How To Simulate It – A Tutorial on the Simulation Proof Technique

Yehuda Lindell

Dept. of Computer Science
Bar-Ilan University, Israel
lindell@biu.ac.il

January 19, 2016

Abstract

One of the most fundamental notions of cryptography is that of *simulation*. It stands behind the concepts of semantic security, zero knowledge, and security for multiparty computation. However, writing a simulator and proving security via the use of simulation is a non-trivial task, and one that many newcomers to the field often find difficult. In this tutorial, we provide a guide to how to write simulators and prove security via the simulation paradigm. Although we have tried to make this tutorial as stand-alone as possible, we assume some familiarity with the notions of secure encryption, zero-knowledge, and secure computation.

Keywords: secure computation, the simulation technique, tutorial

Contents

1 Introduction

What is simulation? Although it means different things in different settings, there is a clear common denominator. Simulation is a way of comparing what happens in the "real world" to what happens in an "ideal world" where the primitive in question is *secure by definition*. For example, the definition of semantic security for encryption compares what can be learned by an adversary who receives a real ciphertext to what can be learned by an adversary who receives nothing. The definition states that an encryption scheme is secure if they can both learn approximately the same amount of information. This is very strange. Clearly, the latter adversary who receives nothing can learn nothing about the plaintext since it receives no information. However, this is exactly the point. Since the adversary who receives nothing can learn nothing by triviality (this is an "ideal world" that is secure by definition), this implies that in the real world, where the adversary receives the ciphertext, nothing is learned as well.

At first, this seems to be a really complicated way of saying something simple. Why not just define encryption to be secure if nothing is learned? The problem is that it's not at all clear how to formalize the notion that "nothing is learned". If we try to say that an adversary who receives a ciphertext cannot output any information about the plaintext, then what happens if the adversary already has information about the plaintext? For example, the adversary may know that it is English text. Of course, this has nothing to do with the security of the scheme since the adversary knew this beforehand and independently of the ciphertext. The simulation-based formulation of security enables us to exactly formalize this. We say that an encryption scheme is secure if the only information derived (or output by the adversary) is that which is based on a priori knowledge. If the adversary receiving no ciphertext is able to output the same information as the adversary receiving the ciphertext, then this is indeed the case.

It is unclear at this point why this is called "simulation"; what we have described is a comparison between two worlds. This will be explained throughout the tutorial (first in Section 3). For now, it suffices to say that security proofs for definitions formulated in this way work by constructing a simulator that resides in the alternative world that is secure by definition, and generates a view for the adversary in the real world that is computationally indistinguishable from its real view. In fact, as we will show, there are three distinct but intertwined tasks that a simulator must fulfill:

1. It must generate a view for the real adversary that is indistinguishable from its real view;
2. It must extract the effective inputs used by the adversary in the execution; and
3. It must make the view generated be consistent with the output that is based on this input.

We will not elaborate on these points here, since it is hard to explain them clearly out of context. However, they will become clear by the end of the tutorial.

Organization. In this tutorial, we will demonstrate the simulation paradigm in a number of different settings, together with explanations about what is required from the simulator and proof. We demonstrate the aforementioned three different tasks of the simulator in simulation-based proofs via a gradual progression. Specifically, in Section 3 we provide some more background to the simulation paradigm and how it expresses itself in the context of encryption. Then, in Section 4, we show how to simulate secure computation protocols for the case of semi-honest adversaries (who follow the protocol specification, but try to learn more than allowed by inspecting the protocol

transcript). We begin with this case since semi-honest simulation is considerably easier than in the malicious case. Next, we demonstrate the three elements of simulation through the following progression. In Section 5 we show how to simulate in the context of zero-knowledge proofs. In this context, the corrupted party (who is the verifier) has no private input nor output. Thus, the simulation consists of the first task only only: *generating a view* that is indistinguishable from the potentially malicious verifier's view in an execution with a real prover. Next, we proceed to secure computation with security in the presence of (static) malicious adversaries. After presenting the definitions in Section 6, we proceed to the problem of secure coin tossing in Section 7. In this task, the parties receive output and the simulator must generate a view that is *consistent with this output*. Thus, an additional element of the simulator's role is added. (In this section, we also demonstrate the hybrid model and the technique of how to write simulation-based proofs in this model.) Then, in Section 8 we consider the oblivious transfer functionality and show how the simulator *extracts the inputs* of the adversary. This completes the three elements of simulation-based proofs. Finally, in Section 9 we show how to simulate in the common reference string model, and in Section 10 we briefly discuss some advanced topics related to simulation: concurrent composition, the random oracle model, and adaptive corruptions.

2 Preliminaries and Notation

For a finite set $S \subseteq \{0,1\}^*$, we write $x \in_R S$ to say that x is distributed uniformly over the set S. We denote by U_n the uniform distribution over the set $\{0,1\}^n$. A function $\mu(\cdot)$ is negligible if for every positive polynomial $p(\cdot)$ and all sufficiently large n's, it holds that $\mu(n) < 1/p(n)$.

Computational indistinguishability. A probability ensemble $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ is an infinite sequence of random variables indexed by $a \in \{0,1\}^*$ and $n \in \mathbb{N}$. In the context of secure computation, the value a will represent the parties' inputs and n will represent the security parameter. Two distribution ensembles $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ and $Y = \{Y(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ are said to be computationally indistinguishable, denoted by $X \stackrel{c}{\equiv} Y$, if for every non-uniform polynomial-time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$ and every $n \in \mathbb{N}$,

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| \leq \mu(n).$$

All parties are assumed to run in time that is polynomial in the security parameter. (Formally, every party considered has a security parameter tape upon which the value 1^n is written. Then the party is polynomial in the input on this tape. We note that this means that a party may not even be able to read its entire input, as would occur in the case where its input is longer than its overall running time.)

Non-uniformity. The above notion of computational indistinguishability is inherently non uniform, and this is not merely because we allow D to be non uniform. In order to see why this is the case, we show what it means if two ensembles are *not* computationally indistinguishable. We first write out the requirement of computational indistinguishability in full (not using the notion "negligible function"). That is, $X \stackrel{c}{\equiv} Y$ if for every non-uniform polynomial-time algorithm D and every polynomial $p(\cdot)$ there exists an $N \in \mathbb{N}$ such that for every $n > N$ and every $a \in \{0,1\}^*$,

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| < \frac{1}{p(n)}.$$

Now, the contradiction of this is that there exists a D and a polynomial $p(\cdot)$ such that for every $N \in \mathbb{N}$ there exists an $n > N$ and an $a \in \{0,1\}^*$ for which

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| \geq \frac{1}{p(n)}.$$

Stated in short, there exists a D and a polynomial $p(\cdot)$ such that for an infinite number of n's there exists an $a \in \{0,1\}^*$ for which

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| \geq \frac{1}{p(n)}.$$

In particular, this means that for every such n there can be a different a. Now, in order to carry out a reduction that breaks some cryptographic primitive or assumption if the ensembles are *not* computationally indistinguishable, it is necessary for the reduction to know the value of a associated with its given n. The value a associated with n must therefore be written on the advice tape of the reduction algorithm, making it inherently non uniform.

Order of quantifiers for computational indistinguishability. We observe that the definition of computational indistinguishability above is *not* the same as saying that for every $a \in \{0,1\}^*$ it holds that $\{X(a,n)\}_{n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \{Y(a,n)\}_{n\in\mathbb{N}}$. In order to see why, observe that this formulation here guarantees that for every a and every non-uniform probabilistic-polynomial time D, there exists a negligible function μ such that for every n, D distinguishes $X(a,n)$ from $Y(a,n)$ with probability at most $\mu(n)$. This means that there can be a different negligible function for every a, and this function can even depend on a. In particular, consider the negligible function μ_a that equals 1 for every $n < 2^{|a|}$ and equals 2^{-n} for every $n \geq 2^{|a|}$, and assume that for every $a \in \{0,1\}^*$ the function μ_a is taken. Such a function meets the definition requirements. However, this notion is too weak to be of use. For example, zero knowledge would become trivial for all languages in $\mathcal{NP}$ since the simulator could output $\perp$ if $n < 2^{|x|}$ where x is the statement being proven, and can just find the witness in the case that $n \geq 2^{|x|}$. This problem does not arise with the actual definition because it requires that there exists a single negligible function for *all* values of $a \in \{0,1\}^*$.

3 The Basic Paradigm – Semantic Security

The birth of complexity-based cryptography (or "provable security") began with the first rigorous definition of the security of encryption [24]. The formulation captures the notion that *nothing* is learned about the plaintext from the ciphertext. As we discussed in the Introduction, this is actually very non-trivial to formalize. Since we have motivated this definition in the Introduction, we proceed directly to present it.

The definition allows the length of the plaintext to depend on the security parameter, and allows for arbitrary distributions over plaintexts (as long as the plaintexts sampled are of polynomial length). The definition also takes into account an arbitrary auxiliary information function h of the plaintext that may be leaked to the adversary through other means (e.g., because the same message x is used for some other purpose as well). The aim of the adversary is to learn some function f of the plaintext, from the ciphertext and the provided auxiliary information. According to the definition, it should be possible to learn the same information from the auxiliary information alone (and from the length of the plaintext), and without the ciphertext.

Definition 3.1 (Def. 5.2.1 in [18]) *A private-key encryption scheme (G, E, D) is* semantically secure *(in the private-key model) if for every non-uniform probabilistic-polynomial time algorithm $\mathcal{A}$ there exists a non-uniform probabilistic-polynomial time algorithm $\mathcal{A}'$ such that for every probability ensemble $\{X_n\}_{n\in\mathbb{N}}$ with $|X_n| \leq \text{poly}(n)$, every pair of polynomially-bounded functions $f, h : \{0,1\}^* \to \{0,1\}^*$, every positive polynomial $p(\cdot)$ and all sufficiently large n:*

$$\Pr_{k\leftarrow G(1^n)}\left[\mathcal{A}(1^n, E_k(X_n), 1^{|X_n|}, h(1^n, X_n)) = f(1^n, X_n)\right]$$
$$< \Pr\left[\mathcal{A}'(1^n, 1^{|X_n|}, h(1^n, X_n)) = f(1^n, X_n)\right] + \frac{1}{p(n)}$$

(The probability in the above terms is taken over X_n as well as over the internal coin tosses of the algorithms G, E and A or A'.)

Observe that the adversary $\mathcal{A}$ is given the ciphertext $E_k(X_n)$ as well as auxiliary information $h(1^n, X_n)$, and attempts to guess the value of $f(1^n, X_n)$. Algorithm $\mathcal{A}'$ also attempts to guess the value of $f(1^n, X_n)$, but is given *only* $h(1^n, X_n)$ and the length of X_n. The security requirement states that $\mathcal{A}'$ can correctly guess $f(1^n, X_n)$ with almost the same probability as $\mathcal{A}$. Intuitively, then, the ciphertext $E_k(X_n)$ does not reveal any information about $f(1^n, X_n)$, for any f, since whatever can be learned by $\mathcal{A}$ (given the ciphertext) can be learned by $\mathcal{A}'$ (without being given the ciphertext).

Semantic security as simulation. Although the definition does not explicitly mention "simulation" or an ideal world, the definition follows this exact paradigm. In the world in which $\mathcal{A}'$ resides, it is given only the auxiliary information and plaintext length, and not the ciphertext. Thus, $\mathcal{A}'$ resides in an ideal world where, trivially, anything that it learns is from the auxiliary information and plaintext length only. The proof that $\mathcal{A}'$ can learn as much as $\mathcal{A}$ can learn is exactly the comparison between the real world and the ideal world, as discussed in the Introduction. [

It is now possible to explain why this ideal/real world comparison is called *simulation*. The reason is that the *proof technique* used to show that a scheme meets a definition formalized in this way is simulation. Let us examine how one would go about proving that an encryption scheme meets Definition 3.1. The main question is how can one construct a machine $\mathcal{A}'$ that outputs $f(1^n, X_n)$ with almost the same probability as $\mathcal{A}$? How can $\mathcal{A}'$ even know what $\mathcal{A}$ does? The answer is that $\mathcal{A}'$ *simulates* an execution of $\mathcal{A}$ and outputs what $\mathcal{A}$ does. If $\mathcal{A}'$ could *perfectly simulate* such an execution – by providing $\mathcal{A}$ with its expected inputs – then $\mathcal{A}'$ would output $f(1^n, X_n)$ with exactly the same probability as $\mathcal{A}$ would. However, clearly $\mathcal{A}'$ cannot do this since it does not receive $E_k(X_n)$ for input. This is solved by having $\mathcal{A}'$ give $\mathcal{A}$ an encryption of garbage instead, as follows:

Simulator $\mathcal{A}'$: Upon input $1^n, 1^{|X_n|}, h = h(1^n, X_n)$, algorithm $\mathcal{A}'$ works as follows

1. $\mathcal{A}'$ runs the key generation algorithm $G(1^n)$ in order to receive k (note that $\mathcal{A}'$ indeed needs to be given 1^n in order to do this).
2. $\mathcal{A}'$ computes $c = E_k\left(0^{|X_n|}\right)$ as an encryption of "garbage" (note that $\mathcal{A}'$ indeed needs to be given $1^{|X_n|}$ in order to do this).
3. $\mathcal{A}'$ runs $\mathcal{A}(1^n, c, 1^{|X_n|}, h)$ and outputs whatever $\mathcal{A}$ outputs.

The simulation that $\mathcal{A}'$ runs is clearly flawed; instead of giving $\mathcal{A}$ an encryption of X_n it gives $\mathcal{A}$ an encryption of zeroes. However, if encryptions are *indistinguishable*, then $\mathcal{A}$ should output $f(1^n, X_n)$ with approximately the same probability when given $E_k(X_n)$ as when given $E_k\left(0^{|X_n|}\right)$. Otherwise, it would be possible to distinguish such encryptions by seeing whether $\mathcal{A}$ succeeds in outputting $f(1^n, X_n)$ or not. Therefore, such a proof proceeds by showing that $\mathcal{A}$ indeed cannot distinguish between two such encryptions. For example, if the encryption works by XORing the plaintext with the output of a pseudorandom generator, then the reduction works by showing that any non-negligible difference between the probability that $\mathcal{A}$ correctly outputs $f(1^n, X_n)$ in the two cases can be converted into a distinguisher that distinguishes the output of the pseudorandom generator from random with non-negligible probability.

This modus operandi is actually typical of all simulation-based proofs. The simulator somehow simulates an execution for the adversary while handing it "garbage" that looks indistinguishable. Then, the proof proceeds by showing that the simulation is "good", or else the given assumption can be broken.

4 Secure Computation – Simulation for Semi-Honest Adversaries

4.1 Background

The model that we consider here is that of two-party computation in the presence of *static semi-honest* adversaries. Such an adversary controls one of the parties (statically, and so at the onset of the computation) and follows the protocol specification exactly. However, it may try to learn more information than allowed by looking at the transcript of messages that it received and its internal state. Note that this is a very weak adversary model; if the adversary does anything not according to specification – even just choosing its random tape in a non-random way – then it may be able to completely break the protocol (and there are actual examples of natural protocols with this property). Nevertheless, a protocol that is secure in the presence of semi-honest adversaries does guarantee that there is no *inadvertent leakage* of information; when the parties involved essentially trust each other but want to make sure that no record of their input is found elsewhere, then this can suffice. Beyond this, protocols that are secure for semi-honest adversaries are often designed as the first step towards achieving stronger notions of security.

We note that it is much easier to define and prove security for semi-honest adversaries than for malicious adversaries, since we know exactly what the adversary will do (it just follows the protocol specification).

4.2 Defining Security for Semi-Honest Adversaries

Two-party computation. A two-party protocol problem is cast by specifying a possibly random process that maps pairs of inputs to pairs of outputs (one for each party). We refer to such a process as a functionality and denote it $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where $f = (f_1, f_2)$. That is, for every pair of inputs $x, y \in \{0,1\}^n$, the output-pair is a random variable $(f_1(x,y), f_2(x,y))$ ranging over pairs of strings. The first party (with input x) wishes to obtain $f_1(x,y)$ and the second party (with input y) wishes to obtain $f_2(x,y)$.

Privacy by simulation. As expected, we wish to formalize the idea that a protocol is secure if whatever can be computed by a party participating in the protocol can be computed based on

its input and output only. This is formalized according to the simulation paradigm by requiring the existence of a simulator who generates the view of a party in the execution. However, since the parties here have input and output, the simulator must be given a party's input and output in order to generate the view. Thus, security here is formalized by saying that a party's view in a protocol execution be simulatable given its *input* and *output*. This formulation implies that the parties learn nothing from the protocol *execution* beyond what they can derive from their input and prescribed output.

One important point to note is that since the parties are semi-honest, it is guaranteed that they use the actual inputs written on their input tapes. This is important since it means that the output is well defined, and not dependent on the adversary. Specifically, for inputs x, y, the output is defined to be $f(x,y)$, and so the simulator can be given this value. As we will see, this is very different in the case of malicious adversaries, for the simple reason that a malicious adversary can ignore the input written on the input tape and can take any other input. (This is similar to the fact that a malicious verifier in zero knowledge can ignore its random tape and use internal hardcoded randomness instead.)

Definition of security. We begin with the following notation:

- Let $f = (f_1, f_2)$ be a probabilistic polynomial-time functionality and let π be a two-party protocol for computing f. (Throughout, whenever we consider a functionality, we always assume that it is polynomially-time computable.)
- The view of the ith party ($i \in \{1,2\}$) during an execution of π on (x,y) and security parameter n is denoted by $\mathsf{view}_i^\pi(x,y,n)$ and equals $(w, r^i; m_1^i, ..., m_t^i)$, where $w \in \{x,y\}$ (its input depending on the value of i), r^i equals the contents of the ith party's *internal* random tape, and m_j^i represents the jth message that it received.
- The output of the ith party during an execution of π on (x,y) and security parameter n is denoted by $\mathsf{output}_i^\pi(x,y,n)$ and can be computed from its own view of the execution. We denote the joint output of both parties by $\mathsf{output}^\pi(x,y,n) = (\mathsf{output}_1^\pi(x,y,n), \mathsf{output}_2^\pi(x,y,n))$.

Definition 4.1 *Let $f = (f_1, f_2)$ be a functionality. We say that π* securely computes *f* in the presence of static semi-honest adversaries *if there exist probabilistic polynomial-time algorithms $\mathcal{S}_1$ and $\mathcal{S}_2$ such that*

$$\begin{aligned} \left\{(\mathcal{S}_1(1^n, x, f_1(x,y)), f(x,y))\right\}_{x,y,n} &\stackrel{c}{\equiv} \left\{(\mathsf{view}_1^\pi(x,y,n), \mathsf{output}^\pi(x,y,n))\right\}_{x,y,n}, \text{ and} \\ \left\{(\mathcal{S}_2(1^n, y, f_2(x,y)), f(x,y))\right\}_{x,y,n} &\stackrel{c}{\equiv} \left\{(\mathsf{view}_2^\pi(x,y,n), \mathsf{output}^\pi(x,y,n))\right\}_{x,y,n}, \end{aligned}$$

where $x, y \in \{0,1\}^*$ such that $|x| = |y|$, and $n \in \mathbb{N}$.

Observe that according to the definition, it is not enough for the simulator S_i to generate a string indistinguishable from $\mathrm{view}_i^\pi(x,y)$. Rather, the *joint distribution* of the simulator's output and the functionality output $f(x,y) = (f_1(x,y), f_2(x,y))$ must be indistinguishable from $(\mathsf{view}_i^\pi(x,y), \mathsf{output}^\pi(x,y))$. This is necessary for probabilistic functionalities. In particular, consider the case that the parties wish to securely compute some randomized functionality $f(x,y)$, where the parties receive different output. For example, let x and y be lists of data elements, and let f be a functionality that outputs an *independent* random sample of $x \cup y$ of some predetermined

size to each party. Now, consider a protocol that securely outputs the *same* random sample to both parties (and where each party's view can be simulated). Clearly, this protocol should *not* be secure. In particular, party P_1 should have no information about the sample received by P_2, and vice versa. Now, consider a simpler definition of security which compares the distribution generated by the simulator only to the view of the adversary (and not the joint distribution). Specifically, the definition requires that:

$$\begin{aligned}\left\{\mathcal{S}_1(1^n, x, f_1(x,y))\right\}_{x,y,n} &\stackrel{c}{\equiv} \left\{\mathsf{view}_1^\pi(x,y,n)\right\}_{x,y,n}, \text{ and}\\ \left\{\mathcal{S}_2(1^n, y, f_2(x,y))\right\}_{x,y,n} &\stackrel{c}{\equiv} \left\{\mathsf{view}_2^\pi(x,y,n)\right\}_{x,y,n}.\end{aligned}$$

It is not difficult to see that the aforementioned protocol that securely computes the same output to both *is* secure under this definition. This is due to the fact that each party's view consists of a random sample of $x \cup y$, as required, and this view can be simulated. The requirement that each sample be independent cannot be expressed by looking at each output separately. This therefore demonstrates that the definition is not satisfactory (since a clearly insecure protocol is "secure by definition"). For this reason, Definition 4.1 is formulated by looking at the *joint distribution*.

A simpler formulation for deterministic functionalities. In the case where the functionality f is deterministic, the aforementioned simpler definition can be used (along with an additional correctness requirement) since the problem described above does not arise. We first present the definition, and then explain why it suffices.

The definition has two requirements **(a)** $\mathsf{correctness}$, meaning that the output of the parties is correct, and **(b)** $\mathsf{privacy}$, meaning that the view of each party can be (separately) simulated. Formally, correctness is the requirement that

$$\left\{\mathsf{output}^\pi(x,y,n)\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{f(x,y)\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}}$$

and privacy is the requirement that there exist probabilistic-polynomial time S_1 and S_2 such that

$$\left\{\mathcal{S}_1(1^n, x, f_1(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\mathsf{view}_1^\pi(x,y,n)\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}}, \tag{4.1}$$

$$\left\{\mathcal{S}_2(1^n, y, f_2(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\mathsf{view}_2^\pi(x,y,n)\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}}. \tag{4.2}$$

For the case of deterministic functionalities f, any protocol that meets the correctness and privacy requirements is secure by Definition 4.1. In order to see this, observe that the distinguisher is given the indices x, y of the ensemble and so can compute $f(x,y)$ by itself. Thus,

$$\left\{\mathcal{S}_1(1^n, x, f_1(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{\mathsf{view}_1^\pi(x,y,n)\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}}. \tag{4.3}$$

implies that

$$\left\{(\mathcal{S}_1(1^n, x, f_1(x,y)), f(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{(\mathsf{view}_1^\pi(x,y,n), f(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}}. \tag{4.4}$$

In addition, the correctness requirement guarantees that $\mathsf{output}^\pi(x,y,n)$ is computationally indistinguishable from $f(x,y)$, implying that

$$\left\{(\mathsf{view}_1^\pi(x,y,n), f(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{(\mathsf{view}_1^\pi(x,y,n), \mathsf{output}^\pi(x,y,n))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}}. \tag{4.5}$$

Combining Equations (4.4) and (4.5), we have that

$$\left\{(\mathcal{S}_1(1^n, x, f_1(x,y)), f(x,y))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}} \stackrel{c}{\equiv} \left\{(\mathsf{view}_1^\pi(x,y,n), \mathsf{output}^\pi(x,y,n))\right\}_{x,y\in\{0,1\}^*;n\in\mathbb{N}},$$

and so the protocol meets Definition 4.1. This argument works for deterministic functionalities, but does not work for probabilistic ones. The reason is that Eq. (4.4) needs to be read as the *same* sample of $f(x,y) = (f_1(x,y), f_2(x,y))$ given to $\mathcal{S}_1$ and appearing in the random variable next to it in the ensemble. However, when we say that the distinguisher can compute $f(x,y)$ by itself, it is *not* true that it can sample $f(x,y)$ so that $f_1(x,y)$ is the same input given to the simulator. This problem does not arise for deterministic functionalities, since $f(x,y)$ is a single well-defined value. Thus, the claim that Eq. (4.3) implies Eq. (4.4) holds only for deterministic functionalities. See [18, Section 7.2.2] for more discussion on these definitions.

The fact that Definition 4.1 implies privacy and correctness is immediate. Thus, for deterministic functionalities, these formulations are equivalent.

Triviality for semi-honest adversaries. We remark that many problems become trivial in the case of semi-honest adversaries. For example, zero knowledge is trivial since the "prover" can just say this is correct. Since all parties are semi-honest, including the prover, this guarantees that the statement is indeed correct. Another example is commitments: in order to "commit" to a value x, the committer can simply store it locally without sending anything. Then, in order to "decommit", the committer can just send the value. This protocol is perfectly hiding. In addition, it is perfectly binding since a semi-honest adversary follows the specification and so will always send the correct value. Finally, if a number of parties wish to toss an unbiased coin, then one of them can simply locally toss a coin and send the result to all others. Since the party tossing the coin is semi-honest, this guarantees that the coin is unbiased. Having said this, we stress that standard secure computation tasks – where multiple parties with inputs wish to compute a joint function of their inputs – are certainly not trivial.

Auxiliary information. In Section 3, and in the definition of security for malicious adversaries in Section 6, auxiliary information is explicitly provided to the adversary. In contrast, here it appears that there is no auxiliary information. However, auxiliary input is implicit in the definition since computational indistinguishability with respect to *non-uniform* adversaries is required. Thus, the distinguisher *is* given auxiliary input. Note that there is no need to provide any auxiliary information to the adversary running the protocol since it is semi-honest and thus follows the exact same instructions irrespective of any auxiliary input.

4.3 Oblivious Transfer for Semi-Honest Adversaries

In this section, we consider a standard two-party functionality, where both parties have private inputs and wish to compute an output. We will show how to securely compute the bit oblivious transfer functionality, defined by $f((b_0,b_1),\sigma) = (\lambda, b_\sigma)$, where $b_0, b_1, \sigma \in \{0,1\}$ [36, 16]. Stated in words, P_1 has a pair of input bits (b_0,b_1) and P_2 has a choice bit σ. The function is such that P_1 receives no output, and in particular learns nothing about σ. In contrast, P_2 receives the bit of its choice b_σ and learns nothing about the other bit $b_{1-\sigma}$. This is called "oblivious transfer" since the first party has two inputs and sends exactly one of the inputs to the receiver, according to the receiver's choice, without knowing which is sent. We present the protocol of [16] in Protocol 4.2, which relies on *enhanced trapdoor permutations*.

Background – enhanced trapdoor permutations [18, Appendix C.1]. Informally, enhanced trapdoor permutation have the property that it is possible to sample from the range, so that given the coins used for sampling it is still hard to invert the value. Formally, a collection of trapdoor permutations is a collection of functions $\{f_\alpha\}_\alpha$ accompanied by four probabilistic-polynomial time algorithms I, S, F, F^{-1} such that:

1. $I(1^n)$ selects a random n-bit index α of a permutation f_α along with a corresponding trapdoor τ. Denote by $I_1(1^n)$ the α-part of the output.
2. $S(\alpha)$ samples an (almost uniform) element in the domain (equivalently, the range) of f_α. We denote by $S(\alpha; r)$ the output of $S(\alpha)$ with random tape r; for simplicity we assume that $r \in \{0,1\}^n$.
3. $F(\alpha, x) = f_\alpha(x)$, for α in the range of I_1 and x in the range of $S(\alpha)$
4. $F^{-1}(\tau, y) = f_\alpha^{-1}(y)$ for y in the range of f_α and (α, τ) in the range of I

Then, the family is a collection of enhanced trapdoor permutations if for every non-uniform probabilistic-polynomial time adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,

$$\Pr\left[\mathcal{A}(1^n, \alpha, r) = f_\alpha^{-1}(S(\alpha; r))\right] \leq \mu(n)$$

where $\alpha \leftarrow I_1(1^n)$ and $r \in_R \{0,1\}^n$ is random. Observe that given α and r, $\mathcal{A}$ can compute $y = S(\alpha; r)$. Thus, $\mathcal{A}$'s task is to invert y, when it is also given the random coins used by S to sample y. See [18, Appendix C.1] for more discussion on the definition and for constructions of enhanced trapdoor permutations.

We will also refer to a hard-core predicate B of a family of enhanced trapdoor permutations [17, Section 2.5]. We say that B is a hard-core predicate of (I, S, F, F^{-1}) if for every non-uniform probabilistic-polynomial time adversary $\mathcal{A}$ there exists a negligible function μ such that for every n,

$$\Pr\left[\mathcal{A}(1^n, \alpha, r) = B\left(\alpha, f_\alpha^{-1}(S(\alpha; r))\right)\right] \leq \frac{1}{2} + \mu(n).$$

The protocol idea. The idea behind the protocol is that P_1 chooses an enhanced trapdoor permutation, and sends the permutation description (without the trapdoor) to P_2. Then, P_2 samples two elements y_0, y_1 where it knows the preimage of y_σ but does *not* know the preimage of $y_{1-\sigma}$. Party P_2 sends y_0, y_1 to P_1, who inverts them both using the trapdoor, and sends b_0 masked by the hard-core bit of $f^{-1}(y_0)$, and b_1 masked by the hard-core bit of $f^{-1}(y_1)$. Party P_2 is able to obtain b_σ since it knows $f^{-1}(y_\sigma)$, but is unable to obtain $b_{1-\sigma}$ since it does not know $f^{-1}(y_{1-\sigma})$ and so cannot guess its hard-core bit with probability non-negligibly greater than $1/2$. In addition, P_1 sees only y_0, y_1 which are identically distributed (even though P_2 generates them differently), and so learns nothing about P_2's bit σ. See Protocol 4.2 for the protocol description.

We prove the following theorem:

Theorem 4.3 *Assume that (I, S, F, F^{-1}) constitutes a family of enhanced trapdoor permutations with a hard-core predicate B. Then, Protocol 4.2 securely computes the functionality $f((b_0, b_1), \sigma) = (\lambda, b_\sigma)$ in the presence of static semi-honest adversaries.*

PROTOCOL 4.2 (Oblivious Transfer [16])

- **Inputs:** P_1 has $b_0, b_1 \in \{0,1\}$ and P_2 has $\sigma \in \{0,1\}$. (Both parties have (I, S, F, F^{-1}) defining a collection of enhanced trapdoor permutations and a hard-core predicate B.)
- **The protocol:**
  1. P_1 runs $I(1^n)$ to obtain a permutation-trapdoor pair (α, τ). P_1 sends α to P_2.
  2. P_2 runs $S(\alpha)$ twice; denote the first value obtained by x_σ and the second by $y_{1-\sigma}$. Then, P_2 computes $y_\sigma = F(\alpha, x_\sigma) = f_\alpha(x_\sigma)$, and sends y_0, y_1 to P_1.
  3. P_1 uses the trapdoor τ and computes $x_0 = F^{-1}(\alpha, y_0) = f_\alpha^{-1}(y_0)$ and $x_1 = F^{-1}(\alpha, y_1) = f_\alpha^{-1}(y_1)$. Then, it computes $\beta_0 = B(\alpha, x_0) \oplus b_0$ and $\beta_1 = B(\alpha, x_1) \oplus b_1$, where B is a hard-core predicate of f. Finally, P_1 sends (β_0, β_1) to P_2.
  4. P_2 computes $b_\sigma = B(\alpha, x_\sigma) \oplus \beta_\sigma$ and outputs the result.

Proof: Since this is the first proof in this tutorial, we prove it in excruciating detail; in later proofs we will not necessarily work through all the fine details. The oblivious transfer functionality is deterministic, and thus it suffices to use the simpler formulation of security. Correctness is immediate, and we therefore proceed to the simulation. We construct a separate simulator for each party ($\mathcal{S}_1$ for P_1's view and $\mathcal{S}_2$ for P_2's view, as in Definition 4.1).

Consider first the case that P_1 is corrupted. Observe that P_1 receives no output. Thus, we merely need to show that a simulator can generate the view of the incoming messages received by P_1. In the protocol, P_1 receives a single message consisting of a pair of values y_0, y_1 in the domain of f_α. Formally, $\mathcal{S}_1$ is given (b_0, b_1) and 1^n and works as follows:

1. $\mathcal{S}_1$ chooses a uniformly distributed random tape r for P_1 (of the length required, which is what is needed to run I).
2. $\mathcal{S}_1$ computes $(\alpha, \tau) \leftarrow I(1^n; r)$, using the r from above.
3. $\mathcal{S}_1$ runs $S(\alpha)$ twice with independent randomness to sample values y_0, y_1.
4. Finally, $\mathcal{S}_1$ outputs $((b_0, b_1), r; (y_0, y_1))$; the pair (y_0, y_1) simulates the incoming message from P_2 to P_1 in the protocol.

Note that $\mathcal{S}_1$ cannot sample y_0, y_1 in the same way as the honest P_2 since it does not know P_2's input σ. Nevertheless, the definition of a collection of trapdoor permutation states that $S(\alpha)$ outputs a value that is almost uniformly distributed in the domain of f_α (and the domain equals the range, since it is a permutation). Thus, it follows that the distribution over $F(\alpha, S(\alpha))$ is statistically close to the distribution over $S(\alpha)$. This implies that

$$\{(F(\alpha, x_0), y_1)\} \stackrel{s}{\equiv} \{(y_0, y_1)\} \stackrel{s}{\equiv} \{(y_0, F(\alpha, x_1))\}$$

where α is in the range of I, and x_0, x_1, y_0, y_1 are all samples of $S(\alpha)$. The view of P_1 includes a pair as above, along with a uniformly generated tape. Note that the pair $(F(\alpha, x_0), y_1)$ is exactly what P_1 sees when P_2 has input $\sigma = 0$, that the pair (y_0, y_1) is the simulator-generated view, and that the pair $(y_0, F(\alpha, x_1))$ is exactly what P_1 sees when P_2 has input $\sigma = 1$. Thus, we conclude that for every $\sigma \in \{0,1\}$,

$$\{\mathcal{S}_1(1^n, (b_0, b_1))\} \stackrel{s}{\equiv} \{\mathsf{view}_1^\pi((b_0, b_1), \sigma)\}$$

as required.

Next, we proceed to the case that P_2 is corrupted, and construct a simulator $\mathcal{S}_2$. In this case, we need to do something very different in the simulation. In particular, we need to construct a view so that the output defined by that view equals the real output of the protocol. (Observe that a party's view includes its input, random tape, and all incoming messages. Thus, by running the protocol instructions on this view, an output is obtained. This output has to be the "correct" one, or the distinguisher can easily see that it is not the view of a real execution.) Recall that $\mathcal{S}_2$ receives P_2's input *and output*, and thus is able to achieve the above. In this protocol, this is achieved by having $\mathcal{S}_2$ set $\beta_\sigma = B(\alpha, x_\sigma) \oplus b_\sigma$, like the real P_1. In contrast, $\mathcal{S}_2$ is unable to compute $\beta_{1-\sigma}$ correctly, since it does not know $b_{1-\sigma}$.

Simulator $\mathcal{S}_2$ receives for input 1^n plus P_2's input and output bits (σ, b_σ). Then:

1. $\mathcal{S}_2$ chooses a uniform random tape for P_2. Since P_2's randomness is for running $S(\alpha)$ twice, we denote the random tape by r_0, r_1.[1]
2. $\mathcal{S}_2$ runs $I(1^n)$ and obtains (α, τ).
3. $\mathcal{S}_2$ computes $x_\sigma = S(\alpha; r_\sigma)$ and $y_{1-\sigma} = S(\alpha; r_{1-\sigma})$, and sets $x_{1-\sigma} = F^{-1}(\tau, y_{1-\sigma})$.
4. $\mathcal{S}_2$ sets $\beta_\sigma = B(\alpha, x_\sigma) \oplus b_\sigma$, where b_σ is P_2's output received by $\mathcal{S}_2$.
5. $\mathcal{S}_2$ sets $\beta_{1-\sigma} = B(\alpha, x_{1-\sigma})$.
6. $\mathcal{S}_2$ outputs $(\sigma, r_0, r_1; \alpha, (\beta_0, \beta_1))$.

First, note that by putting the "σ-value" first, the real view of P_2 in an execution can be written as:

$$\mathsf{view}_2^\pi((b_0, b_1), \sigma) = \big(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus b_{1-\sigma})\big)$$

where $x_0 = S(\alpha; r_0)$ and $x_1 = S(\alpha; r_1)$. In contrast, the output of the simulator written in this way is:

$$\mathcal{S}_2(1^n, \sigma, b_\sigma) = \big(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}))\big)$$

where $x_0 = S(\alpha; r_0)$ and $x_1 = S(\alpha; r_1)$. Thus, these are *identical* when $b_{1-\sigma} = 0$. Formally, when $b_{1-\sigma} = 0$, for every $\sigma, b_\sigma \in \{0,1\}$ and every n:

$$\big\{\mathcal{S}_2(1^n, \sigma, b_\sigma)\big\} \equiv \big\{\mathsf{view}_1^\pi((b_0, b_1), \sigma)\big\}.$$

It therefore remains to show that the view is *indistinguishable* in the case that $b_{1-\sigma} = 1$. The only difference between the two is whether $\beta_{1-\sigma} = B(\alpha, x_{1-\sigma})$ or $\beta_{1-\sigma} = B(\alpha, x_{1-\sigma}) \oplus 1$. Thus, we need to show that for every $\sigma, b_\sigma \in \{0,1\}$,

$$\big\{\big(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}))\big)\big\} \stackrel{\mathrm{c}}{\equiv} \big\{\big(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus 1)\big)\big\}$$

where the distribution on the left is that generated by $\mathcal{S}_2$ and the distribution on the right is the real one when $b_{1-\sigma} = 1$. Assume by contradiction that there exists a non-uniform probabilistic-polynomial time distinguisher D, a polynomial $p(\cdot)$ and an infinite series of tuples (σ, b_σ, n) such that

$$\begin{aligned}&\Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}))) = 1] \\ &\qquad - \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus 1)) = 1] \geq \frac{1}{p(n)}\ .\end{aligned}$$

[1] In almost all cases, the simulation begins by the simulator choosing a uniform random tape for the party.

(Without loss of generality, we assume that for infinitely many n's, D outputs 1 with greater or equal probability when receiving $B(\alpha, x_{1-\sigma})$ than when receiving $B(\alpha, x_{1-\sigma}) \oplus 1$.) We construct a non-uniform probabilistic-polynomial time guessing algorithm $\mathcal{A}$ that uses D to guess the hard-core predicate.

Algorithm $\mathcal{A}$ is given σ, b_σ on its advice tape, and receives $(1^n, \alpha, r)$ for input. $\mathcal{A}$'s aim is to guess $B(\alpha, S(\alpha; r))$. Algorithm $\mathcal{A}$ sets $r_{1-\sigma} = r$ (from its input), chooses a random r_σ, and computes $x_\sigma = S(\alpha; r_\sigma)$ and $\beta_\sigma = B(\alpha, x_\sigma) \oplus b_\sigma$. Finally, $\mathcal{A}$ chooses a random $\beta_{1-\sigma}$, invokes D on input $(\sigma, r_0, r_1; \alpha, (\beta_\sigma, \beta_{1-\sigma}))$ and outputs $\beta_{1-\sigma}$ if D outputs 1, and $1 - \beta_{1-\sigma}$ otherwise. Observe that if $\mathcal{A}$ guesses $\beta_{1-\sigma}$ correctly then it invokes D on $(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma})))$, and otherwise it invokes D on $(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus 1))$. Thus, if D outputs 1, then $\mathcal{A}$ assumes that it guessed $\beta_{1-\sigma}$ correctly (since D outputs 1 with higher probability when given $B(\alpha, x_{1-\sigma})$ than when given $B(\alpha, x_{1-\sigma}) \oplus 1$). Otherwise, it assumes that it guessed $\beta_{1-\sigma}$ incorrectly and so outputs $1 - \beta_{1-\sigma}$. It therefore follows that:

$$
\begin{aligned}
\Pr[\mathcal{A}(1^n, \alpha, r) = B(\alpha, x)] &= \frac{1}{2} \cdot \Pr[\mathcal{A}(1^n, \alpha, r) = B(\alpha, x) \;\mid\; \beta_{1-\sigma} = B(\alpha, x)] \\
&\quad + \frac{1}{2} \cdot \Pr[\mathcal{A}(1^n, \alpha, r) = B(\alpha, x) \;\mid\; \beta_{1-\sigma} \neq B(\alpha, x)] \\
&= \frac{1}{2} \cdot \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}))) = 1] \\
&\quad + \frac{1}{2} \cdot \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus 1)) = 0] \\
&= \frac{1}{2} \cdot \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}))) = 1] \\
&\quad + \frac{1}{2} \cdot (1 - \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus 1)) = 1]) \\
&= \frac{1}{2} + \frac{1}{2} \cdot \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}))) = 1] \\
&\quad - \frac{1}{2} \cdot \Pr[D(\sigma, r_0, r_1; \alpha, (B(\alpha, x_\sigma) \oplus b_\sigma, B(\alpha, x_{1-\sigma}) \oplus 1)) = 1] \\
&\geq \frac{1}{2} + \frac{1}{2p(n)}
\end{aligned}
$$

in contradiction to the assumption that B is a hard-core predicate of f. We conclude that $\mathcal{S}_2$'s output is computationally indistinguishable from the view of P_2 in a real execution. ■

Discussion. We remark that this protocol is a good example of the fact that security in the presence of semi-honest adversaries guarantees nothing if the corrupted party does not behave completely honestly. In particular, if P_2 generates *both* y_0 and y_1 by choosing x_0, x_1 and computing $y_0 = F(\alpha, x_0)$ and $y_1 = F(\alpha, x_1)$, then it will learn both b_0 and b_1. Furthermore, P_1 has no way of detecting this at all.

This concludes our treatment of semi-honest adversaries. As we have seen, proving security for semi-honest adversaries requires constructing a simulator that generates the entire view itself. This view must be a function of the input and output, since the view fully defines the output. Unlike in the case of malicious adversaries who may behave in an arbitrary way, semi-honest adversaries follow the protocol specification exactly. Thus, there is no need to "rewind" them or "interact" with them, in contrast to what we will see in the sequel below.

5 Simulating the View of Malicious Adversaries – Zero Knowledge

In this section we will consider simulation in the context of zero-knowledge proof systems. Unlike what we have seen until now, simulation for zero knowledge considers malicious adversaries (in particular, malicious verifiers) who may behave arbitrarily and not necessarily according to the protocol specification. However, as we have mentioned in the introduction, in zero knowledge there are no private inputs or output. Thus, the simulator needs to generate the view of the verifier in a proof, without the additional complexity of considering inputs and outputs. As we will see below, this can already be challenging.

We will begin by defining zero knowledge and commitments in Sections 5.1 and 5.2, respectively. Then, in Section 5.3, we present a non-constant round zero-knowledge proof for any language in $\mathcal{NP}$. Additional proof techniques are needed to achieve constant-round zero knowledge, as we show in Section 5.4. Finally, we highlight the difference between semi-honest and malicious adversaries by comparing to honest-verifier zero knowledge (which considers semi-honest verifiers) in Section 5.5.

5.1 Defining Zero Knowledge

Notation. Let A be a probabilistic polynomial-time machine. We denote by $A(x, y, r)$ the output of the machine A on input x, auxiliary-input y and random-tape r. In contrast to the rest of this tutorial where the parties are assumed to be polynomial time in a *separate security parameter* n (see Section 2), in this section we set $n = |x|$ and so A runs in time that is polynomial in the length of the statement x. We do this in order to be consistent with the standard definitions of zero knowledge.

Let A and B be interactive machines. We denote by $\mathsf{output}_B(A(x, y, r_A), B(x, z, r_B))$ the output of party B in an interactive execution with party A, on public input x, where A has auxiliary-input y and random-tape r_A, and B has auxiliary input z and random-tape r_B. We will sometimes drop r_A or r_B from this notation, which will mean that the random tape is not fixed but rather chosen at random. For example we denote by $\mathsf{output}_B(A(x, y), B(x, z))$ the random variable $\mathsf{output}_B(A(x, y, U_m), B(x, z, U'_{m'}))$ where m (resp., m') is the number of random bits that A (resp., B) uses on input of size $|x|$.

The definition. Loosely speaking, an interactive proof system for a language L involves a prover P and a verifier V, where upon common input x, the prover P attempts to convince V that $x \in L$. We note that the prover is often given some private auxiliary-input that "helps" it to prove the statement in question to V. Such a proof system has the following two properties:

1. *Completeness:* this states that when honest P and V interact on common input $x \in L$, then V is convinced of the correctness of the statement that $x \in L$ (except with at most negligible probability).

2. *Soundness:* this states that when V interacts with any (cheating) prover P^* on common input $x \notin L$, then V will be convinced *with at most negligible probability.* (Thus V cannot be tricked into accepting a false statement.)

A formal definition of interactive proofs can be found in [17, Section 4.2].

We now recall the definition of zero knowledge [25]. Informally speaking, a proof is zero knowledge if there exists a simulator that can generate the view of the verifier from the statement alone.

We remark that the corrupted verifier may output anything it wishes, including its view. Thus, one may equivalently consider the view of the verifier and its output. For the sake of this tutorial, we will only consider black-box zero knowledge [23, 17], where the simulator receives only oracle access to the verifier. In addition, we will consider only $\mathcal{NP}$ languages. We therefore present this definition only.

Definition 5.1 *Let (P, V) be an interactive proof system for an $\mathcal{NP}$-language L, and let R_L be the associated $\mathcal{NP}$-relation. We say that (P, V) is* black-box computational zero knowledge *if there exists a probabilistic-polynomial time oracle machine $\mathcal{S}$ such that for every non-uniform probabilistic-polynomial time algorithm V^* it holds that:*

$$\left\{\mathsf{output}_{V^*}(P(x,w), V^*(x,z))\right\}_{(x,w)\in R_L, z\in\{0,1\}^*} \stackrel{c}{\equiv} \left\{\mathcal{S}^{V^*(x,z,r,\cdot)}(x)\right\}_{x\in L, z\in\{0,1\}^*}$$

where $V^*(x,z,r,\cdot)$ denotes the next-message function *of the interactive machine V^* when the common input x, auxiliary input z and random-tape r are fixed (i.e., the next message function of V^* receives a message history $\vec{m}$ and outputs $V^*(x,z,r,\vec{m})$).*

In some cases, the simulator (and verifier) are allowed to run in *expected* polynomial-time and not strict polynomial-time. We will relate to this later.

We remark that in our definition above, we fix the random tape of the verifier. With very few exceptions (e.g., the non-black uniform zero-knowledge protocol of [1]), the ability to set the random tape of the adversary does not help. This is due to the fact that the adversary can completely ignore its random tape, and can use a pseudorandom function applied to its history with an internally hardcoded key. Thus, in most cases of simulation, one can just ignore the random tape. Note that if the definition is not black box, then it is necessary to choose a random tape for the adversary. However, in most cases, this can just be chosen to be uniformly distributed of the appropriate length, and then ignored.

5.2 Preliminaries – Commitment Schemes

We will use commitment schemes in a number of places throughout the tutorial. We denote by Com a non-interactive perfectly binding commitment scheme. Let $c = \mathsf{Com}_n(x;r)$ denote a commitment to x using random string r and with security parameter n. We will typically omit the explicit reference to n and will write $c = \mathsf{Com}(x;r)$. Let $\mathsf{Com}(x)$ denote a commitment to x using uniform randomness. Let $\mathsf{decom}(c)$ denote the decommitment value of c; to be specific, if $c = \mathsf{Com}(x;r)$ then $\mathsf{decom}(c) = (x,r)$.

A formal definition of commitment schemes can be found in [17, Section 4.4.1]. Informally, perfect binding is formalized by saying that the sets of all commitments to different values are disjoint; that is, for all $x_1 \neq x_2$ it holds that $C_{x_1} \cap C_{x_2} = \emptyset$ where $C_{x_1} = \{c \mid \exists r : c = \mathsf{Com}(x_1;r)\}$ and $C_{x_2} = \{c \mid \exists r : c = \mathsf{Com}(x_2;r)\}$. Computational hiding can be formalized in multiple ways, and basically states that commitments to different strings are computationally indistinguishable. For bit commitments, this can easily be stated by requiring that $\mathcal{C}_0 \stackrel{c}{\equiv} \mathcal{C}_1$ where $\mathcal{C}_b = \{\mathsf{Com}(b;U_n)\}_{n\in\mathbb{N}}$ is the ensemble of commitments to bit b.

LR-security of commitments. One of the proofs below is made significantly easier by using a definition of security of commitments that is both adaptive and already includes security for multiple

commitments. We present a definition that is based on the LR-oracle definition of encryption [4]. The LR-oracle (Left or Right oracle) definition is formulated by providing the adversary with an oracle that receives two equal-length inputs, and either always returns a commitment to the first (left) input or always returns a commitment to the second (right) input. The task of the adversary is to determine whether it is receiving left or right commitments. This definition is much easier to work with, as we will see below, partly because the hybrid argument relating to multiple commitments is already built in. We first define the oracle as $LR^b_{\mathsf{Com}}(x_0, x_1) = \begin{cases} \mathsf{Com}(x_b) & \text{if } |x_0| = |x_1| \\ \perp & \text{otherwise} \end{cases}$, where $\mathsf{Com}(x)$ denotes a (non-interactive) commitment to x. We define the LR experiment with a non-interactive perfectly-binding commitment scheme Com and an adversary $\mathcal{A}$ who is either given LR^0_{Com} or LR^1_{Com} and attempts to distinguish between these cases. The experiment is as follows:

Experiment $\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n)$**:**

1. Choose a random $b \leftarrow \{0,1\}$.
2. Set $b' \leftarrow \mathcal{A}^{LR^b_{\mathsf{Com}}(\cdot,\cdot)}(1^n)$.
3. Output 1 if and only if $b' = b$.

The following can be proven via a standard hybrid argument:

Theorem 5.2 *If* Com *is a non-interactive perfectly-binding commitment scheme with security for non-uniform adversaries, then for every non-uniform probabilistic-polynomial time adversary* $\mathcal{A}$ *there exists a negligible function* μ *such that*

$$\Pr\left[\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n) = 1\right] \leq \frac{1}{2} + \mu(n).$$

We remark that *non-uniform* security is needed, as we will see below.

5.3 Non-Constant Round Zero Knowledge

Consider the zero-knowledge proofs for $\mathcal{NP}$ of 3-coloring [22] and Hamiltonicity [6]. Both of these protocols work by the prover first sending commitments. Next, the verifier sends a "challenge" asking the prover to open some of the commitments. Finally, the prover sends the appropriate decommitments, and the verifier checks that the results are as expected. In the case of 3-coloring, the prover commits to a random valid coloring, and the verifier asks to open the colors associated with a single edge. In the case of Hamiltonicity, the prover commits to the adjacency matrix of a random permutation of the graph, and the verifier asks to either open the entire graph or to open a simple cycle. In both of these cases, if the prover knows the challenge of the verifier ahead of time, then it can easily prove without knowing the required $\mathcal{NP}$-witness. Let us focus on the 3-coloring case. If the prover does not know a 3-coloring, then it cannot commit to a valid coloring. Thus, there must be at least one edge in the graph which assigns the same color to both endpoints of the edge in the committed coloring by the prover. If the verifier asks to open the colors of this edge, then the prover will be caught cheating. Thus, the prover can cheat with probability at most $1/|E|$ (where E is the set of edges). By repeating the proof $n \cdot |E|$ times (where n is the number of nodes in the graph), we have that the prover can get away with cheating with probability at most $\left(1 - \frac{1}{|E|}\right)^{n \cdot |E|} < e^{-n}$ which is negligible. Thus, this proof is *sound*.

PROTOCOL 5.3 (Zero-Knowledge Proof for 3-Coloring)

- *Common input:* a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
- *Auxiliary input for the prover:* a coloring of the graph $\psi : V \to \{1, 2, 3\}$ such that for every $(v_i, v_j) \in E$ it holds that $\psi(v_i) \neq \psi(v_j)$
- *The proof system:* Repeat the following $n \cdot |E|$ times (using independent randomness each time):
  1. The prover selects a random permutation π over $\{1, 2, 3\}$, defines $\phi(v) = \pi(\psi(v))$ for all $v \in V$, and computes $c_i = \mathsf{Com}(\phi(v_i))$ for all i. The prover sends the verifier the commitments $(c_1, \ldots, c_n)$.
  2. The verifier chooses a random edge $e \in_R E$ and sends e to the prover.
  3. Let $e = (v_i, v_j)$ be the edge received by the prover. The prover sends $\mathsf{decom}(c_i), \mathsf{decom}(c_j)$ to the verifier.
  4. Let $\phi(v_i)$ and $\phi(v_j)$ denote the respective decommitment values from c_i and c_j. The verifier checks that the decommitments are valid, that $\phi(v_i), \phi(v_j) \in \{1, 2, 3\}$, and that $\phi(v_i) \neq \phi(v_j)$. If not, it halts and outputs 0.

  If the checks pass in all iterations, then the verifier outputs 1.

Regarding *zero knowledge*, observe that in each execution a new random coloring of the edges is committed to by the prover, and the verifier only sees the colors of a single edge. Thus, the verifier simply sees two (different) random colors for the endpoints of the edges each time. This clearly reveals nothing about the coloring of the graph. We stress that such an argument is insufficient and we must prove this intuition by constructing a simulator. The idea behind the simulation here is that if the simulator knows the edge to be queried ahead of time, then it can commit to random different colors on the endpoints of that edge and to garbage elsewhere. By the hiding property of the commitment scheme, this will be indistinguishable. As we will see, the simulator will simply repeatedly guess the edge that is to be queried ahead of time until it is correct.

The rewinding technique (with commitments as envelopes). We begin by describing how to construct a simulator when we model the commitments as perfect envelopes that reveal nothing until opened. The key tool for constructing a simulator is that of *rewinding*. Specifically, the simulator invokes the verifier, and guesses a random edge $e = (v_i, v_j) \in_R E$ with the hope that the verifier will query that edge. The simulator then sends the verifier (its oracle in the black-box case) commitments to a coloring whereby v_i and v_j are given two different random colors in $\{1, 2, 3\}$ and zeroes for the rest. If the verifier replies with the edge $e' = e$, then the simulator opens the envelopes for the nodes in e, and the simulation of this iteration is complete. Otherwise, the simulator rewinds the verifier to the beginning of the iteration and tries again, this time choosing a new random edge. This is repeated until $e' = e$ and so the simulator succeeds. (In order to get negligible soundness error, many sequential executions of the protocol are run, and so after it succeeds the simulator proceeds to the next iteration. This essentially means fixing the transcript of incoming messages to this point, and continuing with the residual verifier that is defined by the fixed transcript prefix.) Since the verifier has no way of knowing which edge e the simulator chose (since this fact is hidden inside unopened envelopes), the expected number of repetitions required is $|E|$, and probability

that more than $n \cdot |E|$ repetitions are needed is negligible. Note that the distribution over the view of the verifier in the simulation is identical to its view in a real execution. This is due to the fact that in both a real proof and in a simulation the verifier sees a set of "envelopes" and an opening to two different random colors. The difference between the two is that in a real proof no rewinding took place, in contrast to the simulation. However, this fact is not evident in the verifier's final view, and so they both look the same.

This concept of rewinding is often confusing at first sight. We therefore add two remarks. First, one may wonder how it is possible to "technically" rewind the verifier. In fact, when considering black-box zero knowledge, this is trivial. Specifically, the simulator is given oracle access to the *next message function* $V^*(x, z, r, \cdot)$ of the verifier. This means that it provides a transcript $\vec{m} = (m_1, m_2, \ldots)$ of incoming messages and receives back the next message sent when V^* has input x, auxiliary input z, random tape r and incoming messages $\vec{m}$. Now, rewinding is essentially $\mathcal{S}$ calling its oracle with $(r, (m_1, m_2, m_3))$ and then with $(r, (m_1, m_2, m_3'))$, and so on. It is worthwhile also translating this notion of rewinding into modern computing terms. Virtual machines (VMs) are now very common. Snapshots of a VM can be taken at any time, and it is possible to rewind a VM by simply restoring the snapshot. The VM then continues from exactly the same state as before, and it has no way of knowing that this "rewinding" took place. This is exactly what a simulator does with the verifier.

A second point that is sometimes confusing is why the zero knowledge property, and in particular the existence of a simulator, does not contradict soundness. If the simulator can prove the theorem without knowing the witness (and possibly even if the theorem is not true), then what prevents a cheating prover from doing the same? The answer is that the simulator has additional power that the prover does not have. In our example above, this power is the ability to rewind the verifier; a real prover *cannot* rewind the verifier, in contrast to the simulator. Conceptually, this makes a lot of sense. The motivation behind the simulation paradigm is that whatever the verifier can learn in a real interaction with the prover it can learn *by itself*. The verifier can indeed generate its view by applying the simulator to itself and rewinding, as described above. The prover, who is an external entity to the verifier, cannot do this.

The above analysis and explanation relate to the case that commitments are modeled as ideal envelopes. It is important to stress that this modeling of commitments is an oversimplification that bypasses the main technical difficulties involved when proving that the simulation works. First, it is necessary to show that the view of the verifier is indistinguishable in the simulation and real execution. This requires a reduction to the hiding property of commitments, since the actual distribution is *very* different. In particular, the real prover commits to a valid coloring of the graph, whereas the commitment in the simulation is to zeroes except for the nodes on the opened edge. A second, more subtle, issue is that it is required to prove that the simulation halts successfully within $n \cdot |E|$ attempts, except with negligible probability. In the "envelopes" case, this is immediate. However, when using actual commitments that are just computationally hiding, this needs a proof (perfectly-hiding commitments could be used, but then the protocol would only be computationally sound, and a reduction to the computationally-binding property of the commitment would anyway be needed in order to prove soundness). In order to see the issue that arises here, consider the case of a verifier who can break the commitments. Such a verifier could work as follows: if the committed values constitute a valid coloring then send a random edge; otherwise, if they are all zeroes except for two nodes, then send any edge apart from the one connecting those two nodes. Clearly, when the simulator works with this verifier, it never succeeds. Now, by the hiding property

of commitments, this should not happen. However, it shows that the success of the simulator also depends on the computational hiding of the commitments, and thus a reduction to this property is needed as well. Formally, this can be solved by proving – via a reduction to the hiding property of commitments – that the probability that any given edge is queried by the verifier when it receives valid-prover commitments in the first message is negligibly close to the probability that the edge is queried when it receives (garbage) simulator-generated commitments. Our actual proof will work differently, since we will first show that there exists a hypothetical simulator who receives the correct coloring and generates a distribution identical to a real proof, and then we will show that the actual simulator generates a distribution that is computationally indistinguishable from the hypothetical one. In addition, we prove that the hypothetical simulator halts successfully within $n \cdot |E|$ attempts except with negligible probability. Thus, the fact that the actual simulator generates a distribution that is computationally indistinguishable from the hypothetical one also implies that it halts successfully within $n \cdot |E|$ attempts, except with negligible probability.

On dealing with aborts. At some point in the simulation, it is possible that V^* does not reply with a valid edge. In this case, we have to specify what the simulator should do. In fact, the protocol itself must specify to the honest prover what to do in such a case. One strategy is to state that if the verify returns an illegal value, then the honest prover halts the execution. This certainly works and will be necessary in later cases (e.g., constant-round zero knowledge), where the prover is unable to proceed if the verifier does not respond with a valid value. However, in this specific case, the easiest thing to do is to have the real prover interpret any invalid reply as a default edge. In this case, the simulator will deal with an invalid message in the same way. According to this strategy, there are actually no invalid messages from V^*, and this somewhat simplifies the simulation.

A formal proof of security. We are now ready to prove the zero-knowledge property of the 3-coloring protocol (we do not prove soundness since the focus of this tutorial is on *simulation*).

Theorem 5.4 *Let* Com *be a perfectly-binding commitment scheme with security for non-uniform adversaries. Then, the 3-coloring protocol of [22] is black-box computational zero knowledge.*

Proof: We begin by describing the simulator. $\mathcal{S}$ is given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and oracle access to some probabilistic-polynomial time $V^*(x, z, r, \cdot)$, and works as follows:

1. $\mathcal{S}$ initializes the message history transcript $\vec{m}$ to be the empty string λ.
2. Repeat $n \cdot |E|$ times:
   (a) $\mathcal{S}$ sets $j = 1$.
   (b) $\mathcal{S}$ chooses a random edge $(v_k, v_\ell) \in_R E$ and chooses two random different colors for v_k and v_ℓ. Formally, $\mathcal{S}$ chooses $\phi(v_k) \in_R \{1, 2, 3\}$ and $\phi(v_\ell) \in_R \{1, 2, 3\} \setminus \{\phi(v_k)\}$. For all other $v_i \in V \setminus \{v_k, v_\ell\}$, $\mathcal{S}$ sets $\phi(v_i) = 0$.
   (c) For every $i = 1, \ldots, n$, $\mathcal{S}$ computes $c_i = \mathsf{Com}(\phi(v_i))$.
   (d) $\mathcal{S}$ "sends" the vector $(c_1, \ldots, c_n)$ to V^*. Formally, $\mathcal{S}$ queries $\vec{m}$ concatenated with this vector to its oracle (indeed $\mathcal{S}$ does not interact with V^* and so cannot actually "send" it any message). Let $e \in E$ be the reply back from the oracle.

(e) If $e \neq (v_k, v_\ell)$ then $\mathcal{S}$ sets $j \leftarrow j+1$. If $j = n \cdot |E|$, then $\mathcal{S}$ outputs a fail symbol $\perp$. Else, $\mathcal{S}$ returns to Step 2b and proceeds (i.e., $\mathcal{S}$ tries again for this i). This return to Step 2b is the *rewinding* of V^* by the simulator.

(f) If $e = (v_k, v_\ell)$, then $\mathcal{S}$ completes this iteration by concatenating the commitments $(c_1, \ldots, c_n)$ and $(\mathsf{decom}(c_k), \mathsf{decom}(c_\ell))$ to $\vec{m}$. Formally, $\mathcal{S}$ updates the history string $\vec{m} \leftarrow (\vec{m},\ (c_1, \ldots, c_n),\ (\mathsf{decom}(c_k), \mathsf{decom}(c_\ell)))$.

3. $\mathcal{S}$ outputs whatever V^* outputs on the final transcript $\vec{m}$.

It is clear that $\mathcal{S}$ runs in polynomial-time, since each repetition runs for at most $n \cdot |E|$ iterations, and there are $n \cdot |E|$ repetitions.

In order to prove that $\mathcal{S}$ generates a transcript that is indistinguishable from a real transcript, we need to prove a reduction to the security of the commitment scheme. We begin by constructing an alternative simulator $\mathcal{S}'$ who is given a valid coloring ψ as auxiliary input. We stress that $\mathcal{S}'$ is *not* a valid simulator, since it is given ψ. Rather, it is a thought experiment used in the proof. Now, $\mathcal{S}'$ works in exactly the same way as $\mathcal{S}$ (choosing e at random, rewinding, and so on) except that in every iteration it chooses a random permutation π over $\{1,2,3\}$, sets $\phi(v) = \pi(\psi(v))$, and computes $c_i = \mathsf{Com}(\phi(v_i))$ for *all* i, exactly like the real prover.

We begin by proving that conditioned on $\mathcal{S}'$ not outputting $\perp$, it generates output that is identically distributed to V^*'s output in a real proof. That is, for every V^*, every $(G, \psi) \in R_L$ and every $z \in \{0,1\}^*$,

$$\left\{\mathsf{output}_{V^*}(P(G,\psi), V^*(G,z))\right\} \equiv \left\{{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi) \ \mid \ {\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi) \neq \perp\right\}. \tag{5.1}$$

In order to see this, observe that the distribution over the commitments viewed by V^* is identical to a real proof (since they are commitments to a random permutation of a valid coloring). The only difference is that $\mathcal{S}'$ chooses an edge e ahead of time and only concludes an iteration if the query sent by V^* equals e. However, since e is chosen uniformly every time, and since V^* is rewound to the beginning of each iteration until it succeeds (and we condition on it indeed succeeding), these have identical distributions.

Next, we prove that $\mathcal{S}'$ outputs $\perp$ with at most negligible probability. Observe that the commitments provided by $\mathcal{S}'$ reveal no information whatsoever about the choice of e in that iteration (this is due to the fact that the commitments are the same for *every* choice of e). Thus, the probability that a single iteration succeeds is exactly $1/|E|$, implying that $\mathcal{S}'$ outputs $\perp$ for one of the i's in the simulation with probability $\left(1 - \frac{1}{|E|}\right)^{n \cdot |E|} < e^{-n}$. There are $n \cdot |E|$ iterations, and so by the union bound, $\mathcal{S}'$ outputs $\perp$ somewhere in the simulation with probability less than $n \cdot |E| \cdot e^{-n}$, which is negligible. This implies that[2]

$$\left\{{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi) \ \mid \ {\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi) \neq \perp\right\} \stackrel{\mathrm{c}}{\equiv} \left\{{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi)\right\}. \tag{5.2}$$

Finally, we prove that the outputs of $\mathcal{S}$ and $\mathcal{S}'$ are computationally indistinguishable:

$$\left\{{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi)\right\} \stackrel{\mathrm{c}}{\equiv} \left\{\mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right\}. \tag{5.3}$$

[2]Observe that for all events A and F, $\Pr[A] = \Pr[A \wedge F] + \Pr[A \wedge \neg F] \leq \Pr[F] + \Pr[A \mid \neg F]$. Thus, if F occurs with negligible probability, then $|\Pr[A] - \Pr[A|\neg F]|$ is negligible.

Intuitively, we prove this via a reduction to the security of the commitment scheme. Specifically, assume by contradiction, that there exists a probabilistic-polynomial time verifier V^*, a probabilistic-polynomial time distinguisher D, and a polynomial $p(\cdot)$ such that for an infinite sequence (G, ψ, z) where $(G, \psi) \in R$ and $z \in \{0,1\}^*$,

$$\left|\Pr\left[D\left(G, \psi, z, {\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G, \psi)\right) = 1\right] - \Pr\left[D\left(G, \psi, z, \mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right) = 1\right]\right| \geq \frac{1}{p(n)},$$

where n denotes the number of nodes in G, and R denotes the 3-coloring relation. Without loss of generality, assume that D outputs 1 with higher probability when it receives the output of $\mathcal{S}'$ than when it receives the output of $\mathcal{S}$. We construct a non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the commitment experiment $\mathsf{LR\text{-}commit}$ as defined in Section 5.2. Adversary $\mathcal{A}$ receives (G, ψ, z) on its advice tape (for n, where G has n nodes), and works as follows:

1. $\mathcal{A}$ initializes V^* with input graph G, auxiliary input z and a uniform random tape r.
2. Then, $\mathcal{A}$ runs the instructions of $\mathcal{S}'$ with input (G, ψ) and oracle $V^*(x, z, r; \cdot)$, with some changes. First, note that $\mathcal{A}$ knows ψ and so can compute $\phi(v) = \pi(\psi(v))$ just like $\mathcal{S}'$. Next, $\mathcal{A}$ does not generate the commitments by computing $c_i = \mathsf{Com}(\phi(v_i))$ for all i, as $\mathcal{S}'$ does. Rather, $\mathcal{A}$ works as follows. For every iteration of the simulation:
   (a) For the randomly chosen edge $e = (v_k, v_\ell)$, adversary $\mathcal{A}$ generates commitments $c_k = \mathsf{Com}(\phi(v_k))$ and $c_\ell = \mathsf{Com}(\phi(v_\ell))$ by itself.
   (b) For all other i (i.e., for all $i \in \{1, \ldots, n\} \setminus \{k, \ell\}$), adversary $\mathcal{A}$ queries its LR-oracle with the pair $(0, \phi(i))$. Denote by c_i the commitment received back.

   $\mathcal{A}$ simulates $\mathcal{S}'$ querying V^* with the commitments $(c_1, \ldots, c_n)$ as a result of the above. Observe that $\mathcal{A}$ can decommit to v_k, v_ℓ as needed by $\mathcal{S}'$, since it computed the commitments itself.
3. When $\mathcal{S}'$ concludes, then $\mathcal{A}$ invokes D on $\mathcal{S}'$'s output, and outputs whatever D outputs.

Observe that when $b = 1$ in the LR-oracle experiment, the commitments $c_1, \ldots, c_n$ are generated as valid commitments to a random coloring, and the distribution over V^*'s view is identical to an execution of $\mathcal{S}'$. Thus (conditioning on the b chosen in $\mathsf{LR\text{-}commit}$),

$$\Pr\left[\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n) = 1 \mid \ b = 1\right] = \Pr\left[D\left(G, z, {\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G, \psi)\right) = 1\right].$$

Likewise, when $b = 0$ in the LR-oracle experiment, then the commitments $c_1, \ldots, c_n$ are all 0 except for the commitments c_j, c_k which are to two random different colors. Thus, this is exactly the distribution generated by $\mathcal{S}$, and

$$\Pr\left[\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n) = 1 \mid \ b = 0\right] = \Pr\left[D\left(G, z, \mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right) = 0\right].$$

We have:

$$\begin{aligned}
&\Pr\left[\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n)=1\right]\\
&= \frac{1}{2}\cdot\Pr\left[\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n)=1 \mid \ b=1\right]+\frac{1}{2}\cdot\Pr\left[\mathsf{LR\text{-}commit}_{\mathsf{Com},\mathcal{A}}(1^n)=1 \mid \ b=0\right]\\
&= \frac{1}{2}\cdot\Pr\left[D\left(G,z,{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi)\right)=1\right]+\frac{1}{2}\cdot\Pr\left[D\left(G,z,\mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right)=0\right]\\
&= \frac{1}{2}\cdot\Pr\left[D\left(G,z,{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi)\right)=1\right]+\frac{1}{2}\cdot\left(1-\Pr\left[D\left(G,z,\mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right)=1\right]\right)\\
&= \frac{1}{2}+\frac{1}{2}\cdot\left(\Pr\left[D\left(G,z,{\mathcal{S}'}^{V^*(G,z,r,\cdot)}(G,\psi)\right)=1\right]-\Pr\left[D\left(G,z,\mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right)=1\right]\right)\\
&\geq \frac{1}{2}+\frac{1}{2p(n)}.
\end{aligned}$$

This contradicts the security of Com, as stated in Theorem 5.2. Combining Equations (5.1)–(5.3), we conclude that

$$\left\{\mathsf{output}_{V^*}(P(G,\psi),V^*(G,z))\right\}\stackrel{c}{\equiv}\left\{\mathcal{S}^{V^*(G,z,r,\cdot)}(G)\right\}$$

thereby completing the proof. ■

Discussion on the proof technique. The main technique used in the above proof is to construct an alternative, hypothetical simulator $\mathcal{S}'$ who is given the actual coloring. Of course, $\mathcal{S}'$ could work by just playing the real prover. However, this would not help us prove the indistinguishability of $\mathcal{S}$. Thus, we design $\mathcal{S}'$ to work in exactly the same way as $\mathcal{S}$, except that it generates commitments that are the same as the real prover. In this way, we *separate* the two differences between $\mathcal{S}$ and a real prover: **(a)** the flow of $\mathcal{S}$ that involves choosing e and rewinding, and **(b)** the commitments that are incorrectly generated. The only difference between the real prover and $\mathcal{S}'$ is the flow, and the first part of the proof shows that this results in at most a negligible difference. Then, the second part of the proof, showing that the outputs of $\mathcal{S}'$ and $\mathcal{S}$ are computationally indistinguishable, works via reduction to the commitments. This technique is often used in simulation-based proofs, and in some cases there are series of simulators that bridge the differences between the real execution and the simulation. This is similar to sequences of hybrids in game-based proofs, with the only difference being that the sequence here is from the simulation to the real execution (or vice versa). We recommend reading more about this technique in the tutorial on sequences of games by Shoup [38].

5.4 Constant-Round Zero-Knowledge

Constant-round zero-knowledge introduces a number of difficulties regarding simulation. The protocol itself, as described in [20], is actually very simple and straightforward. However, its proof is far more involved than it seems, and requires new techniques that are important in general. In addition, it highlights difficulties that arise in many places when carrying out simulation.

Background. Before proceeding, we first consider simply running the $n\cdot|E|$ executions of the 3-coloring protocol in parallel, instead of sequentially. At first sight, this does not seem to make a difference to the zero-knowledge property, since the order of execution does not change what is revealed. Despite this, all known simulation attempts fail, and so we simply have no way of proving that this is still zero knowledge. In order to see why, in the suggested parallel protocol,

the prover sends $N \stackrel{\text{def}}{=} n \cdot |E|$ vectors of commitments to random colorings, the verifier responds with N edges, and the prover opens the commitments of the two nodes of the edge. The only way that we know to simulate this protocol is for the simulator to *guess the query edges* ahead of time. However, the probability of correctly guessing N random edges before the verifier sends them is just $|E|^{-n|E|}$ which is exponentially small. It is important to understand that rewinding does *not* solve the problem, since the verifier can choose different random edges each time, even if it has a fixed random tape. For example, the verifier could have a key to a pseudorandom function hardwired, and can choose its randomness in every execution as a function of the (entire) first message that it receives. As a result, when rewinding, effectively new randomness is used each time and the simulator would have to try an exponential number of rewinding attempts. Indeed, Goldreich and Krawczyk showed that this parallel protocol is not *black-box* zero knowledge, and in fact that no constant-round public-coin proof for a language not in $\mathcal{BPP}$ is black-box zero knowledge [21] (where public coin means that the verifier's queries are just random coin tosses). Despite this, we have no proof that it is not zero knowledge in general. We stress again, that the lack of a known attack on a protocol is not sufficient to conclude that it is secure. Thus, an alternative protocol is needed.

The solution presented by Goldreich and Kahan to this problem is to simply have the verifier commit to its query *before* the prover sends its comments. This prevents a malicious verifier from changing its query during rewinding. Details appear in Protocol 5.5.

PROTOCOL 5.5 (The Goldreich-Kahan Proof System [20])
The proof system of [20] works as follows (we provide a clear, yet rather informal description here):

1. The prover sends the first message of a (two-round) perfectly-hiding commitment scheme, denoted $\mathsf{Com}_{\mathrm{h}}$. See [17, Section 4.9.1] for a definition of such commitments.
2. The verifier chooses $N \stackrel{\text{def}}{=} n \cdot |E|$ random edges $e_1, \ldots, e_N \in_R E$. Let $q = (e_1, \ldots, e_N)$ be the query string; the verifier commits to q using the perfectly-hiding commitment $\mathsf{Com}_{\mathrm{h}}$
3. The prover prepares the first message in N parallel executions of the basic three-round proof system in Protocol 5.3 (i.e., commitments to N independent random colorings of the graph), and sends commitments to all using the perfectly-binding commitment scheme Com.
4. The verifier decommits to the string q.
5. If the verifier's decommitment is invalid, then the prover aborts. Otherwise, the prover sends the appropriate decommitments in every execution. Specifically, if e_i is the edge in the ith execution, then the prover decommits to the nodes of that edge in the ith set of commitments to colorings.
6. The verifier outputs 1 if and only if all checks pass (as in the original proof system).

Before discussing zero knowledge, we remark that the commitments from the verifier are perfectly hiding, whereas the commitments from the prover are perfectly binding. This is necessary for proving soundness, but since our focus here is simulation, we will not relate to this issue from here on; see [20] or [17, Section 4.9] for a proof of soundness.

The main difference between this protocol and the simple parallel repetition of the basic 3-coloring protocol is the fact that the verifier is committed ahead of time to its queries. Thus, the simulator can first receive the verifier's commitments, and can then send garbage commitments

and receive back the decommitments. After receiving the decommitments the simulator knows all of the queries, and can *rewind* the verifier back to the point after it sent the commitments and give new prover commitments like in the simulation of a single execution. This works because the verifier is committed to its edge queries before it receives the prover commitments, and so cannot change them. Thus, the simulator can learn the edges (by giving garbage commitments first) and can then provide "good" commitments for which it is able to decommit and complete the proof.

Despite its simplicity, there are a number of issues that must be dealt with when translating this into a formal proof. First, we have to deal with the fact that the verifier may not decommit correctly in Step 4. This may seem simple – if this happens, then just have the simulator abort as well. This makes sense since in a real proof the prover would abort in such a case. However, the problem is that the verifier may sometimes decommit correctly and sometimes not decommit correctly, and this decision may be taken as a result of the messages it receives (specifically, the commitments sent by the prover). If this is the case, and the verifier aborts with some probability p, then the simulator will abort with probability approximately $2p$.[3] Note that this problem is not solved by interpreting an invalid decommit as default edges, as in Protocol 5.3, since the simulator will prepare commitments to default edges if the verifier aborted the first time and will not be able to answer if the verifier does not abort the second time, or vice versa.

The following strategy for the simulator $\mathcal{S}$ addresses this problem:

1. $\mathcal{S}$ invokes V^* and internally hands it an honestly-generated first (receiver) message of the perfectly-hiding commitment protocol.

2. $\mathcal{S}$ receives V^*'s reply consisting of a commitment c.

3. $\mathcal{S}$ sends V^* garbage commitments and receives its decommitment. If the decommitment is not valid, it aborts. Otherwise, denote the decommitted string by $q = (e_1, \ldots, e_N)$.

4. $\mathcal{S}$ rewinds V^* to the beginning (sending the same first message of the perfectly-hiding commitment protocol) and receives its commitment c. (Since V^* has a fixed random tape and this is the first step of the protocol, it always sends the same commitment c.) Then, $\mathcal{S}$ hands simulated prover commitments to V^* that can be answered according to q; i.e., send commitments to random distinct colors on the nodes of the edge committed by the verifier and to 0 elsewhere. If V^* aborts on these commitments, then $\mathcal{S}$ repeats this step with fresh randomness. When V^* provides correct decommitments, $\mathcal{S}$ proceeds to the next step.

5. $\mathcal{S}$ sends V^* decommitments to the nodes on the committed edge, and outputs whatever V^* outputs.

One issue that arises when trying to prove that this simulation strategy works is that the commitment to q is perfectly hiding and thus only *computationally binding*. This means that it is possible that V^* decommits to a valid $q' \neq q$, but in such a case the simulation will fail. This possibility is ruled out by showing that if this occurs with non-negligible probability, then V^* can be used to break the computational binding of the commitments.

[3] This holds since the probability of abort when the simulator sends the first garbage commitments equals p, and probability of abort when the simulator sends the good commitments the second time (if a first abort didn't occur) is also p. Thus, we have that the simulator aborts with probability $p + (1-p) \cdot p = 2p - p^2$.

More importantly, it turns out that this strategy is overly simplistic, for a very important reason. Specifically, it is not necessarily the case that the simulator runs in expected polynomial-time.[4] This may seem surprising. In particular, let ϵ denote the probability that V^* does not abort ($\epsilon = 1 - p$ from above). Then, supposedly, we have that the expected running time of the simulator is $1 - \epsilon + \epsilon \cdot \frac{1}{\epsilon}$ times a fixed polynomial for computing all the commitments and so on. This is the case since with probability $1 - \epsilon$ the verifier V^* aborts and the simulation ends, and with probability ϵ the simulation proceeds to Step 4. However, since each attempt to receive a decommitment from V^* in this step succeeds with probability ϵ only, we expect to have to repeat $1/\epsilon$ times. Thus, the overall expected cost is $\text{poly}(n) \cdot (1 - \epsilon + \epsilon \cdot \frac{1}{\epsilon}) = \text{poly}(n) \cdot (2 - \epsilon)$. Despite being appealing, and true when commitments are modeled as "perfect envelopes", the above analysis is simply false. In particular, the probability that the verifier decommits correctly when receiving the first prover commitments to pure garbage is not necessarily the same as the probability that it decommits correctly when receiving the simulator-generated commitments. In order to see this, if the verifier was all powerful, it could break open the commitments and purposefully make the simulation fail by decommitting when it receives pure garbage (or fully valid commitments) and not decommitting when it receives commitments that can be opened only to its query string. This means that we can only argue that this doesn't happen by a reduction to the commitments, and this also means that there may be a negligible difference. Thus, we actually have that the expected running time of the simulator is

$$\text{poly}(n) \cdot \left(1 - \epsilon(n) + \epsilon(n) \cdot \frac{1}{\epsilon(n) - \mu(n)}\right)$$

for some negligible function μ. Now, it is once again tempting to conclude that the above is polynomial because $\mu(n)$ is negligible, and so $\epsilon(n) - \mu(n)$ is almost the same as $\epsilon(n)$. This is indeed true for "large" values of $\epsilon(n)$. For example, if $\epsilon(n) > 2\mu(n)$ then $\epsilon(n) - \mu(n) > \epsilon(n)/2$. This then implies that $\epsilon(n)/(\epsilon(n) - \mu(n)) < 2$. Unfortunately, however, this is *not* true in general. For example, consider the case that $\mu(n) = 2^{-n}$ and $\epsilon(n) = \mu(n) + 2^{-n/2} = 2^{-n} + 2^{-n/2}$. Then,

$$\frac{\epsilon(n)}{\epsilon(n) - \mu(n)} = \frac{2^{-n} + 2^{-n/2}}{2^{-n/2}} = 2^{n/2} + 1,$$

which is exponential in n. We therefore have that the simulation does not run in expected polynomial-time. This technical problem was observed and solved by [20]. This problem arises in other places, and essentially in any place that rewinding is used where a different (but indistinguishable) distribution is used between rewindings, and some success criteria must be reached in order to proceed (e.g., the party must decommit correctly). For just one example of where this arises in the context of general secure computation, see [31, Section 4.2]. We remark that in the specific case of constant-round zero-knowledge proofs, it is possible to bypass this problem by changing the protocol [37]. However, in other cases – for example, efficient secure two-party computation – it is not necessarily possible without incurring additional cost.

We show how to deal with this problem in the proof of the theorem below.

Theorem 5.6 *Let* $\mathsf{Com}_{\mathrm{h}}$ *and* Com *be perfectly-hiding and perfectly-binding commitment schemes, respectively, with security in the presence of non-uniform probabilistic-polynomial time adversaries. Then, Protocol 5.5 is black-box computational zero knowledge with an expected polynomial-time simulator.*

[4]Note that the simulator, as is, certainly does not run in strict polynomial time. However, this is inherent for black-box constant-round protocols [2], and we show only that it runs in expected polynomial time.

Proof: We first present the simplified strategy above for a black-box simulator $\mathcal{S}$ given oracle access to a verifier V^* (with a fixed input, auxiliary input and random tape), and then explain how to modify it. The simplified simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ hands V^* the first message of $\mathsf{Com}_{\mathrm{h}}$ (formally, this is via the oracle, but we write it this way for conciseness).
2. $\mathcal{S}$ receives from V^* its perfectly-hiding commitment c to some query string $q = (e_1, \ldots, e_N)$, where $N = n \cdot |E|$.
3. $\mathcal{S}$ generates N vectors of n commitments to 0, hands them to V^*, and receives back its reply.
4. If V^* aborts by not replying with a valid decommitment to c (and the decommitment is to a vector of N edges), then $\mathcal{S}$ aborts and outputs whatever V^* outputs. Otherwise, let $q = (e_1, \ldots, e_N)$ be the decommitted value. $\mathcal{S}$ proceeds to the next step.
5. *Rewinding phase – $\mathcal{S}$ repeatedly rewinds V^* back to the point where it receives the prover commitments, until it decommits to q from above:*
   (a) $\mathcal{S}$ generates N vectors of commitments $\vec{c}_1, \ldots, \vec{c}_N$, as follows. Let $e_i = (v_j, v_k)$ in q. Then, the jth and kth commitments in $\vec{c}_i$ are to random distinct colors in $\{1, 2, 3\}$ and all other commitments are to 0. Simulator $\mathcal{S}$ hands all vectors to V^*, and receives back its reply.
   (b) If V^* does not generate a valid decommitment, then $\mathcal{S}$ returns to the previous step (using fresh randomness).
   (c) If V^* generates a valid decommitment to some $q' \neq q$, then $\mathcal{S}$ outputs $\mathsf{ambiguous}$ and halts.
   (d) Otherwise, V^* exits the loop and proceeds to the next step.
6. $\mathcal{S}$ completes the proof by handing V^* decommitments to the appropriate nodes in all of $\vec{c}_1, \ldots, \vec{c}_N$, and outputs whatever V^* outputs.

The intuition behind the simulation is clear. $\mathcal{S}$ repeatedly rewinds until the string q is the one that it initially chose. In this case, it can decommit appropriately and conclude the proof. Intuitively, the fact that the result is computationally indistinguishable from a real proof by an honest prover follows from the hiding property of the perfectly-binding commitments, as in the proof of Theorem 5.4 in Section 5.3.

As we have already demonstrated, this simplified strategy suffers from the problem that $\mathcal{S}$ actually may not run in expected polynomial time. This is solved by ensuring that the simulator $\mathcal{S}$ never runs "too long". Specifically, if $\mathcal{S}$ proceeds to the rewinding phase of the simulation, then it first estimates the value of $\epsilon(n)$ which is the probability that V^* does not abort given garbage commitments. This is done by repeating Step 3 of the simulation (sending fresh random commitments to all zeroes) until $m = O(n)$ successful decommits occurs (for a polynomial $m(n)$; to be exact $m = 12n$ suffices), where a successful decommit is where V^* decommits to q, the string it first decommitted to. We remark that as in the original strategy, if V^* correctly decommits to a different $q' \neq q$ then $\mathcal{S}$ outputs $\mathsf{ambiguous}$. Then, an estimate $\tilde{\epsilon}$ of ϵ is taken to be m/T, where T is the overall number of attempts until m successful decommits occurred. As shown in [20], this suffices to ensure that the probability that $\tilde{\epsilon}$ is not within a constant factor of $\epsilon(n)$ is at most 2^{-n}.

(An exact computation of how to achieve this exact bound using Chernoff can be found in [26, Section 6.5.3].)

Next, $\mathcal{S}$ runs the rewinding phase in Step 5 of the simulation up to n times. Each time, $\mathcal{S}$ limits the number of rewinding attempts in the rewinding phase to $n/\tilde{\epsilon}$ iterations. We have the following cases:

1. If within $n/\tilde{\epsilon}$ rewinding iterations, $\mathcal{S}$ obtains a successful decommitment from V^* to q, then it completes the proof as described. It can do so in this case because the prover commitments enable it to answer the query q.

2. If $\mathcal{S}$ obtains a valid decommitment to some $q' \neq q$ then it outputs ambiguous.

3. If $\mathcal{S}$ does not obtain any correct decommitment within $n/\tilde{\epsilon}$ attempts, then $\mathcal{S}$ aborts this attempted rewinding phase.

As mentioned, the above phase is repeated up to n times, each time using independent coins. If the simulator $\mathcal{S}$ doesn't successfully conclude in any of the n attempts, then it halts and outputs fail. We will show that this strategy ensures that the probability that $\mathcal{S}$ outputs fail is negligible.

In addition to the above, $\mathcal{S}$ keeps a count of its overall running time and if it reaches 2^n steps, then it halts, outputting fail. (This additional time-out is needed to ensure that $\mathcal{S}$ does not run too long in the case that the estimate $\tilde{\epsilon}$ is not within a constant factor of $\epsilon(n)$. Recall that this "bad event" can only happen with probability 2^{-n}.)

We first claim that $\mathcal{S}$ runs in expected polynomial-time.

Claim 5.7 *Simulator $\mathcal{S}$ runs in expected-time that is polynomial in n.*

Proof: Observe that in the first and all later iterations, all of $\mathcal{S}$'s work takes a strict polynomial-time number of steps. We therefore need to bound only the number of rewinding iterations. Before proceeding, however, we stress that rewinding iterations only take place if V^* provides a valid decommitment in the first place. Thus, all rewinding only occur with probability $\epsilon(n)$.

Now, $\mathcal{S}$ first rewinds in order to obtain an estimate $\tilde{\epsilon}$ of $\epsilon(n)$. This involves repeating until $m(n) = 12n$ successful decommitments are obtained. Therefore, the expected number of repetitions in order to obtain $\tilde{\epsilon}$ equals exactly $12n/\epsilon(n)$ (since the expected number of trials for a single success is $1/\epsilon(n)$; observe that in all of these repetitions the commitments are to all zeroes). After the estimate $\tilde{\epsilon}$ has been obtained, $\mathcal{S}$ runs the rewinding phase of Step 5 for a maximum of n times, in each phase limiting the number of rewinding attempts to $n/\tilde{\epsilon}$.

Given the above, we are ready to compute the expected running-time of $\mathcal{S}$. In order to do this, we differentiate between two cases. In the first case, we consider what happens if $\tilde{\epsilon}$ is *not* within a constant factor of $\epsilon(n)$. The only thing we can say about $\mathcal{S}$'s running-time in this case is that it is bound by 2^n (since this is an overall bound on its running-time). However, since this event happens with probability at most 2^{-n}, this case adds only a polynomial number of steps to the overall expected running-time. We now consider the second case, where $\tilde{\epsilon}$ *is* within a constant factor of $\epsilon(n)$ and thus $\epsilon(n)/\tilde{\epsilon} = O(1)$. In this case, we can bound the expected running-time of $\mathcal{S}$ by

$$\text{poly}(n) \cdot \epsilon(n) \cdot \left(\frac{12n}{\epsilon(n)} + n \cdot \frac{n}{\tilde{\epsilon}} \right) = \text{poly}(n) \cdot \frac{\epsilon(n)}{\tilde{\epsilon}} = \text{poly}(n)$$

and this concludes the analysis. ■

Next, we prove that the probability that $\mathcal{S}$ outputs fail is negligible.

Claim 5.8 *The probability that $\mathcal{S}$ outputs* fail *is negligible in n.*

Proof: Notice that the probability that $\mathcal{S}$ outputs fail is less than or equal to the probability that it does not obtain a successful decommitment in any of the n rewinding phase attempts *plus* the probability that it runs for 2^n steps.

We first claim that the probability that $\mathcal{S}$ runs for 2^n steps is negligible. We have already shown in Claim 5.7, that $\mathcal{S}$ runs in expected polynomial-time. Therefore, the probability that an execution will deviate so far from its expectation and run for 2^n steps is negligible. (It is enough to use Markov's inequality to establish this fact.)

We now continue by considering the probability that in all n rewinding phase attempts, $\mathcal{S}$ does not obtain a successful decommitment within $n/\tilde{\epsilon}$ steps. First, recall that $\epsilon(n)$ equals the probability that V^* decommits when given commitments to all zeroes. Next, observe that there exists a negligible function μ such that the probability that V^* decommits when given commitments as in Step 5a is at least $\epsilon(n) - \mu(n)$. If $\epsilon(n)$ is a negligible function then this is immediate (since it just means that V^* decommits with probability at least 0, which is always correct). In contrast, if $\epsilon(n)$ is non-negligible, then this can be proven by a direct reduction to the hiding property of the commitment scheme. In particular, if V^* decommits with probability that is non-negligibly different in both cases, then this in itself can be used to distinguish commitments of one type from another. Having established this, consider the following two possible cases:

1. *Case 1:* $\epsilon(n) \leq 2\mu(n)$: In this case, V^* decommits to its query string with only negligible probability. This means that the probability that $\mathcal{S}$ even reaches the rewinding phase is negligible. Thus, $\mathcal{S}$ only outputs fail with negligible probability.

2. *Case 2:* $\epsilon(n) > 2\mu(n)$: Recall that V^* successfully decommits in any iteration with probability at least $\epsilon(n)-\mu(n)$. Now, since in this case $\epsilon(n) > 2\mu(n)$ we have that $\epsilon(n)-\mu(n) > \frac{\epsilon(n)}{2}$. Thus, the expected number of iterations needed until V^* successfully decommits is $\frac{1}{\epsilon(n)-\mu(n)} < \frac{2}{\epsilon(n)}$. Assuming that $\tilde{\epsilon}$ is within a constant factor of $\epsilon(n)$, we have that $2/\epsilon(n) = O(1/\tilde{\epsilon})$ and so the expected number of rewindings in any given rewinding attempt is bound by $O(1/\tilde{\epsilon})$. Therefore, by Markov's inequality, the probability that $\mathcal{S}$ tries more than $n/\tilde{\epsilon}$ iterations in any given rewinding phase attempt is at most $O(1/n)$. It follows that the probability that $\mathcal{S}$ tries more than this number of iterations in n independent rewinding phases is negligible in n (specifically, it is bound by $O(1/n)^n$).

   This holds under the assumption that $\tilde{\epsilon}$ is within a constant factor of $\epsilon(n)$. However, the probability that $\tilde{\epsilon}$ is *not* within a constant factor of $\epsilon(n)$ is also negligible.

Putting the above together, we have that $\mathcal{S}$ outputs fail with negligible probability only. ■

Next, we prove the following:

Claim 5.9 *The probability that $\mathcal{S}$ outputs* ambiguous *is negligible in n.*

Proof Sketch: Intuitively, if there exists an infinite series of inputs for which $\mathcal{S}$ outputs ambiguous with non-negligible probability, then this can be used to break the computational binding of the $\mathsf{Com}_{\mathrm{h}}$ commitment scheme. The only subtlety is that $\mathcal{S}$ runs in expected polynomial-time, whereas an attacker for the binding of the commitment scheme must run in strict polynomial-time. Nevertheless, this can be overcome by simply truncating $\mathcal{S}$ to twice its expected running time. By

Markov's inequality, this reduces the success probability of the binding attack by at most 1/2, and so this is still non-negligible. ■

It remains to prove that the output distribution generated by $\mathcal{S}$ is computationally indistinguishable from the output of V^* in a real proof with an honest prover. We have already shown that $\mathcal{S}$ outputs fail or ambiguous with only negligible probability. Thus, the only difference between the output distribution generated by $\mathcal{S}$ and the output distribution generated in a real proof is the perfectly binding commitments to the colors. As in the proof of Theorem 5.4, this can be formally proven by constructing an alternative simulator who is given the coloring and works in the same way as $\mathcal{S}$ except that it generates the commitments via its oracle. Then, the LR-commit experiment can be used to show indistinguishability between this and a real proof for non-uniform distinguishers. We omit the details due to the similarity to Theorem 5.4. This completes the proof. ■

Discussion. Beyond the Goldreich-Kahan technique itself, which is of importance and arises in multiple situations where rewinding-based simulation is used, there are two important lessons to be taken away from this proof. First, negligible differences can make a difference, and care must be taken wherever they appear. The intuition that a negligible event does not happen, and that computationally indistinguishable distributions behave the same, is correct only to a point. The case shown here is an excellent example of this. Second, great care must be taken to prove every claim made via a reduction to the primitive that guarantees it. In the constant-round protocol for zero knowledge, it is clear to everyone that in order to prove indistinguishability of the simulation, a reduction to the security of the commitment scheme is necessary. (Although, without doing it carefully, the need for security in the presence of non-uniform adversaries can be missed.) However, it is far less clear that it is necessary to prove that the simulation runs in polynomial-time, that the perfectly-hiding commitment remains computationally binding, and so on. In general, any property that doesn't hold when the cryptographic primitive is completely broken requires a reduction. Thus, when proving security, a good mental experiment to carry out is to consider what happens to the simulation and proof when the adversary is *all powerful*. If some important property needed in the proof no longer holds, then a reduction is needed to prove it. Furthermore, if there is a property of a cryptographic primitive that is not used anywhere in the proof, then one should reconsider whether it is actually needed.

We also remark that the simulator presented here runs in *expected polynomial time* and not strict polynomial time. This is inherent for constant-round black-box zero knowledge, as proven in [2] (perhaps surprisingly, it is not possible to somehow truncate the simulator's execution and obtain only a negligible difference). Thus, in some cases – and in particular when considering constant-round protocols – simulators are relaxed to be allowed to run in expected polynomial time.

Soundness. We reiterate that in order to prove security for zero knowledge, it is necessary to separately prove that soundness holds. We have omitted this here since it is not the focus of the tutorial.

5.5 Honest-Verifier Zero Knowledge

A proof system is honest-verifier zero knowledge if the zero-knowledge property holds for *semi-honest* verifiers. We stress that the proof system must be sound for malicious provers. Otherwise,

as we have mentioned above, it is meaningless (the prover can just say "trust me").

It is instructional to consider honest-verifier zero knowledge as well, since this enables a comparison to the simulation technique above for arbitrary malicious verifiers, and serves as a good contrast between semi-honest simulation as in Section 4 and the remainder of this tutorial that considers malicious adversaries. As we will see, simulation for semi-honest adversaries is very different than for malicious adversaries.

Parallel 3-coloring. Consider the basic 3-coloring protocol described in Protocol 5.3 run $n \cdot |E|$ times in parallel. Specifically, the prover generates $n \cdot |E|$ sets of commitments to random colorings and sends them to the verifier. The verifier chooses $q = (e_1, \ldots, e_N)$ at random and sends q to the prover. Finally, the prover decommits as in the protocol.

The simulator $\mathcal{S}$ for honest-verifier zero knowledge. We proceed directly to describe the simulator for this protocol. Given a graph $G = (V, E)$ with $|V| = n$ and auxiliary input z, the simulator $\mathcal{S}$ works as follows:

1. Let $N = n \cdot |E|$. Then, for $i = 1, \ldots, N$, $\mathcal{S}$ chooses a random edge $e_i \in E$, and sets $q = (e_1, \ldots, e_N)$. Let r_q be the random coin tosses that define q.
2. For every $i = 1, \ldots, N$:
   (a) Let $e_i = (v_j, v_k)$.
   (b) $\mathcal{S}$ chooses random $\phi(v_j) \in_R \{1, 2, 3\}$ and $\phi(v_k) \in_R \{1, 2, 3\} \setminus \{\phi(v_j)\}$. For all other $v_\ell \in V \setminus \{v_j, v_k\}$, $\mathcal{S}$ sets $\phi(v_\ell) = 0$.
   (c) $\mathcal{S}$ sets the commitment vector $\vec{c}_i = (c_i^1, \ldots, c_i^n) = (\mathsf{Com}(\phi(v_1)), \ldots, \mathsf{Com}(\phi(v_n)))$.
   (d) $\mathcal{S}$ sets the decommitment vector $\vec{d_i} = (\mathsf{decom}(c_i^j), \mathsf{decom}(c_i^k))$
3. $\mathcal{S}$ outputs the view of the (semi-honest) verifier, defined by: $\langle G, z, r_q; (\vec{c}_1, \ldots, \vec{c}_N), (\vec{d}_1, \ldots, \vec{d}_N) \rangle$.

Before proving that this is indeed indistinguishable from a real view in a real interaction, observe that there is no rewinding here and the simulator $\mathcal{S}$ just chooses the query string of the verifier. This is allowed since a semi-honest verifier chooses its query string by reading it directly from its random tape. Since $\mathcal{S}$ chose r_q randomly, and writes r_q on the verifier's random tape, it is given that the verifier's query is q. The reason why $\mathcal{S}$ need not rewind at all is because it already knows the query string (indeed, it chose it). In fact, $\mathcal{S}$ here does not "interact" with the verifier at all, unlike the simulator for regular (malicious) zero knowledge interacts with V^*. Rather, $\mathcal{S}$ just generates the transcript of messages, independently of the adversary. This is allowed since the verifier is semi-honest, and so we know exactly what it will do already.

Restating the above, it is not necessary to interact with the adversary or rewind it to somehow guess the query string, since we know exactly how the verifier chooses that string in the semi-honest case. Thus, the problem that the verifier can choose its query in an arbitrary way, and in particular possibly based on the first message does not arise. This means that a simpler protocol suffices, and it is much easier to prove security. Recall that this parallel 3-coloring protocol is *not* black-box zero knowledge for malicious verifiers [21]. Thus, honest-verifier zero knowledge is strictly easier to achieve than black-box zero knowledge.

Theorem 5.10 *If* Com *is a perfectly-binding commitment scheme, then the parallel 3-coloring protocol is honest-verifier zero knowledge.*

Proof Sketch: Assume by contradiction that there exists an infinite series of (G, z) and a distinguisher D, such that D distinguishes the output of $\mathcal{S}$ from a real execution transcript with non-negligible probability. Then, a non-uniform polynomial-time distinguisher $\mathcal{A}$, given a valid coloring ψ of G on its advice tape, can be constructed for the commitment scheme, as follows. $\mathcal{A}$ works exactly like $\mathcal{S}$, but uses its LR-oracle (as in Section 5.3) to generate pairs of commitments to either a real random coloring or to simulator-generated commitments values (the two random colors for the query edge and zeroes otherwise). The distinguisher $\mathcal{A}$ works in a very similar way to in the proof of Theorem 5.4. As with the proof of Theorem 5.4, the distribution generated when the commitments are to the real colorings is exactly that of a real execution, and otherwise it is the simulation. Thus, the distributions are indistinguishable, as required. ■

6 Defining Security for Malicious Adversaries

6.1 Motivation

In this section, we present the definition of security for the case of malicious adversaries who may use any efficient attack strategy and thus may arbitrarily deviate from the protocol specification. In this case, it does not suffice to require the existence of a simulator that can generate the view of the corrupted party, based on its prescribed input and output as is sufficient for the case of semi-honest adversaries. First and foremost, the generation of such a view depends on the actual input used by the adversary, and this input affects the actual output received. Furthermore, in contrast to the case of semi-honest adversaries, the adversary may not use the input that it is provided. Thus, for example, a simulator for the case where P_1 is corrupted cannot just take x and $f(x, y)$ and generate a view (in order to prove that nothing more than the output is learned), because the adversary may not use x at all. Furthermore, beyond the possibility that a corrupted party may learn more than it should, we require that a corrupted party should not be able to cause the output to be incorrectly distributed. This is not captured by considering the view of the adversary alone.

In order to capture these threats, and others, the security of a protocol is analyzed by comparing what an adversary can do in the protocol to what it can do in an ideal scenario that is secure by definition. In this context, the ideal model consists of an *ideal* computation involving an incorruptible *trusted third party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted third party exists) can do no more harm than if it were involved in the ideal computation. See [8, 18] for detailed motivation and discussion on this definitional paradigm.

We remark that in defining security for two parties it is possible to consider only the setting where one of the parties is corrupted, or to also consider the setting where none of the parties are corrupted, in which case the adversary seeing the transcript between the parties should learn nothing. Since this latter case can easily be achieved by using encryption between the parties we present the simpler formulation of security that assumes that exactly one party is always corrupted.[5]

[5]There is no need to consider the case of both parties corrupted, since in such a case there is nothing to protect. In the case of adaptive corruptions (see Section 10.3), there is reason to consider corrupting both. However, this is beyond the scope of this tutorial.

Before proceeding, it is worth contrasting the above to the case of zero knowledge (where malicious verifiers were considered). Recall that in the context of zero knowledge, simulation is used to show that the adversary learns *nothing*. Specifically, the adversary is able to generate its view by itself, without receiving any external information, and thus it learns nothing from the real interaction. This works for zero knowledge where the adversarial party has no private input and is supposed to learn nothing. In fact, in the zero knowledge case, the adversarial verifier does learn that the statement is correct. However, the definition of zero knowledge only states that the adversarial verifier may learn nothing when the statement is in the language, and so is "correct". This makes sense since zero-knowledge proofs are typically used to ensure that parties behave "correctly". Thus, when the verifier is corrupted, the prover is honest and so the statement is supposed to be true. (The case of a corrupted prover who wishes to prove an incorrect statement to an honest verifier is covered separately by soundness.) In the coming sections, we will consider the more general case where parties are supposed to learn output, and also possibly have input. As we will see, this considerably changes the way simulators work, although the techniques shown so far are also needed.

6.2 The Definition

Execution in the ideal model. In the case of no honest majority (and in particular in the two-party case that we consider here), it is in general impossible to achieve guaranteed output delivery and fairness [15]. This "weakness" is therefore incorporated into the ideal model by allowing the adversary in an ideal execution to abort the execution or obtain output without the honest party obtaining its output. Denote the participating parties by P_1 and P_2 and let $i \in \{1,2\}$ denote the index of the corrupted party, controlled by an adversary $\mathcal{A}$. An ideal execution for a function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ proceeds as follows:

Inputs: Let x denote the input of party P_1, and let y denote the input of party P_2. The adversary $\mathcal{A}$ also has an auxiliary input denoted by z. All parties are initialized with the same value 1^n on their security parameter tape (including the trusted party).

Send inputs to trusted party: The honest party P_j sends its prescribed input to the trusted party. The corrupted party P_i controlled by $\mathcal{A}$ may either abort (by replacing the input with a special abort_i message), send its prescribed input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{A}$ and may depend on the input value of P_i and the auxiliary input z. Denote the pair of inputs sent to the trusted party by (x', y') (note that if $i = 2$ then $x' = x$ but y' does not necessarily equal y, and vice versa if $i = 1$).

Early abort option: If the trusted party receives an input of the form abort_i for some $i \in \{1,2\}$, it sends abort_i to the honest party P_j and the ideal execution terminates. Otherwise, the execution proceeds to the next step.

Trusted party sends output to adversary: At this point the trusted party computes $f_1(x', y')$ and $f_2(x', y')$ and sends $f_i(x', y')$ to party P_i (i.e., it sends the corrupted party its output).

Adversary instructs trusted party to continue or halt: $\mathcal{A}$ sends either $\mathsf{continue}$ or abort_i to the trusted party. If it sends $\mathsf{continue}$, the trusted party sends $f_j(x', y')$ to the honest party P_j. Otherwise, if $\mathcal{A}$ sends abort_i, the trusted party sends abort_i to party P_j.

Outputs: The honest party always outputs the output value it obtained from the trusted party. The corrupted party outputs nothing. The adversary $\mathcal{A}$ outputs any arbitrary (probabilistic polynomial-time computable) function of the prescribed input of the corrupted party, the auxiliary input z, and the value $f_i(x', y')$ obtained from the trusted party.

Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be a two-party functionality, where $f = (f_1, f_2)$, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine, and let $i \in \{1,2\}$ be the index of the corrupted party. Then, the ideal execution of f on inputs (x, y), auxiliary input z to $\mathcal{A}$ and security parameter n, denoted by $\text{IDEAL}_{f,\mathcal{A}(z),i}(x, y, n)$, is defined as the output pair of the honest party and the adversary $\mathcal{A}$ from the above ideal execution.

Execution in the real model. We next consider the real model in which a real two-party protocol π is executed (and there exists no trusted third party). In this case, the adversary $\mathcal{A}$ sends all messages in place of the corrupted party, and may follow an arbitrary polynomial-time strategy. In contrast, the honest party follows the instructions of π. We consider a simple network setting where the protocol proceeds in rounds, where in each round one party sends a message to the other party. (In the multiparty setting, this is an unsatisfactory model and one must allow all parties to send messages at the same time. However, in this case, it is standard to assume a *rushing* adversary, meaning that it receives the messages sent by the honest parties before it sends its own.)

Let f be as above and let π be a two-party protocol for computing f, meaning that when P_1 and P_2 are both honest, then the parties output $f_1(x, y)$ and $f_2(x, y)$, respectively, after an execution of π with respective inputs x and y. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine and let $i \in \{1,2\}$ be the index of the corrupted party. Then, the real execution of π on inputs (x, y), auxiliary input z to $\mathcal{A}$ and security parameter n, denoted by $\text{REAL}_{\pi,\mathcal{A}(z),i}(x, y, n)$, is defined as the output pair of the honest party and the adversary $\mathcal{A}$ from the real execution of π.

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

Definition 6.1 *Let f be a two-party functionality and let π be a two-party protocol that computes f.*[6] *Protocol π is said to* securely compute f with abort in the presence of static malicious adversaries *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model, such that for every $i \in \{1,2\}$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}(z),i}(x, y, n)\right\}_{x,y,z,n} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),i}(x, y, n)\right\}_{x,y,z,n}$$

where $x, y \in \{0,1\}^*$ under the constraint that $|x| = |y|$, $z \in \{0,1\}^*$ and $n \in \mathbb{N}$.

The above definition assumes that the parties (and adversary) know the input lengths (this can be seen from the requirement that $|x| = |y|$ is balanced and so all the inputs in the vector of inputs are of the same length). We remark that some restriction on the input lengths is unavoidable

[6] A prerequisite of any secure protocol is that it computes the functionality, meaning that two honest parties receive correct output. As we show at the end of Section 8, this is a *necessary* requirement.

because, as in the case of encryption, to some extent such information is always leaked. We will ignore this throughout, and just assume that the functionality is such that the parties know the lengths of all inputs.

In this tutorial we only consider security with abort. Therefore, in the latter when we say "securely computes" the intention is always *with abort*.

Discussion. Observe that Definition 6.1 implies *privacy* (meaning that nothing but the output is learned), *corrrectness* (meaning that the output is correctly computed) and more. This holds because the IDEAL and REAL distributions include both the corrupted and honest parties' outputs. Specifically, in the ideal model, the adversary cannot learn anything about the honest party's input beyond what is revealed in the output. Now, since the IDEAL and REAL distributions must be indistinguishable, this in particular implies that the output of the adversary in the IDEAL and REAL executions is indistinguishable. Thus, whatever the adversary learns in a real execution can be learned in the ideal model. Regarding correctness, if the adversary can cause the honest party's output to diverge from a correct value in a real execution, then this will result in a non-negligible difference between the distribution over the honest party's output in the real and ideal executions. Observe that correctness in the real model only is rather tricky to define. Is a computation correct if there *exists* some input for the corrupted party such that the output of the honest party is the correct result on that input and its own? This is a very unsatisfactory definition. First, it is possible that such an input exists, but it may be computationally hard to find. Second, it is possible that it is easy to find such an input, but only if the honest party's input is already known.[7] The ideal/real definition solves all of these problems at once. This is because in the ideal model, the adversary has to send its input explicitly to the trusted party, and correctness is judged relative to the actual input sent. This also means that parties actually "know" their inputs in protocols that are secure. See [18, Section 7.2.3] for more discussion.

Remark 6.2 (Deterministic versus probabilistic adversaries): *In all of the proofs in this tutorial – and in most proofs in general – the real-world adversary $\mathcal{A}$ is used in a black-box manner. Thus, the simulator $\mathcal{S}$ who is given input x and auxiliary input z for the corrupted party, can begin by choosing a random string r and defining the residual adversary $\mathcal{A}'(\cdot) \stackrel{\text{def}}{=} \mathcal{A}(x, z, r; \cdot)$, with security parameter 1^n (as on its own security parameter tape). From then on, $\mathcal{S}$ works with $\mathcal{A}'$ and simulates for $\mathcal{A}'$. Due to the above, it suffices to consider a deterministic adversary, with a fixed input, auxiliary input and random tape. This simplifies the treatment throughout.*

Expected polynomial-time simulation. It is sometimes necessary to relax the requirement on the simulator and allow it to run in *expected* polynomial time. As we have mentioned, this is the case for constant-round zero knowledge and thus when using constant-round zero knowledge proofs inside other protocols. However, it is also necessary when constructing constant-round protocols for *general* secure computation (where a protocol for general secure computation can be used to securely compute and polynomial-time computable function). This is due to the fact that such a general protocol can be used to securely compute the "zero-knowledge proof of knowledge" functionality. Thus, if the simulator is black box, it must run in expected polynomial time [2].

[7] This relates to an additional property that is guaranteed by the definition, called *independence of inputs*, meaning that the corrupted party is unable to make its input depend on the honest party's input. For example, in a closed-bid auction, it should not be possible for a corrupted party to make its bid be exactly \$1 greater than the honest party's bid.

6.3 Modular Sequential Composition

A protocol that is secure under *sequential* composition maintains its security when run multiple times, as long as the executions are run sequentially (meaning that each execution concludes before the next execution begins). Sequential composition theorems are theorems that state "if a protocol is secure in the stand-alone model under definition X, then it remains secure under X under sequential composition". Thus, we are interested in proving protocols secure under Definitions 4.1 and 6.1 (for semi-honest, and malicious adversaries), and immediately deriving their security under sequential composition. This is important for two reasons. First, sequential composition constitutes a security goal within itself as security is guaranteed even when parties run many executions, albeit sequentially. Second, sequential composition theorems are useful tools that help in writing proofs of security. Specifically, it enables one to design a protocol using calls to ideal functionalities (as subprotocols), and to analyze its security in this partially ideal setting. This makes protocol design and analysis significantly more simple. Thus, the use of composition theorems in order to help in proving simulation-based proofs of security is one of the most important techniques.

We do not present proofs of the sequential composition theorems for the semi-honest and malicious cases, and we recommend reading these proofs in [18]; see Sections 7.3.1 and 7.4.2 respectively. However, we do present a formal *statement* of the theorem for malicious adversaries as we will use it in the tutorial.

Modular sequential composition. The basic idea behind the formulation of the modular sequential composition theorems is to show that it is possible to design a protocol that uses an ideal functionality as a subroutine, and then analyze the security of the protocol when a trusted party computes this functionality. For example, assume that a protocol is constructed using oblivious transfer as a subroutine. Then, first we construct a protocol for oblivious transfer and prove its security. Next, we prove the security of the protocol that uses oblivious transfer as a subroutine, in a model where the parties have access to a trusted party computing the oblivious transfer functionality. The composition theorem then states that when the "ideal calls" to the trusted party for the oblivious transfer functionality are replaced with real executions of a secure protocol computing this functionality, the protocol remains secure. We begin by presenting the "hybrid model" where parties communicate by sending regular messages to each other (as in the real model) but also have access to a trusted party (as in the ideal model).

The hybrid model. We consider a *hybrid model* where parties both interact with each other (as in the real model) and use trusted help (as in the ideal model). Specifically, the parties run a protocol π that contains "ideal calls" to a trusted party computing some functionalities $f_1, \ldots, f_{p(n)}$. These ideal calls are just instructions to send an input to the trusted party. Upon receiving the output back from the trusted party, the protocol π continues. The protocol π is such that f_i is called before f_{i+1} for every i (this just determines the "naming" of the calls as $f_1, \ldots, f_{p(n)}$ in that order). In addition, if a functionality f_i is *reactive* (meaning that it contains multiple stages), then no messages are sent by the parties directly to each other from the time that the first message is sent to f_i to the time that all stages of f_i have concluded. We stress that the honest party sends its input to the trusted party in the same round and does not send other messages until it receives its output (this is because we consider *sequential composition* here). The trusted party may be used a number of times throughout the execution of π. However, each use is independent (i.e., the trusted party does not maintain any state between these calls). We call the regular

messages of π that are sent amongst the parties standard messages and the messages that are sent between parties and the trusted party ideal messages.

Sequential composition – malicious adversaries. Let $f_1, \ldots, f_{p(n)}$ be probabilistic polynomial-time functionalities and let π be a two-party hybrid-model protocol that uses ideal calls to a trusted party computing $f_1, \ldots, f_{p(n)}$. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine and let i be the index of the corrupted party. Then, the $f_1, \ldots, f_{p(n)}$-hybrid execution of π on inputs (x, y), auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\text{HYBRID}_{\pi,\mathcal{A}(z),i}^{f_1,\ldots,f_{p(n)}}(x, y, n)$, is defined as the output of the honest party and the adversary $\mathcal{A}$ from the hybrid execution of π with a trusted party computing $f_1, \ldots, f_{p(n)}$.

Let $\rho_1, \ldots, \rho_{p(n)}$ be protocols (as we will see ρ_i takes the place of f_i in π). Consider the real protocol $\pi^{\rho_1,\ldots,\rho_{p(n)}}$ that is defined as follows. All standard messages of π are unchanged. When a party is instructed to send an ideal message α to the trusted party to compute f_j, it begins a real execution of ρ_j with input α instead. When this execution of ρ_j concludes with output y, the party continues with π as if y were the output received from the trusted party for f_j (i.e., as if it were running in the hybrid model).

The composition theorem states that if $\rho_1, \ldots, \rho_{p(n)}$ securely compute $f_1, \ldots, f_{p(n)}$ respectively, and π securely computes some functionality g in the $f_1, \ldots, f_{p(n)}$-hybrid model, then $\pi^{\rho_1,\ldots,\rho_{p(n)}}$ securely computes g in the real model. As discussed above, the hybrid model that we consider here is where the protocols are run sequentially. Thus, the fact that sequential composition only is considered is implicit in the theorem, via the reference to the hybrid model.

Theorem 6.3 *Let $p(n)$ be a polynomial, let $f_1, \ldots, f_{p(n)}$ be two-party probabilistic polynomial-time functionalities and let $\rho_1, \ldots, \rho_{p(n)}$ be protocols such that each ρ_i securely computes f_i in the presence of malicious adversaries. Let g be a two-party functionality and let π be a protocol that securely computes g in the $f_1, \ldots, f_{p(n)}$-hybrid model in the presence of malicious adversaries. Then, $\pi^{\rho_1,\ldots,\rho_{p(n)}}$ securely computes g in the presence of malicious adversaries.*

Composition with expected polynomial-time simulation. The composition theorem proven by [8, 18] holds for strict polynomial-time adversaries, and certain difficulties arise when considering expected polynomial-time simulation. This issue was considered by [28], and a far simpler solution was later provided in [19]. Although of importance, we will ignore this issue in this tutorial.

Sequential composition – semi-honest adversaries. A composition theorem that is analogous to Theorem 6.3 also holds for semi-honest adversaries; see [18, Section 7.4.2].

7 Determining Output – Coin Tossing

Previously, we considered the simulation of malicious adversaries in the context of zero knowledge. However, as we have mentioned, zero knowledge is an easier case since the verifier receives no output (if the prover is honest, then the verifier already knows that the statement is true). In this section, we consider the problem of coin tossing. The coin-tossing functionality has no input, but the parties must receive the same uniformly-distributed output. Thus, in this section, we demonstrate simulation in this more difficult scenario, where the view must be generated and correlated to the actual output.

7.1 Coin Tossing a Single Bit

In this section, we present the protocol by Blum for tossing a single coin securely [5]. The protocol securely computes the functionality $f_{\mathsf{ct}}(\lambda, \lambda) = (U_1, U_1)$ where U_1 is a random variable that is uniformly distributed over $\{0,1\}$. We stress that we only consider security with abort here, and thus it is possible for one party to see the output and then abort before the other receives it (e.g., in the case that it is not a favorable outcome for that party). Indeed, it is impossible for two parties to toss a coin fairly so that neither party can cause a premature abort or bias the outcome [15].

Tossing a single coin. The idea behind the protocol is very simple: both parties locally choose a random bit, and the result is the XOR of the two bits. The problem that arises is that if P_1 sends its random bit to P_2 first, then P_2 can cheat and send a bit that forces the output to be the result that it desires. One possible way to solve this problem is to have P_1 and P_2 *simultaneously* send their bits to each other. However, we do not have simultaneous channels that force independence. (Formally, we defined a real model where protocols proceed in rounds and in each round one message is sent from one party to the other.) This is solved by having P_1 send a *commitment* to its bit b_1, rather than b_1 itself. From the hiding property of the commitment scheme, when P_2 sends b_2 it must send it independently of b_1 (since it only receives a commitment). Likewise, from the binding property of the commitment scheme, P_1 cannot change b_1 after it is committed. Therefore, even though P_1 sees b_2 before decommitting, it cannot change the value. See Protocol 7.1 for a description of the protocol.

PROTOCOL 7.1 (Blum's Coin Tossing of a Single Bit)

- **Security parameter:** Both parties have security parameter 1^n
- **The protocol:**
  1. P_1 chooses a random $b_1 \in \{0,1\}$ and a random $r \in \{0,1\}^n$ and sends $c = \mathsf{Com}(b_1; r)$ to P_2.
  2. Upon receiving c, party P_2 chooses a random $b_2 \in \{0,1\}$ and sends b_2 to P_1.
  3. Upon receiving b_2, party P_1 sends (b_1, r) to P_2 and outputs $b = b_1 \oplus b_2$. (If P_2 does not reply, or replies with an invalid value, then P_2 sets $b_2 = 0$.)
  4. Upon receiving (b_1, r) from P_1, party P_2 checks that $c = \mathsf{Com}(b_1; r)$. If yes, it outputs $b = b_1 \oplus b_2$; else it outputs $\perp$.

Before we proceed to proving the security of Protocol 7.1, we discuss the main challenges in carrying out the simulation. This is our first example of a "standard" secure computation. The simulator here is the ideal-model adversary. As such, it *externally interacts* with the trusted party computing the functionality (in this case, $f_{\mathsf{ct}}(\lambda, \lambda) = (U_1, U_1)$), and *internally interacts* with the real-model adversary as part of the simulation. Throughout simulation-based proofs, it is very important to emphasize the difference between such interactions. (Of course, internal interaction is not real, and is just the simulator internally feeding messages to $\mathcal{A}$ that it runs as a subroutine, as in Section 5.) In general, the simulator needs to send the trusted party the corrupted party's input and receive back its output. In this specific case of coin tossing, the parties have no input, and so the adversary just receives the output from the trusted party (formally, the parties send an empty string λ as input so that the trusted party knows to compute the functionality). The challenge

of the simulator is to make the output of the execution that it simulates equal the output that it received from the trusted party.

We elaborate on this challenge: in the simulation, the simulator receives the output bit b from the trusted party and needs to make the result of the execution equal b. Thus, it has to be able to *completely bias* the outcome to be a specific value. This contradicts the basic security of a coin tossing protocol! However, like zero knowledge versus soundness in the case of zero-knowledge proofs, this contradiction is overcome by the fact that the simulator has some *additional power* that a real adversary does not have. As before, the additional power it has here is the ability to rewind the adversary. Intuitively, since we are tossing a single bit, and in each execution the probability that the result equals b is $1/2$, it follows that the simulator can just run the protocol numerous times from scratch, until the result is b. Since we expect to need to rewind only twice, we are guaranteed that the simulator will succeed within n attempts, except with negligible probability. However, another concern arises here. Specifically, a corrupted P_1 may abort and refuse to decommit to its first commitment. Observe that P_1 already knows the output at this point, and so this decision may be a function of what the output will be. Fortunately, in the ideal model, the definition of security allows the corrupted party to obtain the input, and not necessarily provide the output to the honest party. However, the simulation must take great care to not skew the probability of this happening (if in a real execution P_2 receives output with probability p when the output will be 0, and receives output with probability q when the output will be 1, then these probabilities must be negligibly close to p and q in the simulation as well). We now proceed to the actual proof (this proof is based heavily on [18, Section 7.4.3.1]).

Theorem 7.2 *Assume that* Com *is a perfectly-binding commitment scheme. Then, Protocol 7.1 securely computes the bit coin-tossing functionality defined by* $f_{\mathsf{ct}}(\lambda, \lambda) = (U_1, U_1)$.

Proof: It is clear that Protocol 7.1 computes f_{ct}, since when both parties are honest they output $b_1 \oplus b_2$ which is uniformly distributed. We now proceed to prove that the protocol is secure.

Let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time adversary. As discussed in Remark 6.2, we may consider a deterministic $\mathcal{A}$. We first consider the case that P_2 is corrupted. We describe the simulator $\mathcal{S}$:

1. $\mathcal{S}$ sends λ externally to the trusted party computing f_{ct} and receives back a bit b.
2. $\mathcal{S}$ initializes a counter $i = 1$.
3. $\mathcal{S}$ invokes $\mathcal{A}$, chooses a random $b_1 \in_R \{0,1\}$ and $r \in_R \{0,1\}^n$ and internally hands $\mathcal{A}$ the value $c = \mathsf{Com}(b_1; r)$ as if it was sent by P_2.
4. If $\mathcal{A}$ replies with $b_2 = b \oplus b_1$, then $\mathcal{S}$ internally hands $\mathcal{A}$ the pair (b_1, r) and outputs whatever $\mathcal{A}$ outputs. (As in the protocol, if $\mathcal{A}$ does not reply or replies with an invalid value, then this is interpreted as $b_2 = 0$.)
5. If $\mathcal{A}$ replies with $b_2 \neq b \oplus b_1$ and $i < n$, then $\mathcal{S}$ sets $i = i + 1$ and returns back to Step 3.
6. If $i = n$, then $\mathcal{S}$ outputs fail.

We first prove that $\mathcal{S}$ outputs fail with negligible probability. Intuitively, this is the case since $\mathcal{A}$'s response bit b_2 is (computationally) independent of b_1. In order to see this, observe that an

iteration succeeds if and only if $b_1 \oplus b_2 = b$, where b_2 is $\mathcal{A}$'s response to $\mathsf{Com}(b_1)$. We have:

$$\begin{aligned}\Pr[\mathcal{A}(\mathsf{Com}(b_1)) = b_1 \oplus b] &= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Com}(0)) = b] + \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Com}(1)) = 1 \oplus b] \\ &= \frac{1}{2} \cdot \Pr[\mathcal{A}(\mathsf{Com}(0)) = b] + \frac{1}{2} \cdot (1 - \Pr[\mathcal{A}(\mathsf{Com}(1)) = b]) \\ &= \frac{1}{2} + \frac{1}{2} \cdot (\Pr[\mathcal{A}(\mathsf{Com}(0)) = b] - \Pr[\mathcal{A}(\mathsf{Com}(1)) = b])\end{aligned}$$

where the probability is taken over the choice of b_1 and the randomness used to generate the commitment. By the assumption that Com is a perfectly-biding commitment scheme, and thus is computationally hiding, we have that there exists a negligible function μ such that for every $b \in \{0,1\}$

$$\left|\Pr[\mathcal{A}(\mathsf{Com}(0)) = b] - \Pr[\mathcal{A}(\mathsf{Com}(1)) = b]\right| \leq \mu(n),$$

and so

$$\frac{1}{2} \cdot (1 - \mu(n)) \leq \Pr[\mathcal{A}(\mathsf{Com}(b_1)) = b_1 \oplus b] \leq \frac{1}{2} \cdot (1 + \mu(n)). \tag{7.1}$$

(We stress that in Eq. (7.1), the probability is taken over the choice of b_1 and the randomness used to generate $\mathsf{Com}(b_1)$.) Since $\mathcal{S}$ outputs fail if and only if $\mathcal{A}(\mathsf{Com}(b_1)) \neq b_1 \oplus b$ in *all* n iterations, we have that $\mathcal{S}$ outputs fail with probability at most

$$\left(\frac{1}{2} \cdot (1 + \mu(n))\right)^n < \left(\frac{2}{3}\right)^n$$

which is negligible (the inequality holds for all large enough n's).

Next, we show that conditioned on the fact that $\mathcal{S}$ does not output fail, the output distributions IDEAL and REAL are statistically close. Observe that in both the real and ideal (i.e., simulated) executions, the bit b_2 sent by $\mathcal{A}$ is fully determined by b_1, r. Specifically, we can write $b_2 = \mathcal{A}(\mathsf{Com}(b_1; r))$. We therefore have that both distributions are of the form $(b, \mathcal{A}(\mathsf{Com}(b_1; r), b_1, r))$, where $b = b_1 \oplus \mathcal{A}(\mathsf{Com}(b_1; r))$. The difference between the distributions is as follows:

- **Real:** In a real execution, b_1 and r are uniformly distributed.
- **Ideal:** In an ideal execution, a random b is chosen, and then random b_1 and r are chosen under the constraint that $b_1 \oplus \mathcal{A}(\mathsf{Com}(b_1; r)) = b$.

In order to see that these distributions are statistically close, we calculate the probability that every (b_1, r) is chosen according to the distributions. Fix $\hat{b}_1, \hat{r}$. Then, in the real execution it is immediate that $(\hat{b}_1, \hat{r})$ appears with probability exactly $2^{-(n+1)}$.

Regarding the ideal execution, denote by $S_b = \{(b_1, r) \mid b_1 \oplus \mathcal{A}(\mathsf{Com}(b_1; r)) = b\}$. Observe that S_b contains all the pairs (b_1, r) that can lead to an output of b in the ideal execution (since $\mathcal{S}$ concludes when $b_1 \oplus \mathcal{A}(\mathsf{Com}(b_1; r)) = b$). We claim that the fixed $(\hat{b}_1, \hat{r})$ appears in the ideal execution with probability

$$\frac{1}{2} \cdot \frac{1}{|S_b|}. \tag{7.2}$$

This holds because b is uniformly chosen by the trusted party, and then *conditioned on not outputting* fail, simulator $\mathcal{S}$ samples a uniformly distributed element from S_b. (This can be seen by

the fact that $\mathcal{S}$ concludes as soon as it obtains an element of S_b, and in every iteration it chooses a random b_1, r with the "hope" that it is in S_b.)

It remains to show that for every $b \in \{0,1\}$, the set S_b has close to 2^n elements. However, this follows directly from Eq. (7.1). Specifically, Eq. (7.1) states that for every b, the probability that $\mathcal{A}(\mathsf{Com}(b_1; r)) = b_1 \oplus b$ is $\frac{1}{2} \cdot (1 \pm \mu(n))$. However, this probability is exactly the probability that $(b_1, r) \in S_b$. This implies that

$$\frac{1}{2} \cdot (1 - \mu(n)) \leq \frac{|S_b|}{2^{n+1}} \leq \frac{1}{2} \cdot (1 + \mu(n)),$$

and so

$$2^n \cdot (1 - \mu(n)) \leq |S_b| \leq 2^n \cdot (1 + \mu(n)).$$

Combining this with Eq. (7.2), we have that the pair $(\hat{b}_1, \hat{r})$ appears with probability between $2^{-(n+1)} \cdot (1 - \mu(n))$ and $2^{-(n+1)} \cdot (1 + \mu(n))$. This is therefore statistically close to the probability that $(\hat{b}_1, \hat{r})$ appears in a real execution. The real and ideal output distributions are therefore statistically close.[8]

We now turn to the case that P_1 is corrupted. The simulation here needs to take into account the case that $\mathcal{A}$ does not reply with a valid message and so aborts. The simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ sends λ externally to the trusted party computing f_{ct} and receives back a bit b.
2. $\mathcal{S}$ invokes $\mathcal{A}$ and internally receives the message c that $\mathcal{A}$ sends to P_1.
3. $\mathcal{S}$ internally hands $\mathcal{A}$ the bit $b_2 = 0$ as if coming from P_2, and receives back its reply. Then, $\mathcal{S}$ internally hands $\mathcal{A}$ the bit $b_2 = 1$ as if coming from P_2, and receives back its reply. We have the following cases:
   (a) If $\mathcal{A}$ replies with a valid decommitment (b_1, r) such that $\mathsf{Com}(b_1; r) = c$ in both iterations, then $\mathcal{S}$ externally sends $\mathsf{continue}$ to the trusted party. In addition, $\mathcal{S}$ defines $b_2 = b_1 \oplus b$, internally hands $\mathcal{A}$ the bit b_2, and outputs whatever $\mathcal{A}$ outputs.
   (b) If $\mathcal{A}$ does not reply with a valid decommitment in either iteration, then $\mathcal{S}$ externally sends abort_1 to the trusted party. Then, $\mathcal{S}$ internally hands $\mathcal{A}$ a random bit b_2 and outputs whatever $\mathcal{A}$ outputs.
   (c) If $\mathcal{A}$ replies with a valid decommitment (b_1, r) such that $\mathsf{Com}(b_1; r) = c$ only when given b_2 where $b_1 \oplus b_2 = b$, then $\mathcal{S}$ externally sends $\mathsf{continue}$ to the trusted party. Then, $\mathcal{S}$ internally hands $\mathcal{A}$ the bit $b_2 = b_1 \oplus b$ and outputs whatever $\mathcal{A}$ outputs.
   (d) If $\mathcal{A}$ replies with a valid decommitment (b_1, r) such that $\mathsf{Com}(b_1; r) = c$ only when given b_2 where $b_1 \oplus b_2 \neq b$, then $\mathcal{S}$ externally sends abort_1 to the trusted party. Then, $\mathcal{S}$ internally hands $\mathcal{A}$ the bit $b_2 = b_1 \oplus b \oplus 1$ and outputs whatever $\mathcal{A}$ outputs.

We prove that the output distribution is identical. We consider three cases:

[8] It may seem surprising that we obtain statistical closeness, even though we are relying on the computational hiding of the commitment scheme. However, the computational hiding is used only to ensure that $\mathcal{S}$ outputs fail with negligible probability, and holds when considering any polynomial-time $\mathcal{A}$.

1. *Case 3a – $\mathcal{A}$ always replies with a valid decommitment:* In this case, $\mathcal{A}$'s view in a real execution consists of a random bit b_2, and the honest P_2's output equals $b = b_1 \oplus b_2$, where b_1 is the committed value in c. Since b_1 is fully determined by the commitment c before b_2 is chosen by P_1, it follows that b is uniformly distributed.

   In contrast, in an ideal execution, the bit b is uniformly chosen. Then, $\mathcal{A}$'s view consists of $b_2 = b_1 \oplus b$, and the honest P_2's output equals b. Since b_1 is fully determined by the commitment c before any information about b is given to $\mathcal{A}$, it follows that $b_2 = b_1 \oplus b$ is uniformly distributed.

   In both cases, the bits b and b_2 are uniformly distributed under the constraint that $b \oplus b_2 = b_1$. Therefore, the joint distributions over $\mathcal{A}$'s output and the honest party's output are identical in the real and ideal executions.

2. *Case 3b – $\mathcal{A}$ never replies with a valid decommitment:* In this case, $\mathcal{A}$'s view consists of a uniformly distributed bit, exactly like in a real execution. In addition, the honest P_2 outputs $\perp$ in both the real and ideal executions (with probability 1). Thus, the joint distributions over $\mathcal{A}$'s output and the honest party's output are identical in the real and ideal executions.

3. *Case 3c and 3d – $\mathcal{A}$ replies with a valid decommitment for exactly one value $\hat{b}_2 \in \{0,1\}$:* Let b_1 be the value committed in the commitment c sent by $\mathcal{A}$ (since $\mathcal{A}$ is deterministic and this is the first message, this is a fixed value). Then, in the real execution, if P_2 sends $\hat{b}_2$ then $\mathcal{A}$ replies with a valid decommitment and the honest P_2 outputs $b = b_1 \oplus \hat{b}_2$. In contrast, if P_2 sends $\hat{b}_2 \oplus 1$, then $\mathcal{A}$ does not reply with a valid decommitment and P_2 outputs $\perp$.

   Consider now the ideal execution. If $b \oplus b_1 = \hat{b}_2$, then $\mathcal{S}$ hands $\mathcal{A}$ the bit $\hat{b}_2$. In this case, $\mathcal{A}$ replies with a valid decommitment and the honest party P_2 outputs $b = b_1 \oplus \hat{b}_2$. In contrast, if $b \oplus b_1 = \hat{b}_2 \oplus 1$, then $\mathcal{S}$ hands $\mathcal{A}$ the bit $\hat{b} \oplus 1$. In this case, $\mathcal{A}$ does not reply with a valid decommitment and P_2 outputs $\perp$.

   We therefore see that the distribution over the view of $\mathcal{A}$ and the output of P_2 is identical in both cases.

This completes the proof of the theorem. ■

Discussion. The proof of Theorem 7.2 is surprising in its complexity. The intuition behind the security of Protocol 7.1 is very straightforward. Nevertheless, formally justifying this fact is very difficult.[9] Some specific observations are worth making. First, as in the zero-knowledge proofs, the mere fact that the simulator (for the case of P_2 corrupted) runs in polynomial time is not straightforward and requires a reduction to the security of the commitment scheme. Second, in the malicious setting, many additional issues needed to be dealt with:

1. The adversary can send any message and so the simulator must "interact" with it.

2. The adversary may abort in some cases and this must be carefully simulated so that the distribution is not skewed when aborts can happen.

[9] I would like to add a personal anecdote here. The first proof of security that I read that followed the ideal/real simulation paradigm with security for malicious adversaries was this proof by Oded Goldreich (it appeared in a very early draft on Secure Multiparty Computation that can be found at `www.wisdom.weizmann.ac.il/~oded/pp.html`). I remember reading it multiple times until I understood why all the complications were necessary. Thus, for me, this proof brings back fond memories of my first steps in secure computation.

3. The adversary may abort after it receives the output and before the honest party receives the output. This must be correlated with the abort and continue instructions sent to the trusted party, in order to ensure that the honest party aborts with the same probability in the real and ideal executions, and that this behavior matches the view of the adversary.

Third, it is worth comparing this proof to those of zero knowledge in Section 5. In both cases, we deal with a malicious adversary. However, in zero knowledge, there is no "joint distribution" over the output, since there is no output. Thus, it suffices to simulate the view of the verifier V^* alone. Although this is not so easy, it is far less delicate than this proof here. The need to consider the joint distribution over the outputs, and to simulate for the output received from the trusted party (whatever it may be), adds considerable complexity.

Technique discussion. It is worthwhile observing that $\mathcal{S}$ essentially plays the role of the honest party, in that it generates the messages from the honest party that the adversary expects to see. This is true in all simulations. Of course, $\mathcal{S}$ does not actually send the messages that the honest party sends, since $\mathcal{S}$ has to make the output received by $\mathcal{A}$ equal the output sent by the trusted party computing the functionality. This is something that cannot be possible in a real execution, or else a corrupted party could fully determine the output.

Interaction with the trusted party or ideal functionality. As we have seen, the simulator externally interacts with the trusted party computing the functionality. In many papers, the simulator is described as interacting directly with the functionality itself (and not a trusted party computing it). This is merely an issue of terminology and the intention is exactly the same.

7.2 Securely Tossing Many Coins and the Hybrid Model

In this section, we will show how to toss many coins. Of course, we could apply the sequential composition theorem and obtain that in order to toss some $\ell = \text{poly}(n)$ coins, the parties can carry out ℓ sequential executions of Protocol 7.1. However, we wish to toss many coins in a *constant number of rounds*. Formally, the functionality is parameterized by a polynomial ℓ and is defined by $f^{\ell}_{\mathsf{ct}}(\lambda, \lambda) = (U_{\ell(n)}, U_{\ell(n)})$. Note that the security parameter n is also given to the trusted party, and thus it can compute the length $\ell(n)$ itself.

Our main aim in this section is to introduce simulation-based proofs in the hybrid model. As such, we will assume that we are given a constant-round protocol that securely computes the zero-knowledge proof of knowledge functionality for any $\mathcal{NP}$-relation. This functionality is parameterized by a relation $R \in \mathcal{NP}$ and is defined by $f^{R}_{\mathsf{zk}}((x,w),x) = (\lambda, R(x,w))$. Note that f_{zk} receives x from both parties; if different values of x are received then the output is 0. Formally, we define:

$$f^{R}_{\mathsf{zk}}((x,w),x') = \begin{cases} (\lambda, R(x,w)) & \text{if } x = x' \\ (\lambda, 0) & \text{otherwise} \end{cases}$$

We remark that any zero-knowledge proof of knowledge for R, as defined in [17, Section 4.7], securely computes the functionality f^{R}_{zk}. This folklore fact was formally proven in [27]. The existence of a constant-round zero-knowledge proof of knowledge was proven in [30]. Thus, we conclude that f^{R}_{zk} can be securely computed in a constant number of rounds.

As we will see here, working in the hybrid model *greatly simplifies* things. In fact, the proof of security in this section – for a far more complex protocol than for tossing a single coin – is far simpler.

Protocol idea. As in Protocol 7.1, the idea behind the protocol is to have P_1 commit to a random string ρ_1 of length $\ell(n)$, and then for P_2 to reply with another random string ρ_2 of length $\ell(n)$. The result is the XOR $\rho_1 \oplus \rho_2$ of these two strings. Unfortunately, we do not know how to simulate such a protocol. This is due to the fact that when P_2 is corrupted, $\mathcal{S}$ would need to rewind the adversary $\mathcal{A}$ an exponential number of times in order to make $\rho_1 \oplus \rho_2$ equal a specific string ρ provided by the trusted party. This is similar to the problem with simulating the basic 3-round zero-knowledge protocol when running it many times in parallel. We solve this problem by not having P_2 decommit to ρ_1 at all. Rather, it sends ρ_1 and proves in zero knowledge that this is the value in the commitment. In the real world, this is the same as decommitting (up to the negligible probability that P_1 can cheat in the proof). However, in the ideal simulation, the simulator can cheat in the zero-knowledge proof and send $\rho_1 = \rho \oplus \rho_2$, where ρ is the value received from the trusted party, even though the value committed to is completely different.

In the case that P_1 is corrupted, there is another problem that arises. Specifically, in order to simulate, the simulator first needs to learn the value ρ_1 committed before it can set $\rho_2 = \rho \oplus \rho_1$. Thus, it first needs to hand $\mathcal{A}$ a random ρ_2 with the hope that it will decommit ρ_1 and correctly prove the proof. If it does not, then the simulator can just abort. If it does send ρ_1 and correctly proves the zero-knowledge proof, then the simulator can now rewind and hand it $\rho_2 = \rho \oplus \rho_1$. However, what happens if $\mathcal{A}$ aborts given this ρ_2? If the simulator aborts now then the probability of abort is much higher in the ideal execution than in a real execution (because it aborts with the probability that $\mathcal{A}$ aborts when given a random ρ_2 *plus* the probability that $\mathcal{A}$ aborts when receiving $\rho_2 = \rho_1 \oplus \rho$). But, the simulator cannot do anything else since there is only one ρ_2 that can be used at this point. We solve this problem by adding a zero-knowledge proof of knowledge that P_1 proves as soon as it commits to ρ_1. The simulator can then extract ρ_1 and set $\rho_2 = \rho_1 \oplus \rho$ without any rewinding (of course, beyond the internal rewinding needed to prove the security of f_{zk}^R; nevertheless, thanks to the composition theorem we can ignore this here). If $\mathcal{A}$ aborts on this ρ_2 then the simulator aborts; otherwise it does not. This gives the required probability of abort, as we will see. See Protocol 7.3 for the full specification.

Technique discussion – proving in the hybrid model. Before proceeding to prove the security of Protocol 7.3, we explain how a proof of security in the hybrid model works. Recall that the sequential composition theorem states that if a protocol securely computes a functionality f in the g-hybrid model for some functionality g, then it remain secure when using a secure subprotocol that securely computes g. An important observation here is that in the hybrid model with g, there is no "negligible error" or "computational indistinguishability" when computing g. Rather, g is secure by definition, and an incorruptible trusted party computes it. Thus, there is no need to prove a reduction that if an adversary can break the protocol for securely computing f, then there exists an adversary that breaks the subprotocol that securely computes g. As we have seen above, such reductions are often a major effort in the proof, and thus working in a hybrid model saves this effort.

A second important observation is that a protocol that is designed in the g-hybrid model for some g contains instructions for sending inputs to the trusted party computing f. Furthermore,

PROTOCOL 7.3 (Multiple Coin Tossing)

- **Input:** Both parties have input 1^n (where $\ell(n)$ is the number of coins to be tossed).
- **Security parameter:** Both parties have security parameter 1^n.
- **Hybrid functionalities:** Let $L_1 = \{c \mid \exists(x,r) : c = \mathsf{Com}(x;r)\}$ be the language of all valid commitments, and let R_1 be its associated $\mathcal{NP}$-relation (for statement c the witness is x, r such that $c = \mathsf{Com}(x;r)$). Let $L_2 = \{(c,x) \mid \exists r : c = \mathsf{Com}(x;r)\}$ be the language of all pairs of commitments and committed values, and let R_2 be its associated $\mathcal{NP}$-relation (for statement (c,x) the witness is r such that $c = \mathsf{Com}(x;r)$).

  The parties have access to a trusted party that computes the zero-knowledge proof of knowledge functionalities $f_{\mathsf{zk}}^{R_1}$ and $f_{\mathsf{zk}}^{R_2}$ associated with relations R_1 and R_2, respectively.
- **The protocol (for tossing $\ell(n)$ coins):**
  1. P_1 chooses a random $\rho_1 \in \{0,1\}^{\ell(n)}$ and a random $r \in \{0,1\}^{\mathrm{poly}(n)}$ of length sufficient to commit to $\ell(n)$ bits, and sends $c = \mathsf{Com}(\rho_1; r)$ to P_2.
  2. P_1 sends $(c, (\rho_1, r))$ to $f_{\mathsf{zk}}^{R_1}$.
  3. Upon receiving c, party P_2 sends c to $f_{\mathsf{zk}}^{R_1}$ and receives back a bit b. If $b = 0$ then P_2 outputs $\perp$ and halts. Otherwise, it proceeds.
  4. P_2 chooses a random $\rho_2 \in \{0,1\}^{\ell(n)}$ and sends ρ_2 to P_1.
  5. Upon receiving ρ_2, party P_1 sends ρ_1 to P_2 and sends $((c, \rho_1), r)$ to $f_{\mathsf{zk}}^{R_2}$. (If P_2 does not reply, or replies with an invalid value, then P_1 sets $\rho_2 = 0^{\ell(n)}$.)
  6. Upon receiving ρ_1, party P_2 sends (c, ρ_1) to $f_{\mathsf{zk}}^{R_2}$ and receives back a bit b. If $b = 0$ then P_2 outputs $\perp$ and halts. Otherwise, it outputs $\rho = \rho_1 \oplus \rho_2$.
  7. P_1 outputs $\rho = \rho_1 \oplus \rho_2$.

parties receive outputs from the computation of g from the trusted party. This means that an *adversary* for the protocol also sends its inputs to the computation of g in the clear, and expects to receive its outputs back. In the specific example of Protocol 7.3, the functionality used is a zero-knowledge proof of knowledge functionality. Thus, if the adversary controls the party running the prover, then it directly sends the *input and witness* pair (x, w) to f_{zk}. This means that a simulator who internally runs the adversary will receive (x, w) from the adversary and so immediately has the input and witness. Observe that there is no need to run the proof's knowledge extractor and deal with negligible error and polynomial-time issues. The simulator obtains these for free. Likewise, if the adversary controls the party running the verifier, then it expects to receive 1 as output from f_{zk} (in the typical case that an honest party never tries to prove an incorrect statement in the protocol). Thus, the simulator can just hand it 1 as the output from the trusted party, and there is no need for it to run the zero-knowledge simulator and prove a reduction that computational indistinguishability holds. In addition, this "simulation" that works by sending 1 is *perfect*.

In summary, in the simulation, the simulator *plays the trusted party that computes the functionality used in the hybrid model* that interacts with the adversary. The simulator directly receives the input that the adversary sends and can write any output that it likes. (We stress that this should not be confused with the trusted party that the simulator externally interacts with in the ideal model. This interaction is unchanged.) As a result, $\mathcal{S}$ has many types of interactions and it is very helpful to the reader to explicitly differentiate between them within the proof:

1. *External interaction with the trusted party:* this is real interaction where $\mathcal{S}$ sends and receives messages externally.

2. *Internal simulated interaction with the real adversary $\mathcal{A}$:* this is simulated interaction and involves internally invoking $\mathcal{A}$ as a subroutine on incoming messages. This interaction is of two sub-types:

   (a) *Internal simulation of real messages between $\mathcal{A}$ and the honest party.*

   (b) *Internal simulation of ideal messages between $\mathcal{A}$ and the trusted party computing the functionality used as a subprotocol in the hybrid model.*

We attempt to differentiate between these types of interactions in the simulator description.

Theorem 7.4 *Assume that* Com *is a perfectly-binding commitment scheme and let ℓ be a polynomial. Then, Protocol 7.3 securely computes the functionality $f_{\mathsf{ct}}^{\ell}(\lambda,\lambda) = (U_{\ell(n)}, U_{\ell(n)})$ in the $\left(f_{\mathsf{zk}}^{R_1}, f_{\mathsf{zk}}^{R_2}\right)$-hybrid model.*

Proof: As with Protocol 7.1, it is clear that Protocol 7.3 computes f_{ct}^{ℓ} and two honest parties output a uniformly distributed string of length $\ell(n)$. We therefore proceed to prove that the protocol is secure. We construct a simulator who is given an output string ρ and generates a transcript that results in ρ being the output. The simulator utilizes the calls to $f_{\mathsf{zk}}^{R_1}$ and $f_{\mathsf{zk}}^{R_2}$ in order to do this. We first consider the case that P_1 is corrupted, and then the case that P_2 is corrupted.

P_1 corrupted: Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$, and receives the message c that $\mathcal{A}$ sends to P_2, and the message $(c', (\rho_1, r))$ that $\mathcal{A}$ sends to $f_{\mathsf{zk}}^{R_1}$.

2. If $c' \neq c$ or $c \neq \mathsf{Com}(\rho_1; r)$, then $\mathcal{S}$ sends abort_1 to the trusted party computing f_{ct}^{ℓ}, simulates P_2 aborting, and outputs whatever $\mathcal{A}$ outputs. Otherwise, it proceeds to the next step.

3. $\mathcal{S}$ sends 1^n to the external trusted party computing f_{ct}^{ℓ} and receives back a string $\rho \in \{0,1\}^{\ell(n)}$.

4. $\mathcal{S}$ sets $\rho_2 = \rho \oplus \rho_1$ (where ρ is as received from f_{ct}^{ℓ} and ρ_1 is as received from $\mathcal{A}$ as part of its message to $f_{\mathsf{zk}}^{R_1}$), and internally hands ρ_2 to $\mathcal{A}$.

5. $\mathcal{S}$ receives the message ρ_1' that $\mathcal{A}$ sends to P_2, and the message $((c'', \rho_1''), r'')$ that $\mathcal{A}$ sends to $f_{\mathsf{zk}}^{R_2}$. If $c'' \neq c$ or $\rho_1' \neq \rho_1''$ or $c \neq \mathsf{Com}(\rho_1''; r'')$ then $\mathcal{S}$ sends abort_1 to the trusted party computing f_{ct}^{ℓ}, simulates P_2 aborting, and outputs whatever $\mathcal{A}$ outputs.

   Otherwise, $\mathcal{S}$ externally sends $\mathsf{continue}$ to the trusted party, and outputs whatever $\mathcal{A}$ outputs.

We show that the simulation in this case is *perfect*; that is, the joint output distribution in the ideal model with $\mathcal{S}$ is identically distributed to the joint output distribution in an execution of Protocol 7.3 in the f_{zk}-hybrid model with $\mathcal{A}$. In order to show this, we consider three phases of the execution: **(1)** $\mathcal{A}$, controlling P_1, sends c to P_2 and $(c, (\rho_1, r))$ to $f_{\mathsf{zk}}^{R_1}$; **(2)** P_2 sends ρ_2 to P_1; and **(3)** $\mathcal{A}$ sends ρ_2 to P_2 and $((c, \rho_1), r)$ to $f_{\mathsf{zk}}^{R_2}$.

1. *Phase 1:* Since $\mathcal{A}$ is deterministic (see Remark 6.2) and there is no rewinding, the distribution over the first phase is identical in the real and ideal executions. (If these messages cause P_2 to output $\perp$, then this is the entire distribution and so is identical.)

2. *Phase 2:* Assume that the phase 1 messages do not result in P_2 outputting $\perp$. Then, we claim that for *every* triple (c, ρ_1, r) making up the phase 1 messages, the distribution over ρ_2 received by $\mathcal{A}$ is identical in the real and ideal executions. In a real execution, the honest P_2 chooses $\rho_2 \in_R \{0,1\}^{\ell(n)}$ uniformly and independently of (c, ρ_1, r). In contrast, in an ideal execution, $\rho \in_R \{0,1\}^{\ell(n)}$ is chosen uniformly and then ρ_2 is set to equal $\rho \oplus \rho_1$ (where ρ_1 is previously *fixed* since it is committed in a perfectly-binding commitment). Since ρ is chosen independently of ρ_1, we have that $\rho_1 \oplus \rho$ is also uniformly distributed in $\{0,1\}^{\ell(n)}$ and independent of (c, ρ_1, r).

3. *Phase 3:* Assume again that the phase 1 messages do not result in P_2 outputting $\perp$. Then, we claim that for *every* (c, ρ_1, r, ρ_2) making up the phase 1 and 2 messages, it holds that the honest P_2 outputs the exact same value in a real execution with $\mathcal{A}$ and in an ideal execution with $\mathcal{S}$. In order to see this, observe that this phase consists only of $\mathcal{A}$ sending ρ_1' to P_2 and $((c'', \rho_1''), r'')$ to $f_{\mathsf{zk}}^{R_2}$. There are two cases:

   (a) *Case 1 – $c'' = c$ and $\rho_1' = \rho_1''$ and $c = \mathsf{Com}(\rho''; r'')$:* In this case, in a real execution the trusted party computing $f_{\mathsf{zk}}^{R_2}$ will send 1 to P_2 and in an ideal execution $\mathcal{S}$ will send $\mathsf{continue}$ to the trusted party. This holds because both $\mathcal{A}$ and P_2 send the same public statement (c, ρ_1') to $f_{\mathsf{zk}}^{R_2}$ and it holds that $c = \mathsf{Com}(\rho_1'; r'')$. Now, in a real execution, P_2 outputs $\rho_1' \oplus \rho_2$, whereas in an ideal execution P_2 outputs $\rho = \rho_1 \oplus \rho_2$. However, since c is a *perfectly-binding* commitment scheme, we have that $\rho_1' = \rho_1$. This implies that in this case the honest P_2 outputs the same $\rho = \rho_1 \oplus \rho_2$ in the real and ideal executions.

   (b) *Case 2 – $c'' \neq c$ or $\rho_1' \neq \rho_1''$ or $c \neq \mathsf{Com}(\rho''; r'')$:* In this case, in an ideal execution $\mathcal{S}$ will send abort_1 to the trusted party (by its specification), and the honest P_2 will output $\perp$. In a real execution, in this case, the trusted party computing $f_{\mathsf{zk}}^{R_2}$ will send 0 to P_2. This is because either $\mathcal{A}$ and P_2 send different statements to the trusted party ((c, ρ_1') versus (c'', ρ_1'')) or the witness is incorrect and $c \neq \mathsf{Com}(\rho''; r'')$. Thus, the honest P_2 in a real protocol execution will also output $\perp$.

We have shown that the distributions in each phase are identical, conditioned on the previous phases. This therefore proves that the overall joint distribution over $\mathcal{A}$'s view and P_2's output is identical in the real and ideal executions. (Although the simulation is perfect, this does not mean that the real protocol is perfectly secure, since this analysis is in the hybrid model only.)

P_2 is corrupted. Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ sends 1^n to the external trusted party computing f_{ct}^{ℓ} and receives back a string $\rho \in \{0,1\}^{\ell(n)}$. $\mathcal{S}$ externally sends $\mathsf{continue}$ to the trusted party (P_1 always receives output).

2. $\mathcal{S}$ chooses a random $r \in \{0,1\}^{\mathrm{poly}(n)}$ of sufficient length to commit to $\ell(n)$ bits, and computes $c = \mathsf{Com}(0^{\ell(n)}; r)$.

3. $\mathcal{S}$ internally invokes $\mathcal{A}$ and hands it c.

4. $\mathcal{S}$ receives back some ρ_2 from $\mathcal{A}$ (if $\mathcal{A}$ doesn't send a valid ρ_2 then $\mathcal{S}$ sets $\rho_2 = 0^{\ell(n)}$ as in the real protocol).

5. $\mathcal{S}$ sets $\rho_1 = \rho_2 \oplus \rho$ and internally hands $\mathcal{A}$ the message ρ_1 as if coming from P_1.

6. $\mathcal{S}$ receives some pair (c', ρ_1') from $\mathcal{A}$ as it sends to $f_{\mathsf{zk}}^{R_2}$ (as the "verifier"). If $(c', \rho_1') \neq (c, \rho_1)$ then $\mathcal{S}$ internally simulates $f_{\mathsf{zk}}^{R_2}$ sending 0 to $\mathcal{A}$. Otherwise, $\mathcal{S}$ internally simulates $f_{\mathsf{zk}}^{R_2}$ sending 1 to $\mathcal{A}$.

7. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

The *only* difference between a real execution of the protocol (in the f_{zk}-hybrid model) and an ideal execution with the simulator is the commitment c received by $\mathcal{A}$. In a real execution it is a commitment to ρ_1, whereas in the simulation it is a commitment to $0^{\ell(n)}$. It may be tempting to simply say that these distributions are therefore indistinguishable, by the hiding property of the commitment scheme. However, as we have stressed before, a reduction must be given in order to prove this formally. In this specific case, such a reduction is not as straightforward as it may seem. In order to see this, observe that in a reduction, the distinguisher would ask for a commitment to either ρ_1 or to $0^{\ell(n)}$ and then would run the simulator $\mathcal{S}$ with the only difference being that it uses the commitment c received instead of generating itself. Since $\mathcal{S}$ simulates $f_{\mathsf{zk}}^{R_1}$ and $f_{\mathsf{zk}}^{R_2}$, it need not know the randomness used (or even whether it is a commitment to ρ_1 or to $0^{\ell(n)}$). Thus, it can seemingly carry out the reduction. The problem with this is that $\mathcal{S}$ receives ρ externally and sets $\rho_1 = \rho_2 \oplus \rho$. Since ρ_2 is received from $\mathcal{A}$ (controlling P_2) *after* $\mathcal{A}$ receives c, the distinguisher for the commitment scheme only knows the value of ρ_1 *after* it obtains the commitment c in the distinguishing game. Thus, the reduction fails because in the commitment experiment, the pair of values are of course determined ahead of time (it is not possible to commit to either x_1 or x_2 when x_2 is chosen after the commitment is given and may be a function of the commitment value c).

We therefore begin by first showing an alternative way to generate the joint output distribution of $\mathcal{S}$ and the honest P_1 in the ideal model. Let $\mathcal{S}'$ work in the same way as $\mathcal{S}$ except that instead of receiving ρ externally from the trusted party, $\mathcal{S}'$ chooses ρ by itself (uniformly at random) after receiving ρ_2 from $\mathcal{A}$. In addition, the output of the honest party is set to be ρ. Stated differently, $\mathcal{S}'$ outputs the pair $(\rho, \mathsf{output}(\mathcal{A}))$, where $\mathsf{output}(\mathcal{A})$ is the output of $\mathcal{A}$ after the simulation. It is immediate that the output of $\mathcal{S}'$ is *identically distributed* to an ideal execution. That is

$$\left\{\mathcal{S}'(1^n)\right\}_{n\in\mathbb{N}} \equiv \left\{\text{IDEAL}_{f_{\mathsf{ct}}^{\ell},\mathcal{S}}(1^n, 1^n, n)\right\}_{n\in\mathbb{N}}. \tag{7.3}$$

This is due to the fact that the only difference is the point at which ρ is chosen. However, since it is chosen independently in both cases, the output distribution is the same. (Note that $\mathcal{S}'$ is not a valid simulator since the trusted party does not choose the output ρ. Nevertheless, we present $\mathcal{S}'$ as a way of proving the indistinguishability of two distribution, and not as a valid simulator.)

We now wish to show that the output of $\mathcal{S}'$ is computationally indistinguishable from the real output $\text{REAL}_{\pi,\mathcal{A}}(1^n, 1^n, n)$. Since we have already shown that its output is identical to the ideal output distribution, this completes the proof. In order to prove this, we construct an adversary D for the commitment scheme. We use a definition that a commitment to $0^{\ell(n)}$ is computationally indistinguishable from a commitment to a random string R of length $\ell(n)$, even given the random string R. (This follows easily from the standard definition of hiding for commitments, and in particular, from the the LR-oracle formulation in Section 5.2.)

The adversary D is given a commitment c and (random) string R and runs the code of $\mathcal{S}'$ with the following differences. First, instead of computing $c = \mathsf{Com}(0^{\ell(n)}; r)$ by itself, it uses c that it received as input. In addition, instead of choosing ρ uniformly and setting $\rho_1 = \rho \oplus \rho_2$, distinguisher D sets $\rho_1 = R$ and $\rho = \rho_1 \oplus \rho_2$. Apart from that, D follows the instructions of $\mathcal{S}'$. We have the following:

- If D receives the commitment $c = \mathsf{Com}(0^{\ell(n)})$ then its output is identical to the output of $\mathcal{S}'$. In order to see this, observe that the only difference is that D sets $\rho = \rho_2 \oplus \rho_1$ where $\rho_1 = R$ is uniformly distributed (ρ_1 does not appear elsewhere in the execution since c is a commitment to 0). Since ρ_1 is random and independent of everything else, ρ is uniformly distributed, exactly as in an execution of $\mathcal{S}'$. The commitment c is also exactly as generated by $\mathcal{S}'$. Thus, the output distribution is identical.
- If D receives the commitment $c = \mathsf{Com}(\rho_1)$ then its output is identical to the joint output distribution from a real execution. This holds because the commitment from P_1 is a commitment to a random ρ_1, and the same ρ_1 is sent to $\mathcal{A}$ in the last message of the protocol. In addition, the output of the honest party is $\rho_1 \oplus \rho_2$ exactly like in a real execution. Thus, this is just a real execution between an honest P_1 and the adversary $\mathcal{A}$.

It follows from the hiding property of the commitment scheme that the output distributions generated by D are computationally indistinguishable. Therefore, the output of $\mathcal{S}'$ – which is identical to the output in a real execution – is computationally indistinguishable from the joint output of a real execution. That is,

$$\left\{\mathcal{S}'(1^n)\right\}_{n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}}(1^n, 1^n, n)\right\}_{n\in\mathbb{N}}. \tag{7.4}$$

The proof is completed by combining Equations (7.3) and (7.4). ■

Discussion – the power of proving in the hybrid model. We remark that the proof for Protocol 7.3 is considerably more simple than the proof for Protocol 7.1. This may seem somewhat surprising since the protocol is far more complex. However, it is actually not at all surprising since the proof of security is carried out in the f_{zk}-hybrid model. This is a very powerful tool and it makes proving security much easier. For one thing, in this specific case, no rewinding is necessary. Thus, it is not necessary to justify that the simulation is polynomial time, and it is also not necessary to justify that the output distribution is not skewed by the rewinding procedure.

8 Extracting Inputs – Oblivious Transfer

In the coin-tossing functionality, the parties have no input. Thus, the simulator's challenge is to receive the output from the trusted party and to generate a view of a real execution for the adversary that corresponds to the received output. However, in general, functionalities *do* have input, and in this case the output from the trusted party is only defined after the parties provide input. Thus, the simulator must *extract* the input from the adversary, send it to the trusted party and receive back the output. The view generated by the simulator must then correspond to this input and output. As we will see below, this introduces additional challenges.

In this section, we will study the oblivious transfer functionality defined by $f_{\mathsf{ot}}((x_0, x_1), \sigma) = (\lambda, x_\sigma)$ where x_0, x_1 are from a fixed domain and σ is a bit [16, 36]. We present a version of the oblivious transfer protocol of [35] (the original protocol of [35] is in the common reference string model and will be presented in Section 9).

Preliminaries – the RAND procedure. Before presenting the protocol, we will describe and prove an important property of a probabilistic procedure, called $RAND$, that is used in the protocol. Let $\mathbb{G}$ be a multiplicative group of prime order q. Define the probabilistic procedure

$$RAND(g, x, y, z) = (u, v) = \left(g^s \cdot y^t,\ x^s \cdot z^t\right)$$

where $s, t \in_R \mathbb{Z}_q$ are uniformly random.

Claim 8.1 *Let g be a generator of $\mathbb{G}$ and let $x, y, z \in \mathbb{G}$. If (g, x, y, z) do not form a Diffie-Hellman tuple (i.e., there does not exist $a \in \mathbb{Z}_q$ such that $y = g^a$ and $z = x^a$), then $RAND(g, x, y, z)$ is uniformly distributed in $\mathbb{G}^2$.*

Proof: We prove that for every $(a, b) \in \mathbb{G} \times \mathbb{G}$,

$$\Pr[u = a \wedge v = b] = \frac{1}{|\mathbb{G}|^2}, \tag{8.1}$$

where $(u, v) = RAND(g, x, y, z)$ and the probability is taken over the random choices of $s, t \in \mathbb{Z}_q$ (this implies that (u, v) is uniformly distributed). Let $\alpha, \beta, \gamma \in \mathbb{Z}_q$ be values such that $x = g^\alpha$, $y = g^\beta$ and $z = g^\gamma$ and $\gamma \neq \alpha \cdot \beta \bmod q$. (Note that if $\gamma = \alpha \cdot \beta \bmod q$ then this implies that $z = g^\gamma = (g^\alpha)^\beta = x^\beta$ and so $y = g^\beta$ and $z = x^\beta$ in contradiction to the assumption in the claim.) Then,

$$u = g^s \cdot y^t = g^s \cdot (g^\beta)^t = g^{s+\beta \cdot t} \quad \text{and} \quad v = x^s \cdot z^t = (g^\alpha)^s \cdot (g^\gamma)^t = g^{\alpha \cdot s + \gamma \cdot t}\ .$$

Now, let $\delta, \epsilon \in \mathbb{Z}_q$ such that $a = g^\delta$ and $b = g^\epsilon$. Then, since s, t are uniformly distributed in $\mathbb{Z}_q$ it follows that Eq. (8.1) holds if and only if there is a single solution to the equations

$$s + \beta \cdot t = \delta \quad \text{and} \quad \alpha \cdot s + \gamma \cdot t = \epsilon\ .$$

(Observe that g, x, y, z and a, b are fixed. Thus, $\alpha, \beta, \gamma, \delta, \epsilon$ are fixed and s, t are uniformly chosen.) Now, there exists a single solution to these equations if and only if the matrix

$$\begin{pmatrix} 1 & \beta \\ \alpha & \gamma \end{pmatrix}$$

is invertible, which is the case here because its determinant is $\alpha \cdot \beta - \gamma$ and by the assumption $\alpha \cdot \beta \neq \gamma \bmod q$ and so $\alpha \cdot \beta - \gamma \neq 0 \bmod q$. This completes the proof. ■

The protocol idea. We are now ready to present the protocol. The idea behind the protocol is as follows. The receiving party P_2 generates a tuple (g_0, g_1, h_0, h_1) that is not a Diffie-Hellman tuple and sends it to P_1 (along with a proof that it is indeed not a Diffie-Hellman tuple).[10] Next,

[10] The protocol is actually a bit different in that P_2 generates a tuple (g_0, g_1, h_0, h_1) so that $(g_0, g_1, h_0, \frac{h_1}{g_1})$ *is* a Diffie-Hellman tuple. Of course, this implies that (g_0, g_1, h_0, h_1) is *not* a Diffie-Hellman tuple. This method is used since it enables P_2 to prove that (g_0, g_1, h_0, h_1) is not a Diffie-Hellman tuple very efficiently by proving that $(g_0, g_1, h_0, \frac{h_1}{g_1})$ *is* a Diffie-Hellman tuple.

P_2 computes $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$ and sends the pair to P_1. Then, P_1 computes $(u_0, v_0) = RAND(g_0, g, h_0, h)$ and $(u_1, v_1) = RAND(g_1, g, h_1, h)$. Finally, P_1 uses v_0 to mask the input x_0 and uses v_1 to mask x_1. We will prove that if (g_0, g_1, h_0, h_1) is not a Diffie-Hellman tuple, then for every g, h it holds that at least one of $(g_0, g, h_0, h), (g_1, g, h_1, h)$ is not a Diffie-Hellman tuple. Thus, at least one of the values v_0, v_1 is *uniformly distributed* as proven in Claim 8.1, and so a corrupted P_2 can only learn at most one of x_0, x_1. Regarding the case that P_1 is corrupted, we must argue that it cannot learn P_2's input bit σ. However, P_1 only sees $(g_\sigma)^r, (h_\sigma)^r$ and this hides σ by the Decisional Diffie-Hellman assumption. We *stress* that the above "explanation" regarding security explains why P_1 cannot learn P_2's input and why P_2 can learn at most one of x_0, x_1. However, it does *not* show how to simulate, and this requires additional ideas, as we will show. The full description appears in Protocol 8.2.

PROTOCOL 8.2 (Oblivious Transfer)

- **Inputs:** Party P_1's input is a pair (x_0, x_1) and party P_2's input is a bit σ. We assume for simplicity that $x_0, x_1 \in \mathbb{G}$ where $\mathbb{G}$ is defined in the auxiliary input.
- **Auxiliary input:** Both parties hold a security parameter 1^n and $(\mathbb{G}, q, g_0)$, where $\mathbb{G}$ is an efficient representation of a group of prime order q with a generator g_0, and q is of length n. (It is possible to generate this group in the protocol, if needed.)
- **Hybrid functionality:** Let $L = \{(\mathbb{G}, q, g_0, x, y, z) \mid \exists a \in \mathbb{Z}_q : y = (g_0)^a \wedge z = x^a\}$ be the language of all Diffie-Hellman tuples (where $(\mathbb{G}, q, g_0)$ are as above), and let R_L be its associated $\mathcal{NP}$-relation. The parties have access to a trusted party that computes the zero-knowledge proof of knowledge functionality $f_{\mathsf{zk}}^{R_L}$ associated with relation R_L.
- **The protocol:**
  1. Party P_2 chooses random values $y, \alpha \in_R \mathbb{Z}_q$ and computes $g_1 = (g_0)^y$, $h_0 = (g_0)^\alpha$ and $h_1 = (g_1)^{\alpha+1}$ and sends (g_1, h_0, h_1) to party P_1.
  2. P_2 sends statement $\left(\mathbb{G}, q, g_0, g_1, h_0, \frac{h_1}{g_1}\right)$ and witness α to $f_{\mathsf{zk}}^{R_L}$.
  3. P_1 sends statement $\left(\mathbb{G}, q, g_0, g_1, h_0, \frac{h_1}{g_1}\right)$ to $f_{\mathsf{zk}}^{R_L}$ and receives back a bit. If the bit equals 0, then it halts and outputs $\perp$. Otherwise, it proceeds to the next step.
  4. P_2 chooses a random value $r \in_R \mathbb{Z}_q$, computes $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$, and sends (g, h) to P_1.
  5. P_1 computes $(u_0, v_0) = RAND(g_0, g, h_0, h)$, and $(u_1, v_1) = RAND(g_1, g, h_1, h)$. P_1 sends P_2 the values (u_0, w_0) where $w_0 = v_0 \cdot x_0$, and (u_1, w_1) where $w_1 = v_1 \cdot x_1$.
  6. P_2 computes $x_\sigma = w_\sigma / (u_\sigma)^r$.
  7. P_1 outputs λ and P_2 outputs x_σ.

Theorem 8.3 *Assume that the Decisional Diffie-Hellman problem is hard in the auxiliary-input group $\mathbb{G}$. Then, Protocol 8.2 securely computes f_{ot} in the presence of malicious adversaries.*

Proof: We begin by showing that Protocol 8.2 computes f_{ot} (meaning that two honest parties running the protocol compute the correct output). This holds since when both parties are honest,

we have:

$$\frac{w_\sigma}{(u_\sigma)^r} = \frac{v_\sigma \cdot x_\sigma}{(u_\sigma)^r} = \frac{g^s \cdot h^t \cdot x_\sigma}{((g_\sigma)^s \cdot (h_\sigma)^t)^r} = \frac{((g_\sigma)^r)^s \cdot ((h_\sigma)^r)^t \cdot x_\sigma}{((g_\sigma)^s \cdot (h_\sigma)^t)^r} = \frac{(g_\sigma)^{r \cdot s} \cdot (h_\sigma)^{r \cdot t} \cdot x_\sigma}{(g_\sigma)^{r \cdot s} \cdot (h_\sigma)^{r \cdot t}} = x_\sigma.$$

We now proceed to prove security, and separately consider the case that P_1 is corrupted and the case that P_2 is corrupted.

P_1 is corrupted. Recall that in general the simulator $\mathcal{S}$ needs to extract the corrupted party's input in order to send it to the trusted party, and needs to simulate its view so that its output corresponds to the output received back from the trusted party. However, in this case, P_1 receives no output, and so $\mathcal{S}$'s task is somewhat simpler; it needs to extract $\mathcal{A}$'s input while generating a view of an interaction with an honest P_2. Since a corrupted P_1 is not supposed to learn anything about P_2's input, it seems that the following strategy should work:

1. Internally invoke $\mathcal{A}$ and run a complete execution between $\mathcal{A}$ and an honest P_2 with input $\sigma = 0$. Let x_0 be the output that P_2 receives as output from the protocol execution.
2. Rewind and internally invoke $\mathcal{A}$ from scratch and run a complete execution between $\mathcal{A}$ and an honest P_2 with input $\sigma = 1$. Let x_1 be the output that P_2 receives as output from the protocol execution.
3. Send (x_0, x_1) to the external trusted party computing f_{ot}.
4. Output whatever $\mathcal{A}$ output on either one of the two executions above.

Intuitively, this works since $\mathcal{S}$ obtains the output that P_2 would have obtained upon either input. In addition, the view of P_2 does not reveal its input bit (as we have described above), and thus either view can be taken. Unfortunately, this intuition is completely wrong. In order to see why, consider an adversary $\mathcal{A}$ who chooses x_0, x_1 randomly by applying a pseudorandom function to its view until the last step of the protocol (but otherwise works honestly). Furthermore, assume that $\mathcal{A}$ outputs the inputs it chose. Now, $\mathcal{S}$ does not know if the honest P_2 in the ideal model has input $\sigma = 0$ or $\sigma = 1$. If the honest P_2 has input $\sigma = 0$ and $\mathcal{S}$ outputs what $\mathcal{A}$ outputs on the second execution above, then the output that P_2 has in the ideal model will not match either x_0 or x_1 output by $\mathcal{A}$ (except with negligible probability). The same will occur if P_2 has input $\sigma = 1$ and $\mathcal{S}$ outputs what $\mathcal{A}$ outputs on the first execution above. In contrast, in a real execution, P_2 always outputs one of x_0 or x_1 output by $\mathcal{A}$ (depending on its value σ). Thus this strategy completely fails and it is easy to distinguish between a real and ideal execution.

We therefore use a completely different strategy for extracting $\mathcal{A}$'s input that does not involve rewinding. The idea behind the strategy is as follows. As we have mentioned above, if (g_0, g_1, h_0, h_1) is not a Diffie-Hellman tuple, then one of x_0, x_1 is hidden information-theoretically. However, if (g_0, g_1, h_0, h_1) *is* a Diffie-Hellman tuple, then it is actually possible to efficiently recover *both* x_0 and x_1 from P_1's message. Therefore, $\mathcal{S}$ will provide (g_0, g_1, h_0, h_1) that is a Diffie-Hellman tuple and will simply "cheat" by simulating $f_{\mathsf{zk}}^{R_L}$'s response to be 1 even though the statement is false. By the Decisional Diffie-Hellman assumption, this will be indistinguishable, but will enable $\mathcal{S}$ to extract both inputs. $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ internally invokes $\mathcal{A}$ controlling P_1 (we assume that $\mathcal{A}$ is deterministic; see Remark 6.2).

2. $\mathcal{S}$ chooses $y, \alpha \in_R \mathbb{Z}_q$ and computes $g_1 = (g_0)^y$, $h_0 = (g_0)^\alpha$ and $h_1 = (g_1)^\alpha$. (Note that $h_1 = (g_1)^\alpha$ and not $(g_1)^{\alpha+1}$ as an honest P_2 would compute it.)

3. $\mathcal{S}$ internally hands (g_1, h_0, h_1) to $\mathcal{A}$.

4. When $\mathcal{A}$ sends a message intended for $f_{\mathsf{zk}}^{R_L}$. If the message equals $\left(\mathbb{G}, q, g_0, g_1, h_0, \frac{h_1}{g_1}\right)$ then $\mathcal{S}$ internally hands $\mathcal{A}$ the bit 1 as if it came from $f_{\mathsf{zk}}^{R_L}$. If the message equals anything else, then $\mathcal{S}$ simulates $\mathcal{A}$ receiving 0 from $f_{\mathsf{zk}}^{R_L}$.

5. $\mathcal{S}$ chooses a random value $r \in_R \mathbb{Z}_q$, computes $g = (g_0)^r$ and $h = (h_0)^r$, and internally sends (g, h) to $\mathcal{A}$. (This is exactly like an honest P_2 with input $\sigma = 0$.)

6. When $\mathcal{A}$ sends messages $(u_0, w_0), (u_1, w_1)$ then simulator $\mathcal{S}$ computes $x_0 = w_0/(u_0)^r$ and $x_1 = w_1/(u_1)^{r \cdot y^{-1} \bmod q}$. (If the message is not formed correctly, then $\mathcal{S}$ sends abort_1 to the trusted party and outputs whatever $\mathcal{A}$ outputs. Otherwise, it proceeds.)

7. $\mathcal{S}$ sends (x_0, x_1) to the trusted party computing f_{ot}. (Formally, $\mathcal{S}$ receives back output λ and then sends $\mathsf{continue}$ to the trusted party. This isn't really necessary since only P_2 receives output. Nevertheless, formally, $\mathcal{S}$ must send $\mathsf{continue}$ in order for P_2 to receive output.)

8. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and halts.

In order to show that the simulation achieves indistinguishability, we first change the protocol. Denote Protocol 8.2 by π, and denote by π' a protocol that is the same as π except for the two following differences:

1. P_2 chooses $y, \alpha \in_R \mathbb{Z}_q$ and computes $g_1 = (g_0)^y$, $h_0 = (g_0)^\alpha$ and $h_1 = (g_1)^\alpha$, instead of computing $h_1 = (g_1)^{\alpha+1}$.

2. $f_{\mathsf{zk}}^{R_L}$ is modified so that it sends 1 to P_1 if and only if P_1 and P_2 sends the same statement (and irrespective of the witness sent by P_2).

We claim that for every probabilistic-polynomial time non-uniform adversary $\mathcal{A}$ controlling P_1:

$$\left\{\text{REAL}_{\pi,\mathcal{A}(z)}((x_0,x_1),\sigma,n)\right\}_{x_0,x_1,\sigma,z,n} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi',\mathcal{A}(z)}((x_0,x_1),\sigma,n)\right\}_{x_0,x_1,\sigma,z,n} . \tag{8.2}$$

We stress that we *only* claim that the output distributions of π and π' are indistinguishable when P_1 is corrupted. We make no claim when P_2 is corrupted, and indeed it is not true in that case. This suffices since we are currently proving the case that P_1 is corrupted. There is one difference between π and π' and this is how g_1, h_0, h_1 are chosen. (The change to $f_{\mathsf{zk}}^{R_L}$ is just to ensure that the output is always 1 unless $\mathcal{A}$ sends a different statement. Since P_2 is honest, this makes no difference.) However, in order to prove Eq. (8.2), we have to show both that the *joint distribution over* $\mathcal{A}$'s view and P_2's output is indistinguishable in π and π'. Note that the joint distribution including P_2's output must be considered since P_2 computes its output as a function of (u_σ, w_σ) which is computed using (g_1, h_0, h_1) that is generated differently in π'.

We prove this via a straightforward reduction to the DDH assumption in $\mathbb{G}$. We use the variant that states that for every probabilistic-polynomial time non-uniform distinguisher D there exists a negligible function μ such that

$$\left|\Pr[D(\mathbb{G}, q, g_0, g_1, (g_0)^r, (g_1)^r) = 1] - \Pr[D(\mathbb{G}, q, g_0, g_1, (g_0)^r, (g_1)^{r+1}) = 1]\right| \leq \mu(n)$$

where $\mathbb{G}$ is a group of order q with generators g_0, g_1. This follows easily from the standard DDH assumption (using the random self reducibility property of DDH [32]). Assume, by contradiction, that there exists an adversary $\mathcal{A}$ controlling P_1, a distinguisher D_π, a polynomial $p(\cdot)$, and an infinite series of tuples $(\mathbb{G}, q, g, x_0, x_1, \sigma, z, n)$ with $|q| = n$ such that

$$\left|\Pr\left[D_\pi(\text{REAL}_{\pi,\mathcal{A}(z)}((x_0,x_1),\sigma,n))=1\right] - \Pr\left[D_\pi(\text{REAL}_{\pi',\mathcal{A}(z)}((x_0,x_1),\sigma,n))=1\right]\right| \geq \frac{1}{p(n)}.$$

We construct a non-uniform probabilistic-polynomial time distinguisher D who receives input $(\mathbb{G}, q, g_0, g_1, h_0, h_1)$, and a tuple (x_0, x_1, σ, z, n) on its advice tape (where n equals the security parameter used for the DDH instance generation), and works as follows:

1. D invokes $\mathcal{A}$ and an honest P_2 with security parameter 1^n, respective inputs x_0, x_1 and σ, and auxiliary input z for $\mathcal{A}$.
2. D runs the execution between $\mathcal{A}$ and P_2 following the instructions of π' with one change. Instead of P_2 choosing $y, \alpha \in \mathbb{Z}_q$ and generating g_1, h_0, h_1, distinguisher D takes these values from its input. Everything else is the same; observe that P_2 does not use y, α anywhere else inside π' and thus D can carry out the simulation of π' in this way.
3. D invokes D_π on the joint output of $\mathcal{A}$ and the honest P_2 from this execution, and outputs whatever D_π outputs.

Since the only difference between π and π' is how the values g_1, h_0, h_1 are chosen, we have that

$$\Pr[D(\mathbb{G}, q, g_0, g_1, (g_0)^r, (g_1)^r) = 1] = \Pr\left[D_\pi(\text{REAL}_{\pi',\mathcal{A}(z)}((x_0,x_1),\sigma,n))=1\right]$$

and

$$\Pr[D(\mathbb{G}, q, g_0, g_1, (g_0)^r, (g_1)^{r+1}) = 1] = \Pr\left[D_\pi(\text{REAL}_{\pi,\mathcal{A}(z)}((x_0,x_1),\sigma,n))=1\right].$$

Thus,

$$\left|\Pr[D(\mathbb{G}, q, g_0, g_1, (g_0)^r, (g_1)^r) = 1] - \Pr[D(\mathbb{G}, q, g_0, g_1, (g_0)^r, (g_1)^{r+1}) = 1]\right| \geq \frac{1}{p(n)}$$

in contradiction to the assumption that the DDH problem is hard in $\mathbb{G}$. Thus, Eq. (8.2) holds.

Next, we prove that for every $\mathcal{A}$ controlling P_1:

$$\left\{\text{REAL}_{\pi',\mathcal{A}}((x_0,x_1),\sigma,n)\right\}_{x_0,x_1,\sigma\in\{0,1\}^*;n\in\mathbb{N}} \equiv \left\{\text{IDEAL}_{f_{\text{ot}},\mathcal{S}}((x_0,x_1),\sigma,n)\right\}_{x_0,x_1,\sigma\in\{0,1\}^*;n\in\mathbb{N}} \quad (8.3)$$

(i.e., the distributions are *identical*). There are two differences between the description of π' and an ideal execution with $\mathcal{S}$:

1. In an ideal execution, the pair (g, h) in the view of $\mathcal{A}$ is generated by computing $(g_0)^r$ and $(h_0)^r$. In contrast, in π', these values are generated by P_2 computing $(g_\sigma)^r$ and $(h_\sigma)^r$.
2. In an ideal execution, the honest P_2's outputs are determined by the trusted party, based on (x_0, x_1) sent by $\mathcal{S}$ and its input σ (unknown to $\mathcal{S}$). In contrast, in π', the honest P_2's output is determined by the protocol instructions.

Regarding the first difference, we claim that the view of $\mathcal{A}$ in both cases is *identical*. When $\sigma = 0$ then this is immediate. However, when $\sigma = 1$ it also holds. This is because $g_1 = (g_0)^y$ and $h_1 = (h_0)^y$ (where the latter is because $h_1 = (g_1)^\alpha = ((g_0)^y)^\alpha = ((g_0)^\alpha)^y = (h_0)^y$). Thus, $(g_0)^r = (g_1)^{r \cdot y^{-1} \bmod q}$ and $(h_0)^r = (h_1)^{r \cdot y^{-1} \bmod q}$. Since r is uniformly distributed in $\mathbb{Z}_q$, the values r and $r \cdot y^{-1} \bmod q$ are both uniformly distributed. Therefore, $((g_\sigma)^r, (h_\sigma)^r)$ as generated in π' is identically distributed to $((g_0)^r, (h_0)^r)$ as generated by $\mathcal{S}$ in an ideal execution.

Regarding the second difference, it suffices to show that the values (x_0, x_1) sent by $\mathcal{S}$ to the trusted party computing f_{ot} are the exact outputs that P_2 receives in π' on that transcript. There are two cases:

- *Case 1 – P_2 in π' has input $\sigma = 0$:* In this case, in both π' and the ideal execution with $\mathcal{S}$ we have that $g = (g_0)^r$ and $h = (h_0)^r$. Furthermore, in π', party P_2 outputs $x_0 = w_0/(u_0)^r$. Likewise, in an ideal execution with $\mathcal{S}$, the value x_0 is defined by $\mathcal{S}$ to be $w_0/(u_0)^r$. Thus, the value is identical.

- *Case 2 – P_2 in π' has input $\sigma = 1$:* In this case, in π' the pair (g, h) is generated by computing $g = (g_1)^r$ and $h = (h_1)^r$ for a random r, and P_2's output is obtained by computing $x_1 = w_1/(u_1)^r$. In contrast, in an ideal execution with $\mathcal{S}$ the pair (g, h) is generated by computing $g = (g_0)^r$ and $h = (h_0)^r$ for a random r, and P_2's output is defined by $\mathcal{S}$ to be $w_1/(u_1)^{r \cdot y^{-1} \bmod q}$.

  Fix the messages (g_1, h_0, h_1) and (g, h) sent by P_2 to $\mathcal{A}$ in either π' or in an ideal execution with $\mathcal{S}$. In both cases, there exists a unique y such that $g_1 = (g_0)^y$ and $h_1 = (h_0)^y$ (from P_2's instructions in π' and from $\mathcal{S}$'s specification). Let k be the unique value such that $g = (g_1)^k$ and $h = (h_1)^k$. In an execution of π' the value k is set to equal r as chosen by P_2. In contrast, in an ideal execution with $\mathcal{S}$, the value k is set to equal $r \cdot y^{-1} \bmod q$ where r is the value chosen by $\mathcal{S}$. (The fact that this is the correct value of k is justified above.) Now, in *both* a real execution of π' and an ideal execution with $\mathcal{S}$, party P_2's output is determined by $w_1/(u_1)^k$. Thus, the output is the same in both cases.

This completes the proof of Eq. (8.3). The computational indistinguishability for the simulation in the case that P_1 is corrupted is obtained by combining Equations (8.2) and (8.3).

Before proceeding to prove the case that P_2 is corrupted, we remark that it is not possible to prove indistinguishability of the ideal and real executions in a single step. This is due to the fact that $\mathcal{S}$ needs to have y where $g_1 = (g_0)^y$ in order to extract x_1. However, in the DDH reduction, y is not given to the distinguisher (indeed, the DDH problem is easy if y is given). Thus, the proof is carried out in two separate steps.

P_2 is corrupted. We now proceed to the case that P_2 is corrupted. First, $\mathcal{S}$ needs to extract P_2's input bit σ in order to send it to the trusted party and receive back x_σ. As we will show, this is made possible by the fact that in the $f_{\mathsf{zk}}^{R_L}$-hybrid model $\mathcal{S}$ receives the witness α from $\mathcal{A}$ (recall that in this model, $\mathcal{A}$ controlling P_2 must send the valid witness directly to $f_{\mathsf{zk}}^{R_L}$ or P_1 will abort). $\mathcal{S}$ will use α to determine whether $\mathcal{A}$ "used" input $\sigma = 0$ or $\sigma = 1$. Next, $\mathcal{S}$ needs to generate a view for $\mathcal{A}$ that is indistinguishable from a real view. The problem is that $\mathcal{S}$ is given x_σ but *not* $x_{1-\sigma}$. However, Claim 8.1 guarantees that $RAND$ completely hides $x_{1-\sigma}$ (since as we will show the tuple input to $RAND$ in this case is not a Diffie-Hellman tuple). Thus, $\mathcal{S}$ can use any fixed value in place of $x_{1-\sigma}$ and the result is identically distributed. Indeed, in this case we will show that the simulation is *perfect*. We now describe the simulator $\mathcal{S}$:

1. $\mathcal{S}$ internally invokes $\mathcal{A}$ controlling P_2.

2. $\mathcal{S}$ internally obtains (g_1, h_0, h_1) from $\mathcal{A}$, as it intends to send to P_1.

3. $\mathcal{S}$ internally obtains an input tuple and α from $\mathcal{A}$, as it intends to send to $f_{\mathsf{zk}}^{R_L}$.

4. $\mathcal{S}$ checks that the input tuple equals $(\mathbb{G}, q, g_0, g_1, h_0, \frac{h_1}{g_1})$, that $h_0 = (g_0)^\alpha$ and $\frac{h_1}{g_1} = (g_1)^\alpha$. If not, $\mathcal{S}$ externally sends abort_2 to the trusted party computing f_{ot}, outputs whatever $\mathcal{A}$ outputs, and halts. Else, it proceeds.

5. $\mathcal{S}$ internally obtains a pair (g, h) from P_2. If $h = g^\alpha$ then $\mathcal{S}$ sets $\sigma = 0$. Otherwise, it sets $\sigma = 1$.

6. $\mathcal{S}$ externally sends σ to the trusted party computing f_{ot} and receives back x_σ. ($\mathcal{S}$ sends $\mathsf{continue}$ to the trusted party; this isn't really needed since P_1 has only an empty output. Nevertheless, formally it needs to be sent.)

7. $\mathcal{S}$ computes $(u_\sigma, v_\sigma) = RAND(g_\sigma, g, h_\sigma, h)$ and $w_\sigma = v_\sigma \cdot x_\sigma$. In addition, $\mathcal{S}$ sets $(u_{1-\sigma}, w_{1-\sigma})$ to be independent uniformly distributed in $\mathbb{G}^2$.

8. $\mathcal{S}$ internally hands $(u_0, w_0), (u_1, w_1)$ to $\mathcal{A}$.

9. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs and halts.

We construct an alternative simulator $\mathcal{S}'$ in an alternative ideal model with a trusted party who sends *both* of P_1's inputs x_0, x_1 to $\mathcal{S}'$ upon receiving σ. Simulator $\mathcal{S}'$ works in exactly the same way as $\mathcal{S}$ with the exception that it computes $(u_{1-\sigma}, w_{1-\sigma})$ by first computing $(u_{1-\sigma}, v_{1-\sigma}) = RAND(g_{1-\sigma}, g, h_{1-\sigma}, h)$ and $w_{1-\sigma} = v_{1-\sigma} \cdot x_{1-\sigma}$, instead of choosing them uniformly.

First, we claim that the output distribution of the adversary $\mathcal{S}'$ in the alternative ideal model is *identical* to the output of the adversary $\mathcal{A}$ in a real execution with an honest P_1 (it is not necessary to consider P_1's output since it has none in f_{ot}). This follows because $\mathcal{S}'$ generates (u_0, w_0) and (u_1, w_1) exactly like an honest P_1, using the correct inputs (x_0, x_1). In addition, $\mathcal{S}'$ verifies the validity of (g_1, h_0, h_1) using witness α, exactly like $f_{\mathsf{zk}}^{R_L}$. Thus, the result is just a real execution of the protocol. (Observe that the determination of σ by $\mathcal{S}'$ is actually meaningless since it is not used in the generation of (u_0, w_0) and (u_1, w_1).)

We now claim that the output distribution of the adversary $\mathcal{S}'$ in the alternative ideal model is *identical* to the output of the adversary $\mathcal{S}$ in an ideal execution with f_{ot}. Since the only difference is in how $(u_{1-\sigma}, w_{1-\sigma})$ are computed, we need to show that the values $u_{1-\sigma}, w_{1-\sigma}$ generated by $\mathcal{S}'$ are independent uniformly distributed values in $\mathbb{G}$. We stress that the value σ here is the one determined by $\mathcal{S}$ in the simulation in Step 5.

First, consider the case that $\sigma = 0$. By Step 5, this implies that $h = g^\alpha$. We first claim that in this case (g_1, g, h_1, h) is *not* a Diffie-Hellman tuple. This follows from the fact that by Step 4 we have that $\frac{h_1}{g_1} = (g_1)^\alpha$ and so $h_1 = g_1^{\alpha+1}$. This implies that $(g_1, g, h_1, h) = (g_1, g, (g_1)^{\alpha+1}, g^\alpha)$ which is *not* a Diffie-Hellman tuple. Now, by Claim 8.1, since (g_1, g, h_1, h) is not a Diffie-Hellman tuple, it follows that $(u_1, v_1) = RAND(g_1, g, h_1, h)$ is uniformly distributed in $\mathbb{G}^2$, so $(u_1, w_1) = (u_1, v_1 \cdot x_1)$ is uniformly distributed. Thus, (u_1, w_1) are identically distributed in the executions with $\mathcal{S}$ and $\mathcal{S}'$.

Next, consider the case that $\sigma = 1$. By Step 5, this implies that $h \neq g^\alpha$; let $\alpha' \neq \alpha \bmod q$ such that $h = g^{\alpha'}$. As above, we first show that (g_0, g, h_0, h) is *not* a Diffie-Hellman tuple. By Step 4 we have that $h_0 = (g_0)^\alpha$ and so $(g_0, g, h_0, h) = (g_0, g, (g_0)^\alpha, g^{\alpha'})$ where $\alpha \neq \alpha' \bmod q$. Thus, it is not a

Diffie-Hellman tuple. As in the previous claim, using Claim 8.1 we have that in this case (u_0, w_0) as generated by $\mathcal{S}'$ is uniformly distributed and so has the same distribution as (u_0, w_0) generated by $\mathcal{S}$. This completes the proof. ■

Correctness in the case of two honest parties. Recall that Definition 6.1 includes a separate requirement that π computes f, meaning that two honest parties obtain correct output, and indeed our proof of Protocol 8.2 begins by showing that π computes f. In order to see that this separate requirement is necessary, consider the oblivious transfer functionality $f((x_0, x_1), \sigma) = (\lambda, x_\sigma)$ and consider the following protocol π:

1. P_1 sends x_0 to P_2
2. P_2 outputs x_0

We will now show that without the requirement that π computes f, this protocol is secure. Let $\mathcal{A}$ be an adversary. In the case that P_1 is corrupted, we construct an ideal simulator that invokes $\mathcal{A}$ and receives the string x_0 that $\mathcal{A}$ intends to send to P_2. Simulator $\mathcal{S}$ then sends (x_0, x_0) to the trusted party. Clearly, $\mathcal{A}$'s view is identical in both cases, and likewise the output of the honest P_2 is x_0 in both the real and ideal executions. In the case that P_2 is corrupted, the simulator $\mathcal{S}$ sends $\sigma = 0$ to the trusted party and receives back x_0. Simulator $\mathcal{S}$ then internally simulates P_1 sending x_0 to $\mathcal{A}$. Here too, the view of $\mathcal{A}$ is identical in the real and ideal executions. This demonstrates why it is necessary to separately require that π computes f.

9 The Common Reference String Model – Oblivious Transfer

Until now, we have considered the *plain model* with no trusted setup. However, in some cases, a trusted setup is used to obtain additional properties. For example, a common reference string can be used to achieve non-interactive zero knowledge [7], which is impossible in the plain model. In addition, this is used to achieve security under composition, as will be discussed briefly in Section 10.1.

The common reference string model. Let M be a probabilistic-polynomial time machine that generates a common-reference string that is given to both parties. We remark that in the common *random* string model, $M(1^n)$ outputs a uniformly-distributed string of length poly(n), whereas in the common *reference* string the distribution can be arbitrary. Let CRS denote "common reference string".

In the CRS model, in the real model the parties are provided the same string generated by M, whereas in the ideal model the simulator chooses the string. Since the real and ideal models must be indistinguishable, this means that the CRS chosen by the simulator must be indistinguishable from the CRS chosen by M. However, this still provides considerable power to the simulator. For example, assume that the CRS contains an encryption key pk to a CCA-secure public-key encryption scheme. Then, in the real model, neither party knows the associated secret key. In contrast, since the simulator chooses the CRS, it can know the associated secret key and so can decrypt any ciphertext generated by the adversary.

The motivation behind this definition is that if an adversary can attack the protocol in the real model, then it can also attack the protocol in the ideal model with the simulator. The fact

that the simulator can choose the CRS does not change anything in this respect. Indeed, as we have discussed previously, the simulator *must* have additional power beyond that of a legitimate party. (Recall that in the context of zero knowledge, if there is no additional power then the zero-knowledge property will contradict the soundness property, since a cheating prover could run the simulator strategy.) Until now, we have considered a simulator that can rewind the adversary. In the CRS model, it is possible to construct a simulator that does *not* rewind the adversary, since its additional power is in choosing the CRS itself.

There are two ways to define security in the CRS model. The first is to include the CRS in the output distributions. Specifically, one can modify the REAL output distribution to include the CRS generated by M, the output of the adversary $\mathcal{A}$ and the output of the honest party. Then, the IDEAL output distribution includes the output of $\mathcal{S}$ (which include two parts – the CRS generated by $\mathcal{S}$ and the output of the adversary) and the output of the honest party. Alternatively, it is possible to define an ideal CRS functionality $f_{\mathsf{crs}}(1^n, 1^n) = (M(1^n), M(1^n))$. Then, one constructs a protocol and proves its security in the f_{crs}-hybrid model. As we have already seen, in the f-hybrid model, the simulator $\mathcal{S}$ plays the role of f in the simulation of the protocol in. Thus, this means that $\mathcal{S}$ can choose the CRS in the f_{crs}-hybrid model, as we have discussed.

Before proceeding to demonstrate the simulation technique in this model, we remark that the sequential composition theorem of Section 6.3 only holds when each execution of the protocol is independent. Thus, it is *not* possible to generate a single CRS and then run many sequential executions of the protocol using the same CRS, while relying on the composition theorem. Rather, it is either necessary to use a different CRS for each execution (not recommended) or it is necessary to explicitly prove that security holds for many execution. This can be done by defining a *multi-execution functionality* and then prove its security in the f_{crs}-hybrid model. For example, a multi-execution functionality for oblivious transfer could be defined as follows:

> **The multi-execution oblivious transfer $f_{\mathsf{m\text{-}ot}}$ works as follows:** Until one of the parties send end, repeat the following
>
> 1. Wait to receive (x_0, x_1) from P_1, and σ from P_2.
> 2. Send x_σ to P_2.

Typically, such functionalities are not defined in this way, since the CRS model is usually used in the context of *concurrent composition* where executions are run concurrently and not sequentially. In the concurrent setting, parties can send inputs whenever they wish. In order to match executions, a session identifier sid is used; specifically, P_1 sends (sid, x_0, x_1), P_2 sends (sid, σ) and then the functionality sends (sid, x_σ) to P_2. We discuss concurrent composition briefly in Section 10.1.

Oblivious transfer in the CRS model. In Section 8, we described an oblivious transfer protocol that was based on the protocol of Peikert et al. [35]. The original protocol in [35] was designed in the CRS model, and achieves universal composability (see Section 10.1). We can modify Protocol 8.2 to a *two-round protocol* in the CRS model by simply defining the CRS to be $(\mathbb{G}, q, g_0, g_1, h_0, h_1)$ where (g_0, g_1, h_0, h_1) is *not* a Diffie-Hellman tuple; see Protocol 9.1. We will prove that Protocol 9.1 is secure for a single oblivious transfer (we do not prove security under multiple executions since our aim is to demonstrate the use of the CRS and not to show the full power of the protocol).

PROTOCOL 9.1 (Oblivious Transfer [35])

- **Inputs:** Party P_1's input is a pair (x_0, x_1) and Party P_2's input is a bit σ. We assume for simplicity that $x_0, x_1 \in \mathbb{G}$ where $\mathbb{G}$ is defined in the CRS.
- **Auxiliary input:** Both parties hold a security parameter 1^n.
- **Hybrid functionality f_{crs}:** A group $\mathbb{G}$ of order q (of length n) with generator g_0 is sampled, along with three random elements $g_1, h_0, h_1 \in_R \mathbb{G}$ of the group. f_{crs} sends $(\mathbb{G}, q, g_0, g_1, h_0, h_1)$ to P_1 and P_2.
- **The protocol:**
  1. P_2 chooses a random value $r \in_R \mathbb{Z}_q$, computes $g = (g_\sigma)^r$ and $h = (h_\sigma)^r$, and sends (g, h) to P_1.
  2. P_1 computes $(u_0, v_0) = RAND(g_0, g, h_0, h)$, and $(u_1, v_1) = RAND(g_1, g, h_1, h)$. P_1 sends P_2 the values (u_0, w_0) where $w_0 = v_0 \cdot x_0$, and (u_1, w_1) where $w_1 = v_1 \cdot x_1$.
  3. P_2 computes $x_\sigma = w_\sigma / (u_\sigma)^r$.
  4. P_1 outputs λ and P_2 outputs x_σ.

Theorem 9.2 *Assume that the Decisional Diffie-Hellman problem is hard relative to the group sampling algorithm used by f_{crs}. Then, Protocol 9.1 securely computes f_{ot} in the presence of malicious adversaries in the f_{crs}-model.*

Proof Sketch: The proof here is very similar to that of Theorem 8.2. In particular, the fact that Protocol 9.1 computes f_{ot} follows from exactly the same computation.

In the case that P_1 is corrupted, the simulator $\mathcal{S}$ in the proof of Theorem 8.2 chose (g_1, h_0, h_1) so that (g_0, g_1, h_0, h_1) *is* a Diffie-Hellman tuple. Given this fact, and given that it knows y such that $g_1 = (g_0)^y$, simulator $\mathcal{S}$ was able to extract both x_0, x_1 from $\mathcal{A}$. Now, in this case, $\mathcal{S}$ chooses the CRS so that (g_0, g_1, h_0, h_1) *is* a Diffie-Hellman tuple. Also, since it chooses g_1 it knows y such that $g_1 = (g_0)^y$. Thus, $\mathcal{S}$ internally hands this (g_0, g_1, h_0, h_1) to $\mathcal{A}$ when $\mathcal{A}$ calls f_{crs}, as if it was generated by f_{crs}. From then on, $\mathcal{S}$ uses the exact same strategy as the simulator in the proof of Theorem 8.3. The proof of indistinguishability works in exactly the same way.

In the case that P_2 is corrupted, the simulator $\mathcal{S}$ in the proof of Theorem 8.2 was able to extract $\mathcal{A}$'s input σ using the witness α (where $h_0 = (g_0)^\alpha$). Simulator $\mathcal{S}$ obtained α from $\mathcal{A}$'s message to $f_{\mathsf{zk}}^{R_L}$. In this case, $\mathcal{S}$ chooses the CRS. Thus, it generates (g_0, g_1, h_0, h_1) as in the protocol specification and takes α where $h_0 = (g_0)^\alpha$. $\mathcal{S}$ then uses α exactly as the simulator in the proof of Theorem 8.2 in order to extract σ. (Observe that in Protocol 8.2, $h_1 = (g_0)^{\alpha+1}$. This makes the tuple not a Diffie-Hellman tuple, but not a random one either. In contrast, here the tuple is random. Nevertheless, any non-Diffie-Hellman tuple suffices, and the simulator in the proof of Theorem 8.2 only needs the discrete log α of h_0 to base g_0 in order to extract. Specifically, if $h = g^\alpha$ then it determines that the input is $\sigma = 0$, and otherwise it is $\sigma = 1$. This remains the same when h_1 is taken to be a random element.) Based on the above, $\mathcal{S}$ chooses (g_0, g_1, h_0, h_1) as described above, and hands it to $\mathcal{A}$ when it calls f_{crs}. Beyond that, $\mathcal{S}$ works in exactly the same way as $\mathcal{S}$ in the proof of Theorem 8.3. ■

10 Advanced Topics

In this section, we briefly mention some advanced topics, and include pointers for additional reading.

10.1 Composition and Universal Composability

In this tutorial, we focused on the stand-alone model. As discussed in Section 6.3, this implies security under sequential composition. However, in the real-world setting, many secure and insecure protocols are run concurrently, and it is desirable to have security in this setting. The definition of security presented in Section 6 does *not* guarantee security under concurrent composition. There are a number of definitions that have been proposed that achieve this level of security. The most popular is that of universal composability (UC) [9]. This definition expands upon the definition of Section 6 by adding an *environment machine* which is essentially an interactive distinguisher. The environment writes the inputs to the parties' input tapes and reads their outputs. In addition, it externally interacts with the adversary throughout the execution. The environment's "goal" is to distinguish between a real protocol execution and an ideal execution. One very important artifact of this definition is that the simulator can no longer rewind the adversary in the simulation. This is because the real adversary can actually do nothing but fulfill the instructions of the environment. Now, since the environment is an *external* machine that the real and ideal adversaries interact with, this means that the simulator has to simulate for an external adversary. Due to this, rewinding is not possible, and it actually follows that without an honest majority it is impossible to securely compute a large class of functionalities in the UC framework in the plain model without any trusted setup [13]. However, given a trusted setup, like a common reference string as in Section 9, it is possible to securely compute any functionality for any number of corrupted parties under the UC definition [14]. Indeed, the oblivious transfer protocol described in Section 9 has been proven secure in the UC framework [35].

The general UC framework is rather complex, as it enables one to model almost any task and any setting. In case one is interested in standard secure computation tasks, without guaranteeing fairness, it is possible to use the simpler equivalent formalization described in [10].

10.2 Proofs in the Random Oracle Model

In many cases, the random oracle model is used to gain higher efficiency or other properties otherwise unobtainable. The setting of secure computation is no exception. However, beyond its inherent heuristic nature [12], there are some very subtle definitional issues here that must be considered. One issue that arises is whether or not the distinguisher obtains access to the random oracle, and if yes, how. If the distinguisher does not have any access, then this is a very weak definition, and sequential composition will not be guaranteed. If we provide the distinguisher with the same randomly chosen oracle as the parties and the (real and ideal) adversary, then we obtain a *non-programmable* random oracle [33] which may not be strong enough. A third alternative is to provide the distinguisher with the random oracle, but in the ideal world to allow the simulator to still control the oracle. This is a somewhat strange formulation, but something of this type seems necessary in some cases.

In the UC framework, the random oracle can be modeled as an ideal functionality computing a random function. This matches the third alternative in some sense, since the simulator controls the oracle in the case of an ideal execution. It is somewhat different, however, since the environment –

who plays the distinguisher – cannot directly access the oracle.

We will not do more in this tutorial than point out that these issues exist and need to be dealt with carefully if the random oracle is to be used in the context of secure computation. We recommend reading [33] for a basic treatment of modeling random oracles in secure computation, and [39, 40] for a treatment of the issue of oracle-dependent auxiliary input (and more). We conclude by remarking that in [34], it is pointed out that other properties that are sometimes expected (like deniability) are not necessarily obtained in the random oracle model. In many cases of standard secure computation, this is not needed. However, this is another example of why the random oracle model needs to be treated with great care in these settings.

10.3 Adaptive Security

In this tutorial we have considered only the case of *static adversaries* where the subset of corrupted parties is fixed before the protocol execution begins. In contrast, an *adaptive adversary* can choose which parties to corrupt throughout the protocol, based on the messages viewed. A classic example of a protocol that is secure for static adversaries and not for adaptive adversaries is as follows. Consider a very large number of parties (say, linear in the security parameter n), and consider a protocol which begins by securely choosing a random subset of the parties who then carry out the computation for the rest. Assume that the adversary is limited to corrupting a constant fraction of the parties, and assume that $\sqrt{n}$ parties are chosen to compute the result. Then, except with negligible probability, there will be at least one honest party in the chosen $\sqrt{n}$. Thus, as long as a protocol that is secure for any number of corrupted parties is used, we have that security is preserved. This is true for the case of static adversaries. However, an adaptive adversary can wait until the $\sqrt{n}$ parties are chosen, and then adaptively corrupt all of them. Since it only corrupts a constant fraction (less than half for $n > 6$), this is allowed. Clearly, such an adversary completely breaks the protocol, since it controls all the parties who carry out the actual computation.

In order to provide security for such adversaries, it is necessary to be able to simulate even when an adversary corrupts a party midway. The challenge that this raises is that when an adversary corrupts a real party in the middle of an execution, then it obtains its current state. Thus, the simulator must be able to generate a transcript – without knowing a party's input – and later be able to "explain" that transcript as a function of an honest party's instructions on its input, where the input is provided later (upon corruption).

There are two main models that have been considered for the case of adaptive adversaries. In the first, it is assumed that parties cannot securely erase data; this is called the no erasures model. Thus, the adversary obtains the party's entire view – its input, random tape and incoming messages – upon corruption. This forces the simulator to generate such a view, *after* having generated (at least part of) the protocol transcript. Amongst other things, this means that a transcript has to match all possible inputs, and so it must be *non committing*. See [11] for a basic treatment and constructions in the case of an honest majority, see [8] for a definitional treatment in the stand-alone model, and see [14] for constructions in the case of no honest majority.

A weaker model of adaptive security is one which assumes that parties *can* securely erase data; this is called the erasures model. In this case, it is possible for parties to erase some of their data. This makes simulation easier since it is not necessary to generate the entire view, but only the current state. See [3] for a very efficient solution for the case of an honest majority, and see [29] for an example of a two-party protocol that is adaptively secure with erasures. These examples

demonstrate why the erasures model is easier to work with.[11]

Acknowledgements

I would like thank Ben Riva for suggesting that such a tutorial would be very useful, and Oded Goldreich whose years of guidance and education – and his personal example – motivated me to actually write it. Thank you also to my students Ran Cohen, Omer Shlomovits and Avishay Yanay who provided helpful comments on the write up.

References

[1] B. Barak. How to Go Beyond the Black-Box Simulation Barrier. In 42*nd FOCS*, pages 106–115, 2001.

[2] B. Barak and Y. Lindell. Strict Polynomial-Time in Simulation and Extraction. *SIAM Journal on Computing,* 33(4):783–818, 2004. (Extended abstract in the 34*th STOC*, 2002.)

[3] D. Beaver and S. Haber. Cryptographic Protocols Provably Secure Against Dynamic Adversaries. In *EUROCRYPT'92,* Springer-Verlag (LNCS 658), pages 307–323, 1992.

[4] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. In the *38th Symposium on Foundations of Computer Science* (FOCS), 1997.

[5] M. Blum. Coin Flipping by Phone. *IEEE Spring COMPCOM*, pages 133–137, 1982.

[6] M. Blum. How to Prove a Theorem So No One Else Can Claim It. *Proceedings of the International Congress of Mathematicians*, pages 1444–1451, 1986111.

[7] M. Blum, P. Feldman and S. Micali. Non-Interactive Zero-Knowledge and its Applications. In 20*th STOC,* pages 103–112, 1988.

[8] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[9] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In the 42*nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[10] R. Canetti, A. Cohen and Y. Lindell. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *CRYPTO 2015,* Springer (LNCS 9216), pages 3–22, 2015.

[11] In my personal subjective opinion, there has been too much focus on the *no erasures* model. Since achieving security without erasures is so difficult, this has unnecessarily dissuaded people from working in the adaptive model at all, especially when considering efficient protocols. The community would be better served by constructing protocols that are secure with erasures than just those that are secure under static adversaries, since the former captures a very realistic attack threat. Unfortunately, however, it seems that such work is doomed since most people seem to either not care about adaptive security at all, or if they do care then they consider the erasures model to be too weak.

[11] R. Canetti, U. Feige, O. Goldreich and M. Naor. Adaptively Secure Multi-Party Computation. In 28*th STOC*, pages 639–648, 1996.

[12] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited. In the *Journal of the ACM*, 51(4):557–594, 2004. (An extended abstract appeared in the 30*th STOC*, 1998.)

[13] R. Canetti, E. Kushilevitz and Y. Lindell. On the Limitations of Universal Composable Two-Party Computation Without Set-Up Assumptions. *Journal of Cryptology,* 19(2):135-167, 2006. (Extended abstract appeared at *EUROCRYPT 2003.*)

[14] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and Multi-Party Computation. In 34*th STOC*, pages 494–503, 2002. Full version available at `http://eprint.iacr.org/2002/140`.

[15] R. Cleve. Limits on the Security of Coin Flips when Half the Processors are Faulty. In 18*th STOC,* pages 364–369, 1986.

[16] S. Even, O. Goldreich and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM,* 28(6):637–647, 1985.

[17] O. Goldreich. *Foundations of Cryptography Vol. I – Basic Tools.* Cambridge University Press, 2001.

[18] O. Goldreich. *Foundations of Cryptography Vol. II – Basic Applications.* Cambridge University Press, 2004.

[19] O. Goldreich. On Expected Probabilistic Polynomial-Time Adversaries: A Suggestion for Restricted Definitions and Their Benefits. In the *Journal of Cryptology*, 23(1):1–36, 2010. (Extended abstract appeared in TCC 2007.)

[20] O. Goldreich and A. Kahan. How To Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Journal of Cryptology*, 9(3):167–190, 1996.

[21] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, 25(1):169–192, 1996.

[22] 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*, 38(1):691–729, 1991.

[23] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Journal of Cryptology,* 7(1):1–32, 1994.

[24] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Science*, 28(2):270–299, 1984.

[25] S. Goldwasser, S. Micali and C. Rackoff The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing,* 18(1):186–208, 1989.

[26] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions.* Springer, 2010.

[27] C. Hazay and Y. Lindell. A Note on Zero-Knowledge Proofs of Knowledge and the ZKPOK Ideal Functionality. *Cryptology ePrint Archive: Report 2010/552*, 2010.

[28] J. Katz and Y. Lindell. Handling Expected Polynomial-Time Strategies in Simulation-Based Security Proofs. In the *Journal of Cryptology*, 21(3):303–349, 2008. (An extended abstract appeared in TCC 2005.)

[29] Y. Lindell. Adaptively Secure Two-Party Computation with Erasures. In *CT-RSA*, Springer (LNCS 5473), pages 117–132, 2009. Full version in the *Cryptology ePrint Archive*, Report 2009/031.

[30] Y. Lindell. A Note on Constant-Round Zero-Knowledge Proofs of Knowledge. In the *Journal of Cryptology,* 26(4):638-654, 2013.

[31] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In the *Journal of Cryptology*, 28(2):312350, 2015. (Extended abstract in *EUROCRYPT 2007.*)

[32] M. Naor and O. Reingold. Number-Theoretic Constructions Of Efficient Pseudo-Random Functions. In *Journal of ACM*, 51(2):231–262, 2004.

[33] J.B. Nielsen. Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case. In *CRYPTO 2002*, Springer (LNCS 2442), pages 111–126, 2002.

[34] R. Pass. On Deniability in the Common Reference String and Random Oracle Model. In *CRYPTO 2003*, Springer (LNCS 2729), pages 316–337, 2003.

[35] C. Peikert, V. Vaikuntanathan and B. Waters. A Framework for Efficient and Composable Oblivious Transfer. In *CRYPTO 2008*, Springer (LNCS 5157), pages 554–571, 2008.

[36] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981. (See *Cryptology ePrint Archive: Report 2005/187.*)

[37] A. Rosen. A Note on Constant-Round Zero-Knowledge Proofs for NP. In *TCC 2004*, Springer (LNCS 2951), pages 191–202, 2004.

[38] V. Shoup. Sequences of Games: A Tool for Taming Complexity in Security Proofs. *Cryptology ePrint Archive, Report 2004/332*, 2004.

[39] D. Unruh. Random Oracles and Auxiliary Input. In *CRYPTO 2007*, Springer (LNCS 4622), pages 205-223, 2007.

[40] H. Wee. Zero Knowledge in the Random Oracle Model, Revisited. In *ASIACRYPT 2009*, Springer (LNCS 5912), pages 417–434, 2009.