Title

_____

A Thesis

Presented to

The Established Interdisciplinary Committee

for Mathematics and Computer Science

Mathematics and Natural Sciences

Reed College

_____

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Arts

_____

Riley Shahar

May 200x

Approved for the Committee
(Mathematics and Computer Science)

_____   _____

Angélica Osorno   Adam Groce

Table of Contents

Chapter 1

Cryptography

[TODO: An introduction to the chapter; cite [KL14; PS10; Ros21].]

1.1 Foundations

1.1.1 One-Way Functions

Many cryptographic protocols rely on *one-way functions*, which are informally functions that are easy to compute, but hard to invert. The former notion is easy to formalize in terms of time complexity, but the latter is more difficult. We typically ask that any "reasonably efficient" algorithm—called the *adversary*—attempting to invert the function has a negligible chance of success.

In computer science, we generally assume that algorithms are efficient if and only if they are polynomial-time; this assumption has been borne out by decades of practice. This motivates our definition of a "negligible" chance: we say that a function f is *negligible* if $f(n) = o(n^{-k})$ for every k; in other words, if it is asymptotically smaller than any rational function. In this case, we write $f = \mathrm{negl}(n)$ or just $f = \mathrm{negl}$.) The set of negligible functions has all the nice closure properties we expect; in particular, the sum of negligible functions is negligible.

Notation. We will use PPT as shorthand for probabilistic polynomial-time. When we refer to an *adversary*, *distinguisher*, or *simulator*, we always mean a non-uniform PPT algorithm.

Definition 1.1 (one-way function). A function f is *one-way* if:

- (easy to compute) f is PPT-computable;
- (hard to invert) for any adversary $\mathcal{A}$, natural number n, and uniform random choice of input x such that $|x| = n$,

$$\Pr[f(\mathcal{A}(1^n, f(x))) = f(x)] = \mathrm{negl}(n).$$

Note that $|x|$ here is *not* the absolute value, but is instead the length of x as a binary string: if x is a number, then by encoding in binary have that $|x| = \Theta(\log_2 x)$.

The idea is that, given $y = f(x)$, $\mathcal{A}$ attempts to find some x' such that $f(x') = y$. If some adversary can do this with non-negligible probability, then the function is not one-way. While the probability must be negligible in $|x|$, the adversary is given $f(x)$ and 1^n as an input, and hence must run polynomially only in $|f(x)| + n$. This is a common technique called *padding*, wherein algorithms are given an extra input of 1^n to ensure they have enough time to run.

We do not know that one-way functions exist. In fact, while the existence of one-way functions implies that $P \neq NP$, the converse is not known[1]. However, as in the following examples, we have excellent candidates under fairly modest assumptions.

Example 1.2 (Factoring [PS10, subsection 2.3]). Suppose that for any adversary $\mathcal{A}$ and for uniform random choice of primes $p, q < 2^n$,

$$\Pr[\mathcal{A}(pq) = \{p, q\}] = \mathsf{negl}(n).$$

This is the *factoring hardness assumption*, for which there is substantial evidence. Then $(x, y) \mapsto xy$ is one-way[2].

Example 1.3 (Discrete Logarithm [KL14, subsection 8.3.2]). Let $\{G_n\}$ be a fixed sequence of finite groups. The *discrete logarithm hardness assumption* for $\{G_n\}$ is that, for any adversary $\mathcal{A}$ and for uniform random choice of $g \in G_n$ and $h \in \langle g \rangle$ such that $h = g^k$,

$$\Pr[\mathcal{A}(g, h) = k] = \mathsf{negl}(n).$$

Under the discrete logarithm hardness assumption, $(g, k) \mapsto g^k$ is one-way.

The discrete logarithm hardness assumption is known to be false for certain groups, such as the additive groups $\mathbb{Z}_p$ for prime p, in which case $g^k = gk$ and the Euclidean algorithm solves the problem. However, it is believed to hold for groups such as $\mathbb{Z}_p^*$. For a survey of various versions of this assumption, see [SS02].

1.1.2   Proofs by Reduction

Many cryptographic definitions, including Definition 1.1, take the form

for any adversary $\mathcal{A}$, natural number n, and uniform random choice of input x such that $|x| = n$, some predicate on the output of $\mathcal{A}$ has negligible probability.

The basic technique for proving results using these definitions is called *proof by reduction*. The idea is to reduce one problem into another by starting with an arbitrary adversary attacking the second and constructing an adversary attacking the first, such that the probability of their successes is related. If we assume the first problem is hard, then by studying the structure of the reduction we can learn about the hardness of the second problem. As such, we often say that reductions prove *relative hardness results*, so that for instance Example 1.4 below proves the hardness of g relative to f.

More specifically, to prove hardness of a problem Π relative to Π', a proof by reduction generally goes as follows:

---

[1][Imp95] gives a classic discussion of the implications of various resolutions to P vs. NP on cryptography, including the case where $P \neq NP$ but one-way functions nevertheless do not exist.

[2]This statement is slightly imprecise: technically, $(x, y) \mapsto xy$ is *weakly one-way*; to get the stronger notion of Definition 1.1 requires a process called *hardness amplification*. See [PS10, Section 2.4] for details.

1. Fix an arbitrary adversary $\mathcal{A}$ attacking a problem Π.

2. Construct an adversary $\mathcal{A}'$ attacking a problem Π' which:

   (a) Receives an input x' to Π'.

   (b) Translates x' into an input x to Π.

   (c) Simulates $\mathcal{A}(x)$, getting back an output y which solves $\Pi(x)$.

   (d) Translates y into an output y' which solve $\Pi(x')$.

3. Analyze the structure of the translations to conclude that $\mathcal{A}'$ solves Π' with probability related to that with which $\mathcal{A}$ solves Π.

4. Given the hardness assumptions on Π', conclude relative hardness of Π.

The point is that $\mathcal{A}'$'s job is to "simulate" the problem Π to $\mathcal{A}$, using the data it gets from Π' to construct an input to Π. We illustrate this concept now.

Example 1.4 (a straightforward proof by reduction [PS10, subsection 2.4.1])**.** Let f be a one-way function. Then we claim $g : (x, y) \mapsto (f(x), f(y))$ is a one-way function. We can compute g in polynomial time by computing f twice, so it remains to show that g is hard to invert.

Let $\mathcal{A}$ be any adversary. We will construct an adversary $\mathcal{A}'$ such that, if $\mathcal{A}$ can non-negligibly invert g, then $\mathcal{A}'$ can non-negligibly invert f.

The adversary $\mathcal{A}'$ takes input 1^n and y. It then uniformly randomly chooses u of length n and computes $v = f(u)$, which is possible because f is easy to compute. Now $\mathcal{A}'$ computes $(u', x') := \mathcal{A}(1^{2n}, (v, y))$ and outputs x'.

When $\mathcal{A}'$ simulates $\mathcal{A}$, it passes v, which is $f(u)$ for a uniform random u, and y, which is (on well-formed inputs) $f(x)$ for a uniform random x. Thus, this looks like exactly the input that $\mathcal{A}$ would "expect" to receive if it is attempting to break g. As such, whenever $\mathcal{A}$ successfully inverts g, $\mathcal{A}'$ successfully inverts f. Since everything is uniform we may pass to probabilities, and so:

$$
\begin{aligned}
&\Pr[g(\mathcal{A}(1^{2n}, g(u, x))) = g(u, x)] \\
&\quad = \Pr[g(\mathcal{A}(1^{2n}, (f(u), f(x)))) = (f(u), f(x))] \quad \text{by definition of } g \\
&\quad \leq \Pr[f(\mathcal{A}'(1^n, f(x))) = f(x)] \qquad\qquad\qquad \text{by the above argument} \\
&\quad = \mathsf{negl}(n) \qquad\qquad\qquad\qquad\qquad\qquad \text{by the hardness assumption for } f.
\end{aligned}
$$

Thus g is one-way.

Comparing this example to the above schema, we see that the problem Π' is to invert f, while the problem Π is to invert g. The input x' to Π' is y, while the computed input x to Π is (v, y). The output y of $\mathcal{A}$ is (x', u'), while the computed output y' is x'.

Diagramatically, we can represent the algorithm $\mathcal{A}'$ as follows:
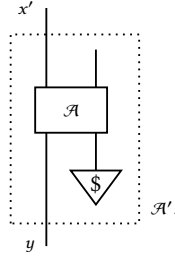
While this is not standard notation in cryptography, it will be useful for our future purposes. We read these diagrams—called *circuit* or *string diagrams*—from bottom to top. This diagram says that $\mathcal{A}'$ is an algorithm which takes y, uniformly randomly generates another input (this is what the $ means), calls $\mathcal{A}$, and returns its first output.

1.1.3   Computational Indistinguishability

Computational indistinguishability formalizes the notion of two probability distributions which "look the same" to adversarial processes. We begin with probability distributions, but because we want to do asymptotic analysis, we will eventually need to switch to working with sequences of probability distributions.

Definition 1.5 (computational advantage)**.** Let X and Y be probability distributions. The *computational advantage* of an adversary $\mathcal{D}$, called the *distinguisher*, over X and Y is

$$\mathrm{ca}_{\mathcal{D}}(X, Y) = \left| \Pr_{x \leftarrow X}[\mathcal{D}(x) = 1] - \Pr_{y \leftarrow Y}[\mathcal{D}(y) = 1] \right|.$$

The idea is that the distinguisher $\mathcal{D}$ is trying to guess whether its input was drawn from X or Y; the computational advantage is how often it can do so.

Proposition 1.6. *Let $\mathcal{D}$ be a fixed distinguisher. Then* $\mathrm{ca}_{\mathcal{D}}$ *is a pseudometric*[3] *on the space of probability distributions over an underlying set A.*

Proof. Symmetry and non-negativity are immediate from the definition, while the triangle inequality follows from the triangle inequality for real numbers.   □

We now turn to the asymptotic case.

Definition 1.7 (probability ensemble)**.** A *probability ensemble* is a sequence $\{X_n\}$ of probability distributions.

We say that two ensembles are computationally indistinguishable if there is no efficient way to tell between them. Formally:

---

[3]A *pseudometric* on a space X is a function $d : X \times X \to \mathbb{R}_{\geq 0}$ which is zero on identical points, symmetric, and satisfies the triangle inequality; in other words, it is a metric which does not necessarily differentiate distinct points.

Definition 1.8 (computational indistinguishability). Two probability ensembles $\{X_n\}$ and $\{Y_n\}$ are *computationally indistinguishable* if for any (non-uniform PPT) distinguisher $\mathcal{D}$ and any natural number n,

$$\mathrm{ca}_{\mathcal{D}}(X_n, Y_n) = \mathrm{negl}(n).$$

In this case, we write $\{X_n\} \overset{c}{\equiv} \{Y_n\}$.

Remark 1.9. A natural thought is to define a metric on probability distributions by

$$\mathrm{ca}(X, Y) = \sup_{\mathcal{D}} \mathrm{ca}_{\mathcal{D}}(X, Y),$$

and extend to ensembles by asking that $\mathrm{ca}(X_n, Y_n) = \mathrm{negl}(n)$. Unfortunately, this does not quite yield the correct notion, as there exist ensembles which are computationally indistinguishable, but have sequences of distinguishers whose advantages for any fixed n converge to 1.

Proposition 1.10. *Computational indistinguishability is an equivalence relation on the space of probability ensembles over a fixed set A.*

Proof. Reflexivity and symmetry follow from the case of distributions. To show transitivity, let $\{X_n\} \overset{c}{\equiv} \{Y_n\}$ and $\{Y_n\} \overset{c}{\equiv} \{Z_n\}$. Let $\mathcal{D}$ be any distinguisher. Then for any n,

$$
\begin{aligned}
\mathrm{ca}_{\mathcal{D}}(X_n, Z_n) &\leq \mathrm{ca}_{\mathcal{D}}(X_n, Y_n) + \mathrm{ca}_{\mathcal{D}}(Y_n, Z_n) && \text{by the triangle inequality} \\
&= \mathrm{negl}(n) + \mathrm{negl}(n) && \text{by assumption} \\
&= \mathrm{negl}(n). && \square
\end{aligned}
$$

It is necessary to be precise about what is being claimed here. Transitivity states that for any *constant, finite sequence* of probability ensembles, if each is computationally indistinguishable from its neighbors, then the two ends of the sequence are computationally indistinguishable. In cryptography, we sometimes want to consider the more general case of a countable sequence of probability ensembles. We can do slightly better than the previous result:

Proposition 1.11. *Let $\{X^k\}$ be a sequence of probability ensembles, so that each $X^k = \{X_n^k\}$ is itself a sequence of probability distributions. Let $\{X^i\} \overset{c}{\equiv} \{X^{i+1}\}$ for each i. Let $\{Y_n = X_n^{K(n)}\}$ for some polynomial K. Then $\{X_n^1\} \overset{c}{\equiv} \{Y_n\}$.*

Proof. Let $\mathcal{D}$ be any distinguisher. Then for any n,

$$
\begin{aligned}
\mathrm{ca}_{\mathcal{D}}(X_n^1, Y_n) &= \mathrm{ca}_{\mathcal{D}}(X_n^1, X_n^{K(n)}) \\
&\leq \mathrm{ca}_{\mathcal{D}}(X_n^1, X_n^2) + \cdots + \mathrm{ca}_{\mathcal{D}}(X_n^{K(n)-1}, X_n^{K(n)}) \\
&= K(n)\mathrm{negl}(n) \\
&= \mathrm{negl}(n).
\end{aligned}
$$

In particular, the last equality follows because K is polynomial. $\square$

On the other hand, the result does not hold for arbitrary K. As we will see, this is a fundamental limitation for cryptographic composition: we only expect composition to work up to polynomial bounds.

One more closure result is valuable:

Proposition 1.12. *Let $\{X_n\} \stackrel{c}{\equiv} \{Y_n\}$, and let $\mathcal{M}$ be a non-uniform PPT algorithm. Then $\{\mathcal{M}(X_n)\} \stackrel{c}{\equiv} \{\mathcal{M}(Y_n)\}$.*

Proof. The proof is by reduction. Let $\mathcal{D}$ be a distinguisher. Then construct $\mathcal{D}'$ which, on input x, simulates $\mathcal{D}(\mathcal{M}(x))$. Then $\mathcal{D}'$ outputs 1 on x if and only if $\mathcal{D}$ outputs 1 on $\mathcal{M}(x)$, so

$$\text{ca}_{\mathcal{D}}(\mathcal{M}(X_n), \mathcal{M}(Y_n)) = \text{ca}_{\mathcal{D}'}(X_n, Y_n) = \text{negl}(n)$$

by the computational indistinguishability assumption. □

Example 1.13 (pseudorandom generators [PS10, Sections 3.2-3.3])**.** We can use computational indistinguishability to formalize the notion of pseudorandomness.

Let $\{\mathcal{X}_n\}$ be a sequence of spaces, and let $\{X_n\}$ be a sequence of probability distributions. We say that $\{X_n\}$ is *pseudorandom for $\mathcal{X}$* if there exists a polynomial p such that

$$\{X_n\} \stackrel{c}{\equiv} \{\mathcal{X}_{p(n)}\},$$

where the latter is equipped with the uniform distribution. In other words, pseudorandom ensembles look uniformly random to distinguishers.

For simplicity, we now work over $\mathcal{X}_n = \mathbb{Z}_2^n$. Let $G : \mathbb{Z}_2^* \to \mathbb{Z}_2^*$ be a *deterministic* function. We say G is a *pseudorandom generator* if

- G is polynomial-time computable;
- for any x, $|G(x)| > |x|$;
- $\{G(\mathbb{Z}_2^n)\}$ is pseudorandom.

The idea is that G gets some input $x \in \mathbb{Z}_2^n$ and produces an output in $\mathbb{Z}_2^{p(n)}$ which looks uniformly random to distinguishers if x is chosen uniformly at random. The polynomial $p(n)$ in the definition of pseudorandomness is now called the *expansion factor*. It this sense, pseudorandom generators allow us to "bootstrap" randomness from random draws even on very small inputs.

As usual, while we have excellent candidates, we have no proof that pseudorandom generators exist. However, there is a known procedure, due to Håstad et al. [Hås+99], to turn any one-way function into a pseudorandom generator.

1.1.4   Interactive and Zero-Knowledge Computation

Cryptographic protocols do not occur in a vacuum; instead, they rely on computations involving multiple parties. We call such situations *interactive computations*. In general, a model of interaction depends on the underlying model of computation; this is for instance the case with the popular notion of interactive Turing machines [Gol01, Definition 4.2.1]. As our approach in this chapter has been model-independent, we can only give an informal discussion of interaction.

An *interactive computation* consists of a finite number of *parties*, which we think of as algorithms $\mathcal{A}_i$, who may potentially communicate by sending messages to each other, and whose behavior may change in response to messages they receive. An *interactive protocol* just consists of descriptions of some interactive algorithms $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$.

We often think of interactive computations as being indexed by a *security parameter* $n \in \mathbb{N}$. Instead of asking each algorithm to be polynomial-time in its inputs, we ask it to be polynomial in n, with the stipulation that the inputs themselves are no more than polynomial in n, so that each algorithm has time to read its own inputs. Intuitively, the security parameter represents a "tuning" of the security of the system, so that a bigger n incurs greater computational cost but gives stronger security guarantees. Often, the security parameter is formalized by ensuring that all parties get an extra input of 1^n at the start of the computation, as we did in Definition 1.1; we assume this formalization in every protocol we give here.

At the start of an interactive computation, there is a *global input x* known to all parties, and each party $\mathcal{A}_i$ may have a *private* or *auxiliary input* x_i known only to itself. We generally assume that there are known sequences of *input spaces* X_n and X_n^i, such that when the security parameter is n, $x \in X_n$ and $x_i \in X_n^i$. At the end of the computation, each party may make some output, the sequence of which we denote $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle (x, x_1, \ldots, x_N)$, so that party i's output is $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle (x, x_1, \ldots, x_N)_i$. When any of these algorithms are potentially probabilistic, we think of this value as a distribution over possible outputs, and we always assume that the internal randomness of the parties is independent.

Example 1.14. Here is a simple interactive protocol. We have two algorithms, $\mathcal{A}$ and $\mathcal{B}$. The input spaces are $X_n^{\mathcal{A}}$ and $X_n^{\mathcal{B}}$; there is no global input (which means the global input is just the security parameter 1^n.) The algorithm $\mathcal{A}$ takes its input $x \in X_n^{\mathcal{A}}$, sends it to $\mathcal{B}$, and outputs the first message it recieves from $\mathcal{B}$. The algorithm $\mathcal{B}$ takes its input $y \in X_n^{\mathcal{B}}$, sends it to $\mathcal{A}$, and outputs the first message it recieves from $\mathcal{A}$. Then we have that $\langle \mathcal{A}, \mathcal{B} \rangle (x, y) = (y, x)$.

The *view* of a party is roughly all of the information it has available to it over the course of the computation. This includes the global input, its private input, any random bits it uses, and all the messages it receives. We denote the view of party i by $\text{view}_i^{\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle}(x, x_1, \ldots, x_N)$. When the algorithms are clear from context, we may omit the superscript. Importantly, while each private input x_k is a parameter of each view view_i, the view does not necessarily include each of these inputs; they are parameters merely because they may affect the messages received by party i.

Example 1.15. In the protocol of Example 1.14,

$$\text{view}_{\mathcal{A}}^{\langle \mathcal{A}, \mathcal{B} \rangle}(1^n, x, y) = \text{view}_{\mathcal{B}}^{\langle \mathcal{A}, \mathcal{B} \rangle}(1^n, x, y) = \{1^n, x, y\}.$$

Suppose that $\mathcal{B}'$ always sends the string 0^n to $\mathcal{A}$, instead of its input. Then

$$\text{view}_{\mathcal{A}}^{\langle \mathcal{A}, \mathcal{B}' \rangle}(1^n, x, y) = \{1^n, x, 0\}, \qquad \text{view}_{\mathcal{B}'}^{\langle \mathcal{A}, \mathcal{B}' \rangle}(1^n, x, y) = \{1^n, x, y\}.$$

Notice that $\text{view}_{\mathcal{B}'}$ does not include the messages which it sends $\mathcal{A}$.

The *running time* of an interactive algorithm $\mathcal{A}$ is now the function $T_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ which, for any n, gives the maximum number of "steps" it takes $\mathcal{A}$ to halt over any choice of:

- global input x and private input y of total length $n = |x| + |y|$;
- other algorithms involved in the computation;
- internal randomness of $\mathcal{A}$ and of any other algorithms involved in the computation.

Essentially, when we say an algorithm is polynomial-time, we mean it is *always* polynomial-time, no matter what. We sometimes assume that each algorithm has a "clock" that it uses to count the number of steps it has taken and ensure it halts in some fixed polynomial number of steps.

We can now formalize the idea of a party "learning something" from an interaction. We say that an interactive protocol $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ is *zero-knowledge for party i* if there exists a non-uniform PPT algorithm $\mathcal{S}$ such that for any choice of inputs $(x, x_1, \ldots, x_N)$,

$$\mathcal{S}(x, x_i) \overset{c}{\equiv} \mathrm{view}_i^{\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle}(x, x_1, \ldots, x_N).$$

The idea is that the "simulator" $\mathcal{S}$ gets only the inputs to $\mathcal{A}_i$ and is responsible for producing a distribution that is indistinguishable from the actual view of $\mathcal{A}_i$. If they can do this, then $\mathcal{A}_i$ must not have learned anything that they could not have computed directly from their inputs.

More often, we want to consider the situation where $\mathcal{A}_i$ is supposed to learn *something* from the computation, but should not learn anything *extra*.

Definition 1.16 (zero-knowledge)**.** Let f be a function. An interactive protocol $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ is *zero-knowledge for party i relative to f* if there exists a (non-uniform PPT) simulator $\mathcal{S}$ such that for any choice of inputs $(x, x_1, \ldots, x_N)$,

$$\mathcal{S}(x, x_i, f(x, x_1, \ldots, x_N)) \overset{c}{\equiv} \mathrm{view}_i^{\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle}(x, x_1, \ldots, x_N).$$

In the above definition, we are asking that the simulator produces a distribution which is negligibly close, in the sense of computational indistinguishability, to the actual view. While this is all that is possible in many situations in practice, we could ask for the stronger condition that the produced distribution is *identical* to the view. We call this notion *perfect* or *information-theoretic* zero-knowledge, and refer to Definition 1.16 as *computational* zero-knowledge when we wish to emphasize the distinction.

Example 1.17. We show that the *trivial protocol*, in which two algorithms $\mathcal{A}$ and $\mathcal{B}$ do nothing, is zero-knowledge for $\mathcal{B}$. Our goal is to give a simulator $\mathcal{S}$ such that for any choice of security parameter n,

$$\mathcal{S}(1^n) \overset{c}{\equiv} \mathrm{view}_{\mathcal{B}}^{\langle \mathcal{A}, \mathcal{B} \rangle}(1^n).$$

Since $\mathcal{B}$ never gets sent any messages, its view is just the input 1^n. We therefore let $\mathcal{S}$ compute the identity, so that the two distributions are both constantly $\{1^n\}$. This shows that the trivial protocol is perfect zero-knowledge.

Example 1.18. Consider the following protocol: $\mathcal{A}$ gets input $x \in \mathbb{Z}_2^n$, which it then sends to $\mathcal{B}$. To show this is not zero-knowledge for $\mathcal{B}$, we must show that for any simulator $\mathcal{S}$, there exists a choice of input x and a distinguisher $\mathcal{D}$ which distinguishes $\mathcal{S}(1^n)$ from $\text{view}_{\mathcal{B}}(1^n, x) = \{1^n, x\}$ with non-negligible probability.

Let $\mathcal{S}$ be fixed. If $\mathcal{S}$ does anything other than outputting 1^n and some $y \in \mathbb{Z}_2^n$, then we will be able to distinguish it syntactically. We may therefore safely assume that $\mathcal{S}$ outputs $(1^n, y)$ for some (potentially random) choice of y. For each n, now choose x such that $\Pr[y = x] \leq 2^{-n}$, which is possible by the pigeonhole principle. Let $\mathcal{D}$ output 1 on input $\{1^n, x\}$, and 0 otherwise. Then

$$\Pr[\mathcal{D}(\mathcal{S}(1^n)) = 1] = \Pr[y = x] \leq 2^{-n},$$

while

$$\Pr[\mathcal{D}(\text{view}_{\mathcal{B}}(1^n, x)) = 1] = 1.$$

Since $|1 - 2^{-n}|$ is not negligible, the protocol is not zero-knowledge for $\mathcal{B}$.

1.1.5 Adversaries and the Real-Ideal Paradigm

Zero-knowledge is a surprisingly general tool for formalizing security definitions, but in some cases it is not enough. For instance, we may want to verify that protocols for electronic coin-flips are fair: in this case, the issue is not that the parties may learn something extra, but that they may be able to unduly influence the outcome of the computation. The general approach taken in the literature is to define security on an ad-hoc basis for each such task by enumerating the properties we want the protocol to have and formalizing them as adversarial games. We will take a more systematic approach, sometimes called the *real-ideal paradigm.*

The idea is to define an *ideal protocol*—also called an *ideal functionality*—which represents the desired behavior of the cryptosystem. The protocol under study—the *real protocol*—is then supposed to emulate the ideal protocol, in the sense that its outcomes should be computationally indistinguishable from the outcomes of the ideal protocol. The subtelty here is our use of the term *outcome*; which necessarily looks different for different protocols: it may just be the information learned by a party, in which case we recover zero-knowledge, it may be some function of the party's outputs, or it may be something else altogether. We will see several different examples of this in Section 1.2, but the important point is that to define security in this paradigm requires both a definition of the ideal protocol and of the data to be compared.

Unlike in the case of zero-knowledge, where we only cared about what parties could learn from the protocol assuming it was executed correctly, in this setting we also want to discuss security against *malicious behavior*, in which one or more parties deliberately try to sabotage the outcome of the protocol. We often refer to these parties as the *adversaries*, and allow them to be non-uniform even when ordinary parties in the protocol are uniform. Since a protocol is not secure if even one possible attack is likely to succeed, we say that a protocol is *secure in the presence of malicious adversaries* if for any choice of algorithms taking over some fixed number of parties, the outcomes of the protocol are

indistinguishable from the ideal. Again, we will see examples of this notion in the next section.

In contrast, sometimes we do want to talk about a protocol being zero-knowledge without dealing with arbitrary adversarial behavior. In this case, we use the term *semi-honest adversaries*, which informally refers to adversaries which follow the prescriptions of the protocol, but attempt to learn as much as they can within those bounds. There are several other notions of adversarial strength considered in the literature—for instance, adaptive vs. static adversaries [Cra+99], Byzantine adversaries [LLR04], and "coercible" parties [CGP15]. We do not explore all these models here.

1.2 Cryptographic Problems

1.2.1 Encryption

Much of the machinery defined in the previous section was originally developed in the 1970s and 80s for the purpose of analyzing *encryption problems*, culminating in the work of Goldwasser and Micali [GM82]. The idea of an encryption problem is that a party Alice has a message m in the *message space* $\mathcal{M}_n$ which they want to send to Bob, but any message they send to Bob must also be sent to the eavesdropping Eve. In the simpler *shared-key encryption problem*, which we consider here, Alice and Bob share some secret key k from the *key space* $\mathcal{K}_n$, which is unknown to Eve.

Definition 1.19 (shared-key encryption scheme)**.** Let $\{\mathcal{M}_n\}$ and $\{\mathcal{K}_n\}$ be sequences of sets. An $(\mathcal{M}, \mathcal{K})$-shared-key encryption scheme is an interactive protocol consisting of three interactive algorithms $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{E}$, where:

- the global input is 1^n, the security parameter;
- $\mathcal{A}$ gets a uniform random key $k \in \mathcal{K}_n$ and a message $m \in \mathcal{M}_n$ as private input;
- $\mathcal{B}$ gets the same[4] key k as private input;
- $\mathcal{E}$ gets no private input;
- $\mathcal{A}$ and $\mathcal{B}$ only send messages to each other if they also send the message to $\mathcal{E}$.

A shared-key encryption scheme is *correct* if $\mathcal{B}$ outputs m at the end of the computation. A shared-key encryption scheme is *secure* if it is zero-knowledge for $\mathcal{E}$; explicitly, if there exists a simulator $\mathcal{S}$ such that for any choice of security parameter n and message m,

$$\mathcal{S}(1^n) \overset{c}{\equiv} \text{view}_{\mathcal{E}}^{\langle \mathcal{A}, \mathcal{B}, \mathcal{E} \rangle}(1^n, k, m),$$

where the randomness of the second distribution is over both the randomness of the algorithms and uniform random choice of k.

---

[4]Notice that this definition includes the stipulation that $\mathcal{A}$ and $\mathcal{B}$ share a uniform random key as input, which is not directly possible using the machinery of Section 1.1.4. One way to formalize this notion is to add a fourth machine $\mathcal{G}$ (the "generator"), which can message $\mathcal{A}$ and $\mathcal{B}$ freely (but not recieve messages from them), whose job is to generate the key and send it to both parties. For our purposes, the important point is that while the input m is seen as a parameter of the system which can be controlled in indistinguishability proofs, the input k is instead always randomly generated.

The point is that the eavesdropper should learn nothing from the interaction, while the intended recipient should learn the message.

Example 1.20. We can construct both a secure-but-not-correct and a correct-but-not-secure encryption scheme using work already done.

- For a secure-but-not-correct scheme, simply have each machine do nothing. The security proof is exactly the same as in Example 1.17.
- For a correct-but-not-secure scheme, have $\mathcal{A}$ send m to $\mathcal{B}$ (and therefore also to $\mathcal{E}$) as a message, and have $\mathcal{B}$ output that message. The insecurity proof is exactly the same as in Example 1.18.

Example 1.21 (the one-time pad)**.** We now give a secure and correct shared key encryption scheme, called the *one-time pad*. Let $\{G_n\}$ be a sequence of finite additive groups[5] such that $|G_n| = \Omega(2^n)$, for instance $G_n = \mathbb{Z}_2^n$. We work over $\mathcal{M}_n = \mathcal{K}_n = G_n$. Given a message m and key k, $\mathcal{A}$ computes $c = m + k$, which it sends to $\mathcal{B}$ (and $\mathcal{E}$). $\mathcal{B}$ then computes $c - k$, which it outputs.

Correctness of this scheme is immediate, as $\mathcal{B}$ outputs $c - k = m + k - k = m$. To prove security, our goal is to construct a simulator $\mathcal{S}$ such that $\mathcal{S}(1^n)$ is indistinguishable from $\text{view}_{\mathcal{E}}(1^n, k, m) = \{1^n, m + k\}$. Because addition by m is a bijection, and k is chosen uniformly at random, the distribution $\{m + k\}$ is just a uniform random sample from G_n. As such, we simply let $\mathcal{S}(1^n)$ draw g uniformly at random from G_n and output $\{1^n, g\}$. This is again a perfectly-secure encryption scheme, since the two distributions are identical.

Because the security is perfect, we don't need the asymptotics, so the one-time pad is more commonly defined on a fixed group G, usually $\mathbb{Z}_2^m$ for some fixed m.

Example 1.22 (the bootstrap one-time-pad)**.** One disadvantage of the one-time pad is that the key must be drawn from the same space as the message. We now show how to rectify this, assuming a pseudorandom generator (Example 1.13) $\mathcal{G}$ with expansion factor p is available. The idea is to use the pseudorandom generator to expand a short key into a longer one.

The protocol is as follows. We let $\mathcal{K}_n = \mathbb{Z}_2^n$ and $\mathcal{M}_n = \mathbb{Z}_2^{p(n)}$. Given a message $m \in \mathbb{Z}_2^{p(n)}$ and a key $k \in \mathbb{Z}_2^n$, $\mathcal{A}$ first computes $\mathcal{G}(k)$ to get a key $k' \in \mathbb{Z}_2^{p(n)}$. It then sends $c = m + k$ to $\mathcal{B}$. Similarly, $\mathcal{B}$ computes k' and then $c - k'$, which it outputs. Since $\mathcal{G}$ is deterministic, both $\mathcal{A}$ and $\mathcal{B}$ get the same value for k', and so the protocol is correct.

Security can be shown by a reduction to the hardness assumption entailed by pseudorandomness of $\mathcal{G}$, but an easier route is available. By definition of pseudorandomness, the distributions $\{\mathcal{G}(\mathbb{Z}_2^n)\}$ and $\{\mathbb{Z}_2^{p(n)}\}$ are computationally indistinguishable. To obtain c in this protocol and in the one-time pad, we perform the same computation on these distributions—merely adding the fixed message m. As such, by Proposition 1.12 the view of $\mathcal{E}$ in this protocol is indistinguishable from the view of $\mathcal{E}$ in the one-time pad. Now we obtain the desired result by security of the one-time pad and transitivity of indistinguishability.

---

[5]We also want that $\{G_n\}$ is *efficiently sampleable*, so that it is possible to generate an element from it uniformly at random in polynomial time.

1.2.2   Interactive Function Computation

Suppose we are given a (potentially stochastic) series of functions

$$f : X_1 \times \cdots \times X_N \to Y_1 \times \cdots \times Y_N.$$

Each such function yields the following cryptographic problem[6]:

> Can N parties, each given a private input $x_i \in X_i$, work together so that the ith party outputs the value $f_i(x_1, \ldots, x_n)$?

Here, f_i is the projection $\mathrm{pr}_i \circ f$. In particular:

Definition 1.23. An interactive protocol $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ *computes the function f* if for any choice of inputs $x_1, \ldots, x_N$,

$$\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle (x_1, \ldots, x_N) = f(x_1, \ldots, x_N),$$

where if f is stochastic the equality should be interpreted in the distributional sense.

Example 1.24. The protocol of Example 1.14 computes the function $f(x, y) = (y, x)$.

On the other hand, there are several possible notions of security in this setting, roughly following the lines discussed in Section 1.1.5. We first consider the more straightforward semi-honest case, in which we ask that no party should learn anything from the computation other than the value they are intended to output.

Definition 1.25. A protocol $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ *securely computes the function f in the presence of semi-honest adversaries* if it computes f and it is zero-knowledge for each ith party relative to f_i.

Example 1.26. Consider the following protocol for $f(x, y, *) = (*, *, xy)$: the first two algorithms each send their inputs two the third, which computes and outputs the product. This protocol is not secure in the presence of semi-honest adversaries, because the third party learns the two factors, not just the product; this insecurity can be proved very similarly to Example 1.18.

On the other hand, consider the following protocol for $f(x, y) = (*, xy)$: the first algorithm sends x to the second, which computes and outputs the product. This protocol is secure in the presence of semi-honest adversaries. The simulator for the second algorithm, which gets its input y and output xy, is responsible for producing a distribution indistinguishable from $\{x, y\}$; it can do this by computing xy/y.

Example 1.27 (oblivious bit-transfer)**.** The *oblivious bit-transfer problem* is as follows. Alice has two bits $b_1, b_2 \in \mathbb{Z}_2$, and Bob has a query $\sigma \in \{1, 2\}$. The goal is for Bob to learn the appropriate bit from Alice, without revealing which bit they asked for. We can formalize this using Definition 1.25: the problem is to securely compute

$$f : \mathbb{Z}_2^2 \times \{1, 2\} \to \{*\} \times \mathbb{Z}_2, \qquad ((b_1, b_2), \sigma) \mapsto (*, b_\sigma)$$

---

[6]Here we need to assume that there is a canonical grading on each input set, for instance given by the length of a bitstring, so that each X_i can also be thought of as a sequence of sets X_n^i in the style of Section 1.1.4.

in the presence of semi-honest adversaries. While the solution is outside our scope, this is possible (under standard complexity-theoretic assumptions) via a protocol originally due to [EGL85].

The situation with malicious adversaries is much more complicated, but we will not need all the details here; they can be found in, for instance, [Lin17, Section 4].

1.2.3 Zero-Knowledge Proof

In an *interactive proof*, one party—the *prover*—tries to convince the other—the *verifier*—that some statement is true. We generally consider proofs of membership predicates over fixed sets (called *languages* in this context), so that the prover is trying to convince the verifier that some global input x is in a fixed set $\mathcal{L}$. The key point is that the prover generally has some computational advantage over the verifier, so that the verifier cannot simply reproduce all the steps taken by the prover.

Informally, there are two correctness properties we want an interactive proof system to have. It should be *sound*: when given an input $x \in \mathcal{L}$, the verifier should except with high probability. It should also be *complete*, which informally means that no adversarial machine should be able to convince the verifier to accept an input not in $\mathcal{L}$ except with low probability. Formally:

Definition 1.28 (interactive proof system). Let $\mathcal{L}$ be a fixed language. An *interactive proof system for $\mathcal{L}$* consists of an algorithm $\mathcal{P}$ and a PPT algorithm $\mathcal{V}$, such that:

- (soundness) for each $x \in \mathcal{L}$,

$$\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x)_{\mathcal{V}} = 1] > \frac{2}{3};$$

- (completeness) for each $x \notin \mathcal{L}$ and each algorithm $\mathcal{P}'$,

$$\Pr[\langle \mathcal{P}', \mathcal{V} \rangle(x)_{\mathcal{V}} = 1] < \frac{1}{3}.$$

Once we achieve any probability of success greater than $\frac{1}{2}$, we can repeat the protocol to achieve any constant probability of success. It is standard to choose $\frac{2}{3}$ as the desired bound for such cases.

Example 1.29 (graph non-isomorphism [Gol01, Section 4.2.2]). We give an interactive proof system for the problem of *graph non-isomorphism*[7], which is formalized as the language

GNI = $\{(G_1, G_2) : G_1$ and G_2 are non-isomorphic finite graphs$\}$.

---

[7]Since we do not know whether graph non-isomorphism is in NP, the reader may now wonder what the computational strength of interactive proof is. It turns out that IP, the class of languages with interactive proof, equals PSPACE, the class of languages whose membership predicates can be computed in polynomial space. This result was first proven by [Sha92].

The idea is that the verifier will construct a random graph isomorphic to one of the chosen graphs, and the prover will have to guess which one. If the graphs are non-isomorphic, then it should always be able to do so; if they are, it can only do so with probability $\frac{1}{2}$.

The algorithms get a pair of graphs (G_1, G_2) as shared input; we let $G_1 = (V_1, E_1)$ and similarly for G_2. The verifier uniform-randomly chooses $\sigma \in \{1, 2\}$ and a relabeling of G_σ. In particular, it uniform-randomly chooses a bijection $\pi : V_\sigma \to \{1, \ldots, |V_\sigma|\}$ and generates the graph

$$G_\sigma' = (\{1, \ldots, |V_\sigma|\}, \{(\pi(v), \pi(w)) : (v, w) \in E_\sigma\}),$$

which it sends to the prover. The prover, which is unbounded, can then check whether $G_\sigma' \cong G_1$ or $G_\sigma' \cong G_2$ and send the result to the verifier; if both hold, then it sends a random bit. The verifier repeats the experiment again, outputting 1 if the prover guessed right both times, and 0 otherwise.

To show soundness, note that if $G_1 \ncong G_2$, then the prover will always guess correctly, in which case the verifier outs 1 with probability 1.

To show completeness, note that G_σ' is a uniform random draw from

$$\{G = (V, E) : G \cong G_\sigma, V = \{1, \ldots, |V_\sigma|\},$$

i.e. from the isomorphism class of G_σ with vertex set $\{1, \ldots, |V_\sigma|\}$. Since G_1 and G_2 have identical isomorphism classes and the same number of vertices, this implies that the distribution of G_σ' is independent of σ. Now let $\mathcal{P}'$ be any algorithm which is given G_1, G_2, and G_σ' and must guess σ. By independence, we have that

$$\Pr[\mathcal{P}'(G_\sigma') = \sigma] = \sum_{\tau \in \{1,2\}} \Pr[\mathcal{P}'(G_\sigma') = \tau \text{ and } \sigma = \tau]$$

$$= \sum_{\tau \in \{1,2\}} \Pr[\mathcal{P}'(G_\sigma') = \tau] \Pr[\sigma = \tau]$$

$$= \frac{\Pr[\mathcal{P}'(G_\sigma' \in \{1, 2\})]}{2}$$

$$\leq \frac{1}{2};$$

since the experiment repeats twice, we get a probability of $\frac{1}{4} < \frac{1}{3}$, as desired.

In cryptography, we are especially concerned with *zero-knowledge proofs*, which are meant to reveal no information other than the truth of the statement under proof. We will need something slightly stronger than for multi-party computation: because this protocol involves the prover giving information in response to queries from the verifier (such as the query graph G_σ' in Example 1.29), we will need to ensure that no verifier can learn anything extra from the prover, even if they give different queries than the protocol prescribes.

Definition 1.30 (zero-knowledge proof). An interactive proof system $(\mathcal{P}, \mathcal{V})$ is *honest-verifier zero-knowledge* it is zero-knowledge for $\mathcal{V}$. It is *semi-honest-verifier zero-knowledge*,

or just *zero-knowledge*, if for each non-uniform PPT $\mathcal{V}'$, the protocol $\langle \mathcal{P}, \mathcal{V}' \rangle$ is zero-knowledge for $\mathcal{V}'$. It is *black-box zero-knowledge* if the simulator can query $\mathcal{V}'$ but must be defined independently of it[8].

 If one-way functions exist, then every problem in NP has a zero-knowledge proof [GMW91]. In fact, rather remarkably, under the same assumption *any* language that admits an interactive proof admits a zero-knowledge proof [Ben+90].

Example 1.31. The protocol of Example 1.29 is an honest-verifier zero-knowledge proof.

1.3 Composition

1.3.1 The Issues at Hand

We now wish to consider whether our security definitions are closed under composition of protocols. There are many different issues to consider in stating such a composition theorem, including:

1. What kinds of protocols are being composed? Our security definitions do not capture security of arbitrary interactive processes, so either we will need a substantially more general definition or we will need to limit our composition theorem to a specific class of protocols.

2. What does it mean to compose these protocols? It is not immediately clear how to compose arbitary interactive algorithms.

3. What kind of composition is allowed? In particular, we can consider *sequential composition*, in which only one protocol is "running" simultaneously, or *parallel composition*, in which many protocols may be running simultaneously. To formally state a parallel composition theorem, we need to either specify a *scheduling model* and deal with low-level issues like atomicity, or find some way to abstract over these details.

4. What kind of security is being preserved? Given a security definition for the component protocols, we need some say to derive the security definition for the composite protocol.

5. What kinds of adversaries does the theorem handle? Composition theorems may look very different for security against uniform and non-uniform adversaries, for instance—these subtle issues can lead to very different results.

6. What protocols are we allowed to compose with? We could be allowed to compose with arbitrary protocols, which might not even be secure, or only with other protocols we already know are secure.

---

[8]There appear to be several distinct choices in the literature for naming conventions, for the computational strength of the prover, and for the level of honesty of the verifier. Our naming convention follows [Vad07]; a good survey of various definitions is [GO94, Section 3].

7. How many times we can compose—for instance, must it be constant in the security parameter?

As these questions demonstrate, composition theorems are really quite difficult to state. However, even without a formal theorem in mind, we can make a few concrete observations.

In regards to Question 6, we need to have some kind of idea of "independence of state" between two protocols before we can compose them. For instance, we certainly should not be allowed to compose the one-time pad with a protocol that publicly broadcasts the key.

In regards to Question 7, we should at best expect to be able to compose polynomially many protocols in the security parameter. This is in part for complexity reasons—a polytime turing machine cannot simulate super-polynomially many protocols—but there is also a security explanation: as a consequence of Proposition 1.11, we should only expect to be able to compose polynomially many computationally indistinguishable distributions before losing the indistinguishability.

We do not make attempt to give a comprehensive review of the composition theorems or counterexamples in the literature. Instead, our goal is to chart a motivating path towards the theory of *universal composability*, and then give a sufficiently detailed overview of that theory to equip the reader to compare it to the cryptographic models we will study in Chapter 3.

1.3.2   Composing Interactive Function Computations

Possibly the easiest non-trivial composition theorem to state is for sequential composition of interactive function computations, but already in this case we will run into several fundamental issues.

Suppose we have two N-party protocols $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ and $\langle \mathcal{B}_1, \ldots, \mathcal{B}_N \rangle$ which securely compute the N-party functions $f : X_1 \times \cdots \times X_N \to Y_1 \times \cdots \times Y_N$ and $g : Y_1 \times \cdots \times Y_N \to Z_1 \times \cdots \times Z_N$ in the presence of semi-honest adversries. We wish to use this data to construct a composite protocol for the composite function $g \circ f$. The natural choice is to have the parties first run the protocol for f, then run the protocol for g. Thus, we can state the following claim:

Claim 1.32. *Suppose* $\{f^i : X_1^i \times \cdots \times X_N^i \to X_1^{i+1} \times \cdots \times X_N^{i+1}, 1 \leq i \leq K\}$ *is a finite sequence of functions which are securely computable in the presence of semi-honest adversaries. Then the composite* $f = f^K \circ \cdots \circ f^1$ *is securely computable in the presence of semi-honest adversaries.*

Proof attempt. For each i, let $\Pi_i = \langle \mathcal{A}_1^i, \ldots, \mathcal{A}_N^i \rangle$ be a protocol which securely computes f^i in the presence of semi-honest adversaries. Form a new protocol Π which performs each of the Π_i in sequence. Correctness is immediate, since the correctness of each Π_i implies that Π computes each step of the composite in turn.

The proof of security is by reduction to the security of the Π_i. By induction, it suffices to consider the case $K = 2$. Suppose that the composite Π does not securely compute $f^2 \circ f^1$. Without loss of generality, suppose that the composite is not zero-knowledge for

party 1. Then for any $\mathcal{S}$, there exists some non-uniform PPT $\mathcal{D}$ which distinguishes, for any choice of inputs $x_i \in X_i^1$,

$$\mathcal{S}(x_1, f_1^2(f^1(x_1, \ldots, x_N))) \qquad \text{and} \qquad \text{view}_1^\Pi(x_1, \ldots, x_2). \tag{1.1}$$

All this works, but we run into issues when we attempt to construct a simulator which contradicts this assumption. Let $\mathcal{S}_i^1$ and $\mathcal{S}_i^2$ be the simulators which witness security of Π_1 and Π_2 with respect to party i. We would like to construct a simulator $\mathcal{S}$, which combines the simulated views of party i in both parts of the computation. The issue is that, in order to fit the form of (1.1), $\mathcal{S}$ should get only $(x_1, f_1^2(f^1(x_1, \ldots, x_N)))$ as input. However, to be able to run $\mathcal{S}_1^1$, it needs $f_1^1(x_1, \ldots, x_N)$, which we have no way to obtain.

I am not aware of a direct way to repair this proof. Instead, to avoid the issue, we need the structure of the composite protocol to mirror the structure of the reduction: there should be some "outermost" protocol which handles calls to the sub-protocols, just as we need to make an outermost simulator which calls the sub-simulators. The right tool for the job is the *oracle algorithm*.

Definition 1.33 (oracle algorithms; oracle protocols)**.** An *oracle algorithm* is an algorithm $\mathcal{A}$ equipped with a "slot" for an *oracle* O, to which it can make *queries* x and receive *responses* $O(x)$. We write $\mathcal{A}^O$ to refer to the oracle-algorithm $\mathcal{A}$ equipped with the specific oracle O.

An *oracle protocol* is a protocol $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ with a slot for an oracle O, where each $\mathcal{A}_i$ may write queries x_i, and if each machine does so, they each receive outputs $O(x_1, \ldots, x_N)_i$. We write $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle^O$.

Let $f : X_1 \times \cdots \times X_N \to Y_1 \times \cdots \times Y_N$ be a function. An f*-oracle* is an oracle which, when queried with $x_1, \ldots, x_n$, responds with $f(x_1, \ldots, x_N)$. We overloadingly write f to refer to an f-oracle.

The idea is to make an oracle protocol Π which, when instantiated with an f-oracle, securely computes g. Then if we have a protocol for securely computing f, we will show that we can substitute it for the oracle in Π. In particular, while we will not be precise about what secure oracle computation means, it is important that the view of an oracle algorithm includes its queries and responses.

Definition 1.34 (oracle reduction)**.** Let f and g be functions. Then g *securely oracle-reduces to* f *in the presence of semi-honest adversaries* if there exists an oracle protocol $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ such that $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle^f$ securely computes g in the presence of semi-honest adversaries.

Theorem 1.35 ([Gol01, Theorem 7.3.3])**.** *Suppose g securely reduces to f in the presence of semi-honest adversaries, and f is securely computable in the presence of semi-honest adversaries. Then g is securely computable in the presence of semi-honest adversaries.*

Proof sketch. Let $\Pi_g = \langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle$ be an oracle protocol such that $\langle \mathcal{A}_1, \ldots, \mathcal{A}_N \rangle^f$ securely computes g in the presence of semi-honest adversaries, and let $\Pi_f = \langle \mathcal{B}_1, \ldots, \mathcal{B}_N \rangle$ be a protocol which securely computes f in the presence of semi-honest adversaries. We

construct a new protocol Π which runs Π_g, but whenever the oracle is queried, it instead replaces the oracle with a complete run of Π_f. Correctness of Π is immediate from its construction; we show security by a reduction.

Let $\mathcal{S}_i^g$ simulate the view of party i in Π_g^f, and let $\mathcal{S}_i^f$ simulate the view of party i in Π_f. We construct a simulator $\mathcal{S}_i$ for the view of party i in Π. Since the inputs to Π_g^f are the same as to Π, we can immediately run $\mathcal{S}_i^g$ to obtain the view of party i in the computation Π_g^f, which in particular includes the queries and responses of the oracle. We can then pass these queries and responses to Π_f, obtaining a view of each of the invocations of the sub-protocol.

To show the this simulator achieves the necessary indistinguishability result, we work in two steps. First, the output of $\mathcal{S}_i$ is indistinguishable from the view of party i in Π with the sub-protocol invocations are replaced with the simulated views; we prove this by the security of Π_f, since otherwise appending the view from Π_g^f would distinguish the view from Π_f with its simulated counterpart. Similarly, this latter distribution is indistinguishable from the view of party i in Π without any such replacement, this time by security of Π_g^f. Filling in these details completes the proof. $\square$

A few remarks on this proof are warranted. First, it fundamentally relies on the semi-honesty of the adversaries, since it means that the execution of the sub-protocols is completely independent from anything other than their inputs. If the adversaries could behave maliciously, and thus use information learned in the larger protocol to affect run the sub-protocols, then the main simulator would be unable to properly invoke the simulator for the sub-protocols.

Second, these techniques seem completely unable to handle parallel composition. The issue is that oracle queries are in some sense immediate—the assumption is that everything else pauses while the oracle does its work. It is much more difficult to handle parallelism, or even more strongly, to handle concurrency, which may in particular be asynchronous.

Third, if we want to handle protocols other than function computations, we will need a more robust notion of composition of ideal functionalities—this proof is tied to the easy-to-understand structure of function computation.

The goal of universal composability is to resolve these issues, but first we will explicitly give an example wherein parallel composition fails.

1.3.3   A Counterexample to Parallel Composition

Before giving the counterexample, we briefly discuss sequential composition of zero-knowledge proofs, which is quite subtle. When the definitions are as given in Definition 1.30, black-box zero knowledge proofs do compose in sequence [GO94]. On the other hand, if we change the definitions so that both the simulator of Definition 1.16 and the adversarial verifier of Definition 1.30 are required to be *uniform* PPT, then computational zero-knowledge does not compose [GK96]. However, if under the same definition we require that the prover is in NP, then uniform computational zero-knowledge proofs do compose in sequence up to a constant number of times; but if the distinguishers in the definition of computational indistinguishability are required to be uniform, then sequen-

tial composition again fails [BV10]. There are a huge number of variations to consider and I believe there are many definitions for which the question of sequential composition is still open.

For parallel composition, the situation is much simpler: we give an argument due to [GK96] that zero-knowledge proofs do not compose in parallel. The idea of the counterexample is as follows. In proof A, the prover poses a randomly chosen computationally intractable challenge to the verifier, and then gives the verifier knowledge if and only if the verifier can solve the challenge. This proof is zero-knowledge because PPT verifiers can solve the challenge only negligibly often. In proof B, the verifier poses a challenge to the prover, which the prover answers. The trick is to choose a class of challenges whose answers are pseudorandom, so that the answer to the challenge on its own does not carry knowledge. However, when proof A and proof B are run together in parallel, the verifier can take the challenge it gets in proof A, get an answer in proof B, and that use that to get knowledge from the prover in proof A.

The difficulty in formalizing this is to find a class of challenges which are computationally intractable, in that PPT algorithms solve them with negligible probability; zero-knowledge, in that answers look pseudorandom; and decidable, in that unbounded algorithms can answer challenges and check answers. Our challenges will be phrased as sets: for each security parameter n, we will agree to some set of sets $S_1^n, \ldots, S_{2^n}^n \subseteq \{0,1\}^{Q(n)}$, where Q is a polynomial. A challenge looks like a value $i \in 1, \ldots, Q(n)$, and a solution is a value $x \in S_i^n$. We now formalize our desiderata as follows.

Definition 1.36. A *non-uniform ensemble* is a sequence $\{S^n\}$, where for each n, $S^n = \{S_1^n, \ldots, S_{2^n}^n\}$ is a set of 2^n sets. A non-uniform ensemble is:

- *polynomially-sized* if there exists a polynomial Q such that for each n and i, $S_i^n \subseteq \{0,1\}^{Q(n)}$;
- *pseudorandom* if for each n and i, S_i^n is pseudorandom (in the sense of Example 1.13);
- *decidable* if there exists an algorithm which, on input (n, i), outputs the elements of S_i^n and then halts.
- *evasive* if for any non-uniform PPT algorithm $\mathcal{A}$,

$$\Pr_{i \in \{1, \ldots, 2^n\}} [\mathcal{A}(i) \in S_i^n] = \mathsf{negl}(n).$$

Theorem 1.37 ([GK96, Theorem 3.2]). *There exists a polynomially-sized, pseudorandom, decidable, evasive non-uniform ensemble.*

Now the actual construction is relatively straightforward. Let S be the ensemble from Theorem 1.37 and Q the size polynomial. Let K be a computable predicate which is known not to be in BPP; such things exist by the time hierarchy theorem from complexity theory. We will give two zero-knowledge proofs for the language $\{0,1\}^*$.

Proof A goes as follows: on input $x \in \{0,1\}^n$, $\mathcal{P}_A$ chooses $i \in \{1, \ldots, 2^n\}$ uniformly at random, which it sends to $\mathcal{V}_A$. Next, $\mathcal{V}_A$ chooses $s \in \{0,1\}^{Q(n)}$ uniformly at random, which it sends to $\mathcal{P}_A$. If $s \in S_i^n$, $\mathcal{P}_A$ sends $K(x)$ to $\mathcal{V}_A$. Then $\mathcal{V}_A$ outputs 1. This proof is zero-knowledge because the probability that any cheating $\mathcal{V}_A'$ sends an $s \in S_i^n$ is negligible, so the probability it learns $K(x)$ is negligible.

Proof B goes as follows: on input $x \in \{0,1\}^n$, $\mathcal{V}_B$ chooses $i \in \{1, \ldots, 2^n\}$ uniformly at random, which it sends to $\mathcal{P}_B$. Next, $\mathcal{P}_B$ sends $s \in S_i^n$ to $\mathcal{V}_B$. Then $\mathcal{V}_B$ outputs 1. This proof is zero-knowledge because S_i^n is pseudorandom, so the simulator can just output a uniform random value and by definition of pseudorandomness no distinguisher can tell whether it is seeing that random value or the value from S_i^n.

However, in the parallel composition, the verifier can wait to send i until it receives the i from $\mathcal{P}_A$. Then, when it gets back $s \in S_i^n$ from $\mathcal{P}_B$, it can send s to $\mathcal{P}_A$ and get back $K(x)$, and hence it learns $K(x)$ with probability 1.

Under computational hardness assumptions, Theorem 1.37 can be extended to, for instance, evasive ensembles which are decidable in NP, and hence under such assumptions zero-knowledge proofs with NP provers also do not compose in parallel.

As a consequence of this construction, it is clear that we need more technology to handle parallel composition in any reasonable way. By far the most popular attempt to do so is universal composability.

1.3.4   Universal Composability

In Section 1.3.2, we identified three critical issues with generalizing the proof of Theorem 1.35: we need to handle malicious adversaries which can forward their views to each other; we need to handle parallel composition during which other protocols can run; and we need a very general way to describe composition and security of a large class of computational protocols. Universal composability (UC), due to Ran Canetti [Can00; Can20], resolves all of these.

The single big idea of UC is to strengthen the notion of emulation required in the real-ideal paradigm. Whereas our formulations of simulation security require only that *by the end of the computation* the real protocol produces an output indistinguishable from the ideal protocol, in UC, this indistinguishability must hold *throughout* the computation. Intuitively, we should expect this to be strong enough to allow us to prove a parallel composition theorem, because in particular the other computation we are composing with will be unable to distinguish our protocol from the ideal. In this section, we will sketch some of the key tools used to make this formal, without attempting to be fully formal ourselves.

In a UC proof, an algorithm called the *environment* represents all the other computational processes happening along with the protocol under study. The environment is also responsible for giving inputs to parties in a protocol. Ultimately, the environment is also the distinguisher, responsible for attempting to determine between the real or ideal protocol. If it cannot do so, then no other protocol is able to do so, so in particular the security guarantees will hold no matter what else is going on. In this way, environments abstract over a lot of the complexity in ordinary cryptographic definitions.

Adversarial behavior is somewhat complicated in UC. In addition to the environment, there is an algorithm called the *adversary* which may write to the *backdoor tapes* of all the parties involved in the protocol. The protocol is responsible for specifying how the parties should behave in respnose to messages writen on their backdoor tapes—in this way, UC conveniently avoids dealing in the formalism with different models of adversaries, since they may be specified on a protocol-by-protocol basis. For instance, a protocol meant to

be secure against semi-honest adversaries may just allow backdoor messages which ask the corrupted party to forwards its state to the environment, while a protocol meant to be secure against malicious adversaries may allow a party to be completely piloted over via instructions sent to its backdoor tape.

When π is a protocol and $\mathcal{A}$ and $\mathcal{E}$ are algorithms, we write $\mathrm{exec}_{\pi,\mathcal{A},\mathcal{E}}$ for the output of $\mathcal{E}$ after an interactive computation with all the algorithms in π and with the adversary $\mathcal{A}$. A protocol π *UC-emulates* another protocol ϕ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that, for any environment $\mathcal{E}$,

$$\mathrm{exec}_{\pi,\mathcal{A},\mathcal{E}} \stackrel{\mathrm{c}}{\equiv} \mathrm{exec}_{\phi,\mathcal{S},\mathcal{E}} \,.$$

The idea is that $\mathcal{E}$ is trying to output a guess of whether it is interacting with π or ϕ; it should be able to do so only negligibly often.

We need a way to express ideal protocols for general cryptographic resources; UC calls these *ideal functionalities*. An ideal protocol for a functionality essentially consists of an algorithm $\mathcal{F}$, which receives its inputs from some "dummy parties" and then sends them the correct outputs back. In order to work with more than just function computation, $\mathcal{F}$ is also constantly talking to the adversary, and so can simulate functionalities like public commitments that do not show up in a function signature. A protocol *UC-realizes* the functionality $\mathcal{F}$ if it UC-simulates the ideal protocol for $\mathcal{F}$.

The next issue is a formal notion of composition. This should look like the "oracle substitution" construction from the proof of Theorem 1.35, but handle more general forms of composition than the sequential composition implied by the oracle reductions. Pick a protocol π and a subset ρ of the parties involved in π. Pick another protocol ϕ such that there is an injective map from the parties in ρ to those in ϕ. Then the protocol $\pi^{\rho\to\phi}$ replaces ρ with ϕ, wiring up the communication with the rest of π according to the injection (note that ϕ may have more machines than ρ; these do not talk to the other parties in π). This is called the *universal composition operation*. It takes rather a lot of work to make this substitution operation precise—for instance, there is a very technical compatibility condition which essentially asks that the interfaces of ϕ and ρ are "functional", in the intuitive sense that they only interact with the rest of the protocol via their inputs and outputs (and maybe as a result of corruption). In [Can20], such protocols are called *subroutine respecting*.

Theorem 1.38 (the univeral composition theorem [Can20, Theorem 22]). *Under technical assumptions, if ρ is a subroutine of π and ϕ UC-emulates ρ, then $\pi^{\rho\to\phi}$ UC-emulates π. In particular, if $\mathcal{F}$ and $\mathcal{G}$ are ideal functionalities such that π UC-realizes $\mathcal{G}$ and ϕ UC-realizes $\mathcal{F}$, then $\pi^{\mathcal{F}\to\phi}$ UC-realizes $\mathcal{G}$.*

All this is just a sketch; there are of course many technical details to work out. For instance, there is an entire low-level model of distributed computation which to me seems inextricable from the formalization of the security definitions. The underlying machine model is a specific kind of *interactive Turing machine*, which has seven different tapes with differing semantics and read/write permissions, and the network is mediated by a *control function*, which can modify or block messages between machines involved in a computation. There is no formal notion of a "secure channel" in UC, nor of other basic

cryptographic resources; instead every resource is meant to be represented as an auxiliary machine which implements some ideal functionality.

To summarize and begin to evaluate the framework, we now return to the questions from Section 1.3.1.

In Question 1, we asked what kinds of protocols are being composed. Protocols in UC are sets of interactive Turing machines behaving according a highly specific execution model. On one hand, Turing machines are a standard and well-understood model of classical computation, and the flexibility of the notion of environment allows for encoding a wide variety of security properties. On the other hand, the technical specificity of the interaction model makes it difficult to formally apply the model; in practice no papers work with the precise model, and it is not clear to me that the model is sufficiently well-understood to justify the common level of hand-waving. Furthermore, this machine model makes it difficult to apply UC to other models of computation. For instance, to deal with quantum cryptography, we need to use a mathematically separate framework based on quantum which redoes much of the work of the UC paper—in fact, there are at least three competing frameworks for doing so [Unr04; BM04; Unr10], each with their own ad-hoc notion of "quantum machine."

In Questions 2 and 3, we asked about the definition and scope of composition. The subroutine substitution operation from UC is extremely general. Canetti argues, and the last twenty years have demonstrated, that together with control functions which model adversarial network conditions, UC composition can handle sequential, parallel, concurrent, and asynchronous composition; that it can handle compositions with variable numbers of rounds and subroutine calls; with coordinated or uncoordinated timings; with adaptively chosen inputs and adversaries; composition with shared and independent state; and more. One good source of examples of these claims is [Can06].

In Question 4, we asked how the system derives security definitions for composite protocols. UC works similarly to the oracle model: there is one "outermost" protocol, and we substitute real protocols for idealized subroutines within the protocol. The key point is the definition of UC-emulation is strong enough to allow a protocol to be substituted for an ideal functionality virtually anywhere and any time. However, this imposes a high proof burden which makes many UC proofs intractable, as even aside from the low-level details of the machine model, constructing a simulator that is in constant conversation with the environment is quite burdensome. The high proof burden also leads to impossibility results that can require somewhat ad-hoc setup assumptions to overcome [CF01; KL07; JM20].

In Question 5, we asked which kinds of adversaries the system accommodates. UC is again quite flexible in this regard. Adversarial behavior is built into the protocol via backdoor tapes and corruption messages, and then a separate party called the adversary activates (in a controlled, stateful way, so that they cannot for instance just corrupt all the parties) corruption messages during the protcol. In [Can00, Section 7.1.1], Canetti gives examples of a wide variety of adversarial models which can be incorporated in this way. Furthermore, this approach allows UC proofs to compose protocols which are secure against different forms of adversary.

In Question 6, we asked what protocols are being composed with. UC does not require that the protocols being composed are secure; in fact there are essentially no requirements

on the "outer" protocol into which the substitution will occur. We also saw previously that any framework for composition needs some way to assert that the protocols being composed are "independent enough" that one does not give away the secret key of the other; UC does this with the notion of "subroutine respecting" protocols, which are one of the only technical limits on composition.

In Question 7, we asked how many times we can compose protocols. We have not discussed nested UC-composition, but the universal composition theorem holds up to polynomially many nested substitutions of subroutines, which as discussed earlier is a tight bound.

Finally, we want to give two pieces of evidence of the naturality of UC. Technically, [Lin03] proved that, if a protocol for multi-party function computation is secure under parallel composition with even a constant number of (not-necessarily-secure) other protocols, then it is UC-secure. Socially, a precise connection has recently been established between UC and the independently-formulated *robust compilation* (RC) framework from programming language security [PKW22]; while RC is a comparatively new tool, this connection suggests a cross-field applicability of even the technical details of UC.

That said, while UC has clear merits, especially in terms of its incredible flexibility, there are serious disadvantages to carrying around that degree of complexity. In particular, UC cannot handle other models of computation, its proves are convoluted and hard to trust, and it carries several impossibility results that seem somewhat artificial. We will conclude the chapter by surveying some alternatives.

Chapter 2

Category Theory

The notion of a *category*, originally developed as an abstraction for certain ideas in pure mathematics, turns out to be the natural algebraic axiomatization of a collection of strongly typed, composable processes, such as functions in a strongly typed programming language. More philosophically, we can think of a category as an *algebra of composition*, and category theory as the mathematical study of composition. In this chapter, we will develop the basic theory of categories, prioritizing examples from computer science where possible.

Basic texts on category theory include [Mac71] and [Rie17], while the connection to computer science is explored in [Pie91] and [BW90]. A more advanced treatment of the connection, especially applications to programming language theory, is [Jac99].

2.1 Basic Notions

2.1.1 Categories

Definition 2.1 (category). A *category* C consists of the following data:

- a collection[1] of objects, overloadingly also called C;
- for each pair of objects $x, y \in C$, a collection of *morphisms* $C(x, y)$;
- for each object $x \in C$, a designated *identity morphism* $x \xrightarrow{1_x} x$;
- for each pair of morphisms $x \xrightarrow{f} y \xrightarrow{g} z$, a designated *composite morphism* $x \xrightarrow{gf} z$.

This data must satisfy the following axioms:

- *unitality*: for any $x \xrightarrow{f} y$, $1_y f = f = f 1_x$;
- *associativity*: for any $x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} w$, $(hg)f = h(gf)$.

Notation. In addition to those used above, many syntaxes are common for basic categorical notions.

---

[1] We use the word *collection* for foundational reasons: in many important examples, the objects and morphisms do not form sets. We ignore such foundational issues here; they are discussed in [Mac71, subsection 1.6].

- A morphism $f \in C(x, y)$ is often written $f \colon x \to y$ or $x \xrightarrow{f} y$; x is called its *domain* or *source* and y is called it *codomain* or *target*.
- Morphisms may be called maps, arrows, or homomorphisms; the class of morphisms $C(x, y)$ may also be written $\mathrm{Hom}_C(x, y)$ or just $\mathrm{Hom}(x, y)$, and is often called a *hom-set*.
- Composition is written gf or $g \circ f$, or sometimes in the left-to-right order fg.
- Identities are written 1_x, id_x, or just x where the context is clear[2].

Example 2.2 (functional programming languages)**.** Consider some strongly-typed functional programming language L, whose functions are never side-effecting. Then under very modest assumptions about L, we can make a category $\mathcal{L}$, as follows:

- the objects of $\mathcal{L}$ are the types of L;
- the morphisms $\mathcal{L}(A, B)$ are the functions of type $A \to B$;
- the identities 1_A are the identity functions $A \to A$;
- composition of morphisms are the usual function composition.

If L is truly non-side-effecting, then it's straightforward to check that this construction does indeed satisfy the axioms of a category; see for instance [BW90, subsection 2.2] to see the necessary assumptions spelled out rigorously.

Categories are also widespread in mathematics, as the following examples show.

Example 2.3 (concrete categories)**.** The following are all categories:

- SET is the category of sets and functions.
- GRP is the category of groups and group homomorphisms.
- RING is the category of rings and ring homomorphisms.
- TOP is the category of topological spaces and homeomorphisms.
- For any field $\Bbbk$, VECT$_\Bbbk$ is the category of vector spaces over $\Bbbk$ and linear transformations.

We call such categories, whose objects are structured sets and whose morphisms are structure-preserving set-functions, *concrete*. On the other hand, many categories look quite different.

Example 2.4. The following are also categories:

- The *empty category* has no objects and no morphisms.
- The *trivial category* has a single object and its identity morphism.
- Any group (or, more generally, monoid) can be thought of as a category with a single object, a morphism for every element, and composition given by the monoid multiplication.
- Any poset (or, more generally, preorder) $(P, \leq)$ can be thought of as a category whose objects are the elements of P, with a unique morphism $x \to y$ if and only if $x \leq y$. In this sense, composition is a "higher-dimensional" transitivity, and identities are higher-dimensional reflexivity.

---

[2]I agree with Harold Simmons, who says that this last is "a notation so ridiculous it should be laughed at in the street" [Sim11, p. 5].

- Associated to any directed graph is the *free category* on the graph, whose objects are nodes and whose morphisms are paths. In particular, the identities are just the empty paths, while composition concatenates two paths.
- Let $M = (Q, \delta)$ be an automaton over an alphabet Σ, so that $\delta : Q \times \Sigma \to Q$ is a transition function (one may replace Q with $\mathcal{P}(Q)$ in the codomain to represent a nondeterministic automaton). There is an associated category $\mathcal{M}$ whose objects are exactly the states and whose morphisms $\mathcal{M}(q_1, q_2)$ are the words $w \in \Sigma^*$ such that, if M is in the state q_1 and receives w as input, it ends in the state q_2. The identity morphism 1_q is the empty word, and composition is concatenation of words[3].
- There is a category whose objects are (roughly) multisets of molecules and whose morphisms are chemical reactions. See [BP17] for a formalization of this notion.

When working with categories, we often want to show that two complex composites equate. In this case, we prefer graphical notation to the more traditional symbolic equalities of Definition 2.1. A diagram in a category C looks something like so[4]:

$$
\begin{array}{ccc}
w & \xrightarrow{f} & x \\
{\scriptstyle h}\downarrow & & \downarrow{\scriptstyle g} \\
y & \xrightarrow{k} & z.
\end{array}
$$

This diagram identifies four objects $w, x, y, z \in C$, and four morphisms $f \in C(w, x)$, $g \in C(x, z)$, $h \in C(w, y)$, and $k \in C(y, z)$.

We say that a diagram *commutes* if, for any pair of paths through the diagram with the same start and end, the composite morphisms are equal. In this language, the previous diagram commutes if and only if $gf = kh$.

Example 2.5. The axioms of Definition 2.1 are expressed by commutativity of the following three diagrams:

$$
x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} w
$$

$$
\begin{array}{ccc}
x & \xrightarrow{1_x} & x \\
& {\scriptstyle f}\searrow & \downarrow{\scriptstyle f} \\
& & y
\end{array}
\qquad
\begin{array}{ccc}
x & \xrightarrow{f} & y \\
& {\scriptstyle f}\searrow & \downarrow{\scriptstyle 1_y} \\
& & y.
\end{array}
$$

The key idea is that commutative diagrams can be "pasted", allowing us to build up complex equalities from simpler ones. For instance, if

$$
\begin{array}{ccc}
w & \xrightarrow{f} & x \\
{\scriptstyle h}\downarrow & & \downarrow{\scriptstyle g} \\
y & \xrightarrow{k} & z
\end{array}
\qquad \text{and} \qquad
\begin{array}{ccc}
x & \xrightarrow{l} & v \\
& {\scriptstyle g}\searrow & \downarrow{\scriptstyle m} \\
& & z
\end{array}
$$

---

[3]I believe this example is due to [Gog+73, Example 2.2].
[4]The notion of a diagram can be made precise fairly easily; see [Rie17, subsection 1.6].

both commute, then by pasting along the shared morphism g, so does

$$
\begin{array}{ccc}
w & \xrightarrow{\ f\ } & x & \xrightarrow{\ l\ } & v \\
{\scriptstyle h}\downarrow & & & & \downarrow{\scriptstyle m} \\
y & & \xrightarrow{\qquad k \qquad} & & z.
\end{array}
$$

This pasting property is essentially just a re-expression of the transitivity and substitution properties of equality, but gives an extraordinarily useful geometric intuition to categorical arguments.

2.1.2   (Iso)morphisms

The philosophy of category theory is that

> *to study an object, one should study its morphisms.*

Indeed, in virtually every natural category, morphisms give enough information to recover the entire data of any object.

Example 2.6. In the following categories, we can reconstruct an object by "probing" it with morphisms from suitable choices of other objects.

- Let X be a set. A function $f : \{*\} \to X$ is exactly a choice of $f(*) \in X$, so the morphisms $\mathrm{SET}(\{*\}, X)$ identify exactly the elements of X, i.e. the entire data of a set.
- Let X be a topological space. A continuous map $f : \{*\} \to X$ picks out the points of X, as before. Let $S = \{0, 1\}$, with $\{1\}$ open; this is the *Sierpinski space*. Then a continuous map $f : X \to S$ consists of a choice of open set $f^{-1}(1) \subseteq X$, so the morphisms $\mathrm{Top}(X, S)$ identify exactly the open sets of X. Together with the points, this is the entire data of a topological space.
- Let G be a group. A group homomorphism $f : \mathbb{Z} \to G$ is determined by a choice of $f(1) \in G$, so these pick out the elements. Letting $-$ be the group homomorphism $\mathbb{Z} \to \mathbb{Z}$ which takes z to $-z$, the composite $f-$ picks out the inverse of the element identified by f. To recover the multiplicative structure, we consider the *free product group* $G \bullet H$, whose elements are words $g_1 h_1 g_2 h_2 \cdots g_n h_n$ modulo the relations of G and H, and whose multiplication is concatenation. There is a canonical map $\varphi : \mathbb{Z} \to \mathbb{Z} \bullet \mathbb{Z}$ given by $1 \mapsto 11'$ (we represent elements in the second copy of $\mathbb{Z}$ with ′s). Given two morphisms $f, g : \mathbb{Z} \to G$, we can define a map $f \bullet g : \mathbb{Z} \bullet \mathbb{Z} \to G$ by $z_1 z_1' \cdots z_n z_n' \mapsto f(z_1) g(z_1') \cdots f(z_n) g(z_n')$. Because the map $(f \bullet g)\varphi : \mathbb{Z} \to G$ picks out exactly the element $f(1)g(1)$, we have recovered the entire structure of G purely by studying $\mathrm{GRP}(\mathbb{Z}, G)$.

These examples are instances of a much more general theory, which we begin to develop here. We first need to formalize what we mean by "recovering the data" of an object.

Definition 2.7 (isomorphism)**.** A morphism $f : x \to y$ in a category C is an *isomorphism* if there exists an *inverse morphism* $g : y \to x$ such that $gf = 1_x$ and $fg = 1_y$. Two objects x and y are *isomorphic*, written $x \cong y$, if there exists an isomorphism between them.

Example 2.8. The general notion of isomorphism recovers the familiar notions in virtually every common setting.

- Every identity morphism is an isomorphism with itself as the inverse.
- Isomorphisms in SET are bijections; in GRP are group isomorphisms; in VECT$_\Bbbk$ are vector space isomorphisms; and in TOP are homeomorphisms.
- Let G be a group with associated category $\mathcal{G}$. Then since composition is group multiplication, every morphism in $\mathcal{G}$ is an isomorphism. (That motivates the following definitions: a *monoid* is a category with one object, while a *group* is a monoid in which every morphism is an isomorphism.)
- Let P be a poset with associated category $\mathcal{P}$. Then antisymmetry of a poset implies that the only isomorphisms in are the identities. (A *preorder* is a category in which every hom-set has at most one element; a *poset* is a preorder in which the only isomorphisms are the identities.)
- Let $M = (Q, \delta)$ be a non-deterministic automaton over the alphabet Σ, so that $\delta : Q \times \Sigma \to \mathcal{P}(\Sigma)$ is the transition function. Recall that the identities in $\mathcal{M}$ are the empty words. As such, an isormorphism between two states q_1 and q_2 is a word w which takes q_1 to q_2, together with a word w' which takes q_2 to q_1, such that the concatenate ww' is the empty string. In other words, w and w' are both empty—so two states are isomorphic if and only if the machine can freely move between them at any point.

Isomorphisms satisfy the basic properties we expect.

Proposition 2.9. *Inverses are unique. Explicitly, if $f : x \to y$ is an isomorphism with inverses $g, h : y \to x$, then $g = h$.*

Proof. We have
$$g = 1_x g = (hf)g = h(fg) = h1_y = h. \qquad \square$$

Notation. We are now justified in unambiguously writing the inverse of an isomorphism f as f^{-1}.

Proposition 2.10. *Isomorphism is an equivalence relation on the class of objects in a category $\mathcal{C}$.*

Proof. We need to show:

- Reflexivity. The identity 1_x is an isomorphism $x \cong x$.
- Symmetry. Given an isomorphism $f : x \to y$, f^{-1} is an isomorphism $y \to x$ with inverse f.
- Transitivity. Given isomorphisms $f : x \to y$ and $g : y \to z$, gf is an isomorphism $x \to z$ with inverse $f^{-1}g^{-1}$. $\qquad\square$

We now take a first step towards justifying the assertion as the beginning of the section.

Proposition 2.11. *Let $x \cong y$ in a category $\mathcal{C}$. Then:*

1. *for every $z \in C$, $C(z,x) \cong C(z,y)$;*

2. *for every $z \in C$, $C(x,z) \cong C(y,z)$.*

Proof. Let $f : x \to y$ be an isomorphism.

First, define a map $f_* : C(z,x) \to C(z,y)$ by post-composition, i.e. $f_*(g) = fg$. We claim that f_*^{-1}, defined similarly, is an inverse of f_*. Letting $h \in C(z,x)$, we have

$$f_*^{-1}(f_*(h)) = f^{-1}(fh) = (f^{-1}f)h = 1_x h = h,$$

and the same on the other side.

Simiarly, define a map $f^* : C(y,z) \to C(x,z)$ by pre-composition, i.e. $f^*(g) = gf$. Then an identical check shows that $(f^{-1})^*$, defined similarly, is an inverse of f^*.   □

To show the other direction, we will need a little bit more machinery. Once shown, this result will indeed imply that the entire structure of an object can be identified by studying its morphisms. We will finally do this in the form of Theorem 2.27.

2.1.3   Functors

Enmeshed in the categorical mindset, we understand that morphisms—relationships— between objects are of crucial importance. Since we now want to study categories, we ask the natural question: what is the right notion of morphism between categories? The answer is a *functor*, which is just a structure-preserving map between categories.

Definition 2.12 (functor)**.** A *functor* $F : C \to \mathcal{D}$ consists of the following data:

- for each object $x \in C$, an object $Fx \in \mathcal{D}$;
- for each morphism $f \in C(x,y)$, a morphism $Ff \in \mathcal{D}(Fx, Fy)$.

This data must preserve the structure of the category, namely identities and composites, meaning:

- for each object $x \in C$, $F1_x = 1_{Fx}$;
- for each pair of morphisms $x \xrightarrow{f} y \xrightarrow{g} z$ in C, $F(gf) = (Fg)(Ff)$.

Example 2.13. In mathematics, functors are ubiquitous as representations of procedures for producing structures of one sort from structures of another. For instance, the following are all functors:

- On any category C, there is an *identity functor* $1_C : C \to C$ which takes each object and morphism to itself.
- There is a functor $\mathcal{P}_\exists : \text{SET} \to \text{SET}$ which takes a set X to its powerset, and a set-function $f : X \to Y$ to the direct image map given by

$$f_\exists(A) = \{y \in Y : \exists a \in A \text{ such that } y = f(a)\}.$$

- There is a distinct functor $\mathcal{P}_\forall : \text{SET} \to \text{SET}$ which takes a set X to its powerset, and a set-function $f : X \to Y$ to the map given by

$$f_\forall(A) = \{y \in Y : \forall x \in X, f(x) = Y \text{ implies } x \in A\}.$$

As these examples show, the action of a functor on morphisms is not determined by its action on objects. (In fact, as usual in category theory, it is the action on morphisms—in particular, on the identities—which determines the action on objects.)

- There is a functor $\text{GL}_n : \text{RING} \to \text{GRP}$ which takes a ring R to the multiplicative group $\text{GL}_n(R)$ of invertible n-by-n matrices with coefficients in R.
- There is another functor $(-)^\times : \text{RING} \to \text{GRP}$ which takes a ring R to the multiplicative group of units in R.
- There is a functor $\text{Maybe} : \text{SET} \to \text{SET}$ which takes a set X to the set $X \sqcup \{\bot\}$, where $\bot$ is a new element, and a function f to its extension by $f(\bot) = \bot$.
- Similarly, there is a functor $\text{List} : \text{SET} \to \text{SET}$ which takes a set X to the set of all finite lists of elements in X, and a set-function f to its mapping over lists, i.e.

$$(\text{List}f)([x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)].$$

In other contexts, this functor is also called the *free monoid* or the *Kleene star*.
- For any field $\Bbbk$, there is a functor $\text{SET} \to \text{VECT}_\Bbbk$ which takes a set X to the $\Bbbk$-span of X, and a set-function f to its linear extension. This is also called the *free vector space*. More generally, any free construction—such as the free group, free ring, etc.—forms a functor.
- Let C be a concrete category, such as those of Example 2.3. Then the *forgetful functor* $U : C \to \text{SET}$ takes each object to its underlying set, and each morphism to its underlying set-function, "forgetting" the additional structure.
- There is also a forgetful functor $\text{RING} \to \text{GRP}$ which takes each ring to its underlying additive group, and each ring homomorphism to its underlying group homomorphism.

Example 2.14. As the following examples show, whenever we can think of each instance of a certain mathematical structure as a category, functors reproduce the right notion of structure-preserving transformation between those structures.

- Let P and Q be posets with associated categories $\mathcal{P}$ and $\mathcal{Q}$. Let $p_1 \leq_P p_2$, so that there is a unique morphism $p_1 \to p_2$ in $\mathcal{P}$. Since F must take this morphism to a morphism $Fp_1 \to Fp_2$, it must hold that $Fp_1 \leq_Q Fp_2$. Furthermore, this is the only requirement on functors, as the statements about identities and composites assert equalities between morphisms, but any two morphisms with the same domain and codomain are equal in a poset. As such, functors between posets are exactly monotone maps.
- Let G and H be groups with associated categories $\mathcal{G}$ and $\mathcal{H}$. A functor $F : \mathcal{G} \to \mathcal{H}$ assigns the single object of $\mathcal{G}$ to the single object of $\mathcal{H}$, and each morphism in $\mathcal{G}$, which is an element $g \in G$, to a morphism (element) $Fg \in H$. That this

preserves composites tells us that it preserves group multiplication, and hence it is a homomorphism. The fact that F preserves identities is extraneous, since every group homomorphism preserves identities. As such, functors between groups are exactly group homomorphisms.

- Let L_1 and L_2 be functional programming languages with associated categories $\mathcal{L}_1$ and $\mathcal{L}_2$. We think of a functor $F : \mathcal{L}_1 \to \mathcal{L}_2$ as an embedding—or, more technically, a *model*—of $\mathcal{L}_1$ in $\mathcal{L}_2$. Specifically, for any function in $\mathcal{L}_1$, F identifies a corresponding function in $\mathcal{L}_2$, and so F allows us to think of computations in L_2 as "simulating" computations in L_1.

The following class of functors are especially important.

Definition 2.15 (hom-functors). Let $x \in C$. There is a functor

$$C(x, -) : C \to \text{SET},$$

the *covariant hom-functor at x*, which takes an object y to the hom-set $C(x, y)$, and a morphism $f : y \to z$ to its action by postcomposition, $f_*(g) = fg$[5].

Since isomorphic objects are meant to look identical to all the machinery of category theory, we should expect the following result.

Proposition 2.16. *Let $F : C \to \mathcal{D}$ be a functor and let $f : x \to y$ be an isomorphism in C. Then $Ff : Fx \to Fy$ is an isomorphism.*

Proof. We have that

$$FfFf^{-1} = F(ff^{-1}) = F1_x = 1_{Fx},$$

and the same works on the other side.   □

Notice that both functorality axioms are exactly what is required to prove this result.

If functors are morphisms between categories, then we should expect that there is a category of categories. This is indeed the case, but we first need to show that functors can be composed.

Proposition 2.17. *Let $F : C \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ be functors. Then there is a* composite *functor $GF : C \to \mathcal{E}$, defined by $(GF)x = G(Fx)$ and $(GF)f = G(Ff)$. Furthermore, this composition is associative and unital, with identities 1_C.*

Example 2.18. The composite of the forgetful functors $\text{RING} \to \text{GRP}$ and $\text{GRP} \to \text{SET}$ is exactly the forgetful functor $\text{RING} \to \text{SET}$.

Definition 2.19. The *category of categories* CAT has categories as objects and functors as morphisms.

The foundationally-inclined reader will correctly object to this definition, which implies that CAT should be an object of itself, leading to issues involving Russell's paradox. There are several resolutions to this—for instance, letting CAT be the category of so-called *locally small* categories, whose hom-sets $C(x, y)$ each form sets. We ignore these issues here.

---

[5]The analogous functor $C(-, x)$ requires a little bit of machinery—the notions of *opposite categories* and *contravariant functors*—which are outside our scope. It will be defined in any introductory text on category theory.

2.1.4   Natural Transformations

The notion of a *natural transformation* can be somewhat mysterious, but is ultimately a workhorse of categorical machinery. We can think of a category C geometrically as a single point, in which case a functor $F : C \to \mathcal{D}$ is an oriented line—an arrow. Two functors $F, G : C \to \mathcal{D}$ look like

$$C \quad \overset{F}{\underset{G}{\rightrightarrows}} \quad \mathcal{D}.$$
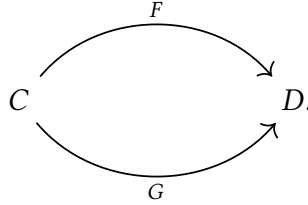
A natural transformation is a square—or, if you prefer, a disk—which "fills in the hole":

$$C \quad \Downarrow \alpha \quad \mathcal{D}.$$
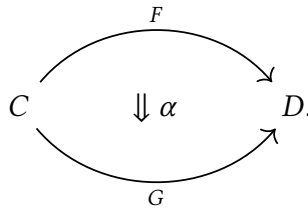
In other words, a natural transformation is a morphism between functors.

More concretely, recall that functors can be thought of as tools which, given a structure of one kind, produce one of another. In this sense, natural transformations are a mechanism for converting between such constructions. For each object $x \in C$, we have two ways to construct an object of $\mathcal{D}$, i.e. Fx and Gx. Of course, objects of $\mathcal{D}$ are related by morphisms, so a natural transformation $\alpha : F \Rightarrow G$ should identify a morphism $\alpha_x : Fx \to Gx$ for each $x \in C$.

This is not quite enough. We want to ensure that the morphisms α_x are somehow "consistent", that the natural transformation represents a procedure to convert between the two constructions F and G "independenty of x". We formalize that intuition now.

Definition 2.20 (natural transformation). Let $F, G : C \to \mathcal{D}$ be functors. A *natural transformation* $\alpha : F \Rightarrow G$ consists of, for every object $x \in C$, a *component* $\alpha_x : Fx \to Gx$ such that, for every morphism $f : x \to y$ in C, the following diagram (a *naturality square*) commutes:

$$
\begin{array}{ccc}
Fx & \xrightarrow{\alpha_x} & Gx \\
\scriptstyle Ff \downarrow & & \downarrow \scriptstyle Gf \\
Fy & \xrightarrow[\alpha_y]{} & Gy.
\end{array}
$$

The idea is that it does not matter whether we first move from x to y via *any morphism* f, or first move from F to G via α; natural transformations commute with any morphism. This is the sense in which natural transformations are natural.

Example 2.21. There are many important examples of natural transformations.

- For any functor F, there is an *identity natural transformation* $1_F : F \Rightarrow F$, whose components are each the identities $(1_F)_x = 1_{Fx}$.
- There is a natural transformation $\alpha : 1_{\text{SET}} \Rightarrow \mathcal{P}_\exists$ with components $\alpha_X : x \mapsto \{x\}$.
- The *dual* of a vector space V over $\Bbbk$ is the vector space of linear maps into $\Bbbk$, i.e. $V^* = \text{VECT}_\Bbbk(V, \Bbbk)$. There is a natural transformation $\alpha : 1_{\text{VECT}} \Rightarrow (-)^{**}$ whose components α_V take any $v \in V$ to the map $\text{ev}_v : \mathbb{V}^* \to \Bbbk$ given by $T \mapsto Tv$.
- There is a natural transformation $\det : \text{GL}_n \Rightarrow (-)^*$, where R^* is the ring of units from Example 2.13, which takes the determinant of an invertible matrix.
- Recall from Example 2.14 that functors between posets are exactly monotone maps. A natural transformation $\alpha : F \Rightarrow G$ between two monotone maps $\mathcal{P} \to \mathcal{Q}$ consists of, for each $p \in \mathcal{P}$, a morphism $Fp \to Gp$. Since $\mathcal{Q}$ is a poset, there is at most one such morphism, and it exists if and only if $Fp \le Gp$. As such, there can only be one such natural transformation, and it exists if and only if $F \le G$ in the pointwise ordering.
- Let $F, G : \mathcal{L}_1 \to \mathcal{L}_2$ be models of a programming language L_1 in L_2. A natural transformation $\alpha : F \Rightarrow G$ is a *transpilation* between the models: it tells us how to convert programs written in the model F into programs written in the model G. The naturality squares assert exactly that this transpilation is *sound*, i.e. that it preserves the meaning of programs.

Example 2.22. Here are three natural transformations common in functional programming.

- Let reverse_X be the function which reverses lists of elements in X, i.e.

$$[x_1, \ldots, x_n] \mapsto [x_n, \ldots, x_1].$$

  Then reverse is a natural transformation $\text{List} \Rightarrow \text{List}$.
- Let head_X be the function which gets the first element of a list if it exists, i.e.

$$[x_1, \ldots, x_n] \mapsto x_1, \qquad [] \mapsto \perp.$$

  Then head is a natural transformation $\text{List} \Rightarrow \text{Maybe}$.
- Let toList_X be the function $\text{Maybe}X \to \text{List}X$ given by

$$x \mapsto [x], \qquad \perp \mapsto [].$$

  Then toList is a natural transformation $\text{Maybe} \Rightarrow \text{List}$.

Each of these are special cases of the so-called *Reynolds abstraction theorem* from programming language theory, which says that (parametrically) polymorphic functions are natural [Rey83]. This theorem is explored in great detail by [Wad89].

If we think of natural transformations as morphisms between functors $\mathcal{C} \to \mathcal{D}$, then following the category-theoretic philosophy, there should be a category of functors. Indeed, natural transformations can be composed, as follows.

Proposition 2.23. *Let $F, G, H : C \to \mathcal{D}$ be functors and let $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$ be natural transformations. Then there is a* vertical composite *natural transformation $\beta\alpha : F \Rightarrow H$, whose components are $(\beta\alpha)_x = \beta_x \alpha_x$. Furthermore, this composition is associative and unital, with identities 1_F.*

The name *vertical composite* comes from the following picture:

As the name implies, there is a horizontal composite, defined in e.g. [Rie17, Lemma 1.7.4].

Definition 2.24 (functor category)**.** Let C and $\mathcal{D}$ be categories. The *functor category* $[C, \mathcal{D}]$ has functors $C \to \mathcal{D}$ as objects and natural transformations as morphisms.

Here is one example of the advantage of working with categorical structure: we already know what the notion of an isomorphism of functors has to be.

Definition 2.25 (natural isomorphism)**.** Let $F, G : C \to \mathcal{D}$ be functors. A natural transformation $\alpha : F \Rightarrow G$ is a *natural isomorphism* if it is an isomorphism in the category $[C, \mathcal{D}]$.

Proposition 2.26. *Let $F, G : C \to \mathcal{D}$ be functors. A natural transformation $\alpha : F \Rightarrow G$ is a natural isomorphism if and only each of its components $\alpha_x : Fx \to Gx$ are isomorphisms in $\mathcal{D}$.*

We can now state the correct form of the converse to Proposition 2.11.

Theorem 2.27. *Let x and y be objects in a category C such that $C(x, -) \cong C(y, -)$. Then $x \cong y$.*

Proof. Let $\eta : C(x, -) \Rightarrow C(y, -)$ be a natural isomorphism. Define

$$t = \eta_x(1_x),$$

which is a morphism $y \to x$, and

$$u = \eta_y^{-1}(1_y),$$

which is a morphism $x \to y$. We claim these are inverses.

Naturality of η applied to u asserts that

$$
\begin{array}{ccc}
C(x, x) & \xrightarrow{\eta_x} & C(y, x) \\
{\scriptstyle u_*}\downarrow & & \downarrow{\scriptstyle u_*} \\
C(x, y) & \xrightarrow{\eta_y} & C(y, y)
\end{array}
$$

commutes. Following 1_x around the top and right, we get

$$u_*(\eta_x(1_x)) = u_*(t) = ut,$$

while on the left and bottom we get

$$\eta_y(u_*(1_x)) = \eta_y(u) = 1_y,$$

so commutativity implies $ut = 1_y$.

Similarly, naturality of η^{-1} applied to t asserts that

$$
\begin{array}{ccc}
C(y,y) & \xrightarrow{\ \eta_y^{-1}\ } & C(x,y) \\
{\scriptstyle t_*}\downarrow & & \downarrow{\scriptstyle t_*} \\
C(y,x) & \xrightarrow[\eta_x^{-1}]{} & C(x,x)
\end{array}
$$

commutes. Following 1_y around the top and right, we get

$$t_*(\eta_y^{-1}(1_y)) = t_*(u) = tu,$$

while on the left and bottom we get

$$\eta_x^{-1}(t_*(1_y)) = \eta_x^{-1}(t) = 1_x,$$

so again $tu = 1_x$. This completes the proof. □

This theorem is a special case of the *Yoneda lemma*, arguably the most important theorem in category theory. The contravariant result, with the functors $C(-, x)$, is also true, but outside our scope. Together, these theorems tell us that objects in a category are indeed determined by their morphisms.

2.2 Monoidal Categories

In ordinary categories, composition is sequential: if morphisms are interpreted as computational processes, the composite gf means roughly "first do f, then do g." In many settings, we want to consider both sequential and parallel composition. The categorical axiomatization of this idea is *monoidal categories*.

2.2.1 The Definition

To model parallel composition, we want an binary operation $\otimes$ which assigns, to each pair of processes (morphisms) $f : x \to y$ and $g : w \to z$, their parallel composite $f \otimes g$. If we think of objects as types, this parallel composite can only run given inputs of both types x and w, to feed to f and g respectively, and should produce two outputs of types y and z. To represent this notion, we also need a way to pair types (objects), which means a

binary operation also called $\otimes$ on morphisms. This dual assignment on both objects and morphisms suggests functoriality: we will ask that $\otimes$ is a functor $C \times C \to C$.

What axioms should this data satisfy? As in most well-behaved algebraic structures, there should be an identity for $\otimes$ on objects, which we will write I. Computationally, we may think of I as a "trivial resource," which may freely be created and has no uses. This I induces an identity, the morphism 1_I, for $\otimes$ on morphisms, so we do not need to add an identity on morphisms as an extra axiom. We would also parallel composition to associate, so that we can sensibly talk about performing n processes in parallel. It is therefore tempting to list the following axioms:

$$I \otimes x = x = x \otimes I; \qquad (x \otimes y) \otimes z = x \otimes (y \otimes z).$$

While this notion, called a *strict monoidal category*, is useful, it is not the most natural axiomatization. For instance, even the category Set, with the ordinary Cartesian product, is not strictly monoidal: the identity is $\{*\}$, but $\{*\} \times X$ is not equal to X, instead merely isomorphic. The point is that there is interesting structure in the way that even isomorphic objects relate to each other; we do not want to lose it by forcing strict equality.

However, we do not want to allow the strcture of these natural isomorphisms to be too strange. For instance, one can imagine two ways to convert from $I \otimes (x \otimes y)$ to $x \otimes y$:

$$I \otimes (x \otimes y) \cong x \otimes y \quad \text{and} \quad I \otimes (x \otimes y) \cong (I \otimes x) \otimes y \cong x \otimes y.$$

The first directly uses unitality, while the second associates and then uses unitality. A *coherence axiom* asserts that choices like this do not matter: every pair of comopsites of our canonical isomorphisms with the same domain and codomain should commute.

We are not quite ready; there is one remaining technical issue, though this paragraph may be safely skipped. It may happen that two domains equate "accidentally", so that, for instance,

$$((x \otimes y) \otimes z) \otimes w = x \otimes (y \otimes (z \otimes w)). \tag{2.1}$$

In this case, the version of the coherence axiom stated above implies that the isomorphisms

$$((x \otimes y) \otimes z) \otimes w \cong (x \otimes y) \otimes (z \otimes w) \quad \text{and} \quad x \otimes (y \otimes (z \otimes w)) \cong (x \otimes y) \otimes (z \otimes w)$$

should commute; they do, after all, have the same domain and codomain. But the first re-associates from the left to the right, and the second re-associates from the right to the left: these are structurally different actions, which only "look the same" because of the accident of Equation 2.1, so our theory should not require them to commute. There is a way to formalize a correct abstract notion of coherence—see for instance [Mac71, subsection VII.2]—but fortunately, Mac Lane's *coherence theorem* enables an easier axiomatization.

We are finally now ready to state the definition of a monoidal category.

Definition 2.28 (monoidal category). A *monoidal category* C consists of the following data:

- an underlying category C;
- a functor $\otimes : C \times C \to C$, called the *tensor product*;

- an object $I \in C$, called the *tensor unit*;
- a natural isomorphism $\alpha_{x,y,z} : (x \otimes y) \otimes z \to x \otimes (y \otimes z)$, called the *associator*;
- a natural isomorphism $\lambda_x : I \otimes x \to x$, called the *left unitor*[6];
- a natural isomorphism $\rho_x : x \otimes I \to x$, called the *right unitor*.

This data must make the following diagrams, called the *triangle* and *pentagon* identities, commute:

$$
\begin{array}{ccc}
(x \otimes I) \otimes y & \xrightarrow{\ \alpha_{x,1_\otimes,y}\ } & x \otimes (I \otimes y) \\
 & \searrow{\rho_x} \quad \swarrow{\lambda_x} & \\
 & x \otimes y &
\end{array}
$$

$$
\begin{array}{ccc}
 & (x \otimes y) \otimes (z \otimes w) & \\
 \nearrow{\alpha_{x\otimes y,z,w}} & & \searrow{\alpha_{x,y,z\otimes w}} \\
((x \otimes y) \otimes z) \otimes w & & x \otimes (y \otimes (z \otimes w)) \\
\downarrow{\alpha_{x,y,z}\otimes 1_w} & & \uparrow{1_x \otimes \alpha_{y,z,w}} \\
(x \otimes (y \otimes z)) \otimes w & \xrightarrow[\ \alpha_{x,y\otimes z,w}\ ]{} & x \otimes ((y \otimes z) \otimes w).
\end{array}
$$

The above diagrams look arbitrary, but as mentioned, they are exactly what is required for the correct notion of coherence. On first exposure to these ideas, it is safe to ignore the exact statement of the identities and work with the intuition that any two ways of associating or unitalizing should be the same.

In the above definition, the natural isomorphisms α, λ, and ρ feel in some sense more like axioms than data. This is another key component of the category-theoretic philosophy, one which should feel comfortable to computer scientists, who often assume the existence of concrete objects which structure our models:

<div align="center">

structure is a kind of data.

</div>

If we think of categories as algebras of structure, it is natural that we should think of axiomatic structure as an algebraic object which may be manipulated[7].

2.2.2   Examples

The notion of a monoidal category is quite general; we survey some important examples here.

---

[6]The letters λ and ρ are chosen for their association with L and R, respectively.

[7]Of course, Definition 2.28 still carries a traditional-looking equational theory in the form of the triangle and pentagon identities. The key difference is that this theory is an assumption about the "two-dimensional" structure of the natural transformations, whereas associativity and unitality are assumptions about the "one-dimensional" structure of the functor $\otimes$. We could continue to generalize, instead asking that these diagrams are themselves witnessed by "three-dimensional" isomorphisms between the natural isomorphisms α, λ, and ρ. Repeating this process *ad infinitum*, the natural endpoint of the structure-as-data philosophy is so-called ∞-*category theory*.

Example 2.29. Let us very explicitly construct the required data to show that SET is a monoidal category under the Cartesian product. The tensor unit is the singleton $\{*\}$. The associator is the natural isomorphism with components

$$\alpha_{X,Y,Z} : (X \times Y) \times Z \to X \times (Y \times Z)$$
$$((x, y), z) \mapsto (x, (y, z)).$$

The left and right unitors are the natural isomorphism with components

$$\lambda_X : \{*\} \times X \to X \qquad\qquad \rho_X : X \times \{*\} \to X$$
$$(*, x) \mapsto x, \qquad\qquad (x, *) \mapsto x.$$

A common complaint about category theory is at play here: we now have a large number of relationships to demonstrate, including functorality of $\times$, naturality of α, λ, and ρ, and the pentagon and triangle identities. The author's opinion is that this work will ultimately save effort, by allowing us to use a powerful abstract theory across any structure we have shown to be monoidal, but if the reader is not convinced, one solution is to work even more generally. For instance, by showing that the Cartesian product satisfies a simple property called the *universal property of the product*, we could automatically conclude on the grounds of a general theorem that it is monoidal. Abstraction of this sort ultimately saves effort, but it is not always comfortable at first. Regardless, in order to exemplify the definition in all its detail, we continue with the explicit demonstration.

To show functorality of $\times$, we need to determine its action on morphisms. Letting $f : X \to Y$ and $g : W \to Z$, we define

$$f \times g : X \times W \to Y \times Z$$
$$(x, y) \mapsto (f(x), g(y)).$$

This is functorial: it takes an identity $1_{(X,W)} = (1_X, 1_W)$ to $1_{X \times W}$, and the composite of two pairs of morphisms to composite of their action on pairs.

To show naturality of α, let $(f, g, h) : (X, Y, Z) \to (X', Y', Z')$ be a morphism in SET^3. We need to show that the following diagram commutes:

$$
\begin{array}{ccc}
(X \times Y) \times Z & \xrightarrow{\ \alpha_{X,Y,Z}\ } & X \times (Y \times Z) \\
\downarrow{\scriptstyle (f \times g) \times h} & & \downarrow{\scriptstyle f \times (g \times h)} \\
(X' \times Y') \times Z' & \xrightarrow[\alpha_{X',Y',Z'}]{} & X' \times (Y' \times Z').
\end{array}
$$

Tracking the action of a triple $((x, y), z)$ through both paths, we see the needed equality:

$$
\begin{array}{ccc}
((x, y), z) & \xmapsto{\ \alpha_{X,Y,Z}\ } & (x, (y, z)) \\
\downarrow{\scriptstyle (f \times g) \times h} & & \downarrow{\scriptstyle f \times (g \times h)} \\
((f(x), g(y)), h(z)) & \xmapsto[\alpha_{X',Y',Z'}]{} & (f(x), (g(y), h(z))).
\end{array}
$$

To show naturality of λ, let $f : X \to Y$. Since the only morphism $\{*\} \to \{*\}$ is $1_{\{*\}}$, naturality is entailed by commutativity of the following diagram:

$$
\begin{array}{ccc}
\{*\} \times X & \xrightarrow{\lambda_X} & X \\
{\scriptstyle 1_{\{*\}} \times f} \downarrow & & \downarrow {\scriptstyle f} \\
\{*\} \times Y & \xrightarrow[\lambda_Y]{} & Y,
\end{array}
\qquad \text{i.e.} \qquad
\begin{array}{ccc}
(*, x) & \xmapsto{\lambda_X} & x \\
{\scriptstyle 1_{\{*\}} \times f} \uparrow & & \downarrow {\scriptstyle f} \\
(*, f(x)) & \xmapsto[\lambda_Y]{} & f(x).
\end{array}
$$

Naturality of ρ is similar. We show the pentagon identity by its action on $(((x, y), z), w)$:

$$
\begin{array}{ccc}
& ((x, y), (z, w)) & \\
{\scriptstyle \alpha_{X \times Y, Z, W}} \nearrow & & \nwarrow {\scriptstyle \alpha_{X, Y, Z \times W}} \\
(((x, y), z), w) & & (x, (y, (z, w))) \\
{\scriptstyle \alpha_{X, Y, Z} \times 1_W} \downarrow & & \uparrow {\scriptstyle 1_X \times \alpha_{Y, Z, W}} \\
((x, (y, z)), w) & \xrightarrow[\alpha_{X, Y \times Z, W}]{} & (x, ((y, z), w)).
\end{array}
$$

The triangle identity is similar.

While we will never again be so explicit, we hope the previous example makes the axioms of a monoidal category more concrete.

Example 2.30. There are many more examples of monoidal categories throughout mathematics.

- $\textsc{Vect}_{\Bbbk}$ is monoidal with the tensor product of vector spaces.
- $\textsc{Cat}$ is monoidal with the product category.
- Let L be a strongly-typed functional programming language with *product types*, which means that for any types A and B, there is a type $A \times B$ whose elements are pairs (a, b) of elements $a \in A$ and $b \in B$. Then the category $\mathcal{L}$ is monoidal in the same way as $\textsc{Set}$.

Example 2.31 (concurrent programming [MM90])**.** Returning to our motivation of parallelism, here is a very different example. Let L be a strongly-typed functional *concurrent* programming language, by which we mean that it can run computations concurrently on different machine threads. Then again under reasonable assumptions, $\mathcal{L}$ is monoidal, with concurrent branching as the tensor product and the do-nothing program as the tensor unit.

2.2.3   String Diagrams

In monoidal categories, there are two "formal mechanisms" for building morphisms: sequential composition $\circ$ and parallel composition $\otimes$. String diagrams are a graphical calculus for morphisms using these mechanisms. String diagrams and related calculi are explored in great detail by [Sel11]; we give a basic outline here.

Consider a monoidal category C with three morphisms $f : x \to y$, $g : w \to z$, and $h : y \otimes z \to u$. We can can form a new morphism $h \circ (f \otimes g) : x \otimes w \to u$. We encode this new morphism in the following *string diagram*, written bottom-up:
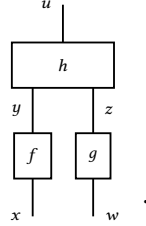

Explicitly, the idea is as follows. A morphism is a labelled box, with "wires"[8] coming into and out of labelled with the domain and codomain. We can hook up two wires representing the same object—this is sequential composition. We can also place boxes or wires side-by-side—this is parallel composition. Accordingly:

where in the second equality we have assumed $y = w$, so that the composition makes sense. As the left hand side of the second equality suggests, we often suppress the label of "intermediate" wires, as they are implicit from the types of the morphisms; in fact, we may even at times suppress the labels of the input and output wires. Finally, if there is no box, then a wire may be read as the identity for its type.

Consider the following diagram:
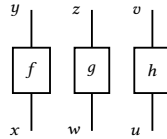
Do we read this as $(f \otimes g) \otimes h$ or $f \otimes (g \otimes h)$? There is not an unambiguous choice, but fortunately the coherence theorem, discussed in Section 2.2.1, means that there is a unique natural isomorphism equating these morphisms. As such, the general rule is that *string diagrams define morphisms up to unique natural isomorphism.*

Similarly, since wires of type I can be created or destroyed at will using λ and ρ, we just do not draw such wires. A morphism with domain or codomain I is represented with

---

[8]As the word "wire" suggests, a string diagram can be thought of as a circuit, where the morphisms/boxes are thought of as gates. This correspondence has recently been made precise by [BS22], but the analogy is much older, and it is a useful intuition even without any rigor. This analogy and many others are discussed in [BS11].

a triangle, so that for instance if $f : I \to x$ and $g : x \to I$, then
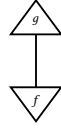
is the morphism $gf : I \to I$.

Sometimes, we work in settings which have some "distinguished" morphisms, in which case we will often write them merely with dots. For instance, recall that a *classical monoid* is a set X together with an associative unital binary operation. Recalling from Example 2.6 that the distinguished unit element $e \in X$ can be associated with the unique set-function $\{*\} \to X$ defined by $* \mapsto e$, we generalize the notion of a monoid as follows.

Definition 2.32 (monoid object)**.** Let C be a monoidal category. A *monoid object* in C is an object m together with distinguished morphisms
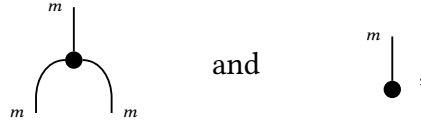
called the *multiplication* and *unit* respectively and traditionally written μ and η. This data must make the equalities

                                                                                                (2.2)

and

                                                                                                (2.3)

hold.

Let us be very explicit about what these equalities say. Equation 2.2 takes in three wires of type m. On the left, it associates them to the left, so we start with $(m \otimes m) \otimes m$. We first multiply on the left while doing nothing on the right, and then multiply the product with the thing on the right: this is the composite morphism

$$(m \otimes m) \otimes m \xrightarrow{\mu \otimes 1_m} m \otimes m \xrightarrow{\mu} m.$$

On the right, the ms are associated to the right, so we have the composite morphism

$$m \otimes (m \otimes m) \xrightarrow{1_m \otimes \mu} m \otimes m \xrightarrow{\mu} m.$$

It may be worrying that these morphisms have different domains, but as discussed, string diagrams merely identify morphisms up to coherence isomorphism. As such, for the axiom to make sense, there should be a canonical natural isomorphism making the domains and codomains of these morphisms equate, and indeed there is: $\alpha_{m,m,m}$ for the domains, and just the identity for the codomains. Thus, Equation 2.2 asserts commutativity of the diagram

$$
\begin{array}{ccc}
(m \otimes m) \otimes m & \xrightarrow{\;\alpha_{m,m,m}\;} & m \otimes (m \otimes m) \\
{\scriptstyle \mu \otimes 1_m}\downarrow & & \downarrow{\scriptstyle 1_m \otimes \mu} \\
m \otimes m & & m \otimes m \\
& {\scriptstyle \mu}\searrow \quad \swarrow{\scriptstyle \mu} & \\
& m. &
\end{array}
$$

Meanwhile, Equation 2.3 features three morphisms. On the left, we have

$$
m \otimes I \xrightarrow{\;1_m \otimes \eta\;} m \otimes m \xrightarrow{\;\mu\;} m,
$$

in the middle we have the identity $1_m : m \to m$, while on the right we have

$$
I \otimes m \xrightarrow{\;\eta \otimes 1_m\;} m \otimes m \xrightarrow{\;\mu\;} m.
$$

Again, the domains are related by the canonical isomorphisms λ and ρ. We can write this equality as commutativity of the diagram

$$
\begin{array}{ccccc}
I \otimes m & \xrightarrow{\;\eta \otimes 1_m\;} & m \otimes m & \xleftarrow{\;1_m \otimes \eta\;} & m \otimes I \\
& {\scriptstyle \lambda_m}\searrow & \downarrow{\scriptstyle \mu} & \swarrow{\scriptstyle \rho_m} & \\
& & m, & &
\end{array}
$$

where we suppress the identity 1_m, which could appear at the bottom of the diagram.

In the following two sections, we will give several examples of definitions—in particular *braided monoidal categories*, *symmetric monoidal categories*, and *monoidal functors*—whose coherence axioms are better understood diagramatically than symbolically. While the axioms themselves are useful to understand, for our purposes it is more important to understand the intuition of the structures in question and their relationship to the graphical calculi. If the reader understands how the diagrams relate to each other, it is generally safe to move on even without a complete understanding of how they are translated into symbolic equalities. The interested reader may find a symbolic statement of the coherence laws in [Mac71, Chapter XI].

2.2.4 Symmetry

While monoidal categories are necessarily associative, nothing in the definition guarantees that the monoidal product is commutative. As usual, it is too stric to ask for commutativity $x \otimes y = y \otimes x$ as an equational axiom. When we want commutativity, we instead

add a natural isomorphism $\gamma_{x,y} : x \otimes y \to y \otimes x$, called the *braiding*, to the data, so named because of its string-diagrammatic representation:

That there are *four* possible braids we could draw involving x and y, each of which is *a priori* a different morphism:

$$\gamma_{x,y} = \quad , \qquad \gamma_{x,y}^{-1} = \quad , \qquad \gamma_{y,x} = \quad , \qquad \gamma_{y,x}^{-1} = \quad .$$

That $\gamma_{x,y}^{-1}$ is indeed an inverse asserts that

as is suggested by our geometric intuitions[9].

What coherence axioms should this satisfy—in other words, what manipulations should we be allowed to make to the our diagrams? It should certainly be coherent with the identity:

$$= \qquad = \qquad \tag{2.4}$$

It should also not matter if we braid twice, or braid once with a tensor, in the sense that:

$$= \qquad \text{and} \qquad = \qquad . \tag{2.5}$$

The previous two axioms define a *braided monoidal category*.

We will care primarily about the stronger case in which

$$= \quad , \tag{2.6}$$

---

[9]It is, in fact, possible to formalize string diagrams geometrically, using the technology of knot theory; this is due to [JS91].

which we may then unambiguously write

$$\bigtimes \; ;$$

we call this map the *symmetry*.

Definition 2.33 (symmetric monoidal category)**.** A *symmetric monoidal category* is a monoidal category C, together with a natural isomorphism $\gamma_{x,y} : x \otimes y \to y \otimes x$, called the *braiding* or *symmetry*, satisfying the coherence laws of Equations 2.4 to 2.6.

Example 2.34. The categories Set, Vect$_{\Bbbk}$, and Cat, with the monoidal structure defined in Section 2.2.2, are all symmetric monoidal.

2.2.5 Monoidal Functors

Let us work out what a "monoidal functor" between monoidal categories should be. Let C and $\mathcal{D}$ be monoidal categories, and annotate their respective data with subscripts, so that for instance $\otimes_C$ is the tensor of C. Let $F : C \to \mathcal{D}$ be a functor.

A sensible choice is to ask for a natural isomorphism $\phi_{x,y} : Fx \otimes_{\mathcal{D}} Fy \to F(x \otimes_C y)$, satisfying certain coherence identities. However, this is often too strong for our purposes. For instance, Maybe does not satisfy this definition: while there is a map

$$\text{Maybe}X \otimes \text{Maybe}Y \to \text{Maybe}(X \otimes Y)$$
$$(x, y) \mapsto (x, y) \quad (x, \bot) \mapsto \bot \quad (\bot, y) \mapsto \bot,$$

and so Maybe respects the monoidal structure in some weaker sense, this map is not an isomorphism. Instead, we will often just ask $\phi_{x,y}$ to be a morrphism, which tells us how to "convert" tensor products in $\mathcal{D}$ into tensor products in the model of $\mathcal{D}$[10]. We also need F to be compatible with the tensor unit, for which we ask for an morphism $\phi : I_{\mathcal{D}} \to FI_C$.

There is a graphical calculus for monoidal functors due to [CS99]; we give a presentation following [Mel06]. The idea is to represent functors as colored boxes which separate the "inside world" of C from the "outside world" of $\mathcal{D}$, so that we may depict the morphism $Ff : Fx \to Fy$ as

.

---
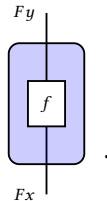
[10]A careful reader may wonder why the morphism goes from $\otimes_{\mathcal{D}}$ to $\otimes_C$, rather than the other way. We do sometimes study the latter under the name *colax monoidal functors*, but the former is far more common. One way to understand this is that the former direction says that tensor in $\mathcal{D}$ is in some sense "more precise" than tensor in C, which tends to be why the functor is interesting in the first place.

If F is monoidal, we write the morphism $\phi_{x,y}$ as
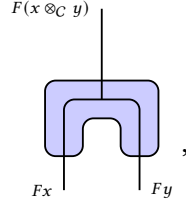
the idea being that at first the blue-shaded wires x and y are connected by white space representing the tensor $\otimes_{\mathcal{D}}$, and then it becomes blue space representing the tensor $\otimes_{\mathcal{C}}$. We often don't write the top part of this morphism, instead doing manipulation inside $\mathcal{C}$, which happens in the blue shading. For instance, coherence with the identity states that

$$(2.7)$$

Similarly, compatibility with the associator asserts that

$$(2.8)$$

while when $\mathcal{C}$ and $\mathcal{D}$ are symmetric monoidal, we might also want compatibility with the symmetry:

$$(2.9)$$

Definition 2.35 (monoidal functor). A functor $F : \mathcal{C} \to \mathcal{D}$ between monoidal categories is *lax monoidal*, or just *monoidal*, if there is a natural transformation $\phi_{x,y} : Fx \otimes_{\mathcal{D}} Fy \to F(x \otimes_{\mathcal{C}} y)$ and a morphism $\phi : I_{\mathcal{D}} \to FI_{\mathcal{C}}$ satisfying Equations 2.7 and 2.8. It is further *symmetric* if $\mathcal{C}$ and $\mathcal{D}$ are symmetric monoidal categories and the data satisfies Equation 2.9. Finally, it is *strong monoidal* if $\phi_{x,y}$ and ϕ are isomorphisms.

Example 2.36. Again, there are many familiar monoidal functors.

- Maybe is (lax) symmetric monoidal, but not strong monoidal.

- For any monoidal category C, $C(I, -)$ is monoidal, with the coherence

$$\phi_{x,y} : C(I, x) \times C(I, y) \rightarrow C(I, x \otimes y)$$

$$(f, g) \mapsto \;\;\raisebox{-1em}{\includegraphics}\;\;.$$

If C is symmetric, then so too is $C(I, -)$.

- The $\Bbbk$-span functor is strong symmetric monoidal. In fact, this is one definition of the tensor of vector spaces:

$$\operatorname{span}_{\Bbbk} X \otimes \operatorname{span}_{\Bbbk} Y = \operatorname{span}_{\Bbbk}(X \times Y).$$
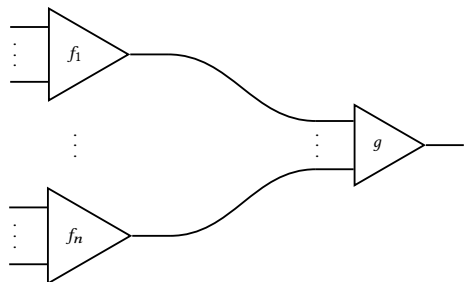
2.2.6   Multicategories

While monoidal categories are extremely elegant structures, as seen in Example 2.29 it can be tedious to construct all the required data. In this section, we give a useful tool for constructing monoidal categories via a related structure called a *multicategory*, which is just a category whose morphisms may take multiple inputs.
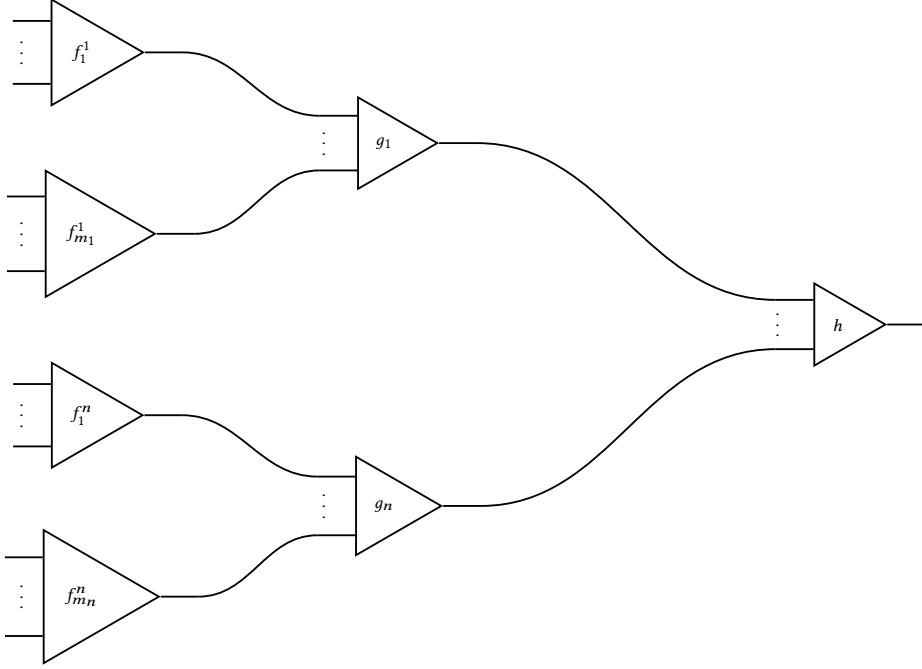
As usual, it is often easier to understand the algebric structure diagramtically than symbolically. A morphism in a multicategory looks like
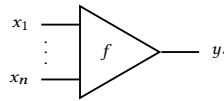
Such morphisms can be composed when the domains and codomains line up, as in

This composition is (strictly) associative, which just means that the composite

is well-defined.

We now state the explicit definition.

Definition 2.37 (multicategory)**.** A *multicategory* consists of the following data:

- a collection of objects C;
- for any finite list of objects $x_1, \ldots, x_n \in C$ and any object $y \in C$, a collection of morphisms $C(x_1, \ldots, x_n; y)$;
- for any finite list of morphisms $f_1, \ldots, f_n$, and any morphism g such that the domain of g has length n and the ith object in its domain is the codomain of f_i, a composite morphism $g \circ (f_1, \ldots, f_n)$ whose domain is the concatenation of the domains of the f_is and whose codomain is the codomain of g;
- for any object $x \in C$, a morphism $1_x \in C(x; x)$.

This data must satisfy the following associativity and unitality axioms:

- for any morphism $(x_1, \ldots, x_n) \xrightarrow{f} y$,

$$1_y \circ f = f = f \circ (1_{x_1}, \ldots, 1_{x_n});$$

- for any morphisms $h, g_1, \ldots, g_n, f_1^1, \ldots, f_1^{m_1}, \ldots, f_n^1, \ldots, f_n^{m_n}$ where the composites are defined,

$$h \circ (g_1 \circ (f_1^1, \ldots, f_1^{m_1}), \ldots, g_n \circ (f_n^1, \ldots, f_n^{m_n}))$$
$$= (h \circ (g_1, \ldots, g_n)) \circ (f_1^1, \ldots, f_1^{m_1}, \ldots, f_n^1, \ldots, f_n^{m_n}).$$

Notation. The *arity* of a morphism in a multicategory is the length of its domain list.

Example 2.38. Any category is a multicategory, with no morphisms of arity greater than one. Such multicategories are called *unary*.

Example 2.39. Any monoidal category has an *underlying multicategory*, where the maps $x_1, \ldots, x_n \to y$ are exactly the maps $x_1 \otimes \cdots \otimes x_n \to y$.

Bibliographic Notes

There are many different ways to visually present string diagrams. In Section 2.2.3, we have followed the style of [BK22] very closely. We program our diagrams in part using the Ti*k*Z code of [BK22], and code due to Zajj Daugherty.

Chapter 3

Categorical Cryptography

A theory of cryptography should define at least four things: computation, protocols, adversarial behavior, and security. A major advantage of categorical models of cryptography is that they conveniently separate these issues. In particular, we have some underlying category of computations, while we represent categories of protocols with functorial constructions on categories; as such, our notions of interaction and security are completely independent of the underlying model of computation.

[todo: lots more intro, cite [BK22]; categorical crypto theories [Hin20; Pav12; Pav14; SV13], caterogies for specific crypto protocols [BKM19; BMR19]; categorical qm [AC04; CP12; HV19; CK17; CG19]]

3.1   Computation

The categorical theory of computation is well-developed, going back at least to the work of Jim Lambek and several contemporaries around the 1970s [Lam74; Lam80; Law69; See84]. The essential idea generalizes Example 2.2: objects are types and morphisms are typed computations. The most disciplined approach is to consider the categorical structure needed to model certain forms of computation, so that for instance models of simply typed computation are *bicartesian closed categories* [Lam74], of linear computation are *star-autonomous categories* [See89], of quantum computation are *compact-closed categories* [AC04], and of probabilistic computation are *Markov categories* [Fri20]. We will not review this approach here. Instead, our focus will be on constructing specific categorical models of forms of computations of interest to cryptographers.

3.1.1   Deterministic Computation

We would like a category of deterministic computations to have computable functions as morphisms. However, the natural choice, taking sets as objects and computable functions as morphisms, is actually not yet precise. The first issue is that there are several distinct notions of computability on uncountable sets. Each such notion forms a category, but formal definitions are outside our scope, as cryptographers tend not to care about

uncountable sets anyway[1].

   We can resolve this issue simply, by limiting ourselves to finite sets, in which case every function is computable (simply by a lookup table):

Definition 3.1 (category of finite sets). The *category of finite sets*, FinSet, has finite sets as objects and functions as morphisms.

   However, we often want to work with larger input spaces. The natural guess is to take countable sets and computable functions. The issue here is one of encoding: there is a canonical notion of computability on the set of finite binary strings, but elements of arbitrary sets do not generally have canonical encodings as binary strings. We could solve this issue by limiting ourselves to working only with binary strings:

Definition 3.2 (category of computable binary functions). The *category of computable binary functions* BinComp has sets of binary strings $A \subseteq \{0, 1\}^*$ as objects and computable functions as morphisms.

   In practice, however, we like to think of computations as working over arbitrary sets, which in particular may have more algebraic structure than sets of binary strings. Our strategy, following [Pav14], will be to work over sets with fixed binary encodings.

Definition 3.3 (binary-encoded set). A *binary-encoded set* is a set X together with an injection $\llbracket - \rrbracket_X : X \to \{0, 1\}^*$, called the *encoding*.

   Note that every binary-encoded set is finite or countable; as such, we avoid the issues with uncountable sets mentioned above.

Notation. When the context is clear, we will generally drop the subscript of $\llbracket - \rrbracket$. We write $\llbracket X \rrbracket$ for the image of $\llbracket - \rrbracket_X$, i.e. $\llbracket X \rrbracket = \{s \in \{0, 1\}^* : s = \llbracket x \rrbracket \text{ for some } x \in X\}$.

   Given a function $f : X \to Y$ of binary-encoded sets, we can define a function

$$\llbracket f \rrbracket : \llbracket X \rrbracket \to \llbracket Y \rrbracket$$
$$\llbracket x \rrbracket \mapsto \llbracket f(x) \rrbracket.$$

This is well-defined exactly because $\llbracket - \rrbracket_X$ is injective.

Definition 3.4 (category of computable functions). A function $f : X \to Y$ of binary-encoded sets is *computable* if $\llbracket f \rrbracket$ is computable. The *category of computable functions*, Comp, has binary-encoded sets as objects and computable functions as morphisms.

   It needs to be shown that this is a category. First, the identities 1_X are computable, as $\llbracket 1_X \rrbracket = 1_{\llbracket X \rrbracket}$ is computable. Second, the composite of computable functions is computable, as the composition of computable binary functions is computable, and composition is preserved by $\llbracket - \rrbracket$. As this argument indicates, there is a functor $\llbracket - \rrbracket : \text{Comp} \to \text{BinComp}$; in fact this functor is an *equivalence of categories*. Nevertheless, the expanded perspective provided by Comp will be convenient.

---

[1] These issues are studied in the field of *computable analysis*; see for instance the PhD thesis of Andre Bauer [Bau00].

Finally, we now define a symmetric monoidal structure on Comp. In particular, for two binary-encoded sets X and Y, we would like to define the product $X \otimes Y$ as the set $X \times Y$, but it is unclear what $[\![-]\!]_{X \times Y}$ should be. We first fix an injective pairing map $\langle -, - \rangle : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}$ which is efficiently computable[2]. We can then define $[\![(x, y)]\!] = \langle [\![x]\!], [\![y]\!] \rangle$; it is a standard check that this defines a symmetric monoidal structure inherited from Set[3].

3.1.2 Probabilistic Computation

Again, there is some subtelty with probabilistic computation. Even in the case of finite sets, not every stochastic function is computable by algorithms with access to fair coin flips[4]. However, there is again a standard notion of computable stochastic function of binary strings, so we can proceed much as before, defining:

Definition 3.5 (category of computable stochastic functions)**.** The category BinCompStoch has sets of binary strings as objects and computable stochastic functions as morphisms.

A stochastic function $f : X \to Y$ between binary-encoded sets is *computable* if $[\![f]\!]$ is computable. The *category of computable stochastic functions* CompStoch has binary-encoded sets as objects and computable stochastic functions as morphisms.

Again, it needs to be shown that this is a category. The identities are computable (and stochastic, since every determinstic function is stochastic), and composition commutes with $[\![-]\!]$, so the composite of computable functions is computable. Furthermore, this category is symmetric monoidal, with pairing of encodings as in Comp.

We give a more abstract characterization of this category. There are only countably many computable probability distributions on $\{0, 1\}^*$, since there are only countably many Turing machines. Fix a choice φ of bijection witnessing this fact. Note further that any probability distribution P on a binary-encoded set X induces a probability distribution $[\![P]\!]$ on $\{0, 1\}^*$ by

$$\Pr_{s \leftarrow [\![P]\!]}[s = s_0] = \Pr_{x \leftarrow P}[[\![x]\!] = s_0].$$

There is now a monad $G_c : \text{Comp} \to \text{Comp}$, which we call the *computable Giry monad*, which takes any binary-encoded set X to the set of computable probability distributions on X, i.e. those such that $[\![P]\!]$ is a computable probability distribution on $\{0, 1\}^*$, with

---

[2]One such map is computed as follows: given inputs (m, n), start by encoding the length of m in $2 \log \log m$ bits: first write a bit of the length, then write a 1 if the length continues and a 0 if it doesn't. Now knowing the length of m, we can append the binary representation of m and then n, which takes $O(\log m + \log n) = O(\log(mn))$ bits. Since $\log \log m = O(\log m)$, in total this algorithm takes $O(\log(mn))$ bits, and just writes across the tape, hence is computable in linear time.

[3]Here is a more abstract way to see this. A suitable pairing function $\langle -, - \rangle$ turns $\{0, 1\}$ into an internal commutative monoid in Set. In other work, we show that the category of subobjects of any internal monoid is a monoidal category [SZ24]. The construction here is approximately an application of that general theorem.

[4]We believe this is a slight conceptual problem with the strategy of [BK22, Section 6], which models unbounded probabilistic computation in the category of finite sets and stochastic functions: this category is too powerful to reasonably model computation. This does not pose a technical issue in their specific example.

encoding given by $[\![P]\!]_{G_cX} = \varphi([\![P]\!]_X)$. Given $f : X \to Y$ and $P \in G_cX$, we define the probability distribution $G_cf(P)$ on Y by

$$\Pr_{y \leftarrow G_cf(P)}[y = y_0] = \Pr_{x \leftarrow P}[f(x) = y_0].$$

The unit of G_c is the function $X \to G_cX$ taking x to the point distribution at x. The multiplication is the function $\mu_X : G_cG_cX \to G_cX$ acting by summation: given a probability distribution Q on G_cX, we define a distribution $\mu_X(Q)$ on X by

$$\Pr_{x \leftarrow \mu_X(Q)}[x = x_0] = \sum_{P_0 \in G_cX} \Pr_{P \leftarrow Q}[P = P_0] \Pr_{x \leftarrow P}[x = x_0], \tag{3.1}$$

which converges because Q is a probability distribution. The proofs of functorality and the monad laws as exactly as for the ordinary Giry monad [Gir82], so we do not give them here. Now CompStoch is in fact (isomorphic to) the Kleisi category of G_c.

3.1.3   Efficient and Effectful Computation

Suppose that we are given some wide subcategory EffBin of BinComp, for instance that of poly-time computable maps. We can define the category EffComp of efficient computations as the wide subcategory of Comp consisting of morphisms f whose encodings $[\![f]\!]$ are in EffBin: this is the *preimage* of EffBin under the functor $[\![-]\!]$.

Definition 3.6. The category P of poly-time computable maps is the wide subcategory of Comp consisting of those morphisms f such that $[\![f]\!]$ is poly-time computable.

   Similarly, suppose that we are give some wide subcategory of BinCompStoch, for instance that of poly-time computable stochastic maps. We can similarly define the cateogry EffCompStoch.

Definition 3.7. The category PPT of poly-time computable stochastic maps is the wide is the wide subcategory of CompStoch consisting of those morphisms f such that $[\![f]\!]$ is probabilistic poly-time-computable.

   In general, we can perform this construction for any complexity class C which is closed under composition.
   Even more generally, let Bin be the category of sets of binary strings and (maybe uncomputable) set-functions between them. Let Enc be the category of binary-encoded sets and (maybe uncomputable) set-functions between them. Then $[\![-]\!]$ is an equivalence of categories Enc $\simeq$ Bin.
   Now let EffBin be any subcategory of Bin. Then the *category of efficient computations* Eff is the subcategory of Enc consisting of morphisms f such that $[\![f]\!]$ is in EffBin, i.e. the preimage of EffBin under $[\![-]\!]$. Finally, let T be any monad on Enc which restricts to a monad on Eff. Then the *category of efficient T-computations* is the Kleisi category of the restriction of T to Eff. When T is symmetric lax monoidal, this category is symmetric monoidal.

Example 3.8. Each example in the previous three sections is a special case of this construction.

- When EffBin consists of computable functions and T is the identity monad, we recover Comp.
- When EffBin consists of computable functions and T is the computable Giry monad, we recover CompStoch.
- When EffBin consists of poly-time computable functions and T is the identity monad, we recover P.
- When EffBin consists of poly-time computable functions and T is the *poly-time Giry monad*, which sends a set X to the set of poly-time computable probability distributions on X, we recover PPT.

The point is that for any notion of efficient computation, and any notion of computational effect (since effects are generally monadic [WT03]), as long as the effect can be efficiently represented, we can use the machinery of binary-encoded sets to define a category of efficient computations carrying the given effect.

3.1.4 Quantum Computation

While an complete introduction to quantum computation is outside our scope, we can sketch a categorical perspective; a standard introduction is [NC10]. The *category of quantum computations* FinHilb is the category of finite-dimensional Hilbert spaces over $\mathbb{C}$ and linear maps. Since nontrivial complex Hilbert spaces have uncountably many vectors, we cannot directly model this category using the machinery of the previous section, as there is no way to encode a complex Hilbert space as an object of Bin. If we had developed a more general theory relying on a notion of computability over uncountable sets, then we could now unify these perspectives; indeed, there have been several attempts to monadically embed quantum computation into classical calculi [AG09; Abr+17]. As we have chosen not to develop such a general theory, in this section we will treat Hilbert spaces as our primitive object.

A *quantum computation* is a sequence of unitary transformations and *measurements*. There are several ways to provide categorical semantics to quantum measurement; we follow [HV19].

Let I be the one-dimensional Hilbert space. Note that the maps $I \to I$ correspond to choices of scalars $\lambda \in \mathbb{C}$; as such, we say that a *scalar* is a map $I \to I$. Given a Hilbert space V, a *state* is a map $I \to V$, so that a state is determined by a choice of vector in V. If $I \xrightarrow{a} V$ and $I \xrightarrow{b} V$ are states, then the projection of a onto b has amplitude

$$I \xrightarrow{a} V \xrightarrow{b^\dagger} I,$$

where $(-)^\dagger$ denotes the adjoint; the corresponding element of $\mathbb{C}$ is the inner product $\langle b, a \rangle$. Now the *Born rule* of quantum mechanics asserts that the probability of measuring the outcome b from the state a is $|\langle b, a \rangle|^2$. Categorically, if $I \xrightarrow{a,b} V$ are states, then the *probability of measuring b from the state a* is the scalar

$$I \xrightarrow{a} V \xrightarrow{b^\dagger} I \xrightarrow{b} V \xrightarrow{a^\dagger} I.$$

This is just a brief sketch of a very rich theory; see especially the work of Bob Coecke such as [AC04; CP12; CK17], or the book by Heunen and Vicary [HV19]. The key point is that any fully categorical treatment of cryptography should obtain quantum cryptography as a special case.

3.2   Protocols

The categorical semantics of interactive computation—in particular, of protocols—originates from the study of quantum cryptography, especially of so-called *resource theories* [CFS16]. The idea is to start with some underlying SMC C of computations—fixed throughout this section—and to construct a category of "protocols built from computations in C." Exactly which functor we choose depends on what we want our protocols to look like.

In all these categories, the basic idea is that we will think of objects as resources and morphisms as protocols, which convert some resources into others. For instance, in the category n-comb C, morphisms will be "protocols with holes"—when instantiated with specific implementations of the resources they are waiting for, they provide some new resource.

3.2.1   Products

While the product category is used implicitly in the definition of monoidal categories, it is worth exploring it explicitly. Given two categories C and $\mathcal{D}$, the *product category* $C \times \mathcal{D}$ has:

- as objects, pairs (X, Y) of objects in C and $\mathcal{D}$;
- as morphisms $(X, Y) \to (X', Y')$, pairs (f, g) of morphisms so that $f : X \to X'$ and $g : Y \to Y'$;
- composition and identities defined componentwise.

When C and $\mathcal{D}$ are categories of computations, we think of $C \times \mathcal{D}$ as a category of non-interfering parallel computations: a computation in $C \times \mathcal{D}$ is a computation in C and a computation in $\mathcal{D}$, but they cannot interact.

3.2.2   States

In the symmetric monoidal category Set, the morphisms $\{*\} \to X$ are in natural correspondence with the elements of X, by the bijection

$$(* \mapsto x) \mapsto x.$$

Similarly, in the symmetric monoidal category $\mathrm{Vect}_{\Bbbk}$, the morphisms $\Bbbk \to V$ are in natural correspondence with vectors in V, since such maps are determined by their action on the vector $1 \in \Bbbk$. This pattern holds more generally, motivating the following definition.

**Definition 3.9** (state). A *state* or *generalized element* of C is a morphism $I \to X$ for some object X.

As we know, it is easy to recognize states string-diagramatically: they are downward-pointing triangles.

Definition 3.10 (resource theory of states). The *resource theory of states* $\mathrm{st}(C)$ is the category whose objects are states in C and whose morphisms $(I \xrightarrow{s} X) \to (I \xrightarrow{t} Y)$ are maps $X \xrightarrow{f} Y$ such that $fs = y$. Composition is as in C.

When C is interpreted a category of types of resources and conversions between them, we can think of $\mathrm{st}(C)$ states as a category of specific resources and of conversions between them, forgetting the type information.

The resource theory of states has a canonical symmetric monoidal structure induced by that of C: the tensor product of states $I \xrightarrow{s} X$ and $I \xrightarrow{t} Y$ is just the state $I \to X \otimes Y$ given by[5]
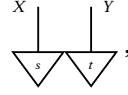
while the tensor product of morphisms is just their tensor in X. The unit is the state 1_I, while the associator and unitor are inherited from C.

It will be useful to be a little more general. Let $F : \mathcal{D} \to C$ be a lax monoidal functor. Then an *F-state* is a map $I \to FX$ in C for some object $X \in \mathcal{D}$. The *resource theory of F-states* $\mathrm{st}(F)$ is the category whose objects are F-states and whose morphisms $(I \xrightarrow{s} FX) \to (I \xrightarrow{t} FY)$ are maps $X \xrightarrow{f} Y$ in $\mathcal{D}$ such that $(Ff)s = t$[6]. Note what then $C = \mathcal{D}$ and $F = 1_C$, we recover $\mathrm{st}(C)$.

We are especially interested in the category $\mathrm{st}(C^2 \xrightarrow{\otimes} C)$. Objects in this category are morphisms $I \to X \otimes Y$ in C, which we can think of as *joint states*. When $C = \mathrm{SET}$, every joint state is *independent*, in that it splits into the tensor of two morphisms $I \to X$ and $I \to Y$, but in more complicated categories like PPT or HILB this may fail, representing a kind of *entanglement*. In this way, we can express the idea that two parties A and B have a shared uniform random key by the map $I \to X \otimes X$ in PPT that sends $*$ to a uniform random choice of pairs (k, k) for $k \in X$. This map does not split into a pair of stochastic maps $I \to X$ and $I \to X$.

More generally, we would like to model computations on n-party states via the category $\mathrm{st}(C^n \xrightarrow{\otimes^{n-1}} C)$, where the ith copy of C represents computations taken by the ith

---

[5]It may worry the careful reader that there are two seemingly distinct, albeit coherently naturally isomorphic, morphisms this diagram could represent:

$$I \xrightarrow{\lambda_I^{-1}} I \otimes I \xrightarrow{s \otimes t} X \otimes Y \quad \text{and} \quad I \xrightarrow{\rho_I^{-1}} I \otimes I \xrightarrow{s \otimes t} X \otimes Y.$$

Fortunately, it is a non-obvious but standard result of Kelly that $\lambda_I = \rho_I$ in any SMC [Kel64], so these morphisms agree.

[6]With a little more machinery: $\mathrm{st}(F)$ is the *category of elements* of the functor $\mathcal{D} \xrightarrow{F} C \xrightarrow{C(I,-)} \mathrm{SET}$. The category of elements of any lax monoidal functor has a canonical monoidal structure on it induced by that of the codomain; this is the monoidal structure with which we endow $\mathrm{st}(F)$. Observe the similarity of the tensor in $\mathrm{st}(F)$ with the coherence map for the functor $C(I, -)$ from Example 2.36.

party in the computation. There is a choice of associativity to be made, but any choice yields an isomorphic category, so we will not worry about it here.

Appendix A

Computer Scientific Foundations

In the main body, we have assumed standard material from a course in computability and complexity, including function asymptotics, the notion of an algorithm, and the complexity class P. We briefly overview these ideas here; a standard text is [Sip13].

A.1 Asymptotics

Function asymptotics formalize the notion of a function approximating another function. In particular, for a pair of functions $f, g : \mathbb{N} \to \mathbb{R}$, we often want to compare f and g on large inputs and only up to a constant factor. This is most common in runtime analysis, the idea being that the running time of algorithms on small inputs is less important to their overall performance than their running time on large inputs. We formalize this notion as follows:

Definition A.1 (Function Asymptotics)**.** Let $f, g : \mathbb{N} \to \mathbb{R}$ be a pair of functions which are both non-negative for sufficiently large inputs. We say that f is *big-Oh of g*, written $f = O(g)$, if there exists a constant $c > 0$ such that for all n sufficiently large,

$$c f(n) \geq g(n).$$

In this case, we also say that g is *big-Omega of f*, written $g = \Omega(f)$.

If $f = O(g)$ and $g = O(f)$, we say that f is *big-Theta of g*, written $f = \Theta(g)$. Explicitly, this means that there exist constants $c_1, c_2 > 0$ such that for all n sufficiently large,

$$c_1 f(n) \leq g(n) \leq c_2 f(n).$$

If $f = O(g)$ but $f \neq \Theta(g)$, we say that f is *little-oh of g*, written $f = o(g)$, and g is *little-omega of f*, written $g = \omega(f)$. Explicitly, this means that for all constants $\epsilon > 0$ and all n sufficiently large,

$$f(n) \leq \epsilon g(n).$$

Notation. By abuse of notation, we often write $f(n) = O(g(n))$ to mean that f is $O(n)$; for example, the statement that n^2 is $O(n^3)$ means that the function $f(n) = n^2$ is $O(g)$, where g is the function $n \mapsto n^3$.

Example A.2. We have that:

- $17n^2$ is $o(n^3)$, $\omega(n)$, and $\Theta(n^2)$;
- $\log n$ is $o(n)$, $\omega(1)$, and $\Theta(\ln n)$;
- e^n is $\omega(n^k)$ for any exponent k;
- e^{-n} is $o(n^{-k})$ for any exponent k.

These last two examples are especially important. No matter how big the power, an exponential will always dominate a polynomial for sufficiently big n. Because of the importance of polynomials in theoretical computer science, we say a function f is *negligible* if $f = o(n^k)$ for all k. In this case, we write $f = \mathrm{negl}(n)$ or just $f = \mathrm{negl}$.

Proposition A.3. *Big-Oh is a preorder on the set of functions* $\mathbb{N} \to \mathbb{N}$. *The induced equivalence relation is exactly big-Theta.*

In the partial order of equivalence classes under Θ, O behaves like $\leq$, o like $<$, and Θ like $=$. As suggested by the notation $f = O(g)$, it is common in some contexts to treat functions as identical with their asymptotic equivalence class.

Proposition A.4. *Let* $f_1 = O(g_1)$ *and* $f_2 = O(g_2)$. *Let c be any nonzero constant. Then,*

$$f_1 + f_2 = O(\max\{g_1, g_2\}), \quad cf_1 = O(g_1), \quad \text{and} \quad f_1 f_2 = O(g_1 g_2).$$

In other words,

$$O(g_1) + O(g_2) = O(\max\{g_1, g_2\}), \quad cO(g) = O(g), \quad \text{and} \quad O(g_1)O(g_2) = O(g_1 g_2).$$

Identical results hold for o and Θ.

Proposition A.4 justifies the universal practice of dropping constants and small additive terms from asymptotics, so that for instance $n^2 + n + \ln n = \Theta(n^2)$.

A.2   Algorithms and Determinism

Our basic notion is of an *algorithm* over a finite alphabet Σ, usually $\mathbb{Z}_2$. An algorithm $\mathcal{A}$ is intuitively some set of steps which take an input word x over Σ, perform some transformations, and output another word $\mathcal{A}(x)$ over Σ. An algorithm may have certain *side effects*, such as sending a message or logging a string, and its behavior may not be deterministic. There are several ways to formalize the notion of algorithm—most common in cryptography are Turing machines—but we will not need to be so precise here.

Algorithms may have multiple possible "branches" in their instructions. Consider the following:

Algorithm A.5. On input x, either output 0 or 1.

We say that algorithms of this sort are *nondeterministic*; in contrast, an algorithm is *deterministic* if its instructions do not include such choices. In particular, we say that an algorithm $\mathcal{A}$ *deterministically computes* a function f if it is deterministic and, for any input

$x \in \Sigma^*$, $\mathcal{A}$ outputs the value $f(x) \in \Sigma^*$. In contrast, $\mathcal{A}$ *nondeterministically computes* f if, for any input x, there exists a particular choice of branches such that $\mathcal{A}$ outputs $f(x)$. Thus Algorithm A.5 nondeterministically computes both the functions $x \mapsto 0$ and $x \mapsto 1$. We sometimes view nondeterministic algorithms as computing functions into the power set of Σ^*, so that Algorithm A.5 computes the function $x \mapsto \{0, 1\}$, and we similarly sometimes write $\mathcal{A}(x) = \{0, 1\}$.

An important middle ground is *probabilistic* algorithms. Again, there are many possible models, but the basic idea is that a probabilistic algorithm has access to some source of randomness—say, an arbitrarily long string of independent and uniform coin tosses—which it can use to choose between branches. In this case, it is not enough for there to be some branch which computes a specific function. Instead, we say that an algorithm *computes f with bounded probability* if for any input x,

$$\Pr[\mathcal{A}(x) = f(x)] > \frac{2}{3},$$

where the probability is taken over the randomness of $\mathcal{A}$[1]. In this case, we often think of $\mathcal{A}(x)$ as a probability distribution on Σ^*.

Instead of thinking of algorithms operating directly on binary strings, we usually think of them as operating on encodings of mathematical objects. For example:

Algorithm A.6. On input x a natural number, output the number $2x$.

We say that Algorithm A.6 deterministically computes $x \mapsto 2x$, even though it technically operates on encodings of naturals. While there are many possible encodings, we assume that a reasonable encoding is chosen, so that for instance numbers are encoded in binary, rather than unary. Such details will not be relevant for us.

One more subtlety is important. In general, we require that the description of any algorithm $\mathcal{A}$ is finite. However, we may also consider *non-uniform* algorithms, which are sequences of algorithms $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots)$ such that, on an input of length n, $\mathcal{A}$ delegates to $\mathcal{A}_n$. Non-uniform computation is generally stronger than uniform computation, as non-uniform algorithms may encode nonfinite information, as long as they only use finitely much of this information for each input length and hence for each computation[2].

A.3 Complexity Theory

Each algorithm has an associated *running time*, which is informally the number of steps the algorithm takes on a given input. In particular, for an algorithm $\mathcal{A}$, we say that its running time is the function $T_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ which takes any natural number n to the maximum number of steps $\mathcal{A}$ takes to terminate on any input of length n. Of course, this

---

[1]The choice of $\frac{2}{3}$ is not particularly important here—generally any constant $c > \frac{1}{2}$ works.

[2]For instance, non-uniform algorithms may solve the halting problem (which asks whether an input algorithm M eventually terminates), which is uniformly undecidable. In particular, since there are only finitely many Turing machines of a given size, a non-uniform algorithm may simply encode in $\mathcal{A}_n$ the answer to the halting problem for each Turing machine of length n.

notion is not yet precise, as we don't know what a "step" is, but it is easy to make precise in any standard model of computation.

In general, the running time may depend on the formal model of computation in which the algorithm is constructed, but the *complexity-theoretic Church-Turing thesis* states that "reasonable" models of classical computation recover the same inhabitants of sufficiently robust complexity classes, in particular of those we are about to define. This hypothesis is a heuristic, but has been born out in practice.

Definition A.7 (polynomial-time; P, NP)**.** An algorithm $\mathcal{A}$ is *polynomial-time* if $T_{\mathcal{A}} = O(n^k)$ for some constant k. The class P consists of all functions which are deterministically computable by polynomial-time algorithms. The class NP consists of all functions which are nondeterministically computable by polynomial-time algorithms[3].

The general idea is that polynomial-time algorithms are "efficient in practice." It may sometimes occur that the constant factors or the exponent are so large as to render the algorithm practically useless, but in most cases functions in P are efficiently solvable for practical applications, including cryptography. We can now state the most important open problem in computer science:

Conjecture A.8. *We have that P ≠ NP.*

While a proof seems completely out of reach, this conjecture is widely believed, and as we will see is necessary for all of modern cryptography; we will assume it here. An introduction to the modern state of P vs. NP is [And17].

Formalizing probabilistic complexity classes is slightly more subtle. Consider the following case:

Algorithm A.9. On input x, output 1 with probability $1 - 2^{-|x|}$; otherwise count from 0 to $2^{|x|}$ and then output 1.

While this algorithm is almost always polynomial-time, it is not polynomial-time when it takes the second branch. The point is that for probabilistic algorithms, $T_{\mathcal{A}}(n)$ is a probability distribution, not just a fixed number. For our purposes, we require that the algorithm *always* runs in polynomial time. As such:

Definition A.10 (probabilistic polynomial-time; BPP)**.** A probabilistic algorithm is *probabilistic polynomial-time* if, for any choice of random bits, $T_{\mathcal{A}} = O(n^k)$ for some constant k. The class BPP consists of all functions which are computable with bounded probability by a probabilistic polynomial-time algorithm.

For non-uniform algorithms, the situation is also slightly more complicated. In particular, it is too much to allow the machines to be arbitrarily large, as they could simply encode lookup tables for every possible input. As such, we ask that the size of each machine is polynomially bounded.

---

[3]In fact, we have defined here the classes FP and FNP of polynomially- and nondeterministically-polynomially-computable *function problems*. Formally, P and NP are classes of *decision problems*, which are just subsets L of Σ^*—the algorithm must output 1 if its input is in L, and 0 otherwise. Function and decision problems are extremely closely related—for instance, P = NP if and only if FP = FNP—and we will not distinguish between them here.

Definition A.11 (non-uniform polynomial-time; P/poly). A non-uniform algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \dots)$ is *polynomial-time* if $T_{\mathcal{A}} = O(n^k)$ for some constant k and the size of each $\mathcal{A}_n$ is $O(n^k)$ for some constant k independent of n. The class P/poly consists of all functions which are computable by non-uniform polynomial-time algorithms.

Non-uniform probabilistic algorithms are similarly defined.

Theorem A.12 (Adelman's theorem). *We have that BPP $\subseteq$ P/poly.*

Bibliography

[Abr+17]   Samson Abramsky et al. "The Quantum Monad on Relational Structures".
           In: *42nd International Symposium on Mathematical Foundations of Computer
           Science (MFCS 2017)*. Ed. by Kim G. Larsen, Hans L. Bodlaender, and Jean-
           Francois Raskin. Vol. 83. Leibniz International Proceedings in Informatics
           (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Infor-
           matik, 2017, 35:1–35:19. ISBN: 978-3-95977-046-0. DOI: 10.4230/LIPIcs.
           MFCS.2017.35. URL: https://drops-dev.dagstuhl.de/entities/
           document/10.4230/LIPIcs.MFCS.2017.35.

[AC04]   S. Abramsky and B. Coecke. "A categorical semantics of quantum protocols".
           In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Sci-
           ence, 2004*. 2004, pp. 415–425. DOI: 10.1109/LICS.2004.1319636.

[ACS22]   Gilad Asharov, Ran Cohen, and Oren Shochat. *Static vs. Adaptive Security
           in Perfect MPC: A Separation and the Adaptive Security of BGW*. Cryptology
           ePrint Archive, Paper 2022/758. https://eprint.iacr.org/2022/758.
           2022. URL: https://eprint.iacr.org/2022/758.

[AG09]   Thorsten Altenkirch and Alexander S. Green. "The Quantum IO Monad". In:
           Semantic Techniques in Quantum Computation. Ed. by Simon Gay and IanEd-
           itors Mackie. Cambridge University Press, 2009, pp. 173–205.

[And17]   Scott Anderson. "P $\overset{?}{=}$ NP". 2017. URL: https://www.scottaaronson.com/
           papers/pnp.pdf.

[Bau00]   Andrej Bauer. PhD thesis. School of Computer Science, 2000.

[Ben+90]   Michael Ben-Or et al. "Everything Provable is Provable in Zero-Knowledge".
           In: *Advances in Cryptology — CRYPTO' 88*. Ed. by Shafi Goldwasser. New York,
           NY: Springer New York, 1990, pp. 37–56. ISBN: 978-0-387-34799-8.

[BK22]   Anne Broadbent and Martti Karvonen. "Categorical composable cryptogra-
           phy". In: *Foundations of software science and computation structures*. Vol. 13242.
           Lecture Notes in Comput. Sci. Springer, Cham, 2022, pp. 161–183. ISBN: 9783030992538.
           DOI: 10.1007/978-3-030-99253-8_9. URL: https://doi.org/10.1007/
           978-3-030-99253-8_9.

[BKM19]   Spencer Breiner, Amir Kalev, and Carl A. Miller. "Parallel Self-Testing of the
           GHZ State with a Proof by Diagrams". In: *Electronic Proceedings in Theoretical
           Computer Science* 287 (Jan. 2019), pp. 43–66. ISSN: 2075-2180. DOI: 10.4204/
           eptcs.287.3. URL: http://dx.doi.org/10.4204/EPTCS.287.3.

[BM04]   Michael Ben-Or and Dominic Mayers. *General Security Definition and Composability for Quantum & Classical Protocols*. 2004. arXiv: `quant-ph/0409062 [quant-ph]`.

[BMR19]   Spencer Breiner, Carl A. Miller, and Neil J. Ross. "Graphical Methods in Device-Independent Quantum C ryptography". In: *Quantum* 3 (May 2019), p. 146. ISSN: 2521-327X. DOI: `10.22331/q-2019-05-27-146`. URL: `https://doi.org/10.22331/q-2019-05-27-146`.

[BP17]   John C. Baez and Blake S. Pollard. "A compositional framework for reaction networks". In: *Reviews in Mathematical Physics* 29.09 (Sept. 2017), p. 1750028. DOI: `10.1142/s0129055x17500283`. URL: `https://doi.org/10.1142%2Fs0129055x17500283`.

[BS11]   J. Baez and M. Stay. "Physics, Topology, Logic and Computation: A Rosetta Stone". In: *New Structures for Physics*. Ed. by Bob Coecke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–172. ISBN: 978-3-642-12821-9. DOI: `10.1007/978-3-642-12821-9_2`. URL: `https://doi.org/10.1007/978-3-642-12821-9_2`.

[BS22]   Guillaume Boisseau and Paweł Sobociński. "String Diagrammatic Electrical Circuit Theory". In: *Electronic Proceedings in Theoretical Computer Science* 372 (Nov. 2022), pp. 178–191. ISSN: 2075-2180. DOI: `10.4204/eptcs.372.13`. URL: `http://dx.doi.org/10.4204/EPTCS.372.13`.

[BV10]   Eleanor Birrell and Salil Vadhan. "Composition of Zero-Knowledge Proofs with Efficient Provers". In: *Theory of Cryptography*. Ed. by Daniele Micciancio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 572–587. ISBN: 978-3-642-11799-2.

[BW90]   Michael Barr and Charles Wells. *Category theory for computing science*. USA: Prentice-Hall, Inc., 1990. ISBN: 0131204866.

[Can00]   Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Paper 2000/067. `https://eprint.iacr.org/2000/067`. 2000. URL: `https://eprint.iacr.org/2000/067`.

[Can06]   Ran Canetti. *Security and Composition of Cryptographic Protocols: A Tutorial*. Cryptology ePrint Archive, Paper 2006/465. `https://eprint.iacr.org/2006/465`. 2006. URL: `https://eprint.iacr.org/2006/465`.

[Can08]   Ran Canetti. *Lecture 11*. Spring 2008. URL: `https://www.cs.tau.ac.il/~canetti/f08-materials/scribe11.pdf`.

[Can20]   Ran Canetti. "Universally Composable Security". In: *J. ACM* 67.5 (Sept. 2020). ISSN: 0004-5411. DOI: `10.1145/3402457`. URL: `https://doi.org/10.1145/3402457`.

[CF01]   Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 19–40. ISBN: 978-3-540-44647-7.

[CFS16]   Bob Coecke, Tobias Fritz, and Robert W. Spekkens. "A mathematical theory of resources". In: *Information and Computation* 250 (2016). Quantum Physics and Logic, pp. 59–86. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2016.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0890540116000353.

[CG19]   Eric Chitambar and Gilad Gour. "Quantum resource theories". In: *Rev. Mod. Phys.* 91 (2 Apr. 2019), p. 025001. DOI: 10.1103/RevModPhys.91.025001. URL: https://link.aps.org/doi/10.1103/RevModPhys.91.025001.

[CGP15]   Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. "Adaptively Secure Two-Party Computation from Indistinguishability Obfuscation". In: *Theory of Cryptography*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 557–585. ISBN: 978-3-662-46497-7.

[CK17]   Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[CP12]   Bob Coecke and Simon Perdrix. "Environment and classical channels in categorical quantum mechanics". In: *Logical Methods in Computer Science* Volume 8, Issue 4 (Nov. 2012). ISSN: 1860-5974. DOI: 10.2168/lmcs-8(4:14)2012. URL: http://dx.doi.org/10.2168/LMCS-8(4:14)2012.

[Cra+99]   Ronald Cramer et al. "Efficient Multiparty Computations Secure Against an Adaptive Adversary". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 311–326. ISBN: 978-3-540-48910-8.

[Cro94]   Roy L. Crole. *Categories for Types*. Cambridge University Press, 1994.

[CS99]   J.R.B. Cockett and R.A.G. Seely. "Linearly distributive functors". In: *Journal of Pure and Applied Algebra* 143.1 (1999), pp. 155–203. ISSN: 0022-4049. DOI: https://doi.org/10.1016/S0022-4049(98)00110-8. URL: https://www.sciencedirect.com/science/article/pii/S0022404998001108.

[EGL85]   Shimon Even, Oded Goldreich, and Abraham Lempel. "A randomized protocol for signing contracts". In: *Commun. ACM* 28.6 (June 1985), pp. 637–647. ISSN: 0001-0782. DOI: 10.1145/3812.3818. URL: https://doi.org/10.1145/3812.3818.

[Fri20]   Tobias Fritz. "A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics". In: *Advances in Mathematics* 370 (Aug. 2020), p. 107239. ISSN: 0001-8708. DOI: 10.1016/j.aim.2020.107239. URL: http://dx.doi.org/10.1016/j.aim.2020.107239.

[Gir82]   Michèle Giry. "A categorical approach to probability theory". In: *Categorical Aspects of Topology and Analysis*. Ed. by B. Banaschewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 68–85. ISBN: 978-3-540-39041-1.

[GK96]   Oded Goldreich and Hugo Krawczyk. "On the Composition of Zero-Knowledge Proof Systems". In: *SIAM Journal on Computing* 25.1 (1996), pp. 169–192. DOI: 10 . 1137 / S0097539791220688. eprint: https : / / doi . org / 10 . 1137 / S0097539791220688. URL: https://doi.org/10.1137/S0097539791220688.

[GL89]   O. Goldreich and L. A. Levin. "A Hard-Core Predicate for All One-Way Functions". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 25–32. ISBN: 0897913078. DOI: 10 . 1145 / 73007 . 73010. URL: https://doi.org/10.1145/73007.73010.

[GM82]   Shafi Goldwasser and Silvio Micali. "Probabilistic encryption & how to play mental poker keeping secret all partial information". In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. STOC '82. San Francisco, California, USA: Association for Computing Machinery, 1982, pp. 365–377. ISBN: 0897910702. DOI: 10.1145/800070.802212. URL: https://doi.org/10.1145/800070.802212.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. "Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems". In: *J. ACM* 38.3 (July 1991), pp. 690–728. ISSN: 0004-5411. DOI: 10 . 1145 / 116825.116852. URL: https://doi.org/10.1145/116825.116852.

[GO94]   Oded Goldreich and Yair Oren. "Definitions and properties of zero-knowledge proof systems". In: *Journal of Cryptology* 7.1 (1994), pp. 1–32.

[Gog+73] J. A. Goguen et al. *A Junction between Computer Science and Category Theory*. Tech. rep. 1973. URL: https : / / dominoweb . draco . res . ibm . com / 49eae98dc5a21de0852574ff005001c8.html.

[Gol01]   O. Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2001. ISBN: 9780521791724.

[Hås+99] Johan Håstad et al. "A Pseudorandom Generator from any One-way Function". In: *SIAM Journal on Computing* 28.4 (1999), pp. 1364–1396. DOI: 10 . 1137/S0097539793244708. eprint: https://doi.org/10.1137/S0097539793244708. URL: https://doi.org/10.1137/S0097539793244708.

[Hin20]   Peter M. Hines. "A Diagrammatic Approach to Information Flow in Encrypted Communication". In: *Graphical Models for Security*. Ed. by Harley Eades III and Olga Gadyatskaya. Cham: Springer International Publishing, 2020, pp. 166–185. ISBN: 978-3-030-62230-5.

[HM03]   Dennis Hofheinz and Jörn Müller-Quade. *A Paradox of Quantum Universal Composability*. 2003. URL: https://www.quiprocone.org/Hot%20Topics%20posters/muellerquade_poster.pdf.

[HV19]   Chris Heunen and Jamie Vicary. "1Basics". In: *Categories for Quantum Theory: An Introduction.* Oxford University Press, Nov. 2019. ISBN: 9780198739623. DOI: 10.1093/oso/9780198739623.003.0009. eprint: https://academic.oup.com/book/0/chapter/367235017/chapter-pdf/50991620/oso-9780198739623-chapter-9.pdf. URL: https://doi.org/10.1093/oso/9780198739623.003.0009.

[Imp95]   R. Impagliazzo. "A personal view of average-case complexity". In: *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference.* 1995, pp. 134–147. DOI: 10.1109/SCT.1995.514853.

[Jac99]   B. Jacobs. *Categorical Logic and Type Theory.* Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999.

[JM20]   Daniel Jost and Ueli Maurer. "Overcoming impossibility results in composable security using interval-wise guarantees". In: *Advances in Cryptology– CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part I 40.* Springer. 2020, pp. 33–62.

[JS91]   André Joyal and Ross Street. "The geometry of tensor calculus, I". In: *Advances in Mathematics* 88.1 (1991), pp. 55–112. ISSN: 0001-8708. DOI: https://doi.org/10.1016/0001-8708(91)90003-P. URL: https://www.sciencedirect.com/science/article/pii/000187089190003P.

[Kel64]   G.M Kelly. "On MacLane's conditions for coherence of natural associativities, commutativities, etc." In: *Journal of Algebra* 1.4 (1964), pp. 397–402. ISSN: 0021-8693. DOI: https://doi.org/10.1016/0021-8693(64)90018-3. URL: https://www.sciencedirect.com/science/article/pii/0021869364900183.

[KL07]   Dafna Kidron and Yehuda Lindell. *Impossibility Results for Universal Composability in Public-Key Models and with Fixed Inputs.* Cryptology ePrint Archive, Paper 2007/478. https://eprint.iacr.org/2007/478. 2007. URL: https://eprint.iacr.org/2007/478.

[KL14]   Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014. ISBN: 9781466570269.

[Lam74]   Joachim Lambek. "Functional completeness of cartesian categories". In: *Annals of Mathematical Logic* 6.3-4 (1974), pp. 259–292.

[Lam80]   Joachim Lambek. "From lambda-calculus to cartesian closed categories". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), pp. 375–402.

[Law69]   F. William Lawvere. "Diagonal arguments and cartesian closed categories". In: *Category Theory, Homology Theory and their Applications II.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1969, pp. 134–145. ISBN: 978-3-540-36101-5.

[Lin03]   Y. Lindell. "General composition and universal composability in secure multi-party computation". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* 2003, pp. 394–403. DOI: 10.1109/SFCS.2003.1238213.

[Lin17]   Yehuda Lindell. "How to Simulate It—A Tutorial on the Simulation Proof Technique". In: *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich.* Springer International Publishing, 2017, pp. 277–346. ISBN: 9783319570488. DOI: [10.1007/978-3-319-57048-8_6](10.1007/978-3-319-57048-8_6).

[LLR04]   Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. *On the Composition of Authenticated Byzantine Agreement.* Cryptology ePrint Archive, Paper 2004/181. https://eprint.iacr.org/2004/181. 2004. URL: https://eprint.iacr.org/2004/181.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics, Vol. 5. New York: Springer-Verlag, 1971, pp. ix+262.

[Mel06]   Paul-André Melliès. "Functorial Boxes in String Diagrams". In: *Computer Science Logic.* Ed. by Zoltán Ésik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–30. ISBN: 978-3-540-45459-5.

[MM90]   José Meseguer and Ugo Montanari. "Petri nets are monoids". In: *Information and Computation* 88.2 (1990), pp. 105–155. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90013-8. URL: https://www.sciencedirect.com/science/article/pii/0890540190900138.

[MR92]   Silvio Micali and Phillip Rogaway. "Secure Computation". In: *Advances in Cryptology — CRYPTO '91.* Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 392–404. ISBN: 978-3-540-46766-3.

[NC10]   Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010.

[Pav12]   Dusko Pavlovic. "Tracing the Man in the Middle in Monoidal Categories". In: *Coalgebraic Methods in Computer Science.* Ed. by Dirk Pattinson and Lutz Schröder. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 191–217. ISBN: 978-3-642-32784-1.

[Pav14]   Dusko Pavlovic. "Chasing Diagrams in Cryptography". In: *Categories and Types in Logic, Language, and Physics: Essays Dedicated to Jim Lambek on the Occasion of His 90th Birthday.* Ed. by Claudia Casadio et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 353–367. ISBN: 978-3-642-54789-8. DOI: [10.1007/978-3-642-54789-8_19](10.1007/978-3-642-54789-8_19). URL: https://doi.org/10.1007/978-3-642-54789-8_19.

[Ped91]   Torben Pryds Pedersen. "Non-interactive and information-theoretic secure verifiable secret sharing". In: *Annual international cryptology conference.* Springer. 1991, pp. 129–140.

[Pie91]   Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Aug. 1991. ISBN: 9780262288460. DOI: 10.7551/mitpress/1524.001.0001. URL: https://doi.org/10.7551/mitpress/1524.001.0001.

[PKW22]   Marco Patrignani, Robert Künnemann, and Riad S. Wahby. *Universal Composability is Robust Compilation*. 2022. arXiv: 1910.08634 [cs.PL].

[PS10]   Raphael Pass and Abhi Shelat. *A Course in Cryptography*. 2010. URL: https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[Rey83]   John C. Reynolds. "Types, Abstraction and Parametric Polymorphism." In: *IFIP Congress*. Ed. by R. E. A. Mason. North-Holland/IFIP, 1983, pp. 513–523. ISBN: 0-444-86729-5. URL: http://dblp.uni-trier.de/db/conf/ifip/ifip83.html#Reynolds83.

[Rie17]   Emily Riehl. *Category theory in context*. en. Courier Dover Publications, Mar. 2017.

[Ros21]   Mike Rosulek. *The Joy of Cryptography*. 2021. URL: https://joyofcryptography.com.

[See84]   R. A. G. Seely. "Locally cartesian closed categories and type theory". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95.1 (1984), pp. 33–48. DOI: 10.1017/S0305004100061284.

[See89]   R. A. G. Seely. "Linear Logic, *-autonomous categories and Cofree Coalgebras". In: *Categories in Computer Science and Logic* (1989), pp. 371–382. DOI: 10.1090/conm/092/1003210.

[Sel11]   P. Selinger. "A Survey of Graphical Languages for Monoidal Categories". In: *New Structures for Physics*. Ed. by Bob Coecke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 289–355. ISBN: 978-3-642-12821-9. DOI: 10.1007/978-3-642-12821-9_4. URL: https://doi.org/10.1007/978-3-642-12821-9_4.

[Sha92]   Adi Shamir. "IP = PSPACE". In: *J. ACM* 39.4 (Oct. 1992), pp. 869–877. ISSN: 0004-5411. DOI: 10.1145/146585.146609. URL: https://doi.org/10.1145/146585.146609.

[Sim11]   Harold Simmons. *An Introduction to Category Theory*. USA: Cambridge University Press, 2011. ISBN: 0521283043.

[Sip13]   Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

[SS02]   Ahmad-Reza Sadeghi and Michael Steiner. *Assumptions Related to Discrete Logarithms: Why Subtleties Make a Real Difference*. Cryptology ePrint Archive, Paper 2002/126. https://eprint.iacr.org/2002/126. 2002. URL: https://eprint.iacr.org/2002/126.

[SV13]   Mike Stay and Jamie Vicary. "Bicategorical Semantics for Nondeterministic Computation". In: *Electronic Notes in Theoretical Computer Science* 298 (2013). Proceedings of the Twenty-ninth Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIX, pp. 367–382. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2013.09.022. URL: https://www.sciencedirect.com/science/article/pii/S1571066113000686.

[SZ24]   Riley Shahar and Steve Zdancewic. "Categorical Phase Semantics for Linear Logic". Forthcoming. 2024.

[Tre09]   Luca Trevisan. *Notes for Lecture 27*. Apr. 2009. URL: https://theory.stanford.edu/~trevisan/cs276/lecture27.pdf.

[Unr04]   Dominique Unruh. *Simulatable security for quantum protocols*. 2004. arXiv: quant-ph/0409125 [quant-ph].

[Unr10]   Dominique Unruh. "Universally Composable Quantum Multi-Party Computation". In: *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT'10. French Riviera, France: Springer-Verlag, 2010, pp. 486–505. ISBN: 3642131891. DOI: 10.1007/978-3-642-13190-5_25. URL: https://doi.org/10.1007/978-3-642-13190-5_25.

[Vad07]   Salil Vadhan. "The Complexity of Zero Knowledge". In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science: 27th International Conference, New Delhi, India, December 12-14, 2007. Proceedings*. New Delhi, India: Springer-Verlag, 2007, pp. 52–70. ISBN: 978-3-540-77049-7. DOI: 10.1007/978-3-540-77050-3_5. URL: https://doi.org/10.1007/978-3-540-77050-3_5.

[Wad89]   Philip Wadler. "Theorems for free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359. ISBN: 0897913280. DOI: 10.1145/99370.99404. URL: https://doi.org/10.1145/99370.99404.

[WT03]   Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Trans. Comput. Logic* 4.1 (Jan. 2003), pp. 1–32. ISSN: 1529-3785. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776.

[Yao82]   Andrew C. Yao. "Theory and application of trapdoor functions". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 1982, pp. 80–91. DOI: 10.1109/SFCS.1982.45.

[Yau16]   Donald Ying Yau. *Colored operads*. American Mathematical Society, 2016.