated in the ideal model of Definition 3.1.2. Actually, there are two minor problems with the above statement:

1. The statement is not quite true. Actually, the protocols generated by Construction 3.2.7 were not full-specified. What was not specified and is crucial here is the order in which messages are sent in Step 3 (i.e., the step in which output bits are recovered).

   To fit our goals, we now further specify Step 3 instructing that Party 1 is the last to send the shares he holds in an output wire of the circuit to the party associated with this wire. Given this specification the above claim can be proven analogously to the proof of Proposition 2.3.16. Recall that they key property used in the proof is the fact that the execution view under this protocol can be simulated in two stages; the first stage depends only on the party's input (here it is the party's' inputs), whereas the second only produces the message corresponding to Step 3. This allows the adversary in the ideal model of Definition 3.1.2 to emulate real executions in the augmented semi-honest model.

2. The statement does not quite suffice. As in the two-party case, we are not interested in the protocols generated by Construction 3.2.7 but rather in protocols generated as follows:

   (a) First, the private computation of an arbitrary functionality is reduced to the private computation of a deterministic functionality, using Eq. (3.11).

   (b) Next, Construction 3.2.7 is applied to the resulting circuit, giving an oracle-aided protocol.

   (c) Then, the oracle was implemented using Constructions 3.2.4 and 2.2.7.

   (d) Finally, the resulting protocol is precompiled as in the proof of Proposition 3.3.2.

However, as observed in the proof of Proposition 2.3.16, none of these pre/post-processing effects the two-stage simulation property. Thus, the statement above does hold also for the protocols produced in the four-step process described above.

Let Π denote a protocol, generated by the above four-step process, for privately computing a given functionality f. Combining the above two parts, we conclude that when feeding Π to the compiler of Construction 3.3.11, the result is a protocol Π' so that executions of Π' in the (real) first malicious model can be emulated in the ideal model of Definition 3.1.2. Thus, Π' securely computes f in the first malicious model.

We are almost done. The only problem is that Π' operates in the single broadcast channel communication model. This problem is resolved by the postcompiler mentioned in Section 3.3.1. Specifically, we implement the broadcast channel over the point-to-point communication model using (authenticated) Byzantine Agreement (cf., Construction 3.3.3). ■

3.3.3 The second complier

We now show how to transform protocols for securely computing some functionality in the *first* malicious model into protocols which securely computing it in the *second* malicious model. We stress that again all communication, both in the input protocol as well as in the one resulting from the compilation, is conducted merely by posting messages on a single broadcast channel.

The current compiler has little to do with anything done in the two-party case. The only similarities are in the technical level; that is, in using secure implementation of the authenticated computation functionality, which in turn amounts to using zero-knowledge proofs. The main novelty is in the use of a new ingredient – Verifiable Secret Sharing (VSS).

Interestingly, we use secure *in the first malicious model* implementations of the authenticated computation functionality (of Eq. (3.16)) and of VSS. It is what we add on top of these implementations which makes the resulting protocol secure *in the second malicious model.* Following is a high-level description of the multi-party protocols generated by the current compiler. Recall that the input to the compiler is a protocol secure in the *first* malicious model, so the random-pad and actions refer to this protocol.[10]

The sharing phase: Each party shares each bit of its input and random-pad, with all the parties so that any strict majority of parties can retrieve the bit. This is done by using a new ingredient – Verifiable Secret Sharing (VSS).

Intuitively, (*minority*) malicious parties are effectively prevented from abort the protocol by the following convention:

- If a party aborts the execution prior to completion of this phase, then the majority players will set its input and random-pad to some default value, and will carry out the execution ("on its behalf").
- If a party aborts the execution after the completion of this phase, then the majority players will reveal its input and random-pad, and will carry out the execution ("on its behalf").

[10] In our application, we will feed the current compiler with a protocol generated by the first compiler. Still the random-pad and actions below refer to the compiled protocol, not the the semi-honest protocol from which it was compiled.

The fact that all communication is over a broadcast channel and the provisions above guarantee that the (honest) majority players will always be in consensus.

Protocol emulation phase: The parties emulate the execution of the original protocol with respect to the input and random-pads shared in the first phase. This will be done using a secure (*in the first malicious model*) implementation of the authenticated-computation functionality of Eq. (3.16).

We start by defining and implementing the only new tool needed – Verifiable Secret Sharing.

3.3.3.1 Verifiable Secret Sharing

Loosely speaking, a Verifiable Secret Sharing scheme is (merely) a secure (in the *first* malicious model) implementation of a secret sharing functionality. Thus, we first define the latter functionality.

Definition 3.3.13 (secret sharing schemes): *Let $t \leq m$ be positive integers. A* t-out-of-m secret sharing scheme *is a pair of algorithms, $G_{m,t}$ and $R_{m,t}$, satisfying the following conditions.*

- (syntax): *The* share-generation algorithm, *$G_{m,t}$, is a* probabilistic *mapping of secret bits to m-sequences of* shares; *that is, $G_{m,t} : \{0,1\} \mapsto (\{0,1\}^*)^m$. The* recovering algorithm, *$R_{m,t}$, maps t-long sequences of pairs in $[m] \times \{0,1\}^*$ into a single bit, where $[m] \stackrel{\text{def}}{=} \{1,...,m\}$.*
- (the recovery condition): *For any $\sigma \in \{0,1\}$, any sequence $(s_1,...,s_m)$ in the range of $G_{m,t}(\sigma)$, and any t-subset $\{i_1,...,i_t\} \subseteq [m]$, it holds that*

$$R_{m,t}((i_1,s_{i_1}),...,(i_t,s_{i_t})) \;=\; \sigma$$

- (the secrecy condition): *For any $(t-1)$-subset $I = \{i_1,...,i_{t-1}\} \subset [m]$, and any $n \in \mathbb{N}$, the distribution of the I-components of $G_{m,t}(\sigma)$ is independent of σ.*

  That is, for $I = \{i_1,...,i_{t-1}\} \subset [m]$, let $g_I(\sigma)$ be defined to equal $((i_1,s_{i_1}),...,(i_{t-1},s_{i_{t-1}}))$, when the value of $G_{m,t}(\sigma)$ is $(s_1,...,s_m)$. Then, we require that for any such I

$$g_I(0) \equiv g_I(1)$$

It is well-known that secret sharing schemes do exists for any value of m and t. However, common presentations neglect some details such as the representation of the field used in the construction. For sake of self-containment, we present a fully specified construction.

Construction 3.3.14 (Shamir's t-out-of-m secret sharing scheme): *Find*[11] *the smallest prime number bigger than m, denoted p, and consider arithmetic over the finite field $\mathrm{GF}(p)$. The share generating algorithm consists of uniformly selecting a degree $t-1$ polynomial over $\mathrm{GF}(p)$ with free term equal σ, and setting the i^{th} share to be the value of this polynomial at i. The recovering algorithm consists of interpolating the unique degree $t-1$ polynomial which matches the given values, and outputting its free term.*

Getting back to our subject matter, we have

[11] By the Fundamental Theorem of Number Theory, $p \leq 2m$. Thus, it can be found by merely (brute-force) factoring all integers between $m+1$ and $2m$.

Definition 3.3.15 (Verifiable Secret Sharing): *A* verifiable secret sharing scheme *with parameters* (m,t) *is an m-party protocol which implements* (i.e., securely computes in the first malicious model) *the share-generation functionality of some t-out-of-m secret sharing scheme. That is, let* $G_{m,t}$ *be a share-generation algorithm of some t-out-of-m secret sharing scheme. Then, the corresponding share-generation functionality which the VSS securely computes is*

$$((\sigma,1^n),1^n,...,1^n) \;\mapsto\; G_{m,t}(\sigma) \tag{3.17}$$

Actually, it will be more convenient to use an augmented notion of Verifiable Secret Sharing. The augmentation supplies each party with auxiliary input which determines the secret σ and allows Party 1 to latter conduct *authenticated computations* depending on this secret. Furthermore, each party is provided with an NP-proof of the validity of its share (relative to public information given to all). From this point on, when we say Verifiable Secret Sharing (or VSS), we mean the notion defined below (rather the the weaker form above).

Definition 3.3.16 (Verifiable Secret Sharing, revised – VSS): *Let* $G_{m,t}$ *be a share-generation algorithm of some t-out-of-m secret sharing scheme, producing shares of length* ℓ. *Let* $\{C_n\}$ *be a bit commitment scheme, and define* $C_n(\sigma_1\cdots\sigma_\ell, r_1\cdots r_\ell) \stackrel{\rm def}{=} C_n(\sigma_1,r_1)\cdots C_n(\sigma_\ell,r_\ell)$. *Consider the corresponding* (augmented) *share-generation functionality*

$$((\sigma,1^n),1^n,...,1^n) \;\mapsto\; ((\overline{s},\overline{\rho}),(s_2,\rho_2,\overline{c}),...,(s_m,\rho_2,\overline{c})) \tag{3.18}$$

$$\text{where } \overline{s} \stackrel{\rm def}{=} (s_1,...,s_m) \leftarrow G_{m,t}(\sigma), \tag{3.19}$$

$$\overline{\rho} = (\rho_1,...,\rho_m) \text{ is uniformly chosen in } \{0,1\}^{m\cdot\ell n}, \tag{3.20}$$

$$\text{and } \overline{c} = (C_n(s_1,\rho_1),...,C_n(s_m,\rho_m)). \tag{3.21}$$

Then any m-party protocol which implements (i.e., securely computes in the first malicious model) Eq. (3.18)–(3.21) *is called a* verifiable secret sharing scheme (VSS) with parameters (m,t).

Observe that each party may demonstrate the validity of its primary share (i.e., the s_i) by revealing the corresponding ρ_i. We shall be particularly interested in VSS schemes with parameters $(m,\lceil m/2\rceil)$ (i.e., $t=\lceil m/2\rceil$). The reason for this is that we assume throughout this section that the malicious parties are in strict minority. Thus, by the secrecy requirement, setting $t\geq m/2$ guarantees that the minority parties are not able to obtain any information about the secret from their shares. On the other hand, by the recovery requirement, setting $t\leq\lceil m/2\rceil$ guarantees that the majority parties are able to efficiently recover the secret from their shares. Thus, in the sequel, whenever we mention VSS without specifying the parameters, we mean the VSS with parameters $(m,\lceil m/2\rceil)$, where m is understood from the context.

Clearly, by Theorem 3.3.12, Verifiable Secret Sharing schemes can be constructed provided that trapdoor permutation exist. Actually, to establish this result we merely need to apply the first compiler to the obvious semi-honest protocol in which Eq. (3.18)–(3.21) is privately computed by merely letting Party 1 invoke the share-generation algorithm $G_{m,t}$ and send the corresponding shares to each of the parties. For sake of subsequent reference we state the result.

Proposition 3.3.17 *Suppose that trapdoor permutation exist. Then, for every* $t\leq m$, *there exists a verifiable secret sharing scheme with parameters* (m,t).

3.3.3.2 The compiler itself

We are now ready to present the compiler. Recall that we are given a multi-party protocol, Π, for the *first* malicious model, and we want to generate an "equivalent" protocol Π' for the *second* malicious model. Also recall that both the given protocol and the one generated operate in a communication model consisting of a single broadcast channel.

Construction 3.3.18 (The second multi-party compiler): *Given an m-party protocol,* Π*, for the* first *malicious model, the compiler produces the following m-party protocol, denoted* Π'*, for the* second *malicious model.*

Inputs: *Party i gets input $x^i \in \{0,1\}^n$.*

Random-pad: *Party i gets (or uniformly selects) a* random-pad*, denoted $r^i \in \{0,1\}^{c(n)}$.*

The sharing phase: *Each party shares each bit of its input and random-pad, with all the parties, using a Verifiable Secret Sharing scheme.*

That is, for $i = 1, ..., m$ and $j = 1, ..., n + c(n)$, Party i invokes a secure implementation of the VSS functionality of Eq. (3.18)–(3.21)*, playing Party 1 with input $(\sigma, 1^n)$, where σ is the $j^{\rm th}$ bit of $x^i r^i$. The other parties play the role of the other parties in* Eq. (3.18)–(3.21) *with input 1^n. (In case the parties supply different values for 1^n, the majority value – supported by the honest parties – is used). Party i obtains output pair, $(\rho_j^i, \overline{\omega}_j^i)$, and each other Party k obtains a pair $(s_{j,i}^k, \overline{c}_j^i)$ so that the $s_{j,i}^k$'s are shares of a secret relative to $G_{m,\lceil m/2 \rceil}$'s random-pad ρ_j^i, and $\overline{c}_j^i$ is a sequence of commitments to the bits of ρ_j^i (using the random strings sequence $\overline{\omega}$).*

Handling Abort: *If a party aborts the execution prior to completion of this phase, then the other parties set its input and random-pad to some default value, and will carry out the execution on its behalf. We stress that since the party's input and random-pad are now fixed and known to all parties, and since the entire execution takes place over a broadcast channel, all subsequent actions of the aborting party are determined. Thus, there is no need to send actual messages on its behalf. Each of the other parties may determine in solitude what these messages are.*

Protocol emulation phase: *The parties emulate the execution of the protocol* Π *with respect to the input and random-pads shared in the first phase. This will be done using a secure* (in the first malicious model) *implementation of the authenticated-computation functionality of* Eq. (3.16).

That is, Party i, which is supposed to send a message in Π*, plays the role of Party 1 in* Eq. (3.16) *and the other parties play the other roles. The inputs α, r, β and the functions h, f, for the functionality of* Eq. (3.16)*, are set as follows:*

- *The string α is set to equal the concatenation of the party's original input and its random-pad, and the string r is set to be the concatenation of the corresponding randomizations obtained by this party when playing the role of Party 1 in the $n + c(n)$ corresponding invocations of the Verifiable Secret Sharing scheme. The value $h(\alpha, r)$ equals the concatenation of the second elements obtained by the other parties in these invocations; that is, the commitment sequences $\overline{c}_j^i$'s. Then*

$$\begin{aligned} \alpha &= (x^i, r^i) \\ r &= ((\rho_1^i, \overline{\omega}_1^i), (\rho_2^i, \overline{\omega}_j^i), ..., (\rho_{n+c(n)}^i, \overline{\omega}_{n+c(n)}^i)) \\ h(\alpha, r) &= (C_n(\rho_1^i, \overline{\omega}_1^i), C_n(\rho_2^i, \overline{\omega}_j^i), ..., C_n(\rho_{n+c(n)}^i, \overline{\omega}_{n+c(n)}^i)), \end{aligned}$$

where $C_n(\sigma_1 \cdots \sigma_\ell, r_1 \cdots r_\ell) \stackrel{\text{def}}{=} C_n(\sigma_1, r_1) \cdots C_n(\sigma_\ell, r_\ell)$.

Note that h indeed satisfies $h(\alpha, r) \neq h(\alpha', r')$ for all $\alpha \neq \alpha'$ and all r, r'.

- *The string β is set to equal the concatenation of all previous messages sent* (over the broadcast channel) *by all other parties.*
- *The function f is set to be the computation which determines the message to be sent in Π. Note that this message is computable in polynomial-time from the party's input* (denoted x^i above)*, its random-pad* (denoted r^i)*, and the previous messages posted so far* (i.e., β).

As in the sharing phase, the inputs with which the other parties are to enter the authenticated-computation functionality can be determined. Thus, in case the parties supply different values for $(h(\alpha, r), \beta)$ the majority value – supported by the honest parties – is used.

Handling Abort: *If a party aborts during the execution of this phase then the majority players will recover its input and random-pad, and carry out the execution on its behalf. We note that the completion of the sharing phase (and the definition of VSS) guarantee that the majority player hold shares which yield the corresponding bits of the input and random-pad of any party. Furthermore, the correct shares are verifiable by each of the other parties, and so reconstruction of the initial secret is efficiently implementable whenever a majority of parties wishes so. Also, by definition of the secure* (in the first malicious model) *implementation of authenticated-computation, it follows that the parties are always in consensus as to whether the emulated sender has aborted. In case it did, each honest party posts all the shares it has of bits (either input of random-pad) of the emulated sender, and recovers using the shares posted by other parties the input and random-pad of the emulated sender. Based on these, all subsequent actions of the aborting party are determined. Thus, as before, there is no need to send actual messages on its behalf. Each of the parties may determine in solitude what these messages are.*

Outputs: *At the end of the emulation phase, each party holds the corresponding output of the party in protocol Π. The party just locally outputs this value.*

We note that in the above we have somewhat modified the definition of VSS and of Eq. (3.16). By the original formulation (following the conventions in Section 2.1), in case the functionality is not defined for some input sequence, the output is defined as a sequence of $\perp$'s. In the compiler above we have adopted an alternative convention by which the input is "corrected" according to some predetermined rule (e.g., majority vote was used above) to an input sequence for which the functionality is defined. One can easily verify that this alternative definition of functionalities can be securely implemented as well (in the semi-honest and first malicious models).

The abort-handling procedure of the protocol-emulation phase is described above as being implemented by having each honerst party reveal the (s_i, ρ_i) pair it has obtained in each VSS, and each party reconstructing the secret bit by first checking the validity of the s_i's against the commitments $C_n(s_i, \rho_i)$, and using the valid s_i's in the reconstruction. An alternative implementation amounts to securely computing the recovery functionality associated with the above VSS; that is,

$$((\overline{s}, \overline{\rho}), (s_2, \rho_2, \overline{c}), ..., (s_m, \rho_2, \overline{c})) \mapsto (\sigma, ..., \sigma) \tag{3.22}$$

where the l.h.s of Eq. (3.22) is in the range of the VSS functionality applied to $((\sigma, 1^n), 1^n, ..., 1^n)$. Clearly, Eq. (3.22) can be securely implemented in the first malicious model, but one may show that the natural implementation is also secure in the *second* malicious model.

Comment: We stress that when one applies the two (multi-party) compilers one after the other, the random-pad to which the second compiler refers is the one of the protocol for the first malicious model (not the one of the original protocol of the semi-honest model). The random-pad of the protocol compiled for the first malicious model includes the coins of the original protocol, the coins generated by the precompiler (i.e., for selecting a public-key instance, and for running the encryption and decryption algorithms), and the coins generated by the first compiler for the input-commit phase and for the implementation of the various functionalities.

Another comment: Applying the two compilers one after the other is indeed wasteful. For example, we enforce proper emulation (via the authenticated-computation functionality) twice; first with respect to the semi-honest protocol, and next with respect to the protocol resulting from the first compiler. Thus, the proper emulation of the action of the semi-honest protocol is enforced twice; first with respect to the random-pad selected in the coin-generation phase of the first compiler, and next with respect to the sharing of it. It follows that more efficient protocols for the second malicious model could be derived by omitting the authenticated-computation protocols generated by the first compiler. Similarly, one can omit the input-commit phase in the first compiler.

3.3.3.3 Analysis of the compiler

Our aim is to establish

Theorem 3.3.19 (Restating the second half of Theorem 3.3.1): *Suppose that trapdoor permutation exist. Then any m-ary functionality can be securely computable in the* second *malicious model* (using only point-to-point communication lines). *Furthermore, security holds even if the adversary can read all communication among honest players.*

As will be shown below, given a protocol as guaranteed by Theorem 3.3.12, the second compiler produces a protocol which securely computes (in the second malicious model) the same functionality. Thus, for any functionality f, the compiler transforms protocols for securely computing f in the first malicious model (in the semi-honest model) into protocols for securely computing f in the second malicious model. This suffices to establish Theorem 3.3.19, yet it does not say what the compiler does when given an arbitrary protocol (i.e., one not provided by Theorem 3.3.12). In order to analyze the action of the second compiler, in general, we introduce the following model which is a hybrid of the semi-honest and the malicious models. We call this new model, the *second-augmented semi-honest* model. Unlike the (first) *augmented semi-honest* model (used in the analysis of the first compiler), the new model allows the dishonest party to select its random-pad arbitrarily, but does not allow it to abort.

Definition 3.3.20 (the second-augmented semi-honest model): *Let Π be a multi-party protocol. A coordinated strategy for parties I is admissible as a* second-augmented semi-honest behavior (w.r.t Π) *if the following holds.*

Entering the execution: *Depending on their initial inputs and in coordination with each other, the parties in I may enter the execution with any input of their choice.*

Selection of random-pad: *Depending on the above and in coordination with each other, the parties in I may set their random-pad arbitrarily.*

Proper message transmission: *In each step of* Π*, depending on its view so far, the designated (by* Π*) party sends a message as instructed by* Π*. We stress that the message is computed as* Π *instructs based on the party's possibly modified input, and its random-pad selected above, possibly not uniformly. That is, in case the party belongs to* I *we refer to its input and random-pad as set above.*

Output: *At the end of the interaction, the parties in* I *produce outputs depending on their entire view of the interaction. We stress that the view consists of their initial inputs, their choice of random-pads, and all messages they received.*

Intuitively, the compiler transforms any protocol Π into a protocol Π' so that executions of Π' in the second malicious model correspond to executions of Π in the second augmented semi-honest model. That is,

Proposition 3.3.21 (general analysis of the second multi-party compiler): *Let* Π' *be the* m*-party protocol produced by the compiler of Construction 3.3.18, when given the protocol* Π*. Then, there exists a polynomial-time computable transformation of of polynomial-size circuit families* A *into polynomial-size circuit families* B *describing admissible behaviors* (w.r.t Π) *in the second-augmented semi-honest model* (of Definition 3.3.20) *so that for every* $I \subset [m]$ *with* $|I| < m/2$

$$\{\text{REAL}_{\Pi,(I,B)}(\overline{x})\}_{\overline{x}} \stackrel{\text{c}}{\equiv} \{\text{REAL}_{\Pi',(I,A)}(\overline{x})\}_{\overline{x}}$$

Proposition 3.3.21 will be applied to protocols which securely compute a functionality in the first malicious model. As we shall see below, for such *specific* protocols, the second augmented semi-honest model (of Definition 3.3.20) can be emulated by the second ideal malicious model (of Definition 3.1.5). Thus, Theorem 3.3.19 will follow. We start by establishing Proposition 3.3.21.

Proof Sketch: Given a circuit, A, representing an adversarial behavior (in the first malicious model), we present a corresponding circuit, B, admissible w.r.t Π for the second augmented semi-honest model. We stress two points. Firstly, whereas A may abort some parties, the adversary B may not do so (as per Definition 3.3.20). Secondly, we may assume that the number of parties controlled by A (and thus by B) is less than $m/2$ (as nothing is required otherwise).

Machine B will use A as well as the ideal-model adversaries (as per Definition 3.1.2) derived from the behavior of A in the various subprotocols invoked by Π'. Furthermore, machine B will also emulate the behavior of the trusted party in these ideal-model emulations (without communicating with any trusted party – there is no trusted party in the augmented semi-honest model). Thus, the following description contains again an implicit special-purpose composition theorem (see discussion in the proof of Proposition 2.3.15).

Entering the execution and selecting a random-pad: B invokes A (on the very input supplied to it), and decides with what input and random-pad to enter the execution of Π. Towards this end, machine B emulates execution of the sharing phase of Π', using A (as subroutine). Machine B supplies A with the messages it expects to see, thus emulating the honest parties in Π', and obtains the messages sent by the parties in I (i.e., those controlled by A).

Specifically, B emulates the executions of the VSS protocol, in attempt to obtain the bits shared by the parties in I. The emulation of each such VSS-execution is done by using the malicious ideal-model adversary derived from (the real malicious adversary) A. We stress that in accordance to the definition of VSS (i.e., security in the first malicious model), the ideal-model adversary derived from A is in the first malicious model, and may abort some parties. Note that (by Definitions 3.1.4 and 3.1.2) this may happen only if the initiator of the VSS is dishonest. In case any of these executions initiated by some party aborts, all input and random-pad bits of this party are set to the default value (as in the corresponding abort-handling of Π'). Details follow.

- In an execution of VSS initiated by an honest party (i.e., in which an honest party plays the role of Party 1 in VSS), machine B obtains the corresponding augmented shares (available to I).[12] Machine B will use an arbitrary value, say 0, as input for the current emulation of the VSS (as the real value is unknown to B). Machine B derives the ideal-model adversary, denoted A', which emulates to the behavior of A – given the history so far – in the corresponding execution of VSS (in Π'). We stress that since the initiating party of the VSS is honest, A' cannot abort any party.

  Invoking the ideal-model adversary A', and emulating both the honest (ideal-model) parties and the trusted party, machine B obtains the outputs of all parties (i.e., and in particular the output of the initiating party). That is, machine B obtains the message that the parties controlled by A' would have sent to the trusted party (i.e., 1^n), emulate the sending of message $(0, 1^n)$ by the initiating party, and emulates the response of the trusted oracle (i.e., uniformly selects $\rho \in \{0,1\}^\ell$ and $\overline{\omega}$, and computes the outputs as in Eq. (3.18)–(3.21)).

  Specifically, when emulating the j^{th} VSS initiated by Party i, machine B generates and records $(\rho_j^i, \overline{\omega}_j^i)$, and concatenates the emulation of the VSS (i.e., the final view of the parties in I as output by A') to the history of the execution of A.

- In an execution of VSS initiated by a party in I (i.e., a dishonest party plays the role of Party 1 in VSS), machine B obtains the corresponding (input or random-pad) bit of the initiator as well as randomization used in the commitment to it. As before, first B derives the ideal-model adversary, denoted A', which corresponds to the behavior of A – given the history so far – in the corresponding execution of the VSS.

  Suppose that we are currently emulating the j^{th} instance of VSS initiating by Party i, and the the j^{th} bit in the initial input/random-pad of Party i is σ. Then, B invokes A' on input $(\sigma, 1^n)$, and emulating both the honest (ideal-model) parties and the trusted party, machine B obtains the outputs of all parties (including the commitment handed to parties not in I). A key point is that machine B has obtained, while emulating the trusted party, the input handed by A' to the trusted party. This bit is recorded as the j^{th} bit of Party i.

  In case the emulated machine did not abort the initiator, machine B records the above bit as well as the randomization used by VSS in committing to it, and concatenates the emulation of the VSS to the history of the execution of A.

  If A aborts Party i in any of the invocation of VSS (initiated by it) then the input and random-pad of Party i are set to the default value (as in Π'). Otherwise, they are defined as the concatenation of the bits of Party i, retrieved as above.

Thus, inputs and random-pads are determined for all parties in I, depending only on their initial inputs. (All this is done before entering the execution of Π.) Furthermore, the view of machine A in the sharing phase of Π' has been emulated, and the randomizations used in the

[12] These will be used in the emulation of future message-transmission steps.

sharing of all values have been recorded by B. (It suffices to record the randomization used by honest parties, and the commitments made by dishonest ones; these will be used in the emulation of the message-transmission steps of Π'.)

Subsequent steps – message transmission: Machine B now enters the execution of Π (with inputs and random-pads for I-parties as determined above). It proceeds in this real execution of Π, along with emulating the corresponding executions of the authenticated-computation functionality of Eq. (3.16) (which are invoked in Π').

In a message-transmission step by an honest party in Π, machine B obtains from this honest party (in the real execution of Π) a message, and emulates an execution of the authenticated-computation protocol resulting in this message as output. In a message-transmission step by dishonest party in Π, machine B computes the message to be sent as instructed by Π, based on the input and random-pad determined above, and the messages obtained so far (in Π). In addition, B emulates an execution of the authenticated-computation protocol resulting in this message as output. The emulation of each execution of the authenticated-computation protocol, which securely computes (in the first malicious model) the functionality Eq. (3.16), is done by using the malicious ideal-model adversary derived from A. The fact that in these emulations machine B also emulates the trusted party allows it to set the outcome of the authenticated-computation protocol to fit the message being delivered. The fact that a (dishonest) party may abort some parties in these emulations of Π' does not effect the real execution of Π (and is merely reflected in the transcript of these emulations). Details follow.

- In a message-transmission step by a honest party in Π, machine B first obtains from this party (in the real execution of Π) a message, denoted `msg`. This completes all that is done in this step w.r.t communication in Π.

  Next, machine B proceeds in emulating the corresponding message-transmission subprotocol of Π'. Firstly, machine B derives the ideal-model adversary, denoted A', which corresponds to the behavior of A – given the history so far – in the corresponding execution of the authenticated-computation protocol (executed by protocol Π'). Invoking the ideal-model adversary A', and emulating both the honest (ideal-model) parties and the trusted party, machine B sets the trusted-party reply to equal `msg`. When emulating the initiator, machine B provides the trusted party with dummy values for the input and random-pad but with correct values for the publically available values (i.e., the previous message posted in the execution of Π').

  The emulation is carried out so to produce output `msg` which does not necessarily equal the output of the authenticated-computation functionality of Eq. (3.16) on the corresponding inputs. However, the machine A' used in the emulation cannot distinguish the two cases (since the inputs which it gets in the two cases – commitments to the values known only to a honest party – are computationally indistinguishable). Finally, B concatenates the emulation of the authenticated-computation protocol to the history of the execution of A. (Note that since the initiator of the authenticated-computation subprotocol is honest, abort is not possible here, by definition of the first ideal model.)

- In a message-transmission step by a dishonest party in Π, machine B first computes the message to be sent according to Π. This message is computed based on the input and random-pad determined (and recorded) above, and the messages received so far (in execution of Π). Denote the resulting message by `msg`. Machine B completes the execution of this step in Π by posting `msg` on the channel.

Next, machine B proceeds in emulating the corresponding message-transmission subprotocol of Π'. Firstly, machine B derives the ideal-model adversary, denoted A'. Invoking A' and emulating both the honest (ideal-model) parties and the trusted party, machine B produces an emulation of the corresponding execution of the authenticated-computation protocol.

By our new convention regarding inputs presented to this protocol, it follows that this emulation either produces the very message `msg` or aborts the sender. In the latter case, we emulate the abort-handling procedure of Π'. In both cases, B concatenates the emulation of the authenticated-computation protocol (and possibly also the abort-handling) to the history of the execution of A.

Note that each message-transmission step is implemented in polynomial-time. Each message posted is computed exactly as instructed by Π. (We stress again that the emulations of aborting in Π' have no effect on the execution of B in Π.)

Output: Machine B just outputs whatever machine A outputs given the execution history composed (emulated) as above.

Clearly, machine B (described above) implements a second-augmented semi-honest behavior with respect to Π. It is left to show that

$$\{\text{REAL}_{\Pi',(I,A)}(\overline{x})\}_{\overline{x}} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi,(I,B)}(\overline{x})\}_{\overline{x}} \tag{3.23}$$

There are two differences between the two ensembles referred to in Eq. (3.23):

1. In the first distribution (i.e., $\text{REAL}_{\Pi',(A,I)}(\overline{x})$), secure (in first malicious model) protocols implementing VSS and authenticated-computation (of Eq. (3.18)–(3.21) and Eq. (3.16), respectively) are executed; whereas in the second distribution (i.e., $\text{REAL}_{\Pi,(B,I)}(\overline{x})$) these executions are emulated using the corresponding ideal-model adversaries.

2. The emulation of Eq. (3.16) in $\text{REAL}_{\Pi,(B,I)}(\overline{x})$ is performed with a potentially wrong input.

However, these differences are computationally indistinguishable, as shown in the analogous part of the proof of Proposition 2.3.15. ■

Proof of Theorem 3.3.19: Given an m-ary functionality f, let Π be an m-party protocol, as guaranteed by Theorem 3.3.12, for securely computing f in the *first* malicious model. (Actually, we need merely a protocol operating in the broadcast channel communication model.) We now apply the compiler of Construction 3.3.18 to Π and derive a protocol Π'. By Proposition 3.3.21, any polynomial-size circuit family A can be efficiently transformed into a polynomial-size circuit family B describing admissible behavior *(w.r.t Π)* in the second-augmented semi-honest model so that for every $I \subset [m]$ with $|I| < m/2$

$$\{\text{REAL}_{\Pi',(I,A)}(\overline{x})\}_{\overline{x}} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi,(I,B)}(\overline{x})\}_{\overline{x}} \tag{3.24}$$

Note that B represents a benign form of adversarial behavior which is certainly allowed by the first malicious model.[13] Thus, by the guarantee regarding Π, the circuit family B can be efficiently

[13] The malicious behavior of B amounts to replacing inputs and selecting a random-pad arbitrarily, rather than uniformly. Otherwise, B follows the protocol Π.

transformed into a polynomial-size circuit family C describing an adversary in the first ideal model so that for every $I \subset [m]$

$$\{\text{REAL}_{\Pi,(I,B)}(\overline{x})\}_{\overline{x}} \stackrel{\text{c}}{\equiv} \{\text{IDEAL}^{(1)}_{f,(I,C)}(\overline{x})\}_{\overline{x}} \tag{3.25}$$

Note that since B does not abort any of the parties (as it operates in the second-augmented semi-honest model), neither does C. Thus, for this C, the first ideal model is equivalent to the second (i.e., $\text{IDEAL}^{(1)}_{f,(I,C)}(\overline{x}) \equiv \text{IDEAL}^{(2)}_{f,(I,C)}(\overline{x})$). Combining all the above, we get (for every $I \subset [m]$ with $|I| < m/2$),

$$\{\text{REAL}_{\Pi',(I,A)}(\overline{x})\}_{\overline{x}} \stackrel{\text{c}}{\equiv} \{\text{IDEAL}^{(2)}_{f,(I,C)}(\overline{x})\}_{\overline{x}} \tag{3.26}$$

We are almost done. The only problem is that Π' operates in the single broadcast channel communication model. This problem is resolved by the postcompiler mentioned in Section 3.3.1. Specifically, we implement the broadcast channel over the point-to-point communication model using (authenticated) Byzantine Agreement (cf., Construction 3.3.3). ■

Chapter 4

Extensions and Notes

This chapter is still in the process of writing, and the following should be considered merely as fragments.

4.1 Reactive systems

The main results of the previous chapters can be extended to reactive systems in which each party interacts with a high-level process (or application). The high-level process supplies each party with a sequence of inputs, one at a time, and expect to receive from the party corresponding outputs. That is, a reactive system goes through (an unbounded number of) iterations of the following type

- Parties are given inputs for the current iteration. We denote the input given to Party i in iteration j by $x_i^{(j)}$.
- Depending on the current inputs, the parties are supposed to compute outputs for the current iteration. That is, the outputs in iteration j are determined by the inputs of the j^{th} iteration (i.e., the $x_i^{(j)}$'s for all i's).

Arguably, a more general reactive system is one in which the outputs of the j^{th} iteration may depend also on previous iterations. In particular, one may require that the outputs of the j^{th} iteration depend on the inputs of the j^{th} iteration, as well as the outputs of all previous iterations. A more flexible and general way of casting a reactive system is to explicitly introduce a global state, and assert that each iteration of the system depends on the global state and the current inputs, while possibly updating the global state. (The global state may include all past inputs and outputs.) Thus, we view reactive systems as iterating the following steps

- Parties are given inputs for the current iteration (i.e., again, in the j^{th} iteration Party i is given input $x_i^{(j)}$). In addition, there is a global state: The global state at the beginning of the j^{th} iteration is denoted $s^{(j)}$.
- Depending on the current inputs and the global state, the parties are supposed to compute outputs for the current iteration as well as update the global state. That is, the outputs in iteration j are determined by the $x_i^{(j)}$'s, for all i's, and $s^{(j)}$. The new global state, $s^{(j+1)}$, is determined similarly (i.e., also based on $x_i^{(j)}$'s and $s^{(j)}$).

But who is to "hold" the global state? As our aim is secure computations, we do not want the global state to be held by any single party. Instead, the global state will be held by all of the parties, in a "secret sharing" manner. Thus, we need to modify the above exposition a little. In each iteration, each party will start with two inputs and end with two outputs: The first input (resp., output) is the outside input (resp., output) of the current iteration, whereas the second input (resp., output) is the party's share in the starting (resp., ending) global state. Thus, we review reactive systems (for the third time), this time as iterating the following

- Parties are given inputs for the current iteration, as well as shares in the current global state. That is, the j^{th} iteration starts when each Party i is given the outside input $x_i^{(j)}$ and holds share in the global state. The latter share is denoted $s_i^{(j)}$.
- Depending on the current inputs and (the shares of) the global state, the parties are supposed to compute outputs for the current iteration as well as the (shares of the) new global state. That is, Party i obtains an output to be handed to the outside as well as an updated share, $s_i^{(j+1)}$, in an updated global state. (We stress that the computation conducted in the j^{th} iteration depends only on the values $(x_i^{(j)}, s_i^{(j)})_{i=1,\ldots,m}$.)

This brings the problem of secure reactive computation quite close to the domain of problems discussed in previous chapters, except that it is not clear what happens if a malicious party corrupts its share in the global state. To overcome the problem we should specify an error correcting mechanism for the shares of the global state. This mechanism should be part of the functionality underlying the computation in each iteration. Details follow.

The actual definition. We start by defining functionalities which capture the second point of view described above. Such functionalities, referred to as *idealized* reactive functionalities captured the mapping of m inputs and one global state into corresponding outputs and a new global state. That is, an idealized reactive m-party functionality is a randomized process mapping $(m+1)$-ary tuples to $(m+1)$-ary tuples, where the last element in each tuple corresponds to the global state. Let f be such a functionality, then the iterative process defined by f should be clear from the above discussion.[1]

In order to define the actual reactive functionality, we specify a randomized sharing process, denoted share, and an *error-correcting* recovery process, denoted $\mathsf{reconst}$. A minimalistic requirement from these processes is that $\mathsf{reconst}(\mathsf{share}(s)) = s$ holds for every s. In general, we will employ a sharing/reconstruction process which has an error correction guarantee corresponding to the one employed by Secret Sharing schemes. Actually, if we employ a sharing process as in a Verifiable Secret Sharing (see Section 3.3.3.1) then the reconstruction process is straightforward (since commitments to each share are public knowledge, as per Definition 3.3.16). For simplicity, we just adopt this convention (i.e., sharing via VSS and reconstructing in the obvious way). Thus, employing a (t,m)-VSS we have for every $(s_1, \ldots, s_m)$ in the range of $\mathsf{share}(\cdot)$ and every $(s'_1, \ldots, s'_m)$ so that $|\{i : s'_i = s_i\}| \geq t$,

$$\mathsf{reconst}(s'_1, \ldots, s'_m) = s \tag{4.1}$$

[1] Starting with a (possibly empty) initial global state, $s^{(1)}$, the iterations are as follows: In the j^{th} iteration, each Party i is handed an input, $x_i^{(j)}$, and hands back the i^{th} element in $\overline{y} \stackrel{\text{def}}{=} f(x_1^{(j)}, \ldots, x_1^{(j)}, s^{(j)})$. The global state in the end of the j^{th} iteration (which serves as the start state for the next iteration; i.e., the state $s^{(j+1)}$) is the last element of $\overline{y}$.

Given a (t, m)-VSS as above, we define for every idealized functionality f, a corresponding reactive m-party functionality, denoted f', with threshold t. Intuitively, f' is an m-ary functionality which takes m pairs, each being an outside input and a share of a global state, and generates m pairs, each being an outside output and a share of a new global state: f' first reconstructs the global state, then applies f to the outside inputs and the global state deriving outputs and a new global state, and finally generates shares for the new global state. That is, for every input-sequence $x_1, ..., x_m$, every global state s, and every $(s'_1, ..., s'_m)$ so that at least t of the s'_i match the corresponding elements of $\mathsf{share}(s)$,

$$\begin{aligned} f'((x_1, s'_1), ..., (x_m, s'_m)) &\stackrel{\text{def}}{=} ((y_1, z_1), ..., (y_m, z_m)) \text{ , where} && (4.2)\\ (y_1, ..., y_m, z) &\leftarrow f(x_1, ..., x_m, \mathsf{reconst}(s'_1, ..., s'_m)) \text{ and} && (4.3)\\ (z_1, ..., z_m) &\leftarrow \mathsf{share}(z) && (4.4) \end{aligned}$$

Interpreting the results. By the results of Chapter 3, the above reactive m-party functionality is securely computable in the two malicious models (as well as in the semi-honest model).[2] Thus, we conclude that any reactive computation can be conducted securely in each of the following three models:

1. A semi-honest model in which any subset may collude (even when the colluding parties are in majority). (Here we use a reactive m-party functionality with threshold m, a corresponding Secret Sharing scheme,[3] and invoke Theorem 3.2.10.)
2. A malicious model in which the honest parties are in majority. (Here we use a reactive m-party functionality with threshold $\lceil m/2 \rceil$, a corresponding VSS, and invoke Theorem 3.3.19.)
3. A malicious model in which the honest parties may be in minority, but abort is not considered a violation of security. (Here we use a reactive m-party functionality with threshold m, a corresponding VSS, and invoke Theorem 3.3.12.)

(We comment that deriving secure reactive systems via an oblivious invocation of the above theorems is somewhat wasteful, since the sharing and reconstruction processes are done in the protocols produced by these theorems and so defining the reactive functionality is doing so will amount to doing sharing/reconstruction twice.)

4.2 Perfect security in the private channels model

As shown in Chapter 3, general secure multi-party computation is achievable provided that trapdoor permutations exist. It can be easily shown that the general results regarding the first malicious model (i.e., Theorem 3.3.12) imply the existence of one-way functions, and so some computational assumption is (currently) necessary in obtaining them. The same holds for the result regarding two-party computation, even in the semi-honest model.[4] However, the results regarding honest majority (i.e., Theorem 3.3.19) do not seem to imply the existence of one-way functions. Thus, the focus of this section is on what can be achieved without making computational assumptions.

[2] Recall that the results for the first malicious model (as well as for the semi-honest model) generalize the results presented in Chapter 2 for the case $m = 2$.

[3] There is no need to use VSS here.

[4] For example, a private implementation of the Oblivious Transfer functionality implies the existence of one-way functions.

Moving away from computational assumptions requires and allows some changes in the model. Firstly, we can no longer say (as said in Section 3.1) that a model in which adversaries can tap the communication lines between honest parties is equivalent to one in which this is not allowed (or considered): We do not know how to emulate private channels on insecure ones without using an initial set-up (such as shared one-time pads per each pair of parties) or computational assumptions. Thus, to allow any non-trivial result we must assume the existence of private channels between pairs of parties. But making such an assumption and refraining from the use tools which provide only computational security, we may aim at a higher level of security – specifically, perfect (or almost-perfect) security. That is, in definitions such as Definitions 3.1.1, 3.1.4 and 3.1.6, we may replace the requirement that the relevant ensembles are computationally indistinguishable by requiring that they are statistically indistinguishable (or even identically distributed). Actually, in the malicious models, one may even allow the adversaries to be computationally unbounded (i.e., be arbitrary circuit families), but still insist that adversaries for the real model be *efficiently* transformed into adversaries for the ideal model.

Main Results: We assume that *honest parties are in majority*, and that the *adversaries cannot tap the communication lines* between the honest parties (i.e., "private channels"). We make no computational assumptions and our notions of privacy and security are information theoretic (as explained above). The main results are

1. Any m-party functionality can be privately computed in the semi-honest model (provided that more than $\frac{1}{2}m$ parties are honest).
2. Any m-party functionality can be securely computed (in the second malicious model), provided that more than $\frac{2}{3}m$ parties are honest.

Postulating also the existence of a broadcast channel (on top of the private point-to-point channels), one can show that any m-party functionality can be securely computed (in the second malicious model), provided that more than $\frac{1}{2}m$ parties are honest [56]. We note that the extra assumption is necessary since the broadcast functionality cannot be securely computed in a malicious model where $m/3$ parties are faulty (and computationally unbounded).

Additional Results: We comment that few functions can be privately computed (in the private channel model) also when the honest parties are not in majority. In case of Boolean function these are exactly the m-ary functions which can be written as the XOR of m predicates, each applied to a single argument [23].

4.3 Other models

Author's Note: *Write about* mobile adversaries, *and adaptive security*

Author's Note: *Other settings include asynchronous, incoercible.*

4.4 Other concerns

Author's Note: *relative fairness in case of dishonest majority...*

4.5 Bibliographic Notes

Main sources. The main results presented in this manuscript are due to Yao [66] and Goldreich, Micali and Wigderson [40, 41], treating two-party and multi-party, respectively. The chronological order is as follows. In the first paper by Goldreich et. al. [40], it was shown how to construct zero-knowledge proofs for any NP-assertion. The conference version of [40] also provided a rough sketch of the compilation of protocols for the semi-honest model into protocols for the malicious model, by "forcing" malicious parties to behave in a semi-honest manner. Assuming the intractability of factoring, Yao's paper [66] asserts the existence of secure protocols for computing any two-party functionality (i.e., Theorem 2.3.1 above). The details of Yao's construction are taken from oral presentations of his work. Finally, the construction of protocols for the semi-honest model (i.e., Theorem 3.2.10 above) is due to the second paper of Goldreich et. al. [41]. Thus, Theorem 3.3.1 is obtained by combining [40, 41].

Our presentation reverses the actual order in which all these results were discovered: Firstly, our treatment of the two-party case is derived, via some degeneration, from the treatment of the multi-party case. Secondly, we start by treating the semi-honest models, and only next compile protocols for this protocol into protocols for the ("full-fledged") malicious models. We note that our treatment is essentially symmetric, whereas Yao's original treatment of the two-party case [66] is asymmetric (with respect to the two parties). The latter asymmetry has its own merits as demonstrated in [53].

In constructing protocols for the semi-honest models, we follows the framework of Goldreich, Micali and Wigderson [41], while adapting important simplifications due to Haber and Micali (priv. comm., 1986) and Goldreich and Vainish [43]. In particular, Haber and Micali suggested to consider arithmetic circuits over GF(2) rather than the (awkward) straight-line programs over permutation groups considered in [41].[5] The reduction of the private computation of the (multi-party) multiplication gate emulation to OT_1^4 is due to [43]; in [41] the former was "implemented" by invoking Yao's general secure two-party computation result.[6]

In presenting the "semi-honest to malicious" compilers (or the paradigm of "forcing" semi-honest behavior), we follow the outline provided in [40, FOCS Ver., Sec. 4] and [41, Sec. 5]. The fundamental role of zero-knowledge proofs for any NP-assertion, coin-tossing-into-the-well, and verifiable secret sharing in these compilers has been noted in both sources.[7] Otherwise, both sources are highly terse regarding these compilers and their analysis. Most of the current text is devoted to filling up the missing details.

Tools. A variety of cryptographic tools is used in establishing the main results of this manuscript. Firstly, we mention the prominent role of Oblivious Transfer in the protocols developed for the semi-honest model.[8] An Oblivious Transfer was first suggested by Rabin [55], but our actual definition and implementation follow the ideas of Even, Goldreich and Lempel [31] (as further developed in the proceedings version of [40]).

Several ingredients play a major role in the compilation of protocols secure in the semi-honest model into generally secure protocols (for the malicious models). These include *zero-knowledge* (ZK)

[5] The reason that this strange computation model was used in [41] has to do with preliminary stages of their research. In general, too little thought was put into the writing of [41], and this specific technical oversight is symptomatic.

[6] Indeed, this brute-force solution of [41] is also indicative of the little thought put into the writing of [41].

[7] The fundamental role of (zero-knowledge) proofs-of-knowledge is not mentioned in the above sources, but was known to the authors at the time. Some indication to this fact can be derived from [24].

[8] This is true also for the original two-party solution of Yao [66]. Subsequent results, by Kilian [49] further demonstrate the importance of Oblivious Transfer in this context.

proofs and *proofs-of-knowledge* (POK), *commitment schemes*, *verifiable secret sharing* (VSS), and *secure coin-flipping*.

Commitment – Commitment schemes are implicit in [10] (and later papers such as [40]). It seems that an explicit definition was first given in [51], which shows how to construct such schemes based on and one-way functions. The construction of commitment schemes based on 1-1 one-way functions is folklore (cf., [36]). The latter construction suffices for the current text, which anyhow assumes the existence of trapdoor permutations.

Coin-flipping – The notion of coin-flipping-into-the-well was introduced and implemented by Blum [10]. We follow him in our presentation of the vanilla versions (i.e., Definition 2.3.2 and Construction 2.3.3).

ZK – Zero-knowledge proof systems were introduced by Goldwasser, Micali and Rackoff [47]. The construction of zero-knowledge proofs for any NP-assertion is due to Goldreich, Micali and Wigderson [40]. Such proofs are the key-stone of the "forcing" paradigm, which in turn underlies the construction of the "semi-honest to malicious" compilers.

POK – The concept of a proof-of-knowledge was introduced in [47], and a satisfactory definition was provided in [5]. It is folklore that all known zero-knowledge proofs for NP are actually proofs-of-knowledge of the NP-witness. To simplify the exposition, we have introduced here the notion of *strong* proofs-of-knowledge, and observed that some known zero-knowledge proofs for NP are actually *strong* proofs-of-knowledge of the NP-witness.

VSS – Verifiable Secret Sharing was introduced by Chor, Goldwasser, Micali and Awerbuch [22], as an enhancement of Shamir's notion of secret sharing [61]. A relaxed notion with secrecy threshold far below the recovery threshold was implemented in [22], based on specific computational number theoretic assumptions. VSS with a single (arbitrary) threshold was first implemented in [40] (by a direct application of the "forcing" paradigm).

In addition, we also used *secure public-key encryption schemes*, as defined by Goldwasser and Micali [46] and implemented based on any trapdoor permutation in [46, 64] (see also [11]), and signature schemes as defined in [48] and implemented based on any trapdoor permutation in [6] (see also [48, 52, 59]).

Other settings. The material in Section 4.1 is based on a terse high-level discussion in [41].

The material in Section 4.2 (i.e., perfect security in the private channels model) is based mainly on [8, 21]. In particular, these papers were the first to obtain general secure multi-party computation without making computational assumptions. In fact, an alternative exposition to ours could have been provided by first presenting results for the private channels model (with or without broadcast), and next compiling these results to a standard point-to-point network by using encryption (to emulate private channels) and possibly signatures (to emulate broadcast). We stress that the latter refers only to multi-party computations with honest majority.

> Author's Note: *Credits for section 4.3 include* – mobile adversaries *[54], asynchronous [7], adaptive [4, 17], incoercible [18]. For 4.4, use [3, 45].*

Definitional treatments. Our definitions follow the treatments of [45, 50, 2, 13, 14, 15], and are most similar to those in [13, 14, 15]. From our point of view, which is focused on the constructions (i.e., the protocols and their proof of security), these alternative definitional treatments are quite similar. However, the reader is warned that, from a definitional point of view, [45, 50, 2, 13, 15] offer quite different perspectives (alas all very appealing). Contrary to the opinion of some of the authors of [45, 50, 2, 13, 15], we do not believe that there exists one CORRECT definitional approach to this complex issue of defining secure multi-party computation. The fact that two appealing and yet fundamentally incomparable notions of security were presented for m-party computations, with $m > 2$, should serve as a warning.

Bibliography

[1] W. Alexi, B. Chor, O. Goldreich and C.P. Schnorr. RSA/Rabin Functions: Certain Parts are As Hard As the Whole. *SIAM Journal on Computing*, Vol. 17, April 1988, pages 194–209.

[2] D. Beaver. Foundations of Secure Interactive Computing. In *Crypto91*, Springer-Verlag Lecture Notes in Computer Science (Vol. 576), pages 377–391.

[3] D. Beaver and S. Goldwasser. Multiparty computation with faulty majority. In *Crypto89*, Springer-Verlag Lecture Notes in Computer Science (Vol. 435), pages 589–590.

[4] D. Beaver and S. Haber. Cryptographic Protocols Provably secure Against Dynamic Adversaries.

[5] M. Bellare and O. Goldreich. On Defining Proofs of Knowledge. In *Crypto92*, Springer-Verlag Lecture Notes in Computer Science (Vol. 740), pages 390–420.

[6] M. Bellare and S. Micali. How to Sign Given Any Trapdoor Function. *Journal of the ACM*, Vol. 39, pages 214–233, 1992.

[7] M. Ben-Or, R. Canetti and O. Goldreich. Asynchronous Secure Computation. In *25th ACM Symposium on the Theory of Computing*, pages 52–61, 1993.

[8] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *20th ACM Symposium on the Theory of Computing*, pages 1–10, 1988.

[9] M. Blum. How to Exchange Secret Keys. *ACM Trans. Comput. Sys.*, Vol. 1, pages 175–193, 1983.

[10] M. Blum. Coin Flipping by Phone. *IEEE Spring COMPCOM*, pages 133–137, February 1982. See also *SIGACT News*, Vol. 15, No. 1, 1983.

[11] M. Blum and S. Goldwasser. An Efficient Probabilistic Public-Key Encryption Scheme which Hides all Partial Information. In *Crypto84*, Lecture Notes in Computer Science (Vol. 196) Springer-Verlag, pages 289–302.

[12] M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computing*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd IEEE Symposium on Foundations of Computer Science*, 1982.

[13] R. Canetti. *Studies in Secure Multi-Party Computation and Applications.* Ph.D. Thesis, Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, June 1995.
Available from from `http://theory.lcs.mit.edu/~tcryptol /BOOKS/ran-phd.html`.

[14] R. Canetti. Modular composition of secure multi-party protocols. Unpublished manuscript, 1997.

[15] R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. Record 98-18 of the *Theory of Cryptography Library*, URL `http://theory.lcs.mit.edu/~tcryptol`. June 1998.

[16] R. Canetti, C. Dwork, M. Naor and R. Ostrovsky. Deniable Encryption. In *Crypto97*, Springer Lecture Notes in Computer Science (Vol. 1294), pages 90–104.

[17] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Multi-party Computation. In *28th ACM Symposium on the Theory of Computing*, pages 639–648, 1996.

[18] R. Canetti and R. Gennaro. Incoercible Multiparty Computation. In *37th IEEE Symposium on Foundations of Computer Science*, pages 504–513, 1996.

[19] R. Canetti, S. Halevi and A. Herzberg. How to Maintain Authenticated Communication in the Presence of Break-Ins. In *16th ACM Symposium on Principles of Distributed Computing*, 1997.

[20] R. Canetti and A. Herzberg. Maintaining Security in the Presence of Transient Faults. In *Crypto94*, Springer-Verlag Lecture Notes in Computer Science (Vol. 839), pages 425–439.

[21] D. Chaum, C. Crépeau and I. Damgård. Multi-party unconditionally Secure Protocols. In *20th ACM Symposium on the Theory of Computing*, pages 11–19, 1988.

[22] B. Chor, S. Goldwasser, S. Micali and B. Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults. In *26th IEEE Symposium on Foundations of Computer Science*, pages 383–395, 1985.

[23] B. Chor and E. Kushilevitz. A Zero-One Law for Boolean Privacy. *SIAM J. on Disc. Math.*, Vol. 4, pages 36–47, 1991.

[24] B. Chor and M.O. Rabin. Achieving independence in logarithmic number of rounds. In *6th ACM Symposium on Principles of Distributed Computing*, pages 260–268, 1987.

[25] R. Cleve. Limits on the Security of Coin Flips when Half the Processors are Faulty. In *18th ACM Symposium on the Theory of Computing*, pages 364–369, 1986.

[26] A. De-Santis, Y. Desmedt, Y. Frankel and M. Yung. How to Share a Function Securely. In *26th ACM Symposium on the Theory of Computing*, pages 522–533, 1994.

[27] Y. Desmedt and Y. Frankel. Threshold Cryptosystems. In *Crypto89*, Springer-Verlag Lecture Notes in Computer Science (Vol. 435), pages 307–315.

[28] W. Diffie, and M.E. Hellman. New Directions in Cryptography. *IEEE Trans. on Info. Theory*, IT-22 (Nov. 1976), pages 644–654.

[29] D. Dolev and A.C. Yao. On the Security of Public-Key Protocols. *IEEE Trans. on Inform. Theory*, Vol. 30, No. 2, pages 198–208, 1983.

[30] S. Even and O. Goldreich. On the Security of Multi-party Ping-Pong Protocols. *24th IEEE Symposium on Foundations of Computer Science*, pages 34–39, 1983.

[31] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *CACM*, Vol. 28, No. 6, 1985, pages 637–647.

[32] U. Feige, A. Fiat and A. Shamir. Zero-Knowledge Proofs of Identity. *Journal of Cryptology*, Vol. 1, 1988, pages 77–94.

[33] P.S. Gemmell. An Introduction to Threshold Cryptography. In *CryptoBytes*, RSA Lab., Vol. 2, No. 3, 1997.

[34] O. Goldreich. *Lecture Notes on Encryption, Signatures and Cryptographic Protocol.* Spring 1989. Available from `http://theory.lcs.mit.edu/~oded/ln89.html`.

[35] O. Goldreich. A Uniform Complexity Treatment of Encryption and Zero-Knowledge. *Journal of Cryptology*, Vol. 6, No. 1, pages 21–53, 1993.

[36] O. Goldreich. *Foundation of Cryptography – Fragments of a Book.* February 1995. Available from `http://theory.lcs.mit.edu/~oded/frag.html`.

[37] O. Goldreich. The Foundations of Modern Cryptograpy. In *Crypto97*, Springer Lecture Notes in Computer Science (Vol. 1294), pages 46–74.

[38] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, Vol. 25, No. 1, February 1996, pages 169–192.

[39] O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st ACM Symposium on the Theory of Computing*, pages 25–32, 1989.

[40] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *Journal of the ACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th IEEE Symposium on Foundations of Computer Science*, 1986.

[41] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th ACM Symposium on the Theory of Computing,* pages 218–229, 1987.

[42] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology*, Vol. 7, No. 1, pages 1–32, 1994.

[43] O. Goldreich and R. Vainish. How to Solve any Protocol Problem – An Efficiency Improvement. In *Crypto87*, Springer Verlag, Lecture Notes in Computer Science (Vol. 293), pages 73–86.

[44] S. Goldwasser. Fault Tolerant Multi Party Computations: Past and Present. In *16th ACM Symposium on Principles of Distributed Computing*, 1997. Also available from `http://www.cs.cornell.edu/Info/People/chandra/podc97/newProgram.html`.

[45] S. Goldwasser and L.A. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *Crypto90*, Springer-Verlag Lecture Notes in Computer Science (Vol. 537), pages 77–93.

[46] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Science*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th ACM Symposium on the Theory of Computing*, 1982.

[47] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th ACM Symposium on the Theory of Computing*, 1985.

[48] S. Goldwasser, S. Micali, and R.L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, April 1988, pages 281–308.

[49] J. Kilian. Basing Cryptography on Oblivious Transfer. In *20th ACM Symposium on the Theory of Computing*, pages 20–31, 1988.

[50] S. Micali and P. Rogaway. Secure Computation. In *Crypto91*, Springer-Verlag Lecture Notes in Computer Science (Vol. 576), pages 392–404.

[51] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, Vol. 4, pages 151–158, 1991. Preliminary version in *Crypto89*, pages 123–132.

[52] M. Naor and M. Yung. Universal One-Way Hash Functions and their Cryptographic Application. *21st ACM Symposium on the Theory of Computing*, 1989, pages 33–43.

[53] R. Ostrovsky, R. Venkatesan and M. Yung, "Secure Commitment Against Powerful Adversary: A Security Primitive based on Average Intractability. In *Proceedings of the 9th Symposium on Theoretical Aspects of Computer Science, STACS92*, pages 439–448.

[54] R. Ostrovsky and M. Yung. How to Withstand Mobile Virus Attacks. In *10th ACM Symposium on Principles of Distributed Computing*, pages 51–59, 1991.

[55] M.O. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[56] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In *21st ACM Symposium on the Theory of Computing*, pages 73–85, 1989.

[57] C. Rackoff and D.R. Simon. Non-Interactive Zero-Knowledge Proof of Knowledge and Chosen Ciphertext Attack. In *Crypto91*, Springer-Verlag Lecture Notes in Computer Science (Vol. 576), pages 433–444.

[58] R. Rivest, A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *CACM*, Vol. 21, Feb. 1978, pages 120–126.

[59] J. Rompel. One-way Functions are Necessary and Sufficient for Secure Signatures. In *22nd ACM Symposium on the Theory of Computing*, 1990, pages 387–394.

[60] C.E. Shannon. Communication Theory of Secrecy Systems. *Bell Sys. Tech. J.*, Vol. 28, pages 656–715, 1949.

[61] A. Shamir. How to Share a Secret. *CACM*, Vol. 22, Nov. 1979, pages 612–613.

[62] A. Shamir, R.L. Rivest, and L. Adleman. Mental Poker. MIT/LCS Report TM-125, 1979.

[63] M. Tompa and H. Woll. Random Self-Reducibility and Zero-Knowledge Interactive Proofs of Possession of Information. University of California (San Diego), Computer Science and Engineering Department, Technical Report Number CS92-244, June 1992. Preliminary version in *28th IEEE Symposium on Foundations of Computer Science*, pages 472–482, 1987.

[64] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[65] A.C. Yao. Protocols for secure computations (extended abstract). In *23rd IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[66] A.C. Yao. How to Generate and Exchange Secrets. In *27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.