Appendix C

On the Foundations of Modern Cryptography

It is possible to build a cabin with no foundations, but not a lasting building.

Eng. Isidor Goldreich (1906–1995)

Summary: Cryptography is concerned with the construction of computing systems that withstand any abuse: Such a system is constructed so to maintain a desired functionality, even under malicious attempts aimed at making it deviate from this functionality.

This appendix is aimed at presenting the foundations of cryptography, which are the paradigms, approaches and techniques used to conceptualize, define and provide solutions to natural security concerns. It presents some of these conceptual tools as well as some of the fundamental results obtained using them. The emphasis is on the clarification of fundamental concepts, and on demonstrating the feasibility of solving several central cryptographic problems. The presentation assumes basic knowledge of algorithms, probability theory and complexity theory, but nothing beyond this.

The appendix augments the treatment of one-way functions, pseudorandom generators and zero-knowledge proofs, given in Sections 7.1, 8.2 and 9.2, respectively. (These augmentations are important for cryptography, but are less central to the main context of this book and thus were omitted from the main text.) Using these basic tools, the appendix provides a treatment of basic cryptographic applications such as Encryption, Signatures, and General Cryptographic Protocols.

C.1 Introduction and Preliminaries

The vast expansion and rigorous treatment of cryptography is one of the major achievements of theoretical computer science. In particular, concepts such as computational indistinguishability, pseudorandomness and zero-knowledge interactive proofs were introduced, classical notions such as secure encryption and unforgeable signatures were placed on sound grounds, and new (unexpected) directions and connections were uncovered. Indeed, modern cryptography is strongly linked to complexity theory (in contrast to "classical" cryptography which is strongly related to information theory).

C.1.1 Modern cryptography

Modern cryptography is concerned with the construction of information systems that are robust against malicious attempts to make these systems deviate from their prescribed functionality. The prescribed functionality may be the private and authenticated communication of information through the Internet, the holding of incoercible and secret electronic voting, or conducting any "fault-resilient" multiparty computation. Indeed, the scope of modern cryptography is very broad, and it stands in contrast to "classical" cryptography (which has focused on the single problem of enabling secret communication over insecure communication media).

The design of cryptographic systems is a very difficult task. One cannot rely on intuitions regarding the "typical" state of the environment in which the system operates. For sure, the adversary attacking the system will try to manipulate the environment into "untypical" states. Nor can one be content with counter-measures designed to withstand specific attacks, since the adversary (which acts after the design of the system is completed) will try to attack the schemes in ways that are different from the ones the designer had envisioned. Although the validity of the foregoing assertions seems self-evident, still some people hope that in practice ignoring these tautologies will not result in actual damage. Experience shows that these hopes rarely come true; cryptographic schemes based on make-believe are broken, typically sooner than later.

In view of the foregoing, we believe that it makes little sense to make assumptions regarding the specific *strategy* that the adversary may use. The only assumptions that can be justified refer to the computational *abilities* of the adversary. Furthermore, the design of cryptographic systems has to be based on *firm foundations*; whereas ad-hoc approaches and heuristics are a very dangerous way to go. A heuristic may make sense when the designer has a very good idea regarding the environment in which a scheme is to operate, yet a cryptographic scheme has to operate in a maliciously selected environment that typically transcends the designer's view.

This appendix is aimed at presenting the foundations for cryptography. The foundations of cryptography are the paradigms, approaches and techniques used to conceptualize, define and provide solutions to natural "security concerns". Solving a cryptographic problem (or addressing a security concern) is a two-stage process consisting of a *definitional stage* and a *constructive stage*. First, in the definitional

stage, the functionality underlying the natural concern is to be identified, and an adequate cryptographic problem has to be defined. Trying to list all undesired situations is infeasible and prone to error. Instead, one should define the functionality in terms of operation in an imaginary ideal model, and require a candidate solution to emulate this operation in the real, clearly defined, model (which specifies the adversary's abilities). Once the definitional stage is completed, one proceeds to construct a system that satisfies the definition. Such a construction may use some simpler tools, and its security is proved relying on the features of these tools. In practice, of course, such a scheme may need to satisfy also some *specific* efficiency requirements.

This appendix focuses on several archetypical cryptographic problems (e.g., encryption and signature schemes) and on several central tools (e.g., computational difficulty, pseudorandomness, and zero-knowledge proofs). For each of these problems (resp., tools), we start by presenting the natural concern underlying it (resp., its intuitive objective), then define the problem (resp., tool), and finally demonstrate that the problem may be solved (resp., the tool can be constructed). In the latter step, our focus is on demonstrating the feasibility of solving the problem, not on providing a practical solution.

Computational Difficulty

The aforementioned tools and applications (e.g., secure encryption) exist only if some sort of computational hardness exists. Specifically, all these problems and tools require (either explicitly or implicitly) the ability to generate instances of hard problems. Such ability is captured in the definition of one-way functions. Thus, one-way functions are the very minimum needed for doing most natural tasks of cryptography. (It turns out, as we shall see, that this necessary condition is "essentially" sufficient; that is, the existence of one-way functions (or augmentations and extensions of this assumption) suffices for doing most of cryptography.)

Our current state of understanding of efficient computation does not allow us to prove that one-way functions exist. In particular, if $\mathcal{P} = \mathcal{NP}$ then no one-way functions exist. Furthermore, the existence of one-way functions implies that $\mathcal{NP}$ is not contained in $\mathcal{BPP} \supseteq \mathcal{P}$ (not even "on the average"). Thus, proving that one-way functions exist is not easier than proving that $\mathcal{P} \neq \mathcal{NP}$; in fact, the former task seems significantly harder than the latter. Hence, we have no choice (at this stage of history) but to assume that one-way functions exist. As justification to this assumption we can only offer the combined beliefs of hundreds (or thousands) of researchers. Furthermore, these beliefs concern a simply stated assumption, and their validity follows from several widely believed conjectures which are central to various fields (e.g., the conjectured intractability of integer factorization is central to computational number theory).

Since we need assumptions anyhow, why not just assume what we want (i.e., the existence of a solution to some natural cryptographic problem)? Well, first we need to know what we want: as stated above, we must first clarify what exactly we want; that is, go through the typically complex definitional stage. But once this stage is completed, can we just assume that the definition derived can be met?

Not really: once a definition is derived, how can we know that it can at all be met? The way to demonstrate that a definition is viable (and that the corresponding intuitive security concern can be satisfied at all) is to construct a solution based on a *better understood* assumption (i.e., one that is more common and widely believed). For example, looking at the definition of zero-knowledge proofs, it is not a-priori clear that such proofs exist at all (in a non-trivial sense). The non-triviality of the notion was first demonstrated by presenting a zero-knowledge proof system for statements, regarding Quadratic Residuosity, which are believed to be hard to verify (without extra information). Furthermore, contrary to prior beliefs, it was later shown that the existence of one-way functions implies that any NP-statement can be proved in zero-knowledge. Thus, facts that were not known at all to hold (and even believed to be false), were shown to hold by reduction to widely believed assumptions (without which most of modern cryptography collapses anyhow). To summarize, not all assumptions are equal, and so reducing a complex, new and doubtful assumption to a widely-believed and simple (or even merely simpler) assumption is of great value. Furthermore, reducing the solution of a new task to the assumed security of a well-known primitive typically means providing a construction that, using the known primitive, solves the new task. This means that we do not only know (or assume) that the new task is solvable but we also have a solution based on a primitive that, being well-known, typically has several candidate implementations.

C.1.2 Preliminaries

Modern Cryptography, as surveyed here, is concerned with the construction of *efficient* schemes for which it is *infeasible* to violate the security feature. Thus, we need a notion of efficient computations as well as a notion of infeasible ones. The computations of the legitimate users of the scheme ought be efficient, whereas violating the security features (by an adversary) ought to be infeasible. We stress that we do not identify feasible computations with efficient ones, but rather view the former notion as potentially more liberal. Let us elaborate.

C.1.2.1 Efficient Computations and Infeasible ones

Efficient computations are commonly modeled by computations that are polynomial-time in the security parameter. The polynomial bounding the running-time of the legitimate user's strategy is *fixed and typically explicit* (and *small*). Indeed, our aim is to have a notion of efficiency that is as strict as possible (or, equivalently, develop strategies that are as efficient as possible). Here (i.e., when referring to the complexity of the legitimate users) we are in the same situation as in any algorithmic setting. Things are different when referring to our assumptions regarding the computational resources of the adversary, where we refer to the notion of feasible, which we wish to be as wide as possible. A common approach is to postulate that feasible computations are polynomial-time too, but here the polynomial is *not a-priori specified* (and is to be thought of as arbitrarily large). In other words, the

adversary is restricted to the class of polynomial-time computations and anything beyond this is considered to be infeasible.

Although many definitions explicitly refer to the convention of associating feasible computations with polynomial-time ones, this convention is *inessential* to any of the results known in the area. In all cases, a more general statement can be made by referring to a general notion of feasibility, which should be preserved under standard algorithmic composition, yielding theories that refer to adversaries of running-time bounded by any specific super-polynomial function (or class of functions). Still, for sake of concreteness and clarity, we shall use the former convention in our formal definitions (but our motivational discussions will refer to an unspecified notion of feasibility that covers at least efficient computations).

C.1.2.2 Randomized (or probabilistic) Computations

Randomized computations play a central role in cryptography. One fundamental reason for this fact is that randomness is essential for the existence (or rather the generation) of secrets. Thus, we must allow the legitimate users to employ randomized computations, and certainly (since we consider randomization as feasible) we must consider also adversaries that employ randomized computations. This brings up the issue of success probability: typically, we require that legitimate users succeed (in fulfilling their legitimate goals) with probability 1 (or negligibly close to this), whereas adversaries succeed (in violating the security features) with negligible probability. Thus, the notion of a negligible probability plays an important role in our exposition.

One requirement of the definition of negligible probability is to provide a robust notion of rareness: A rare event should occur rarely even if we repeat the experiment for a feasible number of times. That is, in case we consider any polynomial-time computation to be feasible, a function $\mu : \mathbb{N} \to \mathbb{N}$ is called negligible if $1 - (1 - \mu(n))^{p(n)} < 0.01$ for every polynomial p and sufficiently big n (i.e., μ is negligible if for every positive polynomial p' the function $\mu(\cdot)$ is upper-bounded by $1/p'(\cdot)$).

We will also refer to the notion of noticeable probability. Here the requirement is that events that occur with noticeable probability, will occur almost surely (i.e., except with negligible probability) if we repeat the experiment for a polynomial number of times. Thus, a function $\nu:\mathbb{N}\to\mathbb{N}$ is called noticeable if for some positive polynomial p' the function $\nu(\cdot)$ is lower-bounded by $1/p'(\cdot)$.

C.1.3 Prerequisites, Organization, and Beyond

Our aim is to present the basic concepts, techniques and results in cryptography, and our emphasis is on the clarification of fundamental concepts and the relationship among them. This is done in a way independent of the particularities of some popular number theoretic examples. These particular examples played a central role in the development of the field and still offer the most practical implementations of all cryptographic primitives, but this does not mean that the presentation has to be linked to them. On the contrary, we believe that concepts are best clarified when presented at an abstract level, decoupled from specific implementations.

The appendix is organized in two main parts, corresponding to the Basic Tools of Cryptography and the Basic Applications of Cryptography.

The basic tools: The most basic tool is computational difficulty, which in turn is captured by the notion of one-way functions. Another notion of key importance is that of computational indistinguishability, underlying the theory of pseudorandomness as well as much of the rest of cryptography. Pseudorandom generators and functions are important tools that are frequently used. So are zero-knowledge proofs, playing a key role in the design of secure cryptographic protocols and in their study.

The basic applications: Encryption and signature schemes are the most basic applications of Cryptography. Their main utility is in providing secret and reliable communication over insecure communication media. Loosely speaking, encryption schemes are used for ensuring the secrecy (or privacy) of the actual information being communicated, whereas signature schemes are used to ensure its reliability (or authenticity). Another basic topic is the construction of secure cryptographic protocols for the implementation of arbitrary functionalities.

The presentation of the basic tools in Sections C.2–C.4 augments (and sometimes repeats parts of) Sections 7.1, 8.2, and 9.2 (which provide a basic treatment of one-way functions, pseudorandom generators, and zero-knowledge proofs, respectively). Sections C.5–C.7, provide a overview of the basic applications; that is, Encryption Schemes, Signature Schemes, and General Cryptographic Protocols.

Suggestions for further reading. This appendix is a brief summary of the author's two-volume work on the subject [87, 88]. Furthermore, the first part (i.e., Basic Tools) corresponds to [87], whereas the second part (i.e., Basic Applications) corresponds to [88]. Needless to say, the interested reader is referred to these textbooks for further detail (and, in particular, for missing references).

Practice. The aim of this appendix is to introduce the reader to the *theoretical foundations* of cryptography. As argued, such foundations are necessary for *sound* practice of cryptography. Indeed, practice requires more than theoretical foundations, whereas the current text makes no attempt to provide anything beyond the latter. However, given a sound foundation, one can learn and evaluate various practical suggestions that appear elsewhere. On the other hand, lack of sound foundations results in inability to critically evaluate practical suggestions, which in turn leads to unsound decisions. *Nothing could be more harmful to the design of schemes that need to withstand adversarial attacks than misconceptions about such attacks.*

C.2 Computational Difficulty

Modern Cryptography is concerned with the construction of systems that are easy to operate (properly) but hard to foil. Thus, a complexity gap (between the ease of

proper usage and the difficulty of deviating from the prescribed functionality) lies at the heart of Modern Cryptography. However, gaps as required for Modern Cryptography are not known to exist; they are only widely believed to exist. Indeed, almost all of Modern Cryptography rises or falls with the question of whether one-way functions exist. We mention that the existence of one-way functions implies that $\mathcal{NP}$ contains search problems that are hard to solve *on the average*, which in turn implies that $\mathcal{NP}$ is not contained in $\mathcal{BPP}$ (i.e., a worst-case complexity conjecture).

Loosely speaking, one-way functions are functions that are easy to evaluate but hard (on the average) to invert. Such functions can be thought of as an efficient way of generating "puzzles" that are infeasible to solve (i.e., the puzzle is a random image of the function and a solution is a corresponding preimage). Furthermore, the person generating the puzzle knows a solution to it and can efficiently verify the validity of (possibly other) solutions to the puzzle. Thus, one-way functions have, by definition, a clear cryptographic flavor (i.e., they manifest a gap between the ease of one task and the difficulty of a related one).

C.2.1 One-Way Functions

We start by reproducing the basic definition of one-way functions as appearing in Section 7.1.1, where this definition is further discussed.

Definition C.1 (one-way functions, Definition 7.1 restated): *A function* $f:\{0,1\}^* \to \{0,1\}^*$ *is called* one-way *if the following two conditions hold:*

1. easy to evaluate: *There exist a polynomial-time algorithm* A *such that* $A(x) = f(x)$ *for every* $x \in \{0,1\}^*$.
2. hard to invert: *For every probabilistic polynomial-time algorithm* A'*, every polynomial* p*, and all sufficiently large* n*,*

$$\mathsf{Pr}[A'(f(x),1^n) \in f^{-1}(f(x))] < \frac{1}{p(n)}$$

where the probability is taken uniformly over $x \in \{0,1\}^n$ *and all the internal coin tosses of algorithm* A'*.*

Some of the most popular candidates for one-way functions are based on the conjectured intractability of computational problems in number theory. One such conjecture is that it is infeasible to factor large integers. Consequently, the function that takes as input two (equal length) primes and outputs their product is widely believed to be a one-way function. Furthermore, factoring such a composite is infeasible if and only if squaring modulo such a composite is a one-way function (see [176]). For certain composites (i.e., products of two primes that are both congruent to 3 mod 4), the latter function induces a permutation over the set of quadratic residues modulo this composite. A related permutation, which is widely believed to be one-way, is the RSA function [186]: $x \mapsto x^e \bmod N$, where $N = P \cdot Q$ is a composite as above, e is relatively prime to $(P-1)\cdot(Q-1)$, and

$x \in \{0,...,N-1\}$. The latter examples (as well as other popular suggestions) are better captured by the following formulation of a collection of one-way functions (which is indeed related to Definition C.1):

Definition C.2 (collections of one-way functions): *A collection of functions,* $\{f_i: D_i \to \{0,1\}^*\}_{i \in \overline{I}}$*, is called* one-way *if there exists three probabilistic polynomial-time algorithms,* I*,* D *and* F*, such that the following two conditions hold*:

1. easy to sample and compute: *On input* 1^n*, the output of* (the index selection) *algorithm* I *is distributed over the set* $\overline{I} \cap \{0,1\}^n$ (*i.e., is an* n*-bit long index of some function*). *On input* (*an index of a function*) $i \in \overline{I}$*, the output of* (the domain sampling) *algorithm* D *is distributed over the set* D_i (*i.e., over the domain of the function*). *On input* $i \in \overline{I}$ *and* $x \in D_i$*,* (the evaluation) *algorithm* F always *outputs* $f_i(x)$.

2. hard to invert:[1] *For every probabilistic polynomial-time algorithm,* A'*, every positive polynomial* $p(\cdot)$*, and all sufficiently large* n*'s*

$$\Pr\left[A'(i, f_i(x)) \in f_i^{-1}(f_i(x))\right] < \frac{1}{p(n)}$$

where $i \leftarrow I(1^n)$ *and* $x \leftarrow D(i)$.

The collection is said to be a collection of permutations *if each of the* f_i*'s is a permutation over the corresponding* D_i*, and* $D(i)$ *is almost uniformly distributed in* D_i.

For example, in case of the RSA, one considers $f_{N,e} : D_{N,e} \to D_{N,e}$ that satisfies $f_{N,e}(x) = x^e \bmod N$, where $D_{N,e} = \{0,...,N-1\}$. Definition C.2 is also a good starting point for the definition of a trapdoor permutation.[2] Loosely speaking, the latter is a collection of one-way permutations augmented with an efficient algorithm that allows for inverting the permutation when given adequate auxiliary information (called a trapdoor).

Definition C.3 (trapdoor permutations): *A collection of permutations as in Definition C.2 is called a* trapdoor permutation *if there are two auxiliary probabilistic polynomial-time algorithms* I' *and* F^{-1} *such that (1) the distribution* $I'(1^n)$ *ranges over pairs of strings so that the first string is distributed as in* $I(1^n)$*, and (2) for every* (i,t) *in the range of* $I'(1^n)$ *and every* $x \in D_i$ *it holds that* $F^{-1}(t, f_i(x)) = x$. (That is, t is a trapdoor that allows to invert f_i.)

For example, in case of the RSA, $f_{N,e}$ can be inverted by raising to the power d (modulo $N = P \cdot Q$), where d is the multiplicative inverse of e modulo $(P-1)\cdot(Q-1)$. Indeed, in this case, the trapdoor information is (N,d).

[1] Note that this condition refers to the distributions $I(1^n)$ and $D(i)$, which are merely required to range over $\overline{I} \cap \{0,1\}^n$ and D_i, respectively. (Typically, the distributions $I(1^n)$ and $D(i)$ are (almost) uniform over $\overline{I} \cap \{0,1\}^n$ and D_i, respectively.)

[2] Indeed, a more adequate term would be a collection of trapdoor permutations, but the shorter (and less precise) term is the commonly used one.

Strong versus weak one-way functions (summary of Section 7.1.2). Recall that the foregoing definitions require that any feasible algorithm *succeeds in inverting* the function *with negligible probability*. A weaker notion only requires that any feasible algorithm *fails to invert* the function *with noticeable probability*. It turns out that the existence of such weak one-way functions implies the existence of strong one-way functions (as in Definition C.1). The construction itself is straightforward, but analyzing it transcends the analogous information theoretic setting. Instead, the security (i.e., hardness of inverting) the resulting construction is proved via a so called "reducibility argument" that transforms the violation of the conclusion (i.e., the security of the resulting construction) into a violation of the hypothesis (i.e., the security of the given primitive). This strategy (i.e., a "reducibility argument") is used to prove all conditional results in the area.

C.2.2 Hard-Core Predicates

Recall that saying that a function f is one-way implies that given y (in the range of f) it is infeasible to find a preimage of y under f. This does not mean that it is infeasible to find out partial information about the preimage(s) of y under f. Specifically it may be easy to retrieve half of the bits of the preimage (e.g., given a one-way function f consider the function g defined by $g(x,r) \stackrel{\text{def}}{=} (f(x),r)$, for every $|x|=|r|$). As will become clear in subsequent sections, hiding partial information (about the function's preimage) plays an important role in more advanced constructs (e.g., secure encryption). This partial information can be considered as a "hard core" of the difficulty of inverting f. Loosely speaking, a *polynomial-time computable* (Boolean) predicate b, is called a hard-core of a function f if no feasible algorithm, given $f(x)$, can guess $b(x)$ with success probability that is non-negligibly better than one half. The actual definition is presented in Section 7.1.3 (i.e., Definition 7.6).

Note that if b is a hard-core of a 1-1 function f that is polynomial-time computable then f is a one-way function. On the other hand, recall that Theorem 7.7 asserts that *for any one-way function f, the inner-product mod 2 of x and r is a hard-core of $f'(x,r) = (f(x),r)$.*

C.3 Pseudorandomness

In practice "pseudorandom" sequences are often used instead of truly random sequences. The underlying belief is that if an (efficient) application performs well when using a truly random sequence then it will perform essentially as well when using a "pseudorandom" sequence. However, this belief is not supported by ad-hoc notions of "pseudorandomness" such as passing the statistical tests in [140] or having large "linear-complexity" (as defined in [108]). Needless to say, using such "pseudorandom" sequences (instead of truly random sequences) in a cryptographic application is very dangerous.

In contrast, truly random sequences can be safely replaced by pseudorandom sequences provided that pseudorandom distributions are defined as being compu-

tationally indistinguishable from the uniform distribution. Such a definition makes the soundness of this replacement an easy corollary. Loosely speaking, pseudorandom generators are then defined as efficient procedures for creating long pseudorandom sequences based on few truly random bits (i.e., a short random seed). The relevance of such constructs to cryptography is in providing legitimate users that share short random seeds a method for creating long sequences that look random to any feasible adversary (which does not know the said seed).

C.3.1 Computational Indistinguishability

A central notion in Modern Cryptography is that of "effective similarity" (a.k.a computational indistinguishability; cf. [104, 228]). The underlying thesis is that we do not care whether or not objects are equal, all we care about is whether or not a difference between the objects can be observed by a feasible computation. In case the answer is negative, the two objects are equivalent as far as any practical application is concerned. Indeed, in the sequel we will often interchange such (computationally indistinguishable) objects. In this section we recall the definition of computational indistinguishability (presented in Section 8.2.3), and consider two variants.

Definition C.4 (computational indistinguishability, Definition 8.4 revised[3]): *We say that* $X = \{X_n\}_{n\in\mathbb{N}}$ *and* $Y = \{Y_n\}_{n\in\mathbb{N}}$ *are* computationally indistinguishable *if for every probabilistic polynomial-time algorithm* D *every polynomial* p*, and all sufficiently large* n*,*

$$|\mathsf{Pr}[D(1^n, X_n)=1] - \mathsf{Pr}[D(1^n, Y_n)=1]| < \frac{1}{p(n)}$$

where the probabilities are taken over the relevant distribution (i.e., either X_n or Y_n) *and over the internal coin tosses of algorithm* D*.*

See further discussion in Section 8.2.3. In particular, recall that for "efficiently constructible" distributions, indistinguishability by a single sample (as defined above) implies indistinguishability by multiple samples (as in Definition 8.5).

Extension to ensembles indexed by strings. Here we refer to a natural extension of Definition C.4: Rather than referring to ensembles indexed by $\mathbb{N}$, we refer to ensembles indexed by an arbitrary set $S \subseteq \{0,1\}^*$. Typically, for an ensemble $\{Z_\alpha\}_{\alpha\in S}$, it holds that Z_α ranges over strings of length that is polynomially-related to the length of α.

[3] For sake of streamlining Definition C.4 with Definition C.5 (and unlike in Definition 8.4), here the distinguisher is explicitly given the index n of the distribution that it inspects. (In typical applications, the difference between Definitions 8.4 and C.4 is immaterial because the index n is easily determined from any sample of the corresponding distributions.)

Definition C.5 *We say that* $\{X_\alpha\}_{\alpha\in S}$ *and* $\{Y_\alpha\}_{\alpha\in S}$ *are* computationally indistinguishable *if for every probabilistic polynomial-time algorithm* D *every polynomial* p*, and all sufficiently long* $\alpha \in S$*,*

$$|\mathsf{Pr}[D(\alpha, X_\alpha)=1] - \mathsf{Pr}[D(\alpha, Y_\alpha)=1]| < \frac{1}{p(|\alpha|)}$$

where the probabilities are taken over the relevant distribution (i.e., either X_α *or* Y_α*) and over the internal coin tosses of algorithm* D*.*

Note that Definition C.4 is obtained as a special case by setting $S = \{1^n : n \in \mathbb{N}\}$.

A non-uniform version. A non-uniform definition of computational indistinguishability can be derived from Definition C.5 by artificially augmenting the indices of the distributions. That is, $\{X_\alpha\}_{\alpha\in S}$ and $\{Y_\alpha\}_{\alpha\in S}$ are computationally indistinguishable in a non-uniform sense if for every polynomial p the ensembles $\{X'_{\alpha'}\}_{\alpha'\in S'}$ and $\{Y'_{\alpha'}\}_{\alpha'\in S'}$ are computationally indistinguishable (as in Definition C.5), where $S' = \{\alpha\beta : \alpha \in S \wedge \beta \in \{0,1\}^{p(|\alpha|)}\}$ and $X'_{\alpha\beta} = X_\alpha$ (resp., $Y'_{\alpha\beta} = Y_\alpha$) for every $\beta \in \{0,1\}^{p(|\alpha|)}$. An equivalent (alternative) definition can be obtained by following the formulation that underlies Definition 8.12.

C.3.2 Pseudorandom Generators

Loosely speaking, a pseudorandom generator is an efficient (deterministic) algorithm that on input a short random *seed* outputs a (typically much) longer sequence that is computationally indistinguishable from a uniformly chosen sequence.

Definition C.6 (pseudorandom generator, Definition 8.1 restated): *Let* $\ell:\mathbb{N}\to\mathbb{N}$ *satisfy* $\ell(n) > n$*, for all* $n \in \mathbb{N}$*. A* pseudorandom generator*, with* stretch function ℓ*, is a* (deterministic) *polynomial-time algorithm* G *satisfying the following:*

1. *For every* $s \in \{0,1\}^*$*, it holds that* $|G(s)| = \ell(|s|)$*.*
2. $\{G(U_n)\}_{n\in\mathbb{N}}$ *and* $\{U_{\ell(n)}\}_{n\in\mathbb{N}}$ *are computationally indistinguishable, where* U_m *denotes the uniform distribution over* $\{0,1\}^m$*.*

Indeed, the probability ensemble $\{G(U_n)\}_{n\in\mathbb{N}}$ *is called* pseudorandom.

We stress that pseudorandom sequences can replace truly random sequences not only in "standard" algorithmic applications but also in cryptographic ones. That is, *any* cryptographic application that is secure when the legitimate parties use truly random sequences, is also secure when the legitimate parties use pseudorandom sequences. The benefit in such a substitution (of random sequences by pseudorandom ones) is that the latter sequences can be efficiently generated using much less true randomness. Furthermore, *in an interactive setting*, it is possible to eliminate all random steps from the on-line execution of a program, by replacing them with the generation of pseudorandom bits based on a random seed selected and fixed off-line (or at set-up time). This allows interactive parties to generate

a long sequence of common secret bits based on a shared random seed which may have been selected at a much earlier time.

Various cryptographic applications of pseudorandom generators will be presented in the sequel, but let us first recall that *pseudorandom generators exist if and only if one-way functions exist* (see Theorem 8.11). For further treatment of pseudorandom generators, the reader is referred to Section 8.2.

C.3.3 Pseudorandom Functions

Pseudorandom generators provide a way to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions, introduced and constructed by Goldreich, Goldwasser, and Micali [91], are even more powerful: they provide efficient direct access to the bits of a huge pseudorandom sequence (which is not feasible to scan bit-by-bit). More precisely, a pseudorandom function is an efficient (deterministic) algorithm that given an n-bit *seed*, s, and an n-bit *argument*, x, returns an n-bit string, denoted $f_s(x)$, such that it is infeasible to distinguish the values of f_s, for a uniformly chosen $s \in \{0,1\}^n$, from the values of a truly random function $F : \{0,1\}^n \to \{0,1\}^n$. That is, the (feasible) testing procedure is given oracle access to the function (but not its explicit description), and cannot distinguish the case it is given oracle access to a pseudorandom function from the case it is given oracle access to a truly random function.

Definition C.7 (pseudorandom functions): *A* pseudorandom function (ensemble), *is a collection of functions* $\{f_s : \{0,1\}^{|s|} \to \{0,1\}^{|s|}\}_{s \in \{0,1\}^*}$ *that satisfies the following two conditions:*

1. (efficient evaluation) *There exists an efficient* (deterministic) *algorithm that given a* seed, s, *and an* argument, $x \in \{0,1\}^{|s|}$, *returns* $f_s(x)$.

2. (pseudorandomness) *For every probabilistic polynomial-time oracle machine, M, every positive polynomial p and all sufficiently large n's*

$$\left| \Pr[M^{f_{U_n}}(1^n) = 1] - \Pr[M^{F_n}(1^n) = 1] \right| < \frac{1}{p(n)}$$

   where F_n denotes a uniformly selected function mapping $\{0,1\}^n$ to $\{0,1\}^n$.

One key feature of the foregoing definition is that pseudorandom functions can be generated and shared by merely generating and sharing their seed; that is, a "random looking" function $f_s : \{0,1\}^n \to \{0,1\}^n$, is determined by its n-bit seed s. Parties wishing to share a "random looking" function f_s (determining 2^n-many values), merely need to generate and share among themselves the n-bit seed s. (For example, one party may randomly select the seed s, and communicate it, via a secure channel, to all other parties.) Sharing a pseudorandom function allows parties to determine (by themselves and without any further communication) random-looking values depending on their current views of the environment (which need not be known a priori). To appreciate the potential of this tool, one should realize that sharing a pseudorandom function is essentially as good as being able

to agree, on the fly, on the association of random values to (on-line) given values, where the latter are taken from a huge set of possible values. We stress that this agreement is achieved without communication and synchronization: Whenever some party needs to associate a random value to a given value, $v \in \{0,1\}^n$, it will associate to v the (same) random value $r_v \in \{0,1\}^n$ (by setting $r_v = f_s(v)$, where f_s is a pseudorandom function agreed upon beforehand). Concrete applications of (this power of) pseudorandom functions appear in Sections C.5.2 and C.6.2.

Theorem C.8 (How to construct pseudorandom functions): *Pseudorandom functions can be constructed using any pseudorandom generator.*

Proof Sketch:[4] Let G be a pseudorandom generator that stretches its seed by a factor of two (i.e., $\ell(n) = 2n$), and let $G_0(s)$ (resp., $G_1(s)$) denote the first (resp., last) $|s|$ bits in $G(s)$. Define

$$G_{\sigma_{|s|}\cdots\sigma_2\sigma_1}(s) \stackrel{\text{def}}{=} G_{\sigma_{|s|}}(\cdots G_{\sigma_2}(G_{\sigma_1}(s))\cdots).$$

We consider the function ensemble $\{f_s : \{0,1\}^{|s|} \to \{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$, where $f_s(x) \stackrel{\text{def}}{=} G_x(s)$. Pictorially, the function f_s is defined by n-step walks down a full binary tree of depth n having labels at the vertices. The root of the tree, hereafter referred to as the level 0 vertex of the tree, is labeled by the string s. If an internal vertex is labeled r then its left child is labeled $G_0(r)$ whereas its right child is labeled $G_1(r)$. The value of $f_s(x)$ is the string residing in the leaf reachable from the root by a path corresponding to the string x.

We claim that this function ensemble $\{f_s\}_{s\in\{0,1\}^*}$ is pseudorandom. The proof uses the hybrid technique (cf. Section 8.2.3): The i^{th} hybrid, H_n^i, is a function ensemble consisting of $2^{2^i\cdot n}$ functions $\{0,1\}^n \to \{0,1\}^n$, each determined by 2^i random n-bit strings, denoted $\overline{s} = \langle s_\beta\rangle_{\beta\in\{0,1\}^i}$. The value of such function $h_{\overline{s}}$ at $x = \alpha\beta$, where $|\beta| = i$, is defined to equal $G_\alpha(s_\beta)$. (Pictorially, the function $h_{\overline{s}}$ is defined by placing the strings in $\overline{s}$ in the corresponding vertices of level i, and labeling vertices of lower levels using the very rule used in the definition of f_s.) The extreme hybrids correspond to our indistinguishability claim (i.e., $H_n^0 \equiv f_{U_n}$ and H_n^n is a truly random function), and neighboring hybrids can be related to our indistinguishability hypothesis (specifically, to the indistinguishability of $G(U_n)$ and U_{2n} under multiple samples). □

Variants. Useful variants (and generalizations) of the notion of pseudorandom functions include Boolean pseudorandom functions that are defined over all strings (i.e., $f_s : \{0,1\}^* \to \{0,1\}$) and pseudorandom functions that are defined for other domains and ranges (i.e., $f_s : \{0,1\}^{d(|s|)} \to \{0,1\}^{r(|s|)}$, for arbitrary polynomially bounded functions $d, r : \mathbb{N} \to \mathbb{N}$). Various transformations between these variants are known (cf. [87, Sec. 3.6.4] and [88, Apdx. C.2]).

[4] See details in [87, Sec. 3.6.2].

Applications and a generic methodology. Pseudorandom functions are a very useful cryptographic tool: One may first design a cryptographic scheme assuming that the legitimate users have black-box access to a random function, and next implement the random function using a pseudorandom function. The usefulness of this tool stems from the fact that having (black-box) access to a random function gives the legitimate parties a potential advantage over the adversary (which does not have free access to this function).[5] The security of the resulting implementation (which uses a pseudorandom function) is established in two steps: First one proves the security of an idealized scheme that uses a truly random function, and next one argues that the actual implementation (which uses a pseudorandom function) is secure (as otherwise one obtains an efficient oracle machine that distinguishes a pseudorandom function from a truly random one).

C.4 Zero-Knowledge

Zero-knowledge proofs provide a powerful tool for the design of cryptographic protocols as well as a good bench-mark for the study of various issues regarding such protocols. Loosely speaking, zero-knowledge proofs are proofs that yield nothing beyond the validity of the assertion. That is, a verifier obtaining such a proof only gains conviction in the validity of the assertion (as if it was told by a trusted party that the assertion holds). This is formulated by saying that anything that is feasibly computable from a zero-knowledge proof is also feasibly computable from the (valid) assertion itself. The latter formulation follows the simulation paradigm, which is discussed next, while reproducing part of the discussion in §9.2.1.1 and making additional comments regarding the use of this paradigm in cryptography.

C.4.1 The Simulation Paradigm

A key question regarding the modeling of security concerns is how to express the intuitive requirement that an adversary "gains nothing substantial" by deviating from the prescribed behavior of an honest user. Our approach is that the adversary *gains nothing* if whatever it can obtain by unrestricted adversarial behavior can also be obtained within essentially the same computational effort by a benign behavior. The definition of the "benign behavior" captures what we want to achieve in terms of security, and is specific to the security concern to be addressed. For example, in the context of zero-knowledge the unrestricted adversarial behavior is captured by an arbitrary probabilistic polynomial-time verifier strategy, whereas the benign behavior is any computation that is based (only) on the assertion itself (while assuming that the latter is valid). Other examples are discussed in Sections C.5.1 and C.7.1.

A notable property of the simulation paradigm, as well as of the entire definitional approach surveyed here, is that this approach is overly liberal with respect to

[5] The aforementioned methodology is sound provided that the adversary does not get the description of the pseudorandom function (i.e., the seed) in use, but has only (possibly limited) oracle access to it. This is different from the so-called Random Oracle Methodology.

its view of the abilities of the adversary as well as to what might constitute a gain for the adversary. Thus, the approach may be considered overly cautious, because it prohibits also "non-harmful" gains of some "far fetched" adversaries. We warn against this impression. Firstly, there is nothing more dangerous in cryptography than to consider "reasonable" adversaries (a notion which is almost a contradiction in terms): typically, the adversaries will try exactly what the system designer has discarded as "far fetched". Secondly, it seems impossible to come up with definitions of security that distinguish "breaking the scheme in a harmful way" from "breaking it in a non-harmful way": what is harmful is application-dependent, whereas a good definition of security ought to be application-independent (as otherwise using the scheme in any new application will require a full re-evaluation of its security). Furthermore, even with respect to a specific application, it is typically very hard to classify the set of "harmful breakings".

C.4.2 The Actual Definition

In §9.2.1.2 zero-knowledge was defined as a property of some prover strategies (within the context of interactive proof systems, as defined in Section 9.1.1). More generally, the term may apply to any interactive machine, regardless of its goal. A strategy A is zero-knowledge on (inputs from) the set S if, for every feasible strategy B^*, there exists a feasible computation C^* such that the following two probability ensembles are computationally indistinguishable (according to Definition C.5):

1. $\{(A,B^*)(x)\}_{x\in S} \stackrel{\text{def}}{=}$ the output of B^* after interacting with A on common input $x \in S$; and
2. $\{C^*(x)\}_{x\in S} \stackrel{\text{def}}{=}$ the output of C^* on input $x \in S$.

Recall that the first ensemble represents an actual execution of an interactive protocol, whereas the second ensemble represents the computation of a stand-alone procedure (called the "simulator"), which does not interact with anybody.

The foregoing definition does *not* account for auxiliary information that an adversary B^* may have prior to entering the interaction. Accounting for such auxiliary information is essential for using zero-knowledge proofs as subprotocols inside larger protocols. This is taken care of by a stricter notion called auxiliary-input zero-knowledge, which was not presented in Section 9.2.

Definition C.9 (zero-knowledge, revisited): *A strategy A is* auxiliary-input zero-knowledge *on inputs from S if, for every probabilistic polynomial-time strategy B^* and every polynomial p, there exists a probabilistic polynomial-time algorithm C^* such that the following two probability ensembles are computationally indistinguishable:*

1. $\{(A,B^*(z))(x)\}_{x\in S,\, z\in\{0,1\}^{p(|x|)}} \stackrel{\text{def}}{=}$ *the output of B^* when having auxiliary-input z and interacting with A on common input $x \in S$; and*
2. $\{C^*(x,z)\}_{x\in S,\, z\in\{0,1\}^{p(|x|)}} \stackrel{\text{def}}{=}$ *the output of C^* on inputs $x \in S$ and $z \in \{0,1\}^{p(|x|)}$.*

Almost all known zero-knowledge proofs are in fact auxiliary-input zero-knowledge. As hinted, *auxiliary-input zero-knowledge is preserved under sequential composition*. A simulator for the multiple-session protocol can be constructed by iteratively invoking the single-session simulator that refers to the residual strategy of the adversarial verifier in the given session (while feeding this simulator with the transcript of previous sessions). Indeed, the residual single-session verifier gets the transcript of the previous sessions as part of its auxiliary input (i.e., z in Definition C.9). For details, see [87, Sec. 4.3.4].

C.4.3 A construction and a generic application

A question avoided so far is whether zero-knowledge proofs exist at all. Clearly, every set in $\mathcal{P}$ (or rather in $\mathcal{BPP}$) has a "trivial" zero-knowledge proof (in which the verifier determines membership by itself); however, what we seek is zero-knowledge proofs for statements that the verifier cannot decide by itself.

Assuming the existence of "commitment schemes" (cf. §C.4.3.1), which in turn exist if one-way functions exist [162, 114], *there exist* (auxiliary-input) *zero-knowledge proofs of membership in any NP-set*. These zero-knowledge proofs, abstractly depicted in Construction 9.10, have the following important property: the prescribed prover strategy is efficient, provided it is given as auxiliary-input an NP-witness to the assertion (to be proved).[6] Indeed, by using the standard Karp-reductions to 3-Colorability, the protocol of Construction 9.10 can be used for obtaining zero-knowledge proofs for any set in $\mathcal{NP}$. Implementing the abstract boxes (referred to in Construction 9.10) by commitment schemes, we get:

Theorem C.10 (On the applicability of zero-knowledge proofs): *If* (non-uniformly hard) *one-way functions exist then every set $S \in \mathcal{NP}$ has an auxiliary-input zero-knowledge interactive proof. Furthermore, the prescribed prover strategy can be implemented in probabilistic polynomial-time, provided it is given as auxiliary-input an NP-witness for membership of the common input in S.*

Theorem C.10 makes zero-knowledge a very powerful tool in the design of cryptographic schemes and protocols (see §C.4.3.3). We comment that the intractability assumption used in Theorem C.10 seems essential.

C.4.3.1 Commitment schemes

Loosely speaking, commitment schemes are two-stage (two-party) protocols allowing for one party to commit itself (at the first stage) to a value while keeping the value secret. In a (second) latter stage, the commitment is "opened" and it is guaranteed that the "opening" can yield only a single value, which is determined

[6] The auxiliary-input given to the prescribed prover (in order to allow for an efficient implementation of its strategy) is not to be confused with the auxiliary-input that is given to malicious verifiers (in the definition of auxiliary-input zero-knowledge). The former is typically an NP-witness for the common input, which is available to the user that invokes the prover strategy (cf. the generic application discussed in §C.4.3.3). In contrast, the auxiliary-input that is given to malicious verifiers models arbitrary partial information that may be available to the adversary.

during the committing phase. Thus, the (first stage of the) commitment scheme is both *binding* and *hiding*.

A simple (uni-directional communication) commitment scheme can be constructed based on any one-way 1-1 function f (with a corresponding hard-core b). To commit to a bit σ, the sender uniformly selects $s \in \{0,1\}^n$, and sends the pair $(f(s), b(s) \oplus \sigma)$. Note that this is both binding and hiding. An alternative construction, which can be based on any one-way function, uses a pseudorandom generator G that stretches its seed by a factor of three (cf. Theorem 8.11). A commitment is established, via two-way communication, as follows (cf. [162]): The receiver selects uniformly $r \in \{0,1\}^{3n}$ and sends it to the sender, which selects uniformly $s \in \{0,1\}^n$ and sends $r \oplus G(s)$ if it wishes to commit to the value one and $G(s)$ if it wishes to commit to zero. To see that this is binding, observe that there are at most 2^{2n} "bad" values r that satisfy $G(s_0) = r \oplus G(s_1)$ for some pair (s_0, s_1), and with overwhelmingly high probability the receiver will not pick one of these bad values. The hiding property follows by the pseudorandomness of G.

C.4.3.2 Efficiency considerations

The number of rounds in a protocol is commonly considered the most important efficiency criterion (or complexity measure), and typically one desires to have it be a constant. However, in order to obtain negligible soundness error, the protocol of Construction 9.10 has to be invoked for a non-constant number of times (and the analysis of the resulting protocol relies on the preservation of zero-knowledge under sequential composition). At first glance, it seems that one can derive a constant-round zero-knowledge proof system (of negligible soundness error) by performing these invocations in parallel (rather than sequentially). Unfortunately, it is not clear that the resulting interactive proof is zero-knowledge. Still, under standard intractability assumptions (e.g., the intractability of factoring), constant-round zero-knowledge proofs (of negligible soundness error) do exist for every set in $\mathcal{NP}$.

C.4.3.3 A generic application

As mentioned, Theorem C.10 makes zero-knowledge a very powerful tool in the design of cryptographic schemes and protocols. This wide applicability is due to two important aspects regarding Theorem C.10: Firstly, Theorem C.10 provides a zero-knowledge proof for every NP-set, and secondly the prescribed prover can be implemented in probabilistic polynomial-time when given an adequate NP-witness. We now turn to a typical application of zero-knowledge proofs.

In a typical cryptographic setting, a user U has a secret and is supposed to take some action based on its secret. The question is how can other users verify that U indeed took the correct action (as determined by U's secret and publicly known information). Indeed, if U discloses its secret then anybody can verify that U took the correct action. However, U does not want to reveal its secret. Using zero-knowledge proofs we can satisfy both conflicting requirements (i.e., having other users verify that U took the correct action without violating U's interest

in not revealing its secret). That is, U can prove in zero-knowledge that it took the correct action. Note that U's claim to having taken the correct action is an NP-assertion (since U's legal action is determined as a polynomial-time function of its secret and the public information), and that U has an NP-witness to its validity (i.e., the secret is an NP-witness to the claim that the action fits the public information). Thus, by Theorem C.10, it is possible for U to efficiently prove the correctness of its action without yielding anything about its secret. Consequently, it is fair to ask U to prove (in zero-knowledge) that it behaves properly, and so to force U to behave properly. Indeed, "forcing proper behavior" is the canonical application of zero-knowledge proofs (see §C.7.3.2).

This paradigm (i.e., "forcing proper behavior" via zero-knowledge proofs), which in turn is based on Theorem C.10, has been utilized in numerous different settings. Indeed, this paradigm is the basis for the wide applicability of zero-knowledge protocols in Cryptography.

C.4.4 Variants and Issues

In this section we consider numerous variants on the notion of zero-knowledge and the underlying model of interactive proofs. These include black-box simulation and other variants of zero-knowledge (cf. Section C.4.4.1), as well as notions such as proofs of knowledge, non-interactive zero-knowledge, and witness indistinguishable proofs (cf. Section C.4.4.2).

Before starting, we call the reader's attention to the notion of computational soundness and to the related notion of argument systems, discussed in §9.1.4.2. We mention that argument systems may be more efficient than interactive proofs as well as provide stronger zero-knowledge guarantees. Specifically, almost-perfect zero-knowledge arguments for $\mathcal{NP}$ can be constructed based on any one-way function [165], where almost-perfect zero-knowledge means that the simulator's output is statistically close to the verifier's view in the real interaction (see a discussion in §C.4.4.1). Note that stronger security guarantee for the prover (as provided by almost-perfect zero-knowledge) comes at the cost of weaker security guarantee for the verifier (as provided by computational soundness). The answer to the question of whether or not this trade-off is worthwhile seems to be application dependent, and one should also take into account the availability and complexity of the corresponding protocols.

C.4.4.1 Definitional variations

We consider several definitional issues regarding the notion of zero-knowledge (as defined in Definition C.9).

Universal and black-box simulation. A strengthening of Definition C.9 is obtained by requiring the existence of a universal simulator, denoted $\mathcal{C}$, that can simulate (the interactive gain of) any verifier strategy B^* when given the verifier's program an auxiliary-input; that is, in terms of Definition C.9, one should replace $C^*(x,z)$ by $\mathcal{C}(x,z,\langle B^*\rangle)$, where $\langle B^*\rangle$ denotes the description of the program of B^*

(which may depend on x and on z). That is, we effectively restrict the simulation by requiring that it be a uniform (feasible) function of the verifier's program (rather than arbitrarily depend on it). This restriction is very natural, because it seems hard to envision an alternative way of establishing the zero-knowledge property of a given protocol. Taking another step, one may argue that since it seems infeasible to reverse-engineer programs, the simulator may as well just use the verifier strategy as an oracle (or as a "black-box"). This reasoning gave rise to the notion of black-box simulation, which was introduced and advocated in [94] and further studied in numerous works. The belief was that inherent limitations regarding black-box simulation represent inherent limitations of zero-knowledge itself. For example, it was believed that the *fact* that the parallel version of the interactive proof of Construction 9.10 cannot be simulated in a black-box manner (unless $\mathcal{NP}$ is contained in $\mathcal{BPP}$) *implies* that this version is not zero-knowledge (as per Definition C.9 itself). However, the (underlying) belief that *any* zero-knowledge protocol can be simulated in a black-box manner was refuted recently by Barak [23].

Honest verifier versus general cheating verifier. Definition C.9 refers to all feasible verifier strategies, which is most natural in the cryptographic setting because zero-knowledge is supposed to capture the robustness of the prover under *any feasible* (i.e., adversarial) attempt to gain something by interacting with it. A weaker and still interesting notion of zero-knowledge refers to what can be gained by an "honest verifier" (or rather a semi-honest verifier)[7] that interacts with the prover as directed, with the exception that it may maintain (and output) a record of the entire interaction (i.e., even if directed to erase all records of the interaction). Although such a weaker notion is not satisfactory for standard cryptographic applications, it yields a fascinating notion from a conceptual as well as a complexity-theoretic point of view. Furthermore, every proof system that is *zero-knowledge with respect to the honest-verifier* can be transformed into a *standard zero-knowledge* proof (without using intractability assumptions and in case of "public-coin" proofs this is done without significantly increasing the prover's computational effort; see [219]).

Statistical versus Computational Zero-Knowledge. Recall that Definition C.9 postulates that for every probability ensemble of one type (i.e., representing the verifier's output after interaction with the prover) there exists a "similar" ensemble of a second type (i.e., representing the simulator's output). One key parameter is the interpretation of "similarity". Three interpretations, yielding different notions of zero-knowledge, have been commonly considered in the literature:

[7] The term "honest verifier" is more appealing when considering an alternative (equivalent) formulation of Definition C.9. In the alternative definition (see [87, Sec. 4.3.1.3]), the simulator is "only" required to generate the verifier's view of the real interaction, where the verifier's view includes its (common and auxiliary) inputs, the outcome of its coin tosses, and all messages it has received.

1. Perfect Zero-Knowledge requires that the two probability ensembles be identically distributed.[8]

2. Statistical (or Almost-Perfect) Zero-Knowledge requires that these probability ensembles be statistically close (i.e., the variation distance between them is negligible).

3. Computational (or rather general) Zero-Knowledge requires that these probability ensembles be computationally indistinguishable.

Indeed, Computational Zero-Knowledge is the most liberal notion, and is the notion considered in Definition C.9. We note that the class of problems having statistical zero-knowledge proofs contains several problems that are considered intractable. The interested reader is referred to [218].

Strict versus expected probabilistic polynomial-time. The notion of probabilistic polynomial-time (which is mentioned both with respect to the verifier and the simulator), has been given two interpretations:

1. Strict probabilistic polynomial-time. That is, there exist a (polynomial in the length of the input) bound on the *number of steps in each possible run* of the machine, regardless of the outcome of its coin tosses.

2. Expected probabilistic polynomial-time. The standard approach is to look at the running-time as a random variable and *bound its expectation* (by a polynomial in the length of the input). However, as observed by Levin (see §10.2.1.1), this definitional approach is quite problematic and an alternative treatment of the aforementioned random variable is preferable.

Consequently, the notion of expected polynomial-time raises a variety of conceptual and technical problems. For that reason, whenever possible, one should prefer the more robust (and restricted) notion of strict (probabilistic) polynomial-time. Thus, with the *exception of constant-round* zero-knowledge protocols, whenever we talked of a probabilistic polynomial-time verifier (resp., simulator) we mean one in the strict sense. In contrast, with a couple of exceptions (e.g., [23]), all results regarding *constant-round* zero-knowledge protocols refer to a strict polynomial-time verifier and an expected polynomial-time simulator, which is indeed a small cheat.

C.4.4.2 Related notions: POK, NIZK, and WI

We briefly discuss the notions of proofs of knowledge (POK), non-interactive zero-knowledge (NIZK), and witness indistinguishable proofs (WI).

[8] The actual definition of Perfect Zero-Knowledge allows the simulator to fail (while outputting a special symbol) with negligible probability, and the output distribution of the simulator is conditioned on its not failing.

Proofs of Knowledge. Loosely speaking, proofs of knowledge (cf. [105]) are interactive proofs in which the prover asserts "knowledge" of some object (e.g., a 3-coloring of a graph), and not merely its existence (e.g., the existence of a 3-coloring of the graph, which in turn is equivalent to the assertion that the graph is 3-colorable). See further discussion in Section 9.2.3. We mention that "proofs of knowledge", and in particular zero-knowledge "proofs of knowledge", have many applications to the design of cryptographic schemes and cryptographic protocols. One famous application of zero-knowledge proofs of knowledge is to the construction of identification schemes (e.g., the Fiat-Shamir scheme).

Non-Interactive Zero-Knowledge. The model of non-interactive zero-knowledge proof systems consists of three entities: a prover, a verifier and a uniformly selected reference string (which can be thought of as being selected by a trusted third party). Both the verifier and prover can read the reference string (as well as the common input), and each can toss additional coins. The interaction consists of a single message sent from the prover to the verifier, who is then left with the final decision (whether or not to accept the common input). The (basic) zero-knowledge requirement refers to a simulator that outputs pairs that should be computationally indistinguishable from the distribution (of pairs consisting of a uniformly selected reference string and a random prover message) seen in the real model.[9] Non-interactive zero-knowledge proof systems have numerous applications (e.g., to the construction of public-key encryption and signature schemes, where the reference string may be incorporated in the public-key). Several different definitions of non-interactive zero-knowledge proofs were considered in the literature (see [87, Sec. 4.10] and [88, Sec. 5.4.4.4]). Constructing non-interactive zero-knowledge proofs seems more difficult than constructing interactive zero-knowledge proofs. Still, based on standard intractability assumptions (e.g., intractability of factoring), it is known how to construct a non-interactive zero-knowledge proof for any NP-set.

Witness Indistinguishability. The notion of witness indistinguishability was suggested in [72] as a meaningful relaxation of zero-knowledge. Loosely speaking, for any NP-relation R, a proof (or argument) system for the corresponding NP-set is called witness indistinguishable if no feasible verifier may distinguish the case in which the prover uses one NP-witness to x (i.e., w_1 such that $(x, w_1) \in R$) from the case in which the prover is using a different NP-witness to the same input x (i.e., w_2 such that $(x, w_2) \in R$). Clearly, any zero-knowledge protocol is witness indistinguishable, but the converse does not necessarily hold. Furthermore, it seems that witness indistinguishable protocols are easier to construct than zero-knowledge ones. Another advantage of witness indistinguishable protocols is that they are closed under arbitrary concurrent composition, whereas (in general) zero-knowledge protocols are not closed even under parallel composition. Witness indistinguishable protocols turned out to be an *important tool in the construction of more complex*

[9] Note that the verifier does not effect the distribution seen in the real model, and so the basic definition of zero-knowledge does not refer to it. The verifier (or rather a process of adaptively selecting assertions to be proved) is referred to in the adaptive variants of the definition.

protocols. We refer, in particular, to the technique of [71] for constructing zero-knowledge proofs (and arguments) based on witness indistinguishable proofs (resp., arguments).

C.5 Encryption Schemes

The problem of providing *secret communication over insecure media* is the traditional and most basic problem of cryptography. The setting of this problem consists of two parties communicating through a channel that is possibly tapped by an adversary. The parties wish to exchange information with each other, but keep the "wire-tapper" as ignorant as possible regarding the contents of this information. The canonical solution to this problem is obtained by the use of encryption schemes. Loosely speaking, an encryption scheme is a protocol allowing these parties to communicate *secretly* with each other. Typically, the encryption scheme consists of a pair of algorithms. One algorithm, called encryption, is applied by the sender (i.e., the party sending a message), while the other algorithm, called decryption, is applied by the receiver. Hence, in order to send a message, the sender first applies the encryption algorithm to the message, and sends the result, called the ciphertext, over the channel. Upon receiving a ciphertext, the other party (i.e., the receiver) applies the decryption algorithm to it, and retrieves the original message (called the plaintext).

In order for the foregoing scheme to provide secret communication, the receiver must know something that is not known to the wire-tapper. (Otherwise, the wire-tapper can decrypt the ciphertext exactly as done by the receiver.) This extra knowledge may take the form of the decryption algorithm itself, or some parameters and/or auxiliary inputs used by the decryption algorithm. We call this extra knowledge the decryption-key. Note that, without loss of generality, we may assume that the decryption algorithm is known to the wire-tapper, and that the decryption algorithm operates on two inputs: a ciphertext and a decryption-key. (This description implicitly presupposes the existence of an efficient algorithm for generating (random) keys.) We stress that the existence of a decryption-key, not known to the wire-tapper, is merely a necessary condition for secret communication.

Evaluating the "security" of an encryption scheme is a very tricky business. A preliminary task is to understand what is "security" (i.e., to properly define what is meant by this intuitive term). Two approaches to defining security are known. The first ("classical") approach, introduced by Shannon [196], is *information theoretic*. It is concerned with the "information" about the plaintext that is "present" in the ciphertext. Loosely speaking, if the ciphertext contains information about the plaintext then the encryption scheme is considered insecure. It has been shown that such high (i.e., "perfect") level of security can be achieved only if the key in use is at least as long as the *total* amount of information sent via the encryption scheme [196]. This fact (i.e., that the key has to be longer than the information exchanged using it) is indeed a drastic limitation on the applicability of such (perfectly-secure) encryption schemes.

The second ("modern") approach, followed in the current text, is based on

computational complexity. This approach is based on the thesis that it *does not matter* whether the ciphertext contains information about the plaintext, but rather whether this information can be *efficiently extracted*. In other words, instead of asking whether it is *possible* for the wire-tapper to extract specific information, we ask whether it is *feasible* for the wire-tapper to extract this information. It turns out that the new (i.e., "computational complexity") approach can offer security even when the key is much shorter than the total length of the messages sent via the encryption scheme.

The computational complexity approach enables the introduction of concepts and primitives that cannot exist under the information theoretic approach. A typical example is the concept of *public-key encryption schemes*, introduced by Diffie and Hellman [62] (with the most popular candidate suggested by Rivest, Shamir, and Adleman [186]). Recall that in the foregoing discussion we concentrated on the decryption algorithm and its key. It can be shown that the encryption algorithm must also get, in addition to the message, an auxiliary input that depends on the decryption-key. This auxiliary input is called the encryption-key. Traditional encryption schemes, and in particular all the encryption schemes used in the millennia until the 1980's, operate with an encryption-key that equals the decryption-key. Hence, the wire-tapper in these schemes must be ignorant of the encryption-key, and consequently the *key distribution* problem arises; that is, how can two parties wishing to communicate over an insecure channel agree on a secret encryption/decryption key. (The traditional solution is to exchange the key through an alternative channel that is secure, though much more expensive to use.) The computational complexity approach allows the introduction of encryption schemes in which the encryption-key may be given to the wire-tapper without compromising the security of the scheme. Clearly, the decryption-key in such schemes is different from the encryption-key, and furthermore it is infeasible to obtain the decryption-key from the encryption-key. Such encryption schemes, called public-key schemes, have the advantage of trivially resolving the key distribution problem (because the encryption-key can be publicized). That is, once some Party X generates a pair of keys and publicizes the encryption-key, any party can send encrypted messages to Party X such that Party X can retrieve the actual information (i.e., the plaintext), whereas nobody else can learn anything about the plaintext.

In contrast to public-key schemes, traditional encryption schemes in which the encryption-key equals the description-key are called private-key schemes, because in these schemes the encryption-key must be kept secret (rather than be public as in public-key encryption schemes). We note that a full specification of either schemes requires the specification of the way in which keys are generated; that is, a (randomized) key-generation algorithm that, given a security parameter, produces a (random) pair of corresponding encryption/decryption keys (which are identical in case of private-key schemes).

Thus, both private-key and public-key encryption schemes consist of three efficient algorithms: a key generation algorithm denoted G, an encryption algorithm denoted E, and a decryption algorithm denoted D. For every pair of encryption and decryption keys (e, d) generated by G, and for every plaintext x, it holds that

$D_d(E_e(x)) = x$, where $E_e(x) \stackrel{\text{def}}{=} E(e,x)$ and $D_d(y) \stackrel{\text{def}}{=} D(d,y)$. The difference between the two types of encryption schemes is reflected in the definition of security: the security of a public-key encryption scheme should hold also when the adversary is given the encryption-key, whereas this is not required for a private-key encryption scheme. In the following definitional treatment we focus on the public-key case (and the private-key case can be obtained by omitting the encryption-key from the sequence of inputs given to the adversary).

C.5.1 Definitions

A good disguise should not reveal the person's height.

Shafi Goldwasser and Silvio Micali, 1982

For simplicity, we first consider the encryption of a single message (which, for further simplicity, is assumed to be of length that equals the security parameter, n).[10] As implied by the foregoing discussion, a public-key encryption scheme is said to be secure if it is infeasible to gain any information about the plaintext by looking at the ciphertext (and the encryption-key). That is, whatever information about the plaintext one may compute from the ciphertext and some a-priori information, can be essentially computed as efficiently from the a-priori information alone. This fundamental definition of security, called semantic security, was introduced by Goldwasser and Micali [104].

Definition C.11 (semantic security): *A* public-key *encryption scheme* (G,E,D) *is* semantically secure *if for every probabilistic polynomial-time algorithm, A, there exists a probabilistic polynomial-time algorithm B such that for every two functions $f,h:\{0,1\}^* \to \{0,1\}^*$ and all probability ensembles $\{X_n\}_{n\in\mathbb{N}}$ that satisfy $|h(x)| = \mathrm{poly}(|x|)$ and $X_n \in \{0,1\}^n$, it holds that*

$$\mathsf{Pr}[A(e,E_e(x),h(x))=f(x)] \;<\; \mathsf{Pr}[B(1^n,h(x))=f(x)] + \mu(n)$$

where the plaintext x is distributed according to X_n, the encryption-key e is distributed according to $G(1^n)$, and μ is a negligible function.

That is, it is feasible to predict $f(x)$ from $h(x)$ as successfully as it is to predict $f(x)$ from $h(x)$ and $(e,E_e(x))$, which means that nothing is gained by obtaining $(e,E_e(x))$. Note that no computational restrictions are made regarding the functions h and f. We stress that the foregoing definition (as well as the next one) refers to public-key encryption schemes, and in the case of private-key schemes algorithm A is not given the encryption-key e.

The following technical interpretation of security states that it is infeasible to distinguish the encryptions of any two plaintexts (of the same length). As we shall see, this definition (also originating in [104]) is equivalent to Definition C.11 (and meeting it requires a probabilistic encryption algorithm).

[10] In the case of public-key schemes no generality is lost by these simplifying assumptions, but in the case of private-key schemes one should consider the encryption of polynomially-many messages (as we do at the end of this section).

Definition C.12 (indistinguishability of encryptions): *A* public-key *encryption scheme* (G, E, D) *has* indistinguishable encryptions *if for every probabilistic polynomial-time algorithm,* A*, and all sequences of triples,* $(x_n, y_n, z_n)_{n\in\mathbb{N}}$*, where* $|x_n| = |y_n| = n$ *and* $|z_n| = \mathrm{poly}(n)$*,*

$$|\mathsf{Pr}[A(e, E_e(x_n), z_n)=1] - \mathsf{Pr}[A(e, E_e(y_n), z_n)=1]| = \mu(n)$$

Again, e *is distributed according to* $G(1^n)$*, and* μ *is a negligible function.*

In particular, z_n may equal (x_n, y_n). Thus, it is infeasible to distinguish the encryptions of any two fixed messages (such as the all-zero message and the all-ones message). Thus, the following motto is adequate too.

> *A good disguise should not allow a mother to distinguish her own children.*
>
> Shafi Goldwasser and Silvio Micali, 1982

Definition C.11 is more appealing in most settings where encryption is considered the end goal. Definition C.12 is used to establish the security of candidate encryption schemes as well as to analyze their application as modules inside larger cryptographic protocols. Thus, the equivalence of these definitions is of major importance.

Equivalence of Definitions C.11 and C.12 – proof ideas. Intuitively, indistinguishability of encryptions (i.e., of the encryptions of x_n and y_n) is a special case of semantic security; specifically, it corresponds to the case that X_n is uniform over $\{x_n, y_n\}$, the function f indicates one of the plaintexts and h does not distinguish them (i.e., $f(w) = 1$ iff $w = x_n$ and $h(x_n) = h(y_n) = z_n$, where z_n is as in Definition C.12). The other direction is proved by considering the algorithm B that, on input $(1^n, v)$ where $v = h(x)$, generates $(e, d) \leftarrow G(1^n)$ and outputs $A(e, E_e(1^n), v)$, where A is as in Definition C.11. Indistinguishability of encryptions is used to prove that B performs as well as A (i.e., for every h, f and $\{X_n\}_{n\in\mathbb{N}}$, it holds that $\mathsf{Pr}[B(1^n, h(X_n))=f(X_n)] = \mathsf{Pr}[A(e, E_e(1^n), h(X_n))=f(X_n)]$ approximately equals $\mathsf{Pr}[A(e, E_e(X_n), h(X_n))=f(X_n)]$).

Probabilistic Encryption: A secure *public-key* encryption scheme must employ a probabilistic (i.e., randomized) encryption algorithm. Otherwise, given the encryption-key as (additional) input, it is easy to distinguish the encryption of the all-zero message from the encryption of the all-ones message.[11] This explains the association of the robust security definitions and the method of *probabilistic encryption*, an association that goes back to the title of the pioneering work of Goldwasser and Micali [104].

[11] The same holds for (stateless) *private-key* encryption schemes, when considering the security of encrypting several messages (rather than a single message as done above). For example, if one uses a deterministic encryption algorithm then the adversary can distinguish two encryptions of the same message from the encryptions of a pair of different messages.

Further discussion: We stress that (the equivalent) Definitions C.11 and C.12 go way beyond saying that it is infeasible to recover the plaintext from the ciphertext. The latter statement is indeed a minimal requirement from a secure encryption scheme, but is far from being a sufficient requirement. Typically, encryption schemes are used in applications where even obtaining partial information on the plaintext may endanger the security of the application. When designing an application-independent encryption scheme, we do not know which partial information endangers the application and which does not. Furthermore, even if one wants to design an encryption scheme tailored to a specific application, it is rare (to say the least) that one has a precise characterization of all possible partial information that endanger this application. Thus, we need to require that it is infeasible to obtain any information about the plaintext from the ciphertext. Furthermore, in most applications the plaintext may not be uniformly distributed and some a-priori information regarding it may be available to the adversary. We require that the secrecy of all partial information is preserved also in such a case. That is, even in presence of a-priori information on the plaintext, it is infeasible to obtain any (new) information about the plaintext from the ciphertext (beyond what is feasible to obtain from the a-priori information on the plaintext). The definition of semantic security postulates all of this. The equivalent definition of indistinguishability of encryptions is useful in demonstrating the security of candidate constructions as well as for arguing about their effect as part of larger protocols.

Security of multiple messages: Definitions C.11 and C.12 refer to the security of an encryption scheme that is used to encrypt a single plaintext (per a generated key). Since the plaintext may be longer than the key[12], these definitions are already non-trivial, and an encryption scheme satisfying them (even in the private-key model) implies the existence of one-way functions. Still, in many cases, it is desirable to encrypt many plaintexts using the same encryption-key. Loosely speaking, an encryption scheme is secure in the multiple-messages setting if conditions as in Definition C.11 (resp., Definition C.12) hold when polynomially-many plaintexts are encrypted using the same encryption-key (cf. [88, Sec. 5.2.4]). *In the public-key model*, security in the single-message setting implies security in the multiple-messages setting. We stress that this is not necessarily true *for the private-key model*.

C.5.2 Constructions

It is common practice to use "pseudorandom generators" as a basis for private-key encryption schemes. We stress that this is a very dangerous practice when the "pseudorandom generator" is easy to predict (such as the "linear congruential generator"). However, this common practice becomes sound provided one uses

[12]Recall that for sake of simplicity we have considered only messages of length n, but the general definitions refer to messages of arbitrary (polynomial in n) length. We comment that, in the general form of Definition C.11, one should provide the length of the message as an auxiliary input to both algorithms (A and B).

pseudorandom generators (as defined in Section C.3.2). An alternative and more flexible construction follows.

Private-Key Encryption Scheme based on Pseudorandom Functions: We present a simple construction that uses pseudorandom functions as defined in Section C.3.3. The key generation algorithm consists of selecting a seed, denoted s, for a (pseudorandom) function, denoted f_s. To encrypt a message $x \in \{0,1\}^n$ (using key s), the encryption algorithm uniformly selects a string $r \in \{0,1\}^n$ and produces the ciphertext $(r, x \oplus f_s(r))$, where $\oplus$ denotes the exclusive-or of bit strings. To decrypt the ciphertext (r, y) (using key s), the decryption algorithm just computes $y \oplus f_s(r)$. The proof of security of this encryption scheme consists of two steps (suggested as a general methodology in Section C.3.3):

1. Proving that an idealized version of the scheme, in which one uses a uniformly selected function $F:\{0,1\}^n \to \{0,1\}^n$, rather than the pseudorandom function f_s, is secure.

2. Concluding that the real scheme is secure (because, otherwise one could distinguish a pseudorandom function from a truly random one).

Note that we could have gotten rid of the randomization (in the encryption process) if we had allowed the encryption algorithm to be history dependent (e.g., use a counter in the role of r). This can be done if all parties that use the same key (for encryption) coordinate their encryption actions (by maintaining a joint state (e.g., counter)). Indeed, when using a private-key encryption scheme, a common situation is that the same key is only used for communication between two specific parties, which update a joint counter during their communication. Furthermore, if the encryption scheme is used for FIFO communication between the parties and both parties can reliably maintain the counter value, then there is no need (for the sender) to send the counter value. (The resulting scheme is related to "stream ciphers" which are commonly used in practice.)

We comment that the use of a counter (or any other state) in the encryption process is not reasonable in the case of public-key encryption schemes, because it is incompatible with the canonical usage of such schemes (i.e., allowing all parties to send encrypted messages to the "owner of the encryption-key" without engaging in any type of further coordination or communication). Furthermore (unlike in the case of private-key schemes), probabilistic encryption is essential for a secure public-key encryption scheme *even in the case of encrypting a single message*. Following Goldwasser and Micali [104], we now demonstrate the use of *probabilistic encryption* in the construction of public-key encryption schemes.

Public-Key Encryption Scheme based on Trapdoor Permutations: We present two constructions that employ a collection of trapdoor permutations, as defined in Definition C.3. Let $\{f_i : D_i \to D_i\}_i$ be such a collection, and let b be a corresponding hard-core predicate. The key generation algorithm consists of selecting a permutation f_i along with a corresponding trapdoor t, and outputting

(i,t) as the key-pair. To encrypt a (*single*) bit σ (using the encryption-key i), the encryption algorithm uniformly selects $r \in D_i$, and produces the ciphertext $(f_i(r), \sigma \oplus b(r))$. To decrypt the ciphertext (y,τ) (using the decryption-key t), the decryption algorithm computes $\tau \oplus b(f_i^{-1}(y))$ (using the trapdoor t of f_i). Clearly, $(\sigma \oplus b(r)) \oplus b(f_i^{-1}(f_i(r))) = \sigma$. Indistinguishability of encryptions is implied by the hypothesis that b is a hard-core of f_i. We comment that this scheme is quite wasteful in bandwidth; nevertheless, the paradigm underlying its construction (i.e., applying the trapdoor permutation to a randomized version of the plaintext rather than to the actual plaintext) is valuable in practice.

A more efficient construction of a public-key encryption scheme, which uses the same key-generation algorithm, follows. To encrypt an ℓ-bit long string x (using the encryption-key i), the encryption algorithm uniformly selects $r \in D_i$, computes $y \leftarrow b(r) \cdot b(f_i(r)) \cdots b(f_i^{\ell-1}(r))$ and produces the ciphertext $(f_i^\ell(r), x \oplus y)$. To decrypt the ciphertext (u,v) (using the decryption-key t), the decryption algorithm first recovers $r = f_i^{-\ell}(u)$ (using the trapdoor t of f_i), and then obtains $v \oplus b(r) \cdot b(f_i(r)) \cdots b(f_i^{\ell-1}(r))$. Note the similarity to the Blum-Micali Construction (depicted in Eq. (8.8)), and the fact that the proof of the pseudorandomness of Eq. (8.8) can be extended to establish the computational indistinguishability of $(b(r) \cdots b(f_i^{\ell-1}(r)), f_i^\ell(r))$ and $(r', f_i^\ell(r))$, for random and independent $r \in D_i$ and $r' \in \{0,1\}^\ell$. Indistinguishability of encryptions follows, and thus the second scheme is secure. We mention that, assuming the intractability of factoring integers, this scheme has a concrete implementation with efficiency comparable to that of RSA.

C.5.3 Beyond Eavesdropping Security

Our treatment so far has referred only to a "passive" attack in which the adversary merely eavesdrops the line over which ciphertexts are sent. Stronger types of attacks (i.e., "active" ones), culminating in the so-called Chosen Ciphertext Attack, may be possible in various applications. Specifically, in some settings it is feasible for the adversary to make the sender encrypt a message of the adversary's choice, and in some settings the adversary may even make the receiver decrypt a ciphertext of the adversary's choice. This gives rise to *chosen plaintext attacks* and to *chosen ciphertext attacks*, respectively, which are not covered by the security definitions considered in Sections C.5.1 and C.5.2. Here we briefly discuss such "active" attacks, focusing on chosen ciphertext attacks (of the strongest type known as "a posteriori" or "CCA2").

Loosely speaking, in a chosen ciphertext attack, the adversary may obtain the decryptions of ciphertexts of its choice, and is deemed successful if it learns something regarding the plaintext that corresponds to some different ciphertext (see [88, Sec. 5.4.4]). That is, the adversary is given oracle access to the decryption function corresponding to the decryption-key in use (and, in the case of private-key schemes, it is also given oracle access to the corresponding encryption function). The adversary is allowed to query the decryption oracle on any ciphertext except for the "test ciphertext" (i.e., the very ciphertext for which it tries to learn something about the corresponding plaintext). It may also make queries that do not correspond to legitimate ciphertexts, and the answer will be accordingly (i.e., a special 'failure'

symbol). Furthermore, the adversary may effect the selection of the test ciphertext (by specifying a distribution from which the corresponding plaintext is to be drawn).

Private-key and public-key encryption schemes secure against chosen ciphertext attacks can be constructed under (almost) the same assumptions that suffice for the construction of the corresponding passive schemes. Specifically:

Theorem C.13 *Assuming the existence of* one-way functions*, there exist* private-key *encryption schemes that are secure against chosen ciphertext attack.*

Theorem C.14 *Assuming the existence of* enhanced[13] trapdoor permutations*, there exist* public-key *encryption schemes that are secure against chosen ciphertext attack.*

Both theorems are proved by constructing encryption schemes in which the adversary's gain from a chosen ciphertext attack is eliminated by making it infeasible (for the adversary) to obtain any useful knowledge via such an attack. In the case of private-key schemes (i.e., Theorem C.13), this is achieved by making it infeasible (for the adversary) to produce legitimate ciphertexts (other than those explicitly given to it, in response to its request to encrypt plaintexts of its choice). This, in turn, is achieved by augmenting the ciphertext with an "authentication tag" that is hard to generate without knowledge of the encryption-key; that is, we use a message-authentication scheme (as defined in Section C.6). In the case of public-key schemes (i.e., Theorem C.14), the adversary can certainly generate ciphertexts by itself, and the aim is to make it infeasible (for the adversary) to produce legitimate ciphertexts without "knowing" the corresponding plaintext. This, in turn, will be achieved by augmenting the plaintext with a non-interactive zero-knowledge "proof of knowledge" of the corresponding plaintext.

Security against chosen ciphertext attack is related to the notion of *non-malleability* of the encryption scheme. Loosely speaking, in a non-malleable encryption scheme it is infeasible for an adversary, given a ciphertext, to produce a valid ciphertext for a related plaintext (e.g., given a ciphertext of a plaintext $1x$, for an unknown x, it is infeasible to produce a ciphertext to the plaintext $0x$). For further discussion see [88, Sec. 5.4.5].

C.6 Signatures and Message Authentication

Both signature schemes and message authentication schemes are methods for "validating" data; that is, verifying that the data was approved by a certain party (or set of parties). The difference between signature schemes and message authentication schemes is that signatures should be "universally verifiable", whereas authentication tags are only required to be verifiable by parties that are also able to generate them.

[13] Loosely speaking, the enhancement refers to the hardness condition of Definition C.2, and requires that it be hard to recover $f_i^{-1}(y)$ also when given the coins used to sample y (rather than merely y itself). See [88, Apdx. C.1].

Signature Schemes: The need to discuss "digital signatures" (cf. [62, 175]) has arisen with the introduction of computer communication to the business environment (in which parties need to commit themselves to proposals and/or declarations that they make). Discussions of "unforgeable signatures" did take place also prior to the computer age, but the objects of discussion were handwritten signatures (and not digital ones), and the discussion was not perceived as related to "cryptography". Loosely speaking, a scheme for unforgeable signatures should satisfy the following:

- each user can *efficiently produce its own signature* on documents of its choice;
- every user can *efficiently verify* whether a given string is a signature of another (specific) user on a specific document; but
- *it is infeasible to produce signatures of other users* to documents they did not sign.

We note that the formulation of unforgeable digital signatures provides also a clear statement of the essential ingredients of handwritten signatures. The ingredients are each person's ability to sign for itself, a universally agreed verification procedure, and the belief (or assertion) that it is infeasible (or at least hard) to forge signatures (i.e., produce some other person's signatures to documents that were not signed by it such that these "unauthentic" signatures are accepted by the verification procedure).

Message authentication schemes: Message authentication is a task related to the setting considered for encryption schemes; that is, communication over an insecure channel. This time, we consider an active adversary that is monitoring the channel and may alter the messages sent over it. The parties communicating through this insecure channel wish to authenticate the messages they send such that their counterpart can tell an original message (sent by the sender) from a modified one (i.e., modified by the adversary). Loosely speaking, a scheme for message authentication should satisfy the following:

- each of the communicating parties can *efficiently produce an authentication tag* to any message of its choice;
- each of the communicating parties can *efficiently verify* whether a given string is an authentication tag of a given message; but
- *it is infeasible for an external adversary* (i.e., a party other than the communicating parties) *to produce authentication tags* to messages not sent by the communicating parties.

Note that, in contrast to the specification of signature schemes, we do not require universal verification: only the designated receiver is required to be able to verify the authentication tags. Furthermore, we do not require that the receiver can not produce authentication tags by itself (i.e., we only require that *external parties* can

not do so). Thus, message authentication schemes cannot convince *a third party* that the sender has indeed sent the information (rather than the receiver having generated it by itself). In contrast, signatures can be used to convince third parties: in fact, a signature to a document is typically sent to a second party so that in the future this party may (by merely presenting the signed document) convince third parties that the document was indeed generated (or sent or approved) by the signer.

C.6.1 Definitions

Formally speaking, both signature schemes and message authentication schemes consist of three efficient algorithms: key generation, signing and verification. As in the case of encryption schemes, the key-generation algorithm, denoted G, is used to generate a pair of corresponding keys, one is used for signing (via algorithm S) and the other is used for verification (via algorithm V). That is, $S_s(\alpha)$ denotes a signature produced by algorithm S on input a signing-key s and a document α, whereas $V_v(\alpha, \beta)$ denotes the verdict of the verification algorithm V regarding the document α and the alleged signature β relative to the verification-key v. Needless to say, for any pair of keys (s, v) generated by G and for every α, it holds that $V_v(\alpha, S_s(\alpha)) = 1$.

The difference between the two types of schemes is reflected in the definition of security. In the case of *signature schemes*, the adversary is given the verification-key, whereas in the case of *message authentication schemes* the verification-key (which may equal the signing-key) is not given to the adversary. Thus, schemes for message authentication can be viewed as a private-key version of signature schemes. This difference yields different functionalities (even more than in the case of encryption): In typical use of a signature scheme, each user generates a pair of signing and verification keys, publicizes the verification-key and keeps the signing-key secret. Subsequently, each user may sign documents using its own signing-key, and these signatures are *universally verifiable* with respect to its public verification-key. In contrast, message authentication schemes are typically used to authenticate information sent among a set of *mutually trusting* parties that agree on a secret key, which is being used both to produce and verify authentication-tags. (Indeed, it is assumed that the mutually trusting parties have generated the key together or have exchanged the key in a secure way, prior to the communication of information that needs to be authenticated.)

We focus on the definition of secure signature schemes, and consider very powerful attacks on the signature scheme as well as a very liberal notion of breaking it. Specifically, the attacker is allowed to obtain signatures to any message of its choice. One may argue that in many applications such a general attack is not possible (because messages to be signed must have a specific format). Yet, our view is that it is impossible to define a general (i.e., application-independent) notion of admissible messages, and thus a general/robust definition of an attack seems to have to be formulated as suggested here. (Note that at worst, our approach is overly cautious.) Likewise, the adversary is said to be successful if it can produce a valid signature to *any* message for which it has not asked for a signature during

its attack. Again, this means that the ability to form signatures to "nonsensical" messages is also viewed as a breaking of the scheme. Yet, again, we see no way to have a general (i.e., application-independent) notion of "meaningful" messages (such that only forging signatures to them will be considered a breaking of the scheme).

Definition C.15 (secure signature schemes – a sketch): *A* chosen message attack *is a process that, on input a verification-key, can obtain signatures* (relative to the corresponding signing-key) *to messages of its choice. Such an attack is said to* succeed (in existential forgery) *if it outputs a valid signature to a message for which it has* not *requested a signature during the attack. A signature scheme is* secure (or unforgeable) *if every* feasible *chosen message attack succeeds with at most negligible probability, where the probability is taken over the initial choice of the key-pair as well as over the adversary's actions.*

We stress that *plain* RSA (alike plain versions of Rabin's scheme [176] and the DSS) is not secure under the above definition. However, it may be secure if the message is "randomized" before RSA (or the other schemes) is applied.

C.6.2 Constructions

Secure *message authentication schemes* can be constructed using pseudorandom functions. Specifically, the key-generation algorithm consists of selecting a seed $s \in \{0,1\}^n$ for such a function, denoted $f_s : \{0,1\}^* \to \{0,1\}^n$, and the (only valid) tag of message x with respect to the key s is $f_s(x)$. As in the case of our private-key encryption scheme, the proof of security of the current message authentication scheme consists of two steps:

1. Proving that an idealized version of the scheme, in which one uses a uniformly selected function $F : \{0,1\}^* \to \{0,1\}^n$, rather than the pseudorandom function f_s, is secure (i.e., unforgeable).

2. Concluding that the real scheme is secure (because, otherwise one could distinguish a pseudorandom function from a truly random one).

Note that this message authentication scheme makes an "extensive use of pseudorandom functions" (i.e., the pseudorandom function is applied directly to the message, which requires a generalized notion of pseudorandom functions (cf. end of Section C.3.3)). More efficient schemes can be constructed either based on a more restricted use of a pseudorandom function or based on other cryptographic primitives.

Constructing secure *signature schemes* seems more difficult than constructing message authentication schemes. Nevertheless, secure signature schemes can be constructed based on the same assumptions.

Theorem C.16 *The following three conditions are equivalent:*

1. *One-way functions exist.*
2. *Secure signature schemes exist.*

3. Secure message authentication schemes exist.

We stress that, unlike in the case of public-key encryption schemes, the construction of signature schemes (which may be viewed as a public-key analogue of message authentication) does not require a trapdoor property.

How to construct secure signature schemes

Three central paradigms used in the construction of secure *signature schemes* are the "refreshing" of the "effective" signing-key, the usage of an "authentication tree", and the "hashing paradigm" (to be all discussed in the sequel). In addition to being used in the proof of Theorem C.16, these three paradigms are of independent interest.

The refreshing paradigm. Introduced in [106], the *refreshing paradigm* is aimed at limiting the potential dangers of chosen message attacks. This is achieved by signing the actual document using a newly (and randomly) generated instance of the signature scheme, and authenticating (the verification-key of) this random instance with respect to the fixed public-key. That is, consider a basic signature scheme (G, S, V) used as follows. Suppose that the user U has generated a key-pair, $(s, v) \leftarrow G(1^n)$, and has placed the verification-key v on a public-file. When a party asks U to sign some document α, the user U generates a new ("fresh") key-pair, $(s', v') \leftarrow G(1^n)$, signs v' using the original signing-key s, signs α using the new signing-key s', and presents $(S_s(v'), v', S_{s'}(\alpha))$ as a signature to α. An alleged signature, (β_1, v', β_2), is verified by checking whether both $V_v(v', \beta_1) = 1$ and $V_{v'}(\alpha, \beta_2) = 1$ hold. Intuitively, the gain in terms of security is that a full-fledged chosen message attack cannot be launched on a fixed instance of (G, S, V) (i.e., on the fixed verification-key that resides in the public-file and is known to the attacker). All that an attacker may obtain (via a chosen message attack on the new scheme) is signatures, relative to the original signing-key s of (G, S, V), to random strings (distributed according to $G(1^n)$) as well as additional signatures that are each relative to a random and independently distributed signing-key.

Authentication trees. The security benefits of the refreshing paradigm are increased when combining it with the use of *authentication trees*. The idea is to use the public verification-key for authenticating several (e.g., two) fresh instances of the signature scheme, use each of these instances for authenticating several additional fresh instances, and so on. Thus, we obtain a tree of fresh instances of the basic signature scheme, where each internal node authenticates its children. We can now use the leaves of this tree for signing actual documents, where each leaf is used at most once. Thus, a signature to an actual document consists of

1. a signature to this document authenticated with respect to the verification-key associated with some leaf, and
2. a sequence of verification-keys associated with the nodes along the path from the root to this leaf, where each such verification-key is authenticated with respect to the verification-key of its parent.

We stress that the same signature, relative to the key of the parent node, is used for authenticating the verification-keys of all its children. Thus[14], each instance of the signature scheme is used for signing at most one string (i.e., a single sequence of verification-keys if the instance resides in an internal node, and an actual document if the instance resides in a leaf). Hence, it suffices to use a signature scheme that is secure as long as it is used for legitimately signing a *single* string. Such signature schemes, called one-time signature schemes, are easier to construct than standard signature schemes, especially if one only wishes to sign strings that are significantly shorter than the signing-key (resp., than the verification-key). For example, using a one-way function f, we may let the signing-key consist of a sequence of n pairs of strings, let the corresponding verification-key consist of the corresponding sequence of images of f, and sign an n-bit long message by revealing the adequate pre-images. (That is, the signing-key consist of a sequence $((s_1^0, s_1^1), ..., (s_n^0, s_n^1)) \in \{0,1\}^{2n^2}$, the corresponding verification-key is $(f(s_1^0), f(s_1^1)), ..., (f(s_n^0), f(s_n^1)))$, and the signature of the message $\sigma_1 \cdots \sigma_n$ is $(s_1^{\sigma_1}, ..., s_n^{\sigma_n})$.)

The hashing paradigm. Note, however, that in the foregoing authentication-tree, the instances of the signature scheme (associated with internal nodes) are used for signing a pair of verification-keys. Thus, we need a one-time signature scheme that can be used for signing messages that are longer than the verification-key. Here is where the *hashing paradigm* comes into play. This paradigm refers to the common practice of signing documents via a two stage process: First the actual document is hashed to a (relatively) short string, and next the basic signature scheme is applied to the resulting string. This practice (as well as other usages of the hashing paradigm) is sound provided that the hashing function belongs to a family of *collision-free hashing* (a.k.a *collision-resistant hashing*) functions. Loosely speaking, given a hash function that is randomly selected in such a family, it is infeasible to find two different strings that are hashed by this function to the same value. We also refer the interested reader to a variant of the *hashing paradigm* that uses the seemingly weaker notion of a family of *Universal One-Way Hash Functions* (see [164] or [88, Sec. 6.4.3]).

C.7 General Cryptographic Protocols

The design of secure protocols that implement arbitrary desired functionalities is a major part of modern cryptography. Taking the opposite perspective, the design of any cryptographic scheme may be viewed as the design of a secure protocol for implementing a corresponding functionality. Still, we believe that it makes sense to

[14] A naive implementation of the foregoing (full-fledged) signature scheme calls for storing in (secure) memory all the instances of the basic (one-time) signature scheme that are generated throughout the entire signing process (which refers to numerous documents). However, we note that it suffices to be able to reconstruct the random-coins used for generating each of these instances, and the former can be determined by a pseudorandom function (applied to the name of the corresponding vertex in the tree). Indeed, the seed of this pseudorandom function will be part of the signing-key of the resulting (full-fledged) signature scheme.

differentiate between basic cryptographic primitives (which involve little interaction) like encryption and signature schemes on one hand, and general cryptographic protocols on the other hand.

In this section, we survey *general* results concerning secure *multi*-party computations, where the *two*-party case is an important special case. In a nutshell, these results assert that one can construct protocols for securely computing *any* desirable multi-party functionality. Indeed, what is striking about these results is their generality, and we believe that the wonder is not diminished by the (various alternative) conditions under which these results hold.

A general framework for casting (m-party) cryptographic (protocol) problems consists of specifying a random process[15] that maps m inputs to m outputs. The inputs to the process are to be thought of as the local inputs of m parties, and the m outputs are their corresponding local outputs. The random process describes the desired functionality. That is, if the m parties were to trust each other (or trust some external party), then they could each send their local input to the trusted party, who would compute the outcome of the process and send to each party the corresponding output. A pivotal question in the area of cryptographic protocols is to what extent can this (imaginary) trusted party be "emulated" by the mutually distrustful parties themselves.

The results surveyed in this section describe a variety of models in which such an "emulation" is possible. The models vary by the underlying assumptions regarding the communication channels, numerous parameters governing the extent of adversarial behavior, and the desired level of emulation of the trusted party (i.e., level of "security"). Our treatment refers to the security of stand-alone executions. The preservation of security in an environment in which many executions of many protocols are attacked is beyond the scope of this section, and the interested reader is referred to [88, Sec. 7.7.2].

C.7.1 The Definitional Approach and Some Models

Before describing the aforementioned results, we further discuss the notion of "emulating a trusted party", which underlies the definitional approach to secure multi-party computation. This approach follows the simulation paradigm (cf. Section C.4.1) which deems a scheme to be secure if whatever a feasible adversary can obtain after attacking it, is also feasibly attainable by a benign behavior. In the general setting of multi-party computation we compare the effect of adversaries that participate in the execution of the actual protocol to the effect of adversaries that participate in an imaginary execution of a trivial (ideal) protocol for computing the desired functionality with the help of a trusted party. If whatever the adversaries can feasibly obtain in the real setting can also be feasibly obtained in

[15] That is, we consider the secure evaluation of randomized functionalities, rather than "only" the secure evaluation of functions. Specifically, we consider an arbitrary (randomized) process F that on input $(x_1,...,x_m)$, first selects at random (depending only on $\ell \stackrel{\text{def}}{=} \sum_{i=1}^{m} |x_i|$) an m-ary function f, and then outputs the m-tuple $f(x_1,...,x_m) = (f_1(x_1,...,x_m),...,f_m(x_1,...,x_m))$. In other words, $F(x_1,...,x_m) = F'(r,x_1,...,x_m)$, where r is uniformly selected in $\{0,1\}^{\ell'}$ (with $\ell' = \text{poly}(\ell)$), and F' is a function mapping $(m+1)$-long sequences to m-long sequences.

the ideal setting then the actual protocol "emulates the ideal setting" (i.e., "emulates a trusted party"), and thus is deemed secure. This basic approach can be applied in a variety of models, and is used to define the goals of security in these models.[16] We first discuss some of the parameters used in defining various models, and next demonstrate the application of this approach in two important cases. For further details, see [88, Sec. 7.2 and 7.5.1].

C.7.1.1 Some parameters used in defining security models

The following parameters are described in terms of the actual (or real) computation. In *some cases*, the corresponding definition of security is obtained by imposing some restrictions or provisions on the ideal model. For example, in the case of two-party computation (see §C.7.1.3), secure computation is possible only if premature termination is *not* considered a breach of security. In that case, the suitable security definition is obtained (via the simulation paradigm) by allowing (an analogue of) premature termination in the ideal model. In *all cases*, the desired notion of security is defined by requiring that for any adequate adversary in the real model, there exist a corresponding adversary in the corresponding ideal model that obtains essentially the same impact (as the real-model adversary).

The communication channels: The standard assumption in cryptography is that the adversary may tap all communication channels (between honest parties), but cannot modify (or omit or insert) messages sent over them. In contrast, one may *postulate* that the adversary cannot obtain messages sent between a pair of honest parties, yielding the so-called private-channel model. Most works in the area assume that communication is *synchronous* and that point-to-point channels exist between every pair of processors (i.e., a *complete network*).

Set-up assumptions: Unless stated differently, no set-up assumptions are made (except for the obvious assumption that all parties have identical copies of the protocol's program).

Computational limitations: Typically, the focus is on computationally-bounded adversaries (e.g., probabilistic polynomial-time adversaries). However, the private-channel model allows for the (meaningful) consideration of computationally-unbounded adversaries.[17]

[16] A few technical comments are in place. Firstly, we assume that the inputs of all parties are of the same length. We comment that as long as the lengths of the inputs are polynomially related, the foregoing convention can be enforced by padding. On the other hand, some length restriction is essential for the security results, because in general it is impossible to hide all information regarding the length of the inputs to a protocol. Secondly, we assume that the desired functionality is computable in probabilistic polynomial-time, because we wish the secure protocol to run in probabilistic polynomial-time (and a protocol cannot be more efficient than the corresponding centralized algorithm). Clearly, the results can be extended to functionalities that are computable within any given (time-constructible) time bound, using adequate padding.

[17] We stress that, also in the case of computationally-unbounded adversaries, security should be defined by requiring that for every real adversary, whatever the adversary can compute after

Restricted adversarial behavior: The parameters of the model include questions like whether the adversary is "active" or "passive" (i.e., whether a dishonest party takes active steps to disrupt the execution of the protocol or merely gathers information) and whether or not the adversary is "adaptive" (i.e., whether the set of dishonest parties is fixed before the execution starts or is adaptively chosen by an adversary during the execution).

Restricted notions of security: One important example is the willingness to tolerate "unfair" protocols in which the execution can be suspended (at any time) by a dishonest party, provided that it is detected doing so. We stress that in case the execution is suspended, the dishonest party does not obtain more information than it could have obtained when not suspending the execution. (What may happen is that the honest parties will not obtain their desired outputs, but will detect that the execution was suspended.) We stress that the motivation to this restricted model is the impossibility of obtaining general secure two-party computation in the unrestricted model.

Upper bounds on the number of dishonest parties: These are assumed in some models, when required. For example, in some models, secure multi-party computation is possible only if a majority of the parties is honest.

C.7.1.2 Example: Multi-party protocols with honest majority

Here we consider an active, non-adaptive, computationally-bounded adversary, and do not assume the existence of private channels. Our aim is to define multi-party protocols that remain secure provided that the honest parties are in majority. (The reason for requiring an honest majority will be discussed at the end of this subsection.)

We first observe that in any multi-party protocol, each party may change its local input before even entering the execution of the protocol. However, this is unavoidable also when the parties utilize a trusted party. Consequently, such an effect of the adversary on the real execution (i.e., modification of its own input prior to entering the actual execution) is not considered a breach of security. In general, whatever cannot be avoided when the parties utilize a trusted party, is not considered a breach of security. We wish secure protocols (in the real model) to suffer only from whatever is unavoidable also when the parties utilize a trusted party. Thus, the basic paradigm underlying the definitions of *secure multi-party computations* amounts to requiring that the only situations that may occur in the real execution of a secure protocol are those that can also occur in a corresponding ideal model (where the parties may employ a trusted party). In other words, the

participating in the execution of the actual protocol is computable *within comparable time* by an imaginary adversary participating in an imaginary execution of the trivial ideal protocol (for computing the desired functionality with the help of a trusted party). That is, although no computational restrictions are made on the real-model adversary, it is required that the ideal-model adversary that obtains the same impact does so within comparable time (i.e., within time that is polynomially related to the running time of the real-model adversary being simulated).

"effective malfunctioning" of parties in secure protocols is restricted to what is postulated in the corresponding ideal model.

When defining secure multi-party protocols (with honest majority), we need to pin-point what cannot be avoided in the ideal model (i.e., when the parties utilize a trusted party). Since we are interested in executions in which the majority of parties are honest, we consider an ideal model in which any minority group (of the parties) may collude as follows:

1. Firstly this dishonest minority shares its original inputs and decides together on replaced inputs to be sent to the trusted party. (The other parties send their respective original inputs to the trusted party.)

2. Upon receiving inputs from all parties, the trusted party determines the corresponding outputs and sends them to the corresponding parties. (We stress that the information sent between the honest parties and the trusted party is not seen by the dishonest colluding minority.)

3. Upon receiving the output-message from the trusted party, each honest party outputs it locally, whereas the dishonest colluding minority may determine their outputs based on all they know (i.e., their initial inputs and their received outputs).

A *secure multi-party computation with honest majority* is required to emulate this ideal model. That is, the effect of any feasible adversary that controls a minority of the parties in a real execution of such a (real) protocol, can be essentially simulated by a (different) feasible adversary that controls the corresponding parties in the ideal model.

Definition C.17 (secure protocols – a sketch): *Let f be an m-ary functionality and Π be an m-party protocol operating in the real model.*

- *For a real-model adversary A, controlling some minority of the parties* (and tapping all communication channels)*, and an m-sequence $\overline{x}$, we denote by* $\mathrm{REAL}_{\Pi,A}(\overline{x})$ *the sequence of m outputs resulting from the execution of Π on input $\overline{x}$ under the attack of the adversary A.*

- *For an ideal-model adversary A', controlling some minority of the parties, and an m-sequence $\overline{x}$, we denote by* $\mathrm{IDEAL}_{f,A'}(\overline{x})$ *the sequence of m outputs resulting from the foregoing three-step ideal process, when applied to input $\overline{x}$ under the attack of the adversary A'.*

We say that Π securely implements f with honest majority *if for every feasible real-model adversary A, controlling some minority of the parties, there exists a feasible ideal-model adversary A', controlling the same parties, such that the probability ensembles* $\{\mathrm{REAL}_{\Pi,A}(\overline{x})\}_{\overline{x}}$ *and* $\{\mathrm{IDEAL}_{f,A'}(\overline{x})\}_{\overline{x}}$ *are computationally indistinguishable* (*as in Definition C.5*).

Thus, security means that the effect of each minority group in a real execution of a secure protocol is "essentially restricted" to replacing its own local inputs

(independently of the local inputs of the majority parties) before the protocol starts, and replacing its own local outputs (depending only on its local inputs and outputs) after the protocol terminates. (We stress that in the real execution the minority parties do obtain additional pieces of information; yet in a secure protocol they gain nothing from these additional pieces of information, because they can actually reproduce those by themselves.)

The fact that Definition C.17 refers to a model without private channels is reflected in the fact that our (sketchy) definition of the real-model adversary allowed it to tap the channels, which in turn effects the set of possible ensembles $\{\text{REAL}_{\Pi,A}(\overline{x})\}_{\overline{x}}$. When defining security in the private-channel model, the real-model adversary is not allowed to tap channels between honest parties, and this again effects the possible ensembles $\{\text{REAL}_{\Pi,A}(\overline{x})\}_{\overline{x}}$. On the other hand, when we wish to define security with respect to passive adversaries, both the scope of the real-model adversaries and the scope of the ideal-model adversaries changes. In the real-model execution, all parties follow the protocol but the adversary may alter the output of the dishonest parties arbitrarily depending on their intermediate internal states (during the entire execution). In the corresponding ideal-model, the adversary is not allowed to modify the *inputs* of dishonest parties (in Step 1), but is allowed to modify their outputs (in Step 3).

We comment that a definition analogous to Definition C.17 can be presented also in the case that the dishonest parties are not in minority. In fact, such a definition seems more natural, but the problem is that such a definition cannot be satisfied. That is, most (natural) functionalities do not have a protocol for computing them securely in the case that at least half of the parties are dishonest and employ an adequate adversarial strategy. This follows from an impossibility result regarding two-party computation, which essentially asserts that there is no way to prevent a party from prematurely suspending the execution. On the other hand, secure multi-party computation with dishonest majority is possible if premature suspension of the execution is not considered a breach of security (see §C.7.1.3).

C.7.1.3 Another example: Two-party protocols allowing abort

In light of the last paragraph, we now consider multi-party computations in which premature suspension of the execution is not considered a breach of security. For simplicity, we focus on the special case of two-party computations (As in §C.7.1.2, we consider a non-adaptive, active, computationally-bounded adversary.)

Intuitively, in any two-party protocol, each party may suspend the execution at any point in time, and furthermore it may do so as soon as it learns the desired output. Thus, in case the output of each parties depends on both inputs, it is always possible for one of the parties to obtain the desired output while preventing the other party from fully determining its own output.[18] The same phenomenon occurs even in case the two parties just wish to generate a common random value. Thus, when considering active adversaries in the two-party setting, we do not consider

[18] In contrast, in the case of an honest majority (cf., §C.7.1.2), the honest party that fails to obtain its output is not alone. It may seek help from the other honest parties, which together and being in majority can do things that dishonest minorities cannot do: See §C.7.3.2.

such premature suspension of the execution a breach of security. Consequently, we consider an ideal model where each of the two parties may "shut-down" the trusted (third) party at any point in time. In particular, this may happen after the trusted party has supplied the outcome of the computation to one party but before it has supplied the outcome to the other. Thus, an execution in the corresponding ideal model proceeds as follows:

1. Each party sends its input to the trusted party, where the dishonest party may replace its input or send no input at all (which can be treated as sending a default value).

2. Upon receiving inputs from both parties, the trusted party determines the corresponding pair of outputs, and sends the first output to the first party.

3. In case the first party is dishonest, it may instruct the trusted party to halt, otherwise it always instructs the trusted party to proceed. If instructed to proceed, the trusted party sends the second output to the second party.

4. Upon receiving the output-message from the trusted party, an honest party outputs it locally, whereas a dishonest party may determine its output based on all it knows (i.e., its initial input and its received output).

A secure two-party computation allowing abort is required to emulate this ideal model. That is, as in Definition C.17, security is defined by requiring that for every feasible real-model adversary A, there exists a feasible ideal-model adversary A', controlling the same party, such that the probability ensembles representing the corresponding (real and ideal) executions are computationally indistinguishable. This means that each party's "effective malfunctioning" in a secure protocol is restricted to supplying an initial input of its choice and aborting the computation at any point in time. (Needless to say, the choice of the initial input of each party may *not* depend on the input of the other party.)

We mention that an alternative way of dealing with the problem of premature suspension of execution (i.e., abort) is to restrict our attention to single-output functionalities; that is, functionalities in which only one party is supposed to obtain an output. The definition of secure computation of such functionalities can be made identical to Definition C.17, with the exception that no restriction is made on the set of dishonest parties (and in particular one may consider a single dishonest party in the case of two-party protocols). For further details, see [88, Sec. 7.2.3].

C.7.2 Some Known Results

We next list some of the models for which general secure multi-party computation is known to be attainable (i.e., models in which one can construct secure multi-party protocols for computing any desired functionality). We mention that the first results of this type were obtained by Goldreich, Micali, Wigderson and Yao [96, 230, 97].

In the standard channel model. *Assuming the existence of enhanced*[19] *trapdoor permutations*, secure multi-party computation is possible in the following three models (cf. [96, 230, 97] and details in [88, Chap. 7]):

1. Passive adversary, for any number of dishonest parties (see [88, Sec. 7.3]).
2. Active adversary that may control only a minority of the parties (see [88, Sec. 7.5.4]).
3. Active adversary, for any number of dishonest parties, provided that suspension of execution (as discussed in §C.7.1.3) is not considered a violation of security (see [88, Sec. 7.4 and 7.5.5]).

In all these cases, the adversary is computationally-bounded and non-adaptive. On the other hand, the adversary may tap the communication lines between honest parties (i.e., we do not assume "private channels" here). The results for active adversaries assume a broadcast channel. Indeed, the latter can be implemented (while tolerating any number of dishonest parties) using a signature scheme and assuming that each party knows (or can reliably obtain) the verification-key corresponding to each of the other parties.

In the private channels model. Making no computational assumptions and allowing computationally-unbounded adversaries, but *assuming private channels*, secure multi-party computation is possible in the following two models (cf. [32, 50]):

1. Passive adversary that may control only a minority of the parties.
2. Active adversary that may control only less than one third of the parties.

In both cases the adversary may be adaptive.

C.7.3 Construction Paradigms and Two Simple Protocols

We briefly sketch a couple of paradigms used in the construction of secure multi-party protocols. We focus on the construction of secure protocols for the model of computationally-bounded and non-adaptive adversaries [96, 230, 97]. These constructions proceed in two steps (see details in [88, Chap. 7]): First a secure protocol is presented for the model of passive adversaries (for any number of dishonest parties), and next such a protocol is "compiled" into a protocol that is secure in one of the two models of active adversaries (i.e., either in a model allowing the adversary to control only a minority of the parties or in a model in which premature suspension of the execution is not considered a violation of security). These two steps are presented in the following two corresponding subsections, in which we also present two relatively simple protocols for two specific tasks, which in turn are used extensively in the general protocols.

Recall that in the model of passive adversaries, all parties follow the prescribed protocol, but at termination the adversary may alter the outputs of the dishonest

[19] See Footnote 13.

parties depending on their intermediate internal states (during the entire execution). Below, we refer to protocols that are secure in the model of passive (resp., active) adversaries by the term passively-secure (resp., actively-secure).

C.7.3.1 Passively-secure computation with shares

For any $m \geq 2$, suppose that m parties, each having a private input, wish to obtain the value of a predetermined m-argument function evaluated at their sequence of inputs. Below, we outline a passively-secure protocol for achieving this goal. We mention that the design of passively-secure multi-party protocol for any functionality (allowing different outputs to different parties as well as handling also randomized computations) reduces easily to the aforementioned task.

We assume that the parties hold a circuit for computing the value of the function on inputs of the adequate length, and that the circuit contains only `and` and `not` gates. The key idea is having each party "secretly share" its input with everybody else, and having the parties "secretly transform" shares of the input wires of the circuit into shares of the output wires of the circuit, thus obtaining shares of the outputs (which allows for the reconstruction of the actual outputs). The value of each wire in the circuit is shared such that all shares yield the value, whereas lacking even one of the shares keeps the value totally undetermined. That is, we use a simple secret sharing scheme such that a bit b is shared by a random sequence of m bits that sum-up to $b \bmod 2$. First, each party shares each of its input bits with all parties (by secretly sending each party a random value and setting its own share accordingly). Next, all parties jointly scan the circuit from its input wires to its output wires, processing each gate as follows:

- When encountering a gate, the parties already hold shares of the values of the wires entering the gate, and their aim is to obtain shares of the value of the wires exiting the gate.
- For a `not`-gate this is easy: the first party just flips the value of its share, and all other parties maintain their shares.
- Since an `and`-gate corresponds to multiplication modulo 2, the parties need to securely compute the following randomized functionality (in which the x_i's denote shares of one entry-wire, the y_i's denote shares of the second entry-wire, the z_i's denote shares of the exit-wire, and the shares indexed by i are held by Party i):

$$((x_1, y_1), ..., (x_m, y_m)) \mapsto (z_1, ..., z_m), \text{ where} \tag{C.1}$$

$$\sum_{i=1}^{m} z_i = \left(\sum_{i=1}^{m} x_i\right) \cdot \left(\sum_{i=1}^{m} y_i\right) \tag{C.2}$$

  That is, the z_i's are random subject to Eq. (C.2).

Finally, the parties send their shares of each circuit-output wire to the designated party, which reconstructs the value of the corresponding bit. Thus, the parties have

propagated shares of the circuit-input wires into shares of the circuit-output wires, by repeatedly conducting passively-secure computation of the m-ary functionality of Eq. (C.1) & (C.2). That is, securely evaluating the entire (arbitrary) circuit "reduces" to securely conducting a specific (very simple) multi-party computation. But things get even simpler: the key observation is that

$$\left(\sum_{i=1}^{m} x_i\right) \cdot \left(\sum_{i=1}^{m} y_i\right) = \sum_{i=1}^{m} x_i y_i + \sum_{1 \le i < j \le m} (x_i y_j + x_j y_i) \,. \tag{C.3}$$

Thus, the m-ary functionality of Eq. (C.1) & (C.2) can be computed as follows (where all arithmetic operations are mod 2):

1. Each Party i locally computes $z_{i,i} \stackrel{\text{def}}{=} x_i y_i$.
2. Next, each pair of parties (i.e., Parties i and j) securely compute random shares of $x_i y_j + y_i x_j$. That is, Parties i and j (holding (x_i, y_i) and (x_j, y_j), respectively), need to securely compute the randomized two-party functionality $((x_i, y_i), (x_j, y_j)) \mapsto (z_{i,j}, z_{j,i})$, where the z's are random subject to $z_{i,j} + z_{j,i} = x_i y_j + y_i x_j$. Equivalently, Party j uniformly selects $z_{j,i} \in \{0,1\}$, and Parties i and j securely compute the following deterministic functionality

   $$((x_i, y_i), (x_j, y_j, z_{j,i})) \mapsto (z_{j,i} + x_i y_j + y_i x_j, \lambda), \tag{C.4}$$

   where λ denotes the empty string.
3. Finally, for every $i = 1, ..., m$, the sum $\sum_{j=1}^{m} z_{i,j}$ yields the desired share of Party i.

The foregoing construction is analogous to a construction that was outlined in [97]. A detailed description and full proofs appear in [88, Sec. 7.3.4 and 7.5.2].

The foregoing construction reduces the passively-secure computation of any m-ary functionality to the implementation of the simple 2-ary functionality of Eq. (C.4). The latter can be implemented in a passively-secure manner by using a 1-out-of-4 Oblivious Transfer. Loosely speaking, a 1-out-of-k Oblivious Transfer is a protocol enabling one party to obtain one of k secrets held by another party, without the second party learning which secret was obtained by the first party. That is, it allows a passively-secure computation of the two-party functionality

$$(i, (s_1, ..., s_k)) \mapsto (s_i, \lambda). \tag{C.5}$$

Note that any function $f : [k] \times \{0,1\}^* \to \{0,1\}^* \times \{\lambda\}$ can be computed in a passively-secure manner by invoking a 1-out-of-k Oblivious Transfer on inputs i and $(f(1,y), ..., f(k,y))$, where i (resp., y) is the initial input of the first (resp., second) party.

A passively-secure 1-out-of-k Oblivious Transfer. Using a collection of enhanced trapdoor permutations, $\{f_\alpha : D_\alpha \to D_\alpha\}_{\alpha \in I}$ and a corresponding hard-core predicate b, we outline a passively-secure implementation of the functionality of Eq. (C.5), when restricted to single-bit secrets.

Inputs: The first party, hereafter called the receiver, has input $i \in \{1,2,...,k\}$. The second party, called the sender, has input $(\sigma_1,\sigma_2,...,\sigma_k) \in \{0,1\}^k$.

Step S1: The sender selects at random a permutation f_α along with a corresponding trapdoor, denoted t, and sends the permutation f_α (i.e., its index α) to the receiver.

Step R1: The receiver uniformly and independently selects $x_1,...,x_k \in D_\alpha$, sets $y_i = f_\alpha(x_i)$ and $y_j = x_j$ for every $j \neq i$, and sends $(y_1,y_2,...,y_k)$ to the sender.

Thus, the receiver knows $f_\alpha^{-1}(y_i) = x_i$, but cannot predict $b(f_\alpha^{-1}(y_j))$ for any $j \neq i$. Needless to say, the last assertion presumes that the receiver follows the protocol (i.e., we only consider passive-security).

Step S2: Upon receiving $(y_1,y_2,...,y_k)$, using the inverting-with-trapdoor algorithm and the trapdoor t, the sender computes $z_j = f_\alpha^{-1}(y_j)$, for every $j \in \{1,...,k\}$. It sends the k-tuple $(\sigma_1 \oplus b(z_1), \sigma_2 \oplus b(z_2), ..., \sigma_k \oplus b(z_k))$ to the receiver.

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

We first observe that this protocol correctly computes 1-out-of-k Oblivious Transfer; that is, the receiver's local output (i.e., $c_i \oplus b(x_i)$) indeed equals $(\sigma_i \oplus b(f_\alpha^{-1}(f_\alpha(x_i)))) \oplus b(x_i) = \sigma_i$. Next, we offer some intuition as to why this protocol constitutes a privately-secure implementation of 1-out-of-k Oblivious Transfer. Intuitively, the sender gets no information from the execution because, for any possible value of i, the senders sees the same distribution; specifically, a sequence of k uniformly and independently distributed elements of D_α. (Indeed, the key observation is that applying f_α to a uniformly distributed element of D_α yields a uniformly distributed element of D_α.) As for the receiver, intuitively, it gains no computational knowledge from the execution because, for $j \neq i$, the only information that the receiver has regarding σ_j is the triplet $(\alpha, x_j, \sigma_j \oplus b(f_\alpha^{-1}(x_j)))$, where x_j is uniformly distributed in D_α, and from this information it is infeasible to predict σ_j better than by a random guess.[20] (See [88, Sec. 7.3.2] for a detailed proof of security.)

C.7.3.2 From passively-secure protocols to actively-secure ones

We show how to transform any passively-secure protocol into a corresponding actively-secure protocol. The communication model in both protocols consists of a single broadcast channel. Note that the messages of the original protocol may be assumed to be sent over a broadcast channel, because the adversary may see them anyhow (by tapping the point-to-point channels), and because a broadcast

[20] The latter intuition presumes that sampling D_α is trivial (i.e., that there is an easily computable correspondence between the coins used for sampling and the resulting sample), whereas in general the coins used for sampling may be hard to compute from the corresponding outcome. This is the reason that an enhanced hardness assumption is used in the general analysis of the foregoing protocol.

channel is trivially implementable in the case of passive adversaries. As for the resulting actively-secure protocol, the broadcast channel it uses can be implemented via an (authenticated) Byzantine Agreement protocol, thus providing an emulation of this model on the standard point-to-point model (in which a broadcast channel does not exist). We mention that authenticated Byzantine Agreement is typically implemented using a signature scheme (and assuming that each party knows the verification-key corresponding to each of the other parties).

Turning to the transformation itself, the main idea is using zero-knowledge proofs (as described in §C.4.3.3) in order to force parties to behave in a way that is consistent with the (passively-secure) protocol. Actually, we need to confine each party to a unique consistent behavior (i.e., according to some fixed local input and a sequence of coin tosses), and to guarantee that a party cannot fix its input (and/or its coin tosses) in a way that depends on the inputs (and/or coin tosses) of honest parties. Thus, some preliminary steps have to be taken before the step-by-step emulation of the original protocol may start. Specifically, the compiled protocol (which, like the original protocol, is executed over a broadcast channel) proceeds as follows:

1. *Committing to the local input*: Prior to the emulation of the original protocol, each party commits to its input (using a commitment scheme as defined in §C.4.3.1). In addition, using a zero-knowledge proof-of-knowledge (see Section 9.2.3), each party also proves that it knows its own input; that is, it proves that it can decommit to the commitment it sent. (These zero-knowledge proof-of-knowledge prevent dishonest parties from setting their inputs in a way that depends on inputs of honest parties.)

2. *Generation of local random tapes*: Next, all parties jointly generate a sequence of random bits for each party such that only this party knows the outcome of the random sequence generated for it, and everybody else gets a commitment to this outcome. These sequences will be used as the random-inputs (i.e., sequence of coin tosses) for the original protocol. Each bit in the random-sequence generated for Party X is determined as the exclusive-or of the outcomes of instances of an (augmented) coin-tossing protocol (cf. [88, Sec. 7.4.3.5]) that Party X plays with each of the other parties. The latter protocol provides the other parties with a commitment to the outcome obtained by Party X.

3. *Effective prevention of premature termination*: In addition, when compiling (the passively-secure protocol to an actively-secure protocol) *for the model that allows the adversary to control only a minority of the parties*, each party shares its input and random-input with all other parties using a "Verifiable Secret Sharing" (VSS) protocol (cf. [88, Sec. 7.5.5.1]). Loosely speaking, a VSS protocol allows sharing a secret in a way that enables each participant to verify that the share it got fits the publicly posted information, which includes commitments to all shares, where a sufficient number of the latter allow for the efficient recovery of the secret. The use of VSS guarantees that if Party X prematurely suspends the execution, then the honest parties can

together reconstruct all Party X's secrets and carry on the execution while playing its role. This step effectively prevents premature termination, and is not needed in a model that does not consider premature termination a breach of security.

4. *Step-by-step emulation of the original protocol*: Once all the foregoing steps are completed, the new protocol emulates the steps of the original protocol. In each step, each party augments the message determined by the original protocol with a zero-knowledge proof that asserts that the message was indeed computed correctly. Recall that the next message (as determined by the original protocol) is a function of the sender's own input, its random-input, and the messages it has received so far (where the latter are known to everybody because they were sent over a broadcast channel). Furthermore, the sender's input is determined by its commitment (as sent in Step 1), and its random-input is similarly determined (in Step 2). Thus, the next message (as determined by the original protocol) is a function of publicly known strings (i.e., the said commitments as well as the other messages sent over the broadcast channel). Moreover, the assertion that the next message was indeed computed correctly is an NP-assertion, and the sender knows a corresponding NP-witness (i.e., its own input and random-input as well as the corresponding decommitment information). Thus, the sender can prove in zero-knowledge (to each of the other parties) that the message it is sending was indeed computed according to the original protocol.

The above compilation was first outlined in [96, 97]. A detailed description and full proofs appear in [88, Sec. 7.4 and 7.5].

A secure coin-tossing protocol. Using a commitment scheme, we outline a secure (ordinary as opposed to augmented) coin-tossing protocol.

Step C1: Party 1 uniformly selects $\sigma \in \{0,1\}$ and sends Party 2 a commitment, denoted c, to σ.

Step C2: Party 2 uniformly selects $\sigma' \in \{0,1\}$, and sends σ' to Party 1.

Step C3: Party 1 outputs the value $\sigma \oplus \sigma'$, and sends σ along with the decommitment information, denoted d, to Party 2.

Step C4: Party 2 checks whether or not (σ, d) fit the commitment c it has obtained in Step 1. It outputs $\sigma \oplus \sigma'$ if the check is satisfied and halts with output $\perp$ otherwise, where $\perp$ indicates that Party 1 has effectively aborted the protocol prematurely.

Outputs: Party 1 always outputs $b \stackrel{\text{def}}{=} \sigma \oplus \sigma'$, whereas Party 2 either outputs b or $\perp$.

Intuitively, Steps C1–C2 may be viewed as "tossing a coin into the well". At this point (i.e., after Step C2), the value of the coin is determined (essentially

as a random value), but only one party (i.e., Party 1) "can see" (i.e., knows) this value. Clearly, if both parties are honest then they both output the same uniformly chosen bit, recovered in Steps C3 and C4, respectively. Intuitively, each party can guarantee that the outcome is uniformly distributed, and Party 1 can cause premature termination by improper execution of Step 3. Formally, we have to show how the effect of any real-model adversary can be simulated by an adequate ideal-model adversary (which is allowed premature termination). This is done in [88, Sec. 7.4.3.1].

C.7.4 Concluding Remarks

In Sections C.7.1-C.7.2 we have mentioned numerous definitions and results regarding secure multi-party protocols, where some of these definitions are incomparable to others (i.e., they neither imply the others nor are implies by them). For example, in §C.7.1.2 and §C.7.1.3, we have presented two alternative definitions of "secure multi-party protocols", one requiring an honest majority and the other allowing abort. These definitions are incomparable and there is no generic reason to prefer one over the other. Actually, as mentioned in §C.7.1.2, one could formulate a natural definition that implies both definitions (i.e., waiving the bound on the number of dishonest parties in Definition C.17). Indeed, the resulting definition is free of the annoying restrictions that were introduced in each of the two aforementioned definitions; the "only" problem with the resulting definition is that it cannot be satisfied (in general). Thus, for the first time in this appendix, we have reached a situation in which a natural (and general) definition cannot be satisfied, and we are forced to choose between two weaker alternatives, where each of these alternatives carries fundamental disadvantages.

In general, Section C.7 carries a stronger flavor of compromise (i.e., recognizing inherent limitations and settling for a restricted meaningful goal) than previous sections. In contrast to the impression given in other parts of this appendix, it is now obvious that we cannot get all that we may want (and this is without mentioning the problems involved in preserving security under concurrent composition; cf. [88, Sec. 7.7.2]). Instead, we should study the alternatives, and go for the one that best suits our real needs.

Indeed, as stated in Section C.1, the fact that we can define a cryptographic goal does not mean that we can satisfy it as defined. In case we cannot satisfy the initial definition, we should search for relaxations that can be satisfied. These relaxations should be defined in a clear manner such that it would be obvious what they achieve (and what they fail to achieve). Doing so will allow a sound choice of the relaxation to be used in a specific application. This seems to be a good point to end the current appendix.

> *A good compromise is one in which the most important interests of all parties are satisfied.*
>
> Adv. Klara Goldreich-Ingwer (1912–2004)